# Formal Modelling and Verification of Distributed Railway Control Systems

Per Lange Laursen & Van Anh Thi Trinh


Kongens Lyngby 2019

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

# Summary (English)

This project investigates the modelling and model checking of an existing distributed railway control system algorithm with regard to safety. The control system in question is modelled and verified using both UPPAAL and UMC. Based on the this, other variants are created to both explore how easily modifications to a model can be made and how the model can be changed in order to improve the verification time and memory usage.

Experiments show that the first UPPAAL model has been implemented in such a way that the addition of another train operation was relatively easy to implement, and by restricting the allowed sequence of train operations, the verification time and memory usage were greatly improved. When comparing the two modelling languages' verification of the first variant of the control system, UPPAAL turned out to be much more efficient in terms of verification time. The efficiency of UPPAAL even made it possible to verify the railway control system for a real-world railway network. Therefore, UPPAAL is a promising choice of model checking tool when verifying distributed railway control systems for small local networks in the real world.

# Summary (Danish)

Dette projekt undersøger modellering og model tjek af en eksisterende distribueret jernbane-styresystem-algoritme med hensyn til sikkerhed. Det pågældende styresystem er modelleret og verificeret ved brug af både UPPAAL og UMC. Baseret på dette, er også andre varianter lavet for at undersøge hvor let modifikationer kan laves på en model, og hvordan modellen kan ændres for at forbedre verifikationstiden og hukommelsesforbruget.

Eksperimenter viser, at den første UPPAAL-model er blevet implementeret på en måde, der gør det relativt let at introducere en ny tog-operation, og ved at begrænse de mulige sekvenser af tog-operationer, blev verifikationstiden og hukommelsesforbruget markant forbedret. Ved sammenligning af de to modelleringssprogs verificering af den første variant af styresystemet, ses det at UPPAAL er meget mere effektiv med hensyn til verifikationstid. Effektiviteten af UPPAAL gjorde det endda muligt at verificere jernbane-styresystemet for et eksisterende jernbanenetværk. UPPAAL er altså et lovende valg af værktøj til model tjekning af distribuerede jernbane-styresystemer for små lokale netværk fra den virkelige verden.

# Preface

This thesis was written as a conclusion of the master of science degree in *Computer Science and Engineering* at the Technical University of Denmark. The project was carried out in the spring semester of 2019 and supervised by Associate Professor Anne Elisabeth Haxthausen.

The project was created as a continuation of an ongoing investigation of verifying a distributed railway control system using model checking. Railway control systems are safety-critical systems for which it is crucial that certain safety criteria are met. Therefore, we hope that our findings in this project can contribute to further investigation of the distributed railway control system; and that it ultimately leads to an improvement of the railway control systems that exist in our society.

We would like to thank the supervisor of this project, Associate Professor Anne Elisabeth Haxthausen from the DTU Department of Applied Mathematics and Computer Science for her supervision. Her guidance supported us in the completion of the project by encouraging and inspiring us and by continuously providing us with valuable feedback.

Lyngby, 17-June-2019

Per Lange Laursen & Van Anh Thi Trinh

# Contents

Chapter 1

# Introduction

The development of technology and the use thereof have increased steadily for many years such that it is almost impossible to imagine a world without it. From the entertainment industry to the healthcare industry and many more, software has been developed to improve the life and the well-being of people. To reduce the risk of including bugs and errors in software products, they can be tested in many different ways, but the absence of flaws in some software systems is so crucial that *formal verification methods* in particular are favoured since they provide much stronger assertions.

In some cases, errors can simply be corrected when they are found, but this can often be expensive and in the most serious cases, errors can even endanger people. Therefore, it is especially important that *safety-critical systems* work precisely as intended, which is why some standards, like the CENELEC standard EN 50128 for *railway control systems* recommend the use of formal verification methods for the verification of these kinds of systems [ECfES11].

As the name indicates, railway control systems are systems that control railways to ensure the safety of trains and their passengers. The most common railway control systems are *centralized*, which means that a single computer is responsible for controlling the entire system. This solution can, however, be quite expensive to implement and maintain for small local railway networks because communication in centralized control systems require a large number of copper

cables. Furthermore, if the network is ever changed, the new routes that arise entail a required update of the whole system. Because of this, it has long been of interest to examine the possibilities of using *distributed* railway control systems [HP00, FH18, FHN17].

In geographically distributed railway control systems, the geographically distributed railway control components cooperate through communication with each other in order to ensure the safety of the system. These control systems are cheaper to implement, because the need of copper cables is eliminated since communication can happen through wireless links. Additionally, with *plug-and-play* installation, changes in the network only require local changes and the certification process is also simplified [FGH+16]. However, the introduced communication among the distributed railway components also makes it more difficult to verify the system.

*Model checking* in particular, is a popular choice of formal verification method for railway control systems and it has been an ongoing topic for a long time and still is to this day [GH18, VHP17, FH18, BtBF+18]. When model checking a system, one must generally first model the system and then specify the desired properties of it. How this is done depends on the model checking tool that is used. The choice of this tool can greatly affect how well or how easily the system as well as the properties can be formulated and verified since the various tools differ in terms of expressive power, supported features and capability.

## 1.1 Related Work

So far, there exists many examples of formal verification of centralized railway control systems e.g. [HKL+10, JMN+14, LCPT16, VHP17], but there are only a few examples for distributed railway control systems, e.g. [FHN17, HP00, GH18].

The engineering concept considered in this project is based on the one described in [HP00], which is based on the real RELIS 2000 system of INSY GmbH. This concept was first modelled and verified in [HP00] using RSL and the RAISE theorem prover. It has later again been modelled and verified in [GH18], this time using RSL-SAL [PG07] in combination with the SAL symbolic model checker.

The distributed railway control system that was modelled and verified using UMC [TBFGM11] in [FHN17] is an example of an entirely different algorithm that is based on a two-phase commit protocol.

## 1.2 Problem Description

The aim of this project is to model a distributed railway control system, whose safety should be verified for specific railway networks and routes. The generic system is hence given configuration data for a specific network and then verified for that network. The system is first modelled using UPPAAL [BDL04] and then using UMC, but the focus will primarily be on UPPAAL. The experiences obtained through the use of these two model checking tools will then be reported and compared. Verification experiments are also conducted with different railway configurations in order to examine what kind of systems the model checkers are capable of verifying the models for in terms of size and complexity.

## 1.3 Overview

- **Chapter 2 - Background** gives a brief description of the used tools: UPPAAL and UMC.
- **Chapter 3 - Method** describes the general approach with which the models are created and verified.
- **Chapter 4 - Case Study** presents the case study that this project is based on and the specific related requirements.
- **Chapter 5 - Design** describes the general idea of how the first control system variant will be modelled and introduces additional requirements that will simplify the modelling.
- **Chapter 6 - UPPAAL Model of First Control System Variant** describes the modelling of the first control system variant using UPPAAL.
- **Chapter 7 - Formulating Properties for the UPPAAL Model** presents the properties that should be verified for the model described in chapter 6.
- **Chapter 8 - Verifying Safety Properties for the UPPAAL Model** presents various small railway configurations for which the properties presented in chapter 7 should be verified.
- **Chapter 9 - UPPAAL Models for Other Control System Variants** introduces other distributed control system variants that are developed from the variant described in chapter 6.
- **Chapter 10 - UMC Model of First Control System Variant** describes the modelling of the first control system variant using UMC.

- **Chapter 11 - Revised UPPAAL Model: UPPAAL $\times$ UMC** presents another way to model the first control system variant in UPPAAL based on the modelling of it in UMC.
- **Chapter 12 - Eclipse Plugin for Generating Models** describes the implementation of a tool that can be used to automatically generate model code for the different control system variants and for specific railway configurations.
- **Chapter 13 - Verification Experiments** describes and compares verification experiments for the various control system variants and for different railway configurations.
- **Chapter 14 - Discussion** discusses the differences between the model checking tools with regard to the modelling of the control systems and also compares the experiment results with a previous study.
- **Chapter 15 - Conclusion** summarizes the experiences and findings that have been obtained.

Chapter 2

# Background

This chapter gives a general overview over the model checking tools that are used in this project: UPPAAL and UMC. More detailed information about UPPAAL can be found at [Uni] while details about UMC can be found at [Maz].

## 2.1 UPPAAL

UPPAAL was developed as a collaboration between the Department of Information Technology at Uppsala University in Sweden and the Department of Computer Science at Aalborg University in Denmark.

Systems in UPPAAL are modelled as collections of timed automata - or *state machines* - that among other features also support the use of (bounded) integer variables, channel synchronization and urgency. The state machines of a system are created as *instances* of *templates*, which means that a template must be created for every type of state machine that exists in the system, while state machines of the same kind are instances created from the same template. Hence, a template is created by creating a general state machine, but besides this, it also includes a set of *local declarations*, which can be used to store, retrieve and manipulate data.

The state machines themselves consist of *locations*, which are traditionally called states, and *edges*, which are traditionally called transitions. In UPPAAL, the terms 'states' and 'transitions' are instead used to refer to the overall states of the whole system while a transition is the change from one such state to another.

Besides the templates and their local declarations, it is also possible to define *global declarations*. These are very similar to local declarations except for the fact that the data that is declared here is accessible to all instances of a system and variables here can also be changed by any instance.

Finally, to define a system, instances of the templates must be declared in a *system declaration* file.

The modelling of a system can be done using the tool's *editor* while verification can be done using the tool's *verifier*. It is also in the verifier that the desired properties of a system are formulated as queries written in a sub-language of CTL.

## 2.2 UMC

UMC is a model checking tool developed at the FM&&T Laboratory of ISTI-CNR and it is created for modelling and validating UML models in both a state-based and event-based manner.

Similarly to systems modelled in UPPAAL, systems modelled in UMC are modelled as state machines. UMC state machines of the same type are modelled as *classes* from which concrete *objects* are created. These can interact with each other through signal sending and operation calling.

Classes consist of local variables, *states* and *transitions*. In this context, 'states' and 'transitions' refer to the states and transitions of a state machine while the state of the whole system is referred to as a *configuration* and the change from one configuration to another is referred to as an *evolution*.

UMC properties are formulated as formulae in UCTL, which is an extension of CTL that includes both state-based and event-based temporal logic. These formulae rely on the formulation of *abstractions*, which are rules that label configurations and evolutions.

Chapter 3

# Method

This chapter describes the approach that is used to create the first variant of the control system using UPPAAL and how this model can be used to create other variants in the same way. Also the general approach for model checking and experimenting with the variants will be explained.

## 3.1 First Model

The first task of this project is to create a generic model for a distributed railway control system using UPPAAL that can be verified for specific railway networks and routes - i.e. specific configurations. This control system is referred to as the *first control system variant* and the model is referred to as the *first model*. It is designed as the baseline for other variants, which will be tested and compared to the first variant.

The first model is created based on the case study presented in chapter 4. It is then repeatedly refined with a balance between how it *should* be modelled according to the case study and how it *could* be modelled to improve verification efficiency. Generally, it is desired to imitate the real-life problem as well as possible. However, if it is possible to introduce abstractions that simplify or

improve the modelling and/or verification without affecting the core design of the railway control system, this has been done.

Another factor that plays an important role in the decision making during the modelling process is the tool. What *can* be modelled with UPPAAL will control the modelling of the system as well.

## 3.2 Other Models

The first model is designed and implemented in a way that will allow the other variants to be created with only slight changes rather than requiring completely new implementations. The other versions should therefore not differ in terms of implementation details like the choice of data structures. Figure 3.1 shows an overview of all the models created in this project and they will be explained in more details in this section.

**Second Variant in UPPAAL**
The first variation of the first model - i.e. the second variant - is a more *restricted* version. The idea of this is similar to [Gei18, ch. 5.5] where the number of branches in the model's computation tree is reduced. Compared to the first model, there is hence focus on stricter transition guards, so this model's state space will be a subset of the first model's state space.

**Third Variant in UPPAAL**
The third variant is an *extension* of the first model. This model is supposed to preserve all original states while introducing additional functionality.

**Fourth Variant in UPPAAL**
The fourth variant is identical to the first model, with the only change being how configuration data is stored and utilized. It is hence the original case study, but the model is *re-designed* such that certain data is stored differently. The primary purpose of this variant is to determine whether verification time and memory usage can be improved by using a different way to store data.

**First Variant in UMC**
The fifth created model is not a new variant of the control system, but the first control system variant modelled using UMC. The idea with this version is to create it as similar to the first UPPAAL model as possible.

**Fifth Variant in UPPAAL**
The final variant is a *revised* version of the first model. This variant will focus

on mimicking the structure of the UMC model, which could not be modelled exactly like the first model in UPPAAL. The purpose with this is to make a more fair comparison between the first control system variant in UPPAAL and the first control system variant in UMC.

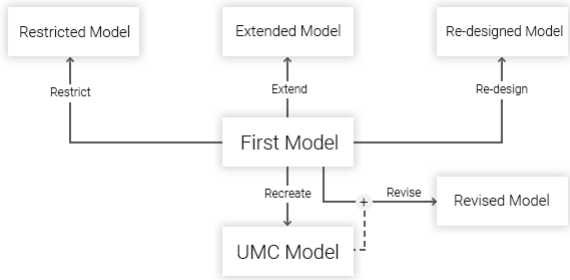



**Figure 3.1:** Created models

## 3.3 General Approach

For each model, the general approach is to create the *model* and formulate the *properties* that are desired for the system (see figure 3.2). The model will then be model checked to satisfy the properties for different concrete configuration data, which should cover all possible situations that may occur in a railway network. If a check fails, the model and/or the properties must be looked into again and changed accordingly. This is repeated until all the checks succeed. When all checks succeed, concrete configuration data is created and used for *experimenting*.

While the checks cover different small scenarios used to find errors in the implementation or formulations of properties, the experiments are used to examine how efficient the verification of the model is. For this reason, the railway networks formulated for this do not focus on capturing different scenarios, but rather on being of different sizes and with different number of trains.

A railway network that imitates a real-world railway network is also modelled to examine how well the models can be verified for real situations.

After the experiments have been performed, it is likely that one discovers parts of the model that can be *optimized*. In these cases, the model is modified again and the whole process is repeated.

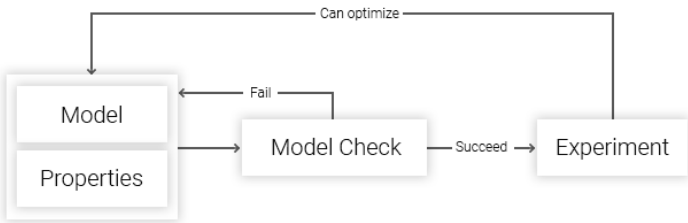



**Figure 3.2:** General approach for modelling and verification

Chapter 4

# Case Study

The design and control strategy of the distributed railway control system in this project is similar to the one described in [HP00] and [GH18], but with minor deviations.

This chapter describes the various railway components that are involved in the distributed railway control system and the strategy that is implemented to ensure the safety of the system. At the end of the chapter in section 4.4, an example of how the control system works is shown.

## 4.1 Railway Networks

A railway network consists of *segments*, which are railway sections that *trains* can move on. A *control box* and a *sensor* are placed at each segment end. The sensors detect passing trains while the trains communicate with the control boxes in order to be able to pass them and thereby enter new segments. Control boxes that are placed at the meetings of three segment ends - i.e. at *points* - are also called *switch boxes* because they are also responsible for switching the point. Finally, every train is equipped with a *train control computer*, which is responsible for the actual communication between the train and the control boxes.

Train control computers and control boxes constitute the control components of the railway control system.

An overview of the components introduced in this section can be seen in figure 4.1. It is assumed that all the railway networks used with this system are small and local railway networks.

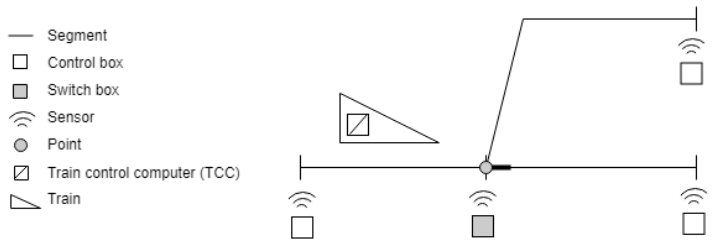



**Figure 4.1:** Railway components

## 4.2 Control Goal

As explained in [Gei18, p. 20], the implemented control strategy must ensure the safety of trains. This means that trains should never *collide* or *derail*. Collision is modelled as two trains being on the same segment at the same time. Derailment is modelled by a train moving from one segment to another segment that is not connected to the former or by a train passing a switching point. It is hence the proper communication between trains - or more precisely, their control computers - and control boxes, which ensure the safety of a system.

## 4.3 Railway Components

**Segments**
Segments are the part of the railway network that determine how a train can physically move from one place to another and consequently which routes are possible for a train to take. Movement on a segment is here allowed in both directions.

Segment ends are either closed ends, meaning that a train cannot move beyond this point, or they can be connected to other segment ends. Each segment

end meets with at most two other segment ends and is connected to at most one segment end. The design here deviates slightly from the one described in [GH18], since that design only included closed segment ends or precisely three meeting segment ends. In the design here, it is also possible that only two segment ends meet. This will allow multiple trains to move on long paths that have no stations.

**Sensors**
The sections in which segment ends meet are called *critical sections*[1]. In some railway control systems, axle counters are placed around these sections to count the number of axles passing by. This information can then be used to determine when a train has left a critical section again after having entered it. Similarly to [GH18, p. 45], a critical section here is modelled as a single geometric point rather than a whole section. A sensor placed at this point is in an *active* state if a train has entered the critical section and in a *passive* state otherwise.

**Control Boxes**
Each closed segment end and group of meeting segment ends in a railway network is associated with the control box placed there. In order for a train to enter a new segment, it first needs to *reserve* that segment at the segment's associated control boxes. This means that when a train needs a *full reservation* of a segment, it needs two reservations since each segment is associated with two control boxes - one at each of its ends. Only when the full reservation is obtained is the train allowed to enter the segment.

To ensure that a point is not switched while a train is moving in the critical section, the point must have been *locked* by the associated switch box. This too must be requested by a train and only the owner of the lock may pass the point.

These interactions with the train mean that a control box must have information about reservations of its associated segments, which train its point (in case of switch boxes) is locked for (if it is locked) and which segments it currently connects (if any). Finally, a control box also knows whether its associated sensor currently detects a passing train or not.

**Points**
Points are placed at the meetings of three segment ends and can be used to switch between two different connections. The *stem* is the fixed part of a point, which means that this part must always be included in the point connection while the other part of the connection may be any of the two other segments. The *plus* position is the position of the point when it connects the stem with the segment that is placed in straight extension of the stem, and the minus position

[1] 'Sections' are here meant as areas and not railway segments.

is the position of the point when it connects the stem with the other (diverging) segment. These segments will be referred to as a point's *stem segment*, *plus segment* and *minus segment* (see figure 4.2).

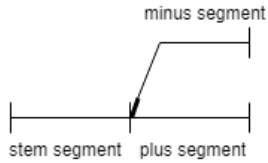



**Figure 4.2:** Segments associated with a point

**Trains and Train Control Computers**
A train has a train control computer (TCC), which is the component that communicates with control boxes as explained earlier. A train's TCC stores information about the train's route, its current position and its currently owned reservations and locks.

The route is represented by lists of

- segments that the train will move on
- control boxes at the ends of the route segments
- switch boxes from the control box list indicating the boxes that need to be switched/locked

These lists are ordered according to the train's route and they are used to know which reservations and locks to obtain and where to get them. It is assumed that a train never passes a segment more than once, so the route segments should all be different.

A train's position is defined by the segment that it is currently occupying, but not where on the segment. Since the railway networks that will be checked are small local railway networks, it can be assumed that a train is always shorter than the shortest segment in the network. A train can therefore be on at most two segments at a time, which is when it is in a critical section. Because the exact position of a train within a segment is not known, it is not certain when a collision happens either, which is why collision is traditionally modelled as two trains being on the same segment as mentioned earlier. This means that two trains on the same segment would be assumed to collide even though they could be very far away from each other, but because it is *possible* that they collide.

The operations that a TCC can perform are the requesting of a segment reservation at a control box and the requesting of a lock at a switch box. The locking here includes the switching of a point into a desired position, since it does not make sense for a TCC to request a connection without getting it locked as well. While the introduced operations are initiated by the TCC, the train itself also triggers an operation when it passes a sensor because the reservations (and lock) obtained for passing the associated control box have to be cleared when the train leaves the segment.

In real systems, a TCC can also block its train's engine if the train enters a critical section or leaves a station without permission [HP00, p. 4]. In such cases, the emergency brake is triggered, but these scenarios will not be included in the model here. It is therefore assumed that trains do not move without permission.

### 4.3.1 Operations

In this section, the introduced operations have been listed in tables with their requirement details. The *sender* is the component that initiates the communication and the *receiver* is the component that the sender component communicates with for the operation. For the operation to complete successfully, the requirements must be fulfilled.

The sender requirements must be fulfilled before the sender even attempts to start the operation while the receiver requirements must be fulfilled for the receiver to send back an acknowledgement, otherwise a negative acknowledgement is returned.

**Request reservation of segment**
As long as a train has more segments in its route that it has not yet passed or reserved, it may request a reservation of these segments. A segment reservation request is sent to the control box at one of the segment ends. A segment has been fully reserved by a train if the train has reserved the segment at the control boxes at the two ends of the segment. Table 4.1 shows the requirements for a successful reservation.

If a reservation is successful, both the TCC and the control box update their state spaces to reflect the newly created reservation.

| Sender | TCC |
|---|---|
| Receiver | Control box |
| TCC Requirements | • The TCC does not currently have a reservation of the segment at the control box<br>• The control box is a part of the train's route<br>• The segment is a part of the train's route |
| Control box requirements | • The control box is placed at one of the segment's ends<br>• The segment is not already reserved at the control box |

**Table 4.1:** Requirements for reservation of a segment

**Request switching and locking of point for the connection between two segments**
When a train passes a point, it is important that the point is in the correct position leading the train in the direction of its route. The responsible switch box may have to switch the point, wait for it to finish switching and then lock the point, so it cannot be switched again before the train has passed the point (see table 4.2).

| Sender | TCC |
|---|---|
| Receiver | Switch box |
| TCC Requirements | • The TCC does not currently have a lock at the switch box<br>• The switch box is a part of the train's route<br>• The segments that the TCC wants to have connected are adjacent segments in the train's route<br>• The TCC must have the reservations for the segments that it wants to have connected at the switch box |
| Control box requirements | • Exactly one of the segments that the TCC wants to have connected is the switch box' plus segment or minus segment<br>• Exactly one of the segments that the TCC wants to have connected is the switch box' stem segment<br>• No train is in the switch box' critical section - i.e. its sensor is passive<br>• The point is not already locked |

**Table 4.2:** Requirements for switching and locking a point

For a point to be switched, it must also not already be in the process of switching. This, however, is never possible since a point can only be switched by its associated switch box. The case that a switch box tries to switch a point that is being switched by another switch box will therefore not exist in the system. A switch box would also never initiate a switch if its last switch initiation has not finished.

Like for reservations of segments, if a locking is successful, both the TCC and the switch box update their state spaces to reflect that the switch box has locked its point for the train.

**Pass control box**
When a train moves from one segment to another segment, it always passes a control box when it enters and leaves the critical section. Before doing this, the TCC has to check that it has a full reservation of the upcoming segment. If the upcoming control box is a switch box, it also has to ensure that the associated point has been switched if necessary and locked (see table 4.3).

| Sender | Train (TCC) |
|---|---|
| Receiver | Sensor |
| TCC requirements | • The train's TCC must have the lock at the upcoming control box if the control box is a switch box<br>• The upcoming control box is not the last one on the train's route<br>• The train's TCC must have the reservation for the next segment at the control boxes at each end of the segment |

**Table 4.3:** Requirements for passing a control box

When a train leaves the critical section, the sensor that it passes becomes passive so the associated control box knows that it now has to clear the reservations of the currently connected segments as well as the lock. The TCC either uses the Global Positioning System (GPS) or track component signals to determine where it is and thereby when it has left the critical section [HP00, p. 1]. It then also updates its own state space with regard to its new position and its reservations and locks.

## 4.4 Example

In this section, an example of a process will be presented using a small railway network with a single train. The example is an extended version of [Gei18, ch. 4.2] where the use of a regular control box - i.e. a control box that is not a switch box - is also demonstrated.

### 4.4.1 Initial State

In the initial state (see fig. 4.3), a train t0 is positioned on the segment s0, which is the first segment in its route. To be able to be on this segment, the segment must be reserved at the upcoming control box (cb1), otherwise other trains would be able to enter the segment.

It can be seen in the example that the control box route includes the very first control box behind the train as well (cb0), even though this is actually

never used. The very first control box could hence have been left out, but it is convenient to include it if the train ever needs to return to its initial location after having arrived at its destination.

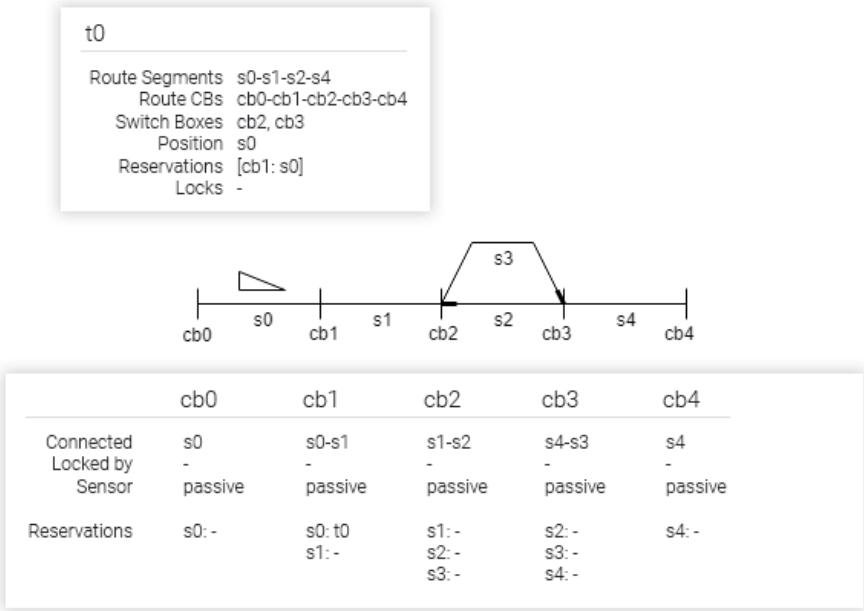



**Figure 4.3:** Initial state

### 4.4.2 Steps

1. t0 sends a request to cb1 for the reservation of s1.

2. Since s1 is not already reserved for another train, cb1 returns an acknowledgement to t0 and updates its state space by saving the new reservation of s1 for t0.

3. t0 receives the acknowledgement and saves the new reservation.

4. t0 sends a request to cb2 for the reservation of s1.

5. Since s1 is not already reserved for another train, cb2 returns an acknowledgement to t0 and updates its state space by saving the new reservation of s1 for t0.

6. t0 receives the acknowledgement and saves the new reservation.

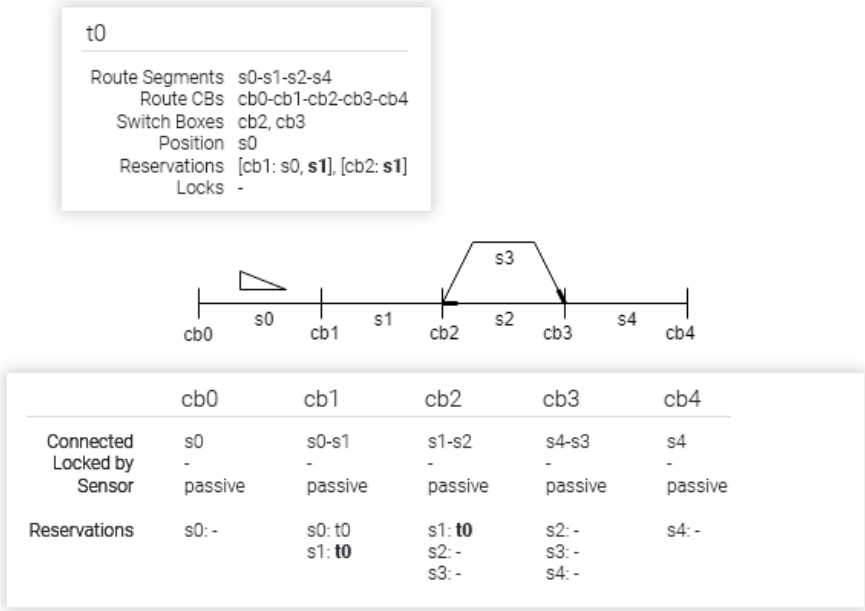



**Figure 4.4:** The state after steps 1-6

7. t0 enters the critical section of cb1. This is possible because cb1 is a regular control box and because t0 has the full reservation of s1.

8. cb1's sensor registers that a train has entered its critical section so the sensor becomes active.

9. t0 leaves the critical section of cb1 and updates its state space by setting its position to s1 and by removing the reservations that it had at cb1.

10. cb1's sensor registers that a train has left its critical section so the sensor becomes passive.

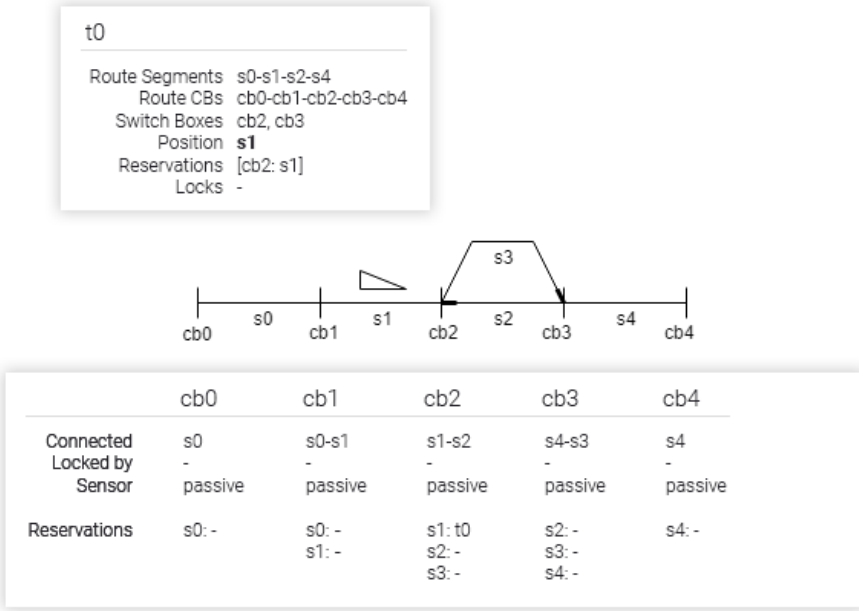



**Figure 4.5:** The state after steps 7-11

11. cb1 updates its state space by removing t0's two reservations.

12. t0 sends a request to cb2 for the reservation of s2.

13. Since s2 is not already reserved for another train, cb2 returns an acknowledgement to t0 and updates its state space by saving the new reservation of s2 for t0.

14. t0 receives the acknowledgement and saves the new reservation.

15. t0 sends a request to cb3 for the reservation of s2.

16. Since s2 is not already reserved for another train, cb3 returns an acknowledgement to t0 and updates its state space by saving the new reservation of s2 for t0.

17. t0 receives the acknowledgement and saves the new reservation.

18. t0 requests a lock for the connection between s1 and s2 at cb2 since cb2 is a switch box.
19. Since cb2's point is not already locked, cb2's sensor is passive and cb2 is associated with the two segments, which are already connected, cb2 locks its point, sets its locked-by value to t0 and returns an acknowledgement to t0.
20. t0 receives the acknowledgement and saves the lock.

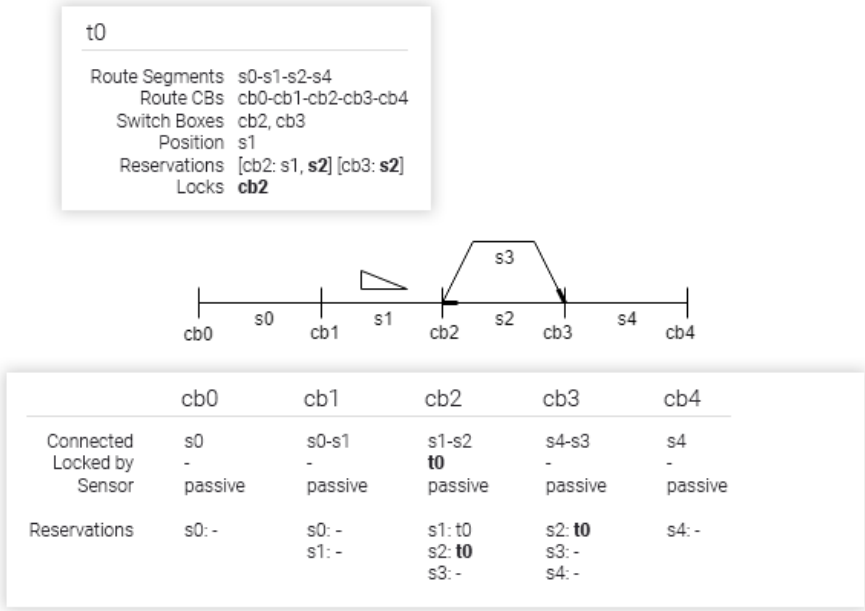



**Figure 4.6:** The state after steps 12-20

21. t0 enters the critical section of cb2. This is possible because it has the lock at cb2 and the full reservation of s2.
22. cb2's sensor registers that a train has entered its critical section so the sensor becomes active.
23. t0 leaves the critical section of cb2 and updates its state space by setting its position to s2 and by removing the reservations and lock that it had at cb2.

24. cb2's sensor registers that a train has left its critical section so the sensor becomes passive.

25. cb2 updates its state space by removing t0's two reservations and by removing the lock.

26. t0 sends a request to cb3 for the reservation of s4.

27. Since s4 is not already reserved for another train, cb3 returns an acknowledgement to t0 and updates its state space by saving the new reservation of s4 for t0.

28. t0 receives the acknowledgement and saves the new reservation.

29. t0 sends a request to cb4 for the reservation of s4.

30. Since s4 is not already reserved for another train, cb4 returns an acknowledgement to t0 and updates its state space by saving the new reservation of s4 for t0.

31. t0 receives the acknowledgement and saves the new reservation.

32. t0 requests a lock for the connection between s2 and s4 at cb3 since cb3 is a switch box.

33. Since cb3's point is not already locked, cb3's sensor is passive and cb3 is associated with the two segments, which are not yet connected, cb3 switches and locks its point, sets its locked-by value to t0 and returns an acknowledgement to t0.

34. t0 receives the acknowledgement and saves the lock.

35. t0 enters the critical section of cb3. This is possible because it has the lock at cb3 and the full reservation of s4.

36. cb3's sensor registers that a train has entered its critical section so the sensor becomes active.

37. t0 leaves the critical section of cb3 and updates its state space by setting its position to s4 and by removing the reservations and lock that it had at cb3. Train t0 has arrived at its destination.

38. cb3's sensor registers that a train has left its critical section so the sensor becomes passive.

39. cb3 updates its state space by removing t0's two reservations and by removing the lock.

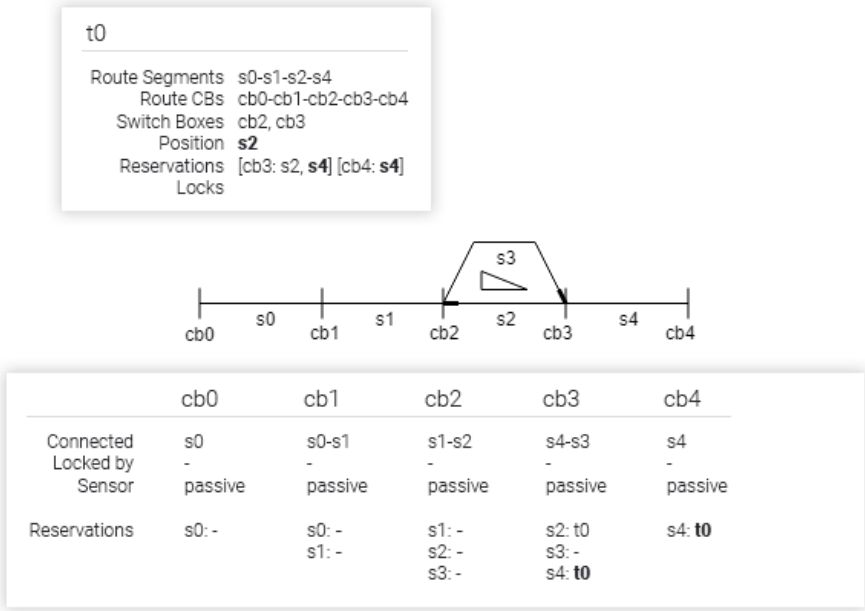



**Figure 4.7:** The state after steps 21-31

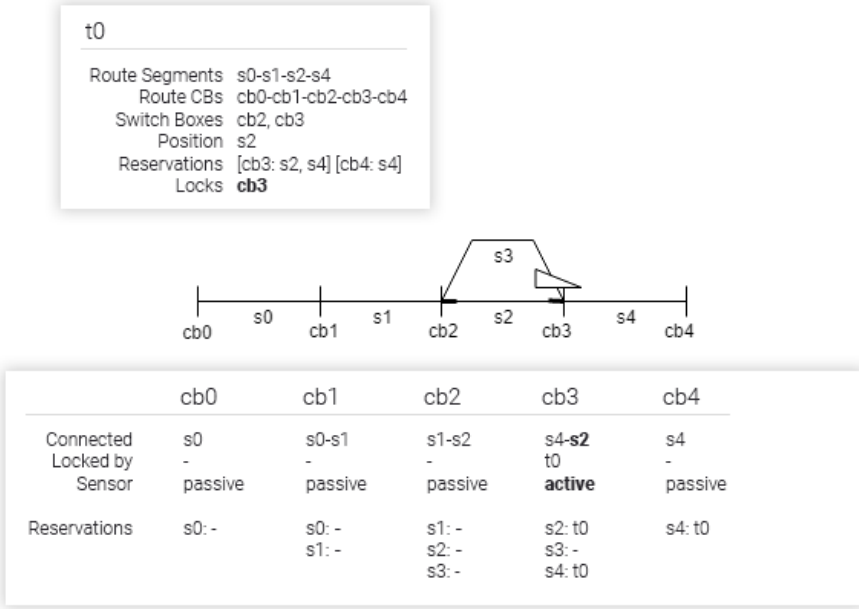



**Figure 4.8:** The state after steps 32-36

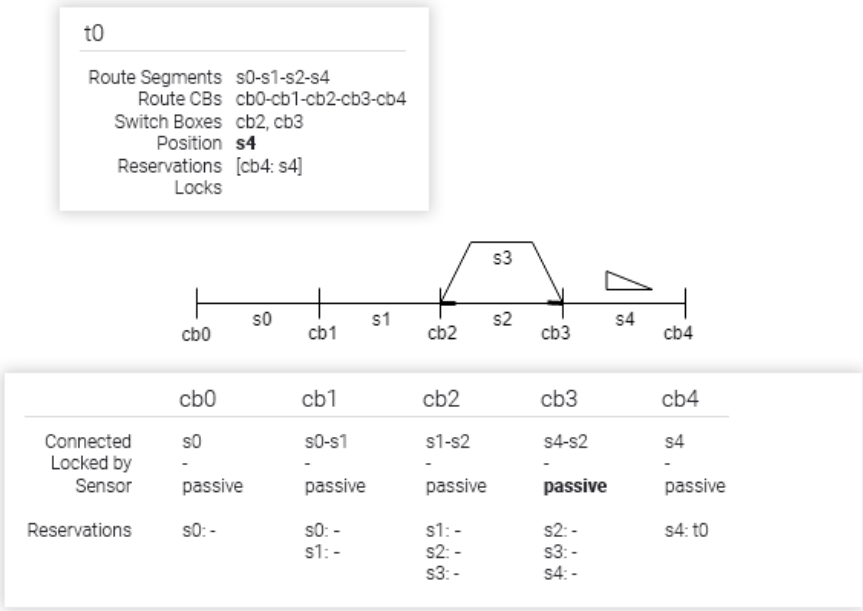



**Figure 4.9:** The state after steps 37-39

Chapter 5

# Design

In the previous chapter the railway components were described as well as how they interact with each other in order to ensure the safety of trains. This chapter discusses the design decisions that were made during the modelling of these components and the railway control system in general when modelling the system in UPPAAL.

## 5.1 Modelling Railway Components

The railway components of a distributed railway control system were trains, TCCs, regular control boxes, switch boxes, sensors, points and segments. Based on the introduced operations, TCCs and control boxes should naturally be modelled explicitly, since they are directly involved in most of these communication schemes. By modelling them as separate UPPAAL templates, it is possible to implement communication between them as channel synchronization while the local variables can be used as their state spaces.

Although the 'pass' operation is not actually a communication between a train and a sensor but an action that is detected by a sensor, it can still be modelled as a synchronization on a channel. It would therefore make sense to model trains

and sensors as templates as well. Furthermore, because the safety of trains depend on a train's actual position in a railway network, this too is important to model since the current position stored by a TCC is not necessarily in accordance with reality. For example, if a train is passing a switch box and the segments that the TCC wanted to have connected are not actually connected, the TCC may think that the train ended up on the segment that it was supposed to end up on while the train is actually on another segment. This means that when a train leaves a critical section, not only are the TCC and control box state spaces updated, but a train's actual position is updated as well. Trains and TCCs could therefore be modelled separately, but since a TCC is a part of a train anyway and the only train property that is needed in this model is its position, it is more convenient and efficient to model them together, but still distinguish between the TCC position and the actual train position.

As for sensors, these do not need to be modelled separately either because the communication between a sensor and its control box happens almost instantly since the connection is wired. This means that a sensor can be seen as a part of its control box similarly to how a TCC is a part of a train. The communication between a train and a sensor is hence simplified to the communication between the train and the sensor's control box.

Control boxes and switch boxes can been modelled separately as well, but switch boxes are also control boxes so creating a separate template for them could easily result in inconsistency between the control box template and the part that needs to be similar in the switch box template. Instead, the control box template can be modelled such that it also has switch box functionalities, which should then be skipped or ignored if the control box is not a switch box.

Points did not appear directly as a sender or a receiver in the introduced operations, but they actually take part in communication with switch boxes when a switch box switches a point. Modelling a point as an explicit template simplifies the modelling of its state, i.e. whether it is in its plus position, minus position or in the middle of switching. By also introducing intermediate states, it is possible to easily check that there are no trains in a critical section while the point there is switching since it does take some time for a point to switch.

Lastly, since segments do not actually interact with any of the other railway components or have their own state spaces, they do not need to be modelled with an explicit template. They are instead simply be modelled by integers for their IDs.

The design is summarized in figure 5.1 where nested boxes indicate the modelling of multiple components as one.

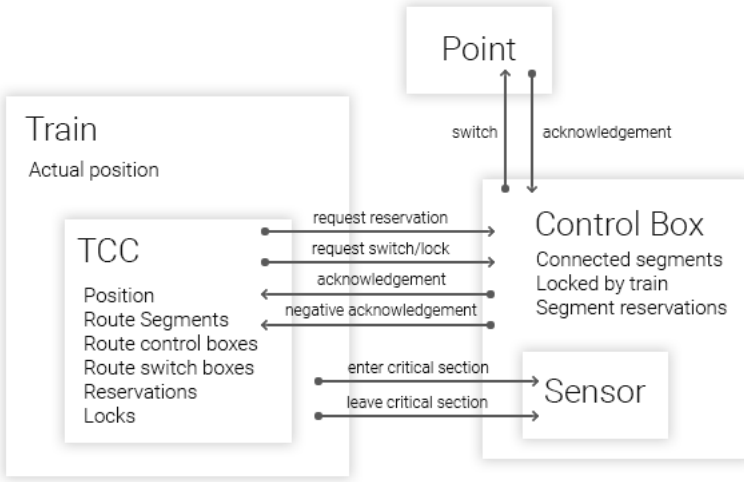



**Figure 5.1:** Design of railway components and their interactions

## 5.2 Additional Requirements

The distributed railway control system described in the previous chapter is a general and flexible system with requirements that mainly ensure the safety of trains. However, the purpose of requirements such as *'The segment is a part of the train's route'*, is to create certain limitations for TCCs because they do not need complete freedom of choice. For example, concerning the mentioned reservation request requirement, a TCC never needs the reservation of a segment that is not a part of its route anyway.

Adding these kinds of requirements reduces the number of possible executions in a system so with regard to model checking, this can also reduce the state space remarkably. Besides the requirements presented in section 4.3.1, which are the original requirements from [Gei18], the model here will include additional requirements that can contribute to this state space reduction.

### 5.2.1 Order of Reservations and Locks

As discussed in [Gei18, p. 29], *when* reservations and locks are obtained by a train affects other trains' progress in the railway network. Reserving and locking

too far ahead can possibly block the progress of other trains while only reserving at the control box just ahead can result in a livelock if two trains obtain the reservation of the same segment at different control boxes. One thing that affects when a TCC makes a request is the *order* in which reservations and locks are obtained.

In the model described so far, a TCC can request the reservation of any segment in its route and request the switching/locking of any point in its route. Although these reservations and locks will be required at some point along the way, it does not actually make sense for a TCC to request a reservation of a segment far ahead if it cannot obtain or has not obtained the reservations of the segments before that. Additional requirements that can limit the choice of segment to reserve and point to lock can therefore be formulated as:

- A TCC only requests a reservation of a segment if it has fully reserved all segments in its route between that segment and its current position.
- A TCC only requests a reservation of a segment at a control box if the control box is at the segment end from which the train will enter the segment, or if the segment has already been reserved at the control box at the end from which the train will enter the segment.
- A TCC only requests a switching/locking of a point if it has obtained all the needed locks in its route between that point and its current position.

The requirements mean that a TCC will now always make requests in the order of the train's route, which again means that a train will always know exactly which segment to reserve next and where to reserve it and which lock it should request next.

Another thing that can affect when a TCC makes a request is *how many* reservations and locks a TCC can have at a time.

### 5.2.2 Number of Reservations and Locks

The number of allowed reservations and locks affects when a TCC can make a request because it limits the number of requests. A train would for instance not be able to reserve its whole route from the beginning if it has a long route while being limited to have only one reservation at a time. This limit can therefore prevent trains from blocking the progress of other trains far away.

To allow variations of the control algorithm with regard to the number of reservations and locks that a train may possess, a limit setting is used for each such that one can adjust the limits as needed.

Because it does not make sense for a TCC to have a reservation of a segment at only one control box (besides the one it is standing on), the limit for segment reservations should be the number of *full* segment reservations. For example, if the segment reservation limit is two, a TCC is allowed to have zero, one or two full reservations. This corresponds to up to four reservations besides half the reservation of the segment that the train is currently positioned on.

Since locks are not related to each other as reservations of the same segment at different control boxes are, the lock limit simply is the number of maximum locks that a TCC may have. Although this does not need to be related to the reservation limit, a lock limit is always indirectly bound by the reservation limit because of the lock request requirement: *'The TCC must have the reservations for the segments that it wants to have connected at the switch box'*. A lock limit larger than a reservation limit will therefore not result in more states than a lock limit equal to the reservation limit.

The two limits result in the additional requirements for the requesting of a segment reservation and switching/locking respectively:

- A TCC only requests a reservation of a segment if it does not already have the maximum allowed number of reservations.
- A TCC only requests the switching/locking of a point if it does not already have the maximum allowed number of locks.

Chapter 6

# UPPAAL Model of First Control System Variant

Based on the requirements specified in chapter 4 and 5, the railway control system can finally be modelled in UPPAAL. This chapter first describes the modelling of the railway control system in details. Section 6.8 then describes the requirements that the configuration data has to fulfil in order for it to be well-formed and thereby used in the first model.

## 6.1 Templates

The UPPAAL model consists of the four templates:

- *Train*: Models a train and its train control computer
- *CB*: Models a control box
- *Point*: Models a railway point
- *Initializer*: An auxiliary template used to initialize the network

The three first templates model the different railway components as described in the previous chapter. In order to model a concrete network, instances of each template must be created. For example, an instance of the `Train` template is created for each train/TCC and an instance of the `CB` template is created for each control box/sensor.

The last template is not a railway component like the other templates, but an auxiliary template, which is used to initialize a network in a single step. This template is explained in more details in section 6.2.7. The state machines of the templates are shown in the following sections and the declarations can be seen in appendix A.

## 6.2 Global Declarations

The global declarations consists of configuration data specific to a network as well as type declarations, and channels. These are described in this section while an example of concrete configuration data can be seen in appendix H.

### 6.2.1 Size Constants

The configuration data that has to be declared first concerns the size of the network since the size is used for the remaining declarations. The size is defined by the number of trains `NTRAIN`, the number of control boxes `NCB`, the number of points `NPOINT`, the number of segments `NSEG` and the number of segments in the longest route in the network `NROUTELENGTH`.

The sizes are defined as constants in order to be computable at compile time and thereby usable in the other declarations.

```
const int NTRAIN = ...; //Number of trains
const int NCB = ...; //Number of control boxes
const int NPOINT = ...; //Number of points
const int NSEG = ...; //Number of segments
const int NROUTELENGTH = ...; //Maximum length of a route (in number of
    segments)
```

### 6.2.2 Types

Every railway component template takes an ID as parameter, which is used to identify the different instances. In order to have an arbitrary number of

instances generated automatically, each ID is a bounded integer ranging from 0 to the relevant size constant minus 1 since the numbering is zero-based. Most of the type declarations are bounded integers that can be used to improve the readability of IDs. For example, the type for train IDs `t_id` is used as the ID parameter type for the `Train` template, and `cB_id` and `p_id` can similarly be used as the ID parameter types for `CB` and `Point` respectively. A type for segment IDs `seg_id` is also defined and later used to both identify and represent segments.

Each of the four types also have another variant, `tV_id`, `cBV_id`, `pV_id` and `segV_id`. The difference between these and their original types is that their ranges start at -1 rather than 0. These variants are used as array types in the remaining configuration data where the number -1 is interpreted as 'no value'.

```
typedef int[0, NTRAIN-1] t_id; //Train IDs
typedef int[0, NCB-1]  cB_id; //Control box IDs
typedef int[0, NPOINT-1] p_id; //Point IDs
typedef int[0, NSEG-1] seg_id; //Segment IDs
typedef int[-1, NTRAIN-1] tV_id; //Train IDs with -1
typedef int[-1, NCB-1] cBV_id; //Control Box IDs with -1
typedef int[-1, NPOINT-1] pV_id; //Switch IDs with -1
typedef int[-1, NSEG-1] segV_id; //Segment IDs with -1
typedef int[0, NROUTELENGTH] cBRoute_i;
    //Indices of arrays of length NROUTELENGTH
typedef int[0, NROUTELENGTH-1] segRoute_i;
    //Indices of arrays of length NROUTELENGTH-1
```

The two types with the suffix '`_i`' are bounded integers in the ranges 0 to `NROUTELENGTH` and `NROUTELENGTH-1`. These can be used as the types for pointers into arrays of such lengths.

A final record type `reservation` is used for setting the initial reservations since a train that is placed on a segment must always have the reservation for that segment at the upcoming control box. This type consists of the ID `cb` of the control box and the ID `seg` of the reserved segment.

```
typedef struct {
    cB_id cb;
    seg_id seg;
} reservation;
```

### 6.2.3 Limits

The maximum number of reservations and the maximum number of locks that a train may have at a time are defined by the constants `resLimit` and `lockLimit` respectively.

`resLimit` denotes the number of *full* reservations that a train is allowed to have at a time. The smallest non-empty network consists only of a single segment and a train. In this network the train does not need any full reservations, so the lower bound of `resLimit` is 0. Since a train can only reserve the segments in its route, the upper bound of `resLimit` is `NROUTELENGTH`.

Since railway networks do not necessarily need to have any switch boxes, the lower bound of `lockLimit` is also 0 and the upper bound is `NROUTELENGTH-1`. This is because `NROUTELENGTH` includes a control box per segment in a route, but a train never needs a lock at the control box behind it on its first segment, nor at the last control box in its route.

```
const int[0,NROUTELENGTH] resLimit = ...;
const int[0,NROUTELENGTH-1] lockLimit = ...;
```

### 6.2.4 Train and TCC Data

The remaining configuration data is mainly data belonging to TCCs and control boxes.

As previously seen, TCCs store routes in terms of segments and control boxes. To store collections of data, UPPAAL only offers arrays, whose sizes must be stated in the declarations of them. Declaring route arrays in the `Train` template then means that all `Train` instances must have route arrays of the same lengths.

To make sure that every train's route can fit into the array, the array length should be the maximum route length found among all trains. A train's segment list is therefore implemented as an `NROUTELENGTH` long array of `segV_id`s. It contains the segments' IDs and it uses -1 as padding if the route is shorter than `NROUTELENGTH`. A train's control box list is for the same reason implemented as an `NROUTELENGTH+1` long array of type `cBV_id`. The control box list is longer than the segment list because there is always one control box more in a route than there are segments.

Ideally, this data should only be a part of `Train`s' local declarations and therefore given as arguments when instantiating them. However, this means that `Train` instances can no longer be instantiated automatically, which also complicates the verification process slightly as explained in chapter 7. Because of these drawbacks, all segment arrays are saved in a global array `segRoutes` and all control box arrays are saved in the global array `boxRoutes`. Each `Train` instance can then retrieve its segment list and control box list during an initialization step - similar to how new route data must be installed on a TCC before it can be used.

```
const segV_id segRoutes[NTRAIN][NROUTELENGTH] = ...;
const cBV_id boxRoutes[NTRAIN][NROUTELENGTH+1] = ...;
```

### 6.2.5 Control Box Data

The data concerning control boxes also models the layout of the railway network, i.e. how segments are or can be connected. Control boxes are either placed by closed segment ends, by two meeting segment ends or by three meeting segment ends. Each control box is therefore represented by an array of length three of type `segV_id`.

The values in a control box array is used to determine which segment end(s) a control box is placed by and hence which kind of placement it has. The valid definitions and corresponding meaning can be seen in table 6.1.

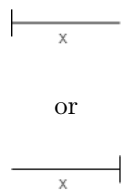
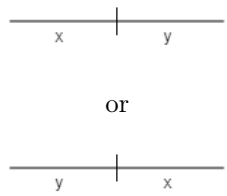
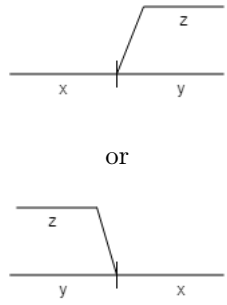
| Definition | Placement | Description |
|---|---|---|
| `{x,-1,-1}` |   | The control box is placed at the segment's closed end |
| `{x,y,-1}` |   | The control box is placed at the meeting of two segments |
| `{x,y,z}` |   | The control box is a switch box |

**Table 6.1:** Valid control box definitions

All control box definitions are saved in one global array `cBs` and each `CB` instance retrieves its own data during the initialization step.

```
const segV_id cBs[NCB][3]= ...;
```

The initial reservations at control boxes are saved in the array `initialRes` of type `reservation`. This array has length `NTRAIN` because there is initially one reservation per train, i.e. `initialRes[i]` represents the initial reservation of the `Train` with ID `i`. This information is also retrieved during the initialization step.

```
const reservation initialRes[NTRAIN] = ...;
```

### 6.2.6 Point Data

To create the connection between switch boxes and their associated points, each switch box has the ID of the point that it controls. An `NCB` long array `points` of type `pV_id` is used to store these. If the `CB` with ID `i` has an associated `Point`, `points[i]` is the ID of that `Point`, otherwise `points[i]` is -1.

Whether a point initially connects the stem with the plus segment or the minus segment is set in the `NPOINT` long boolean array `pointInPlus`. If `pointInPlus[i]` is `true`, the `Point` with ID `i` connects the stem with the plus segment, otherwise it connects it with the minus segment. As the only global declaration, this is not a constant since it is possible for a point to change its position.

```
const pV_id points[NCB] = ...;
bool pointInPlus[NPOINT] = ...;
```

Intuitively, `NPOINT` should have the value 0 if a network does not have any points, but since this constant is included in integer bounds like `[0, NPOINT-1]` and as an array size, it cannot be 0. In these cases, one can instead give it any other positive value (e.g. 1) and put any value(s) into `pointInPlus`. This is because the values in `pointInPlus` do not matter if the `points` array is correctly initialized with only the value -1. Even though the value of `NPOINT` does not matter if all values in `pointInPlus` are -1, it is most optimal to set `NPOINT` to the lowest possible number, since `NPOINT` `Point` instance(s) will still be created.

### 6.2.7 Channels

As mentioned before, the different instances of the templates communicate through synchronization on channels. This means that two instances can only communicate if the matching synchronization edges are enabled at the same time. It is therefore not possible to send a message at all if no instance is ready to receive it. As a result of this, it is assumed that messages are never lost and there is no delay in communication.

Every operation is implemented as an array of channels, with a channel for each instance of the template that receives the operation emission. Some operation may also require that additional data is sent during the synchronization. This too can be implemented by using a specific channel for it.

For example, for the reserve operation, a `Train` needs to send along the ID of the segment it wants to reserve as well as its own ID. This means that the `Train` in question has to emit on the channel for itself, the specific `CB` that should receive the emission, and the specific segment that is requested. There are hence `NCB` $\times$ `NTRAIN` $\times$ `NSEG` channels in total for the reservation of segments.

Similarly, there are `NCB` $\times$ `NTRAIN` $\times$ `NSEG` $\times$ `NSEG` channels for the `reqLock` operation. These are used for a `Train` to request a lock at a `CB` for a connection between two segments. The `Train` ID again has to be sent for identification and the segment IDs are sent since the receiving `CB` may have to switch its associated `Point`.

The last channel arrays that `Train`s can use to initiate synchronization with `CB`s are `pass[NCB]` and `passed[NCB]`. This communication models the detection of a train passing a control box, i.e. when it enters and leaves a critical section.

When a `Train` has requested a reservation of a segment or requested a lock, the receiving `CB` must respond with either an acknowledgement or a negative acknowledgement. This is necessary, since the `Train` needs to know if it should update its state space or not. For this, the channels `OK[i]` and `notOK[i]` are used. `i` is here the ID of the receiving `Train` instance.

For the communication between `CB`s and `Point`s, there are two channel arrays: `switchPoint` and `OKp`. If `i` is the ID of a `CB`'s `Point`, the `CB` switches the `Point` by synchronizing with it on the channel `switchPoint[i]`. When the `Point` has completed switching, it can then send back an acknowledgement on the channel `OKp[i]`. This acknowledgement is used for the `CB` to know when it can send an acknowledgement back to the `Train` during the processing of a lock request.

```
chan reqSeg[NCB][NTRAIN][NSEG];
```

```
chan reqLock[NCB][NTRAIN][NSEG][NSEG];
chan OK[NTRAIN];
chan notOK[NTRAIN];
chan pass[NCB];
chan passed[NCB];
chan switchPoint[NPOINT];
chan OKp[NCB];
urgent broadcast chan start;
```

A final channel `start` is used for the initialization step. This channel is an urgent broadcast channel since this ensures that all instances initialize immediately[1] and in one step, which will eliminate all the interleavings that would otherwise be possible if instances had to initialize in turn.

### 6.2.8 Functions

When a train moves past a switch box, it is not certain which segment it will move on next since this depends on the current position of the associated point. In the real world, a train would simply follow the connection of the physical tracks, but since there are no physical tracks defined here, this concept is modelled with a global function `nextSegment`.

Given the ID of the `CB` that is being passed by a `Train` and the segment that the `Train` is coming from, the function returns the segment that the `Train` will be moving onto next. For a regular `CB` with ID `cb`, this means that the function will return either `cBs[cb][0]` or `cBs[cb][1]` - whichever segment that is not the same as the one the `Train` is coming from. If instead the `CB` is a switch box (`points[cb] > -1`), then it is also possible that `cBs[cb][2]` is returned. This depends on the associated `Point`'s position (`pointInPlus[points[cb]]`).

```
int nextSegment(cB_id cb, seg_id s){
    int s1 = cBs[cb][0];
    int s2 = cBs[cb][1];
    if(points[cb] > -1 && !pointInPlus[points[cb]]){
        s2 = cBs[cb][2];
    }

    if(s == s1){
        return s2;
    } else {
        return s1;
    }
}
```

[1] The `urgent` keyword is not necessary here, but if a real-time aspect is later introduced, the initialization synchronization has to be urgent or committed in order to ensure no delay.

## 6.3 The Initializer Template


**Figure 6.1:** The `Initializer` state machine

The `Initializer` template is fairly simple; it has no data and only the two locations `Uninitialized` and `Initialized` and an edge between them (see figure 6.1). The whole initialization step is started when the `Initializer` emits on the `start` broadcast channel from the `Uninitialized` location. It then enters the `Initialized` location and is not used any further. Since `start` is a broadcast channel, all the instances of the other templates initialize at the same time when the `Initializer` emits on the channel.

## 6.4 The Train Template

This section first introduces the state space of the `Train` template in 6.4.1 and explains how it is initialized. Section 6.4.2 then describes the modelling of the various train operations.

### 6.4.1 State Space and Initialization

As seen in figure 6.2, the state machine for the `Train` template starts in the location `Initial`, and initializes through the edge to the location `SingleSegment`, which happens when it receives an emission on the `start` channel and if the guard function `isWellFormed` returns true. This guard is explained in details in section 6.8. The initialization process is executed in the function `initialize`.

**Route Information**
The `initialize` function includes, among other things, the copying of a train's route segments and route control boxes from the global arrays to the local arrays `segments` and `boxes` respectively.

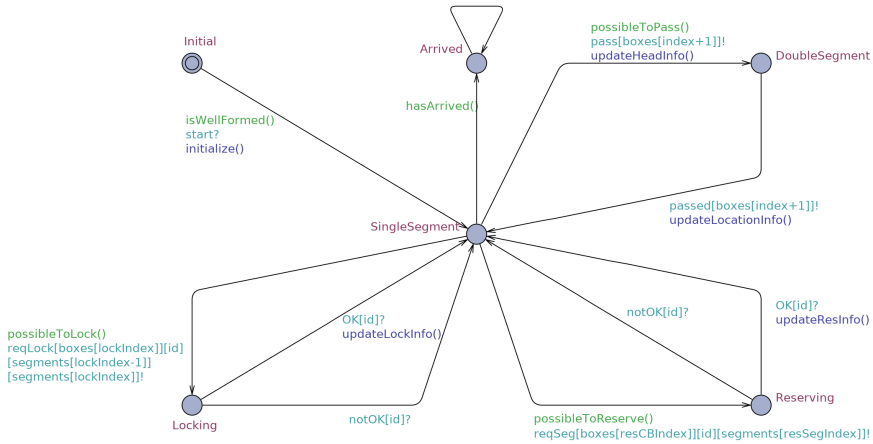



**Figure 6.2:** The `Train` state machine

```
segV_id segments[NROUTELENGTH];
cBV_id boxes[NROUTELENGTH+1];
```

In order for a `Train` to know when it needs a lock to pass a control box, it also needs information from the global `points` array. It saves this information in its boolean array `requiresLock`. If a control box at index `i` in `boxes` is a switch box, `requiresLock[i]` is set to true.

```
bool requiresLock[NROUTELENGTH+1];
```

The function also finds the actual length `routeLength` of the route, which is used to determine when a train has arrived at its destination.

```
int[0,NROUTELENGTH] routeLength;
```

### Train Position

After having copied the segment route, `initialize` saves the first segment ID in the route as the current segment `curSeg`. `curSeg` represents the train's *actual* current position. A train is hence always assumed to start on the first segment in its route. The position that the TCC *believes* that the train has is indicated by the variable `index` so `segments[index]` is the segment that the TCC believes that the train is currently on.

```
cBRoute_i curSeg;
segRoute_i index = 0;
```

Besides curSeg, a variable headSeg is also used in relation to a Train's actual position. This variable is only used when a Train is in a critical section and therefore possibly positioned on two different segments. In this case, headSeg is set to the ID of the segment that the train is about to move to while curSeg then becomes the segment ID of the segment that the train's tail is possibly on. In all other situations, headSeg has the value -1.

```
segV_id headSeg = -1;
```

### Reservation States

The variable resSegIndex is also set to 1 in the initialization function. This index points to the segment in the segments array (i.e. segments[resSegIndex]) that the Train has not yet reserved but wants to reserve next. Its type is cBRoute_i because it may also exceed the segments array. If it exceeds the length of segments, then this will be interpreted as no segments left to be reserved.

```
cBRoute_i resSegIndex = 0;
```

### Lock States

Before the initialization is complete, the Train must also set its variable lockIndex. This is similar to resSegIndex, but it points to the control box in boxes that the Train needs a lock for next (i.e. boxes[lockIndex]). This is done in the function updateLockIndex.

```
cBRoute_i lockIndex = 1;
```

updateLockIndex goes through the requiresLock array and increments lockIndex in each iteration until it finds a switch box. The initial value of lockIndex is also 1 because this points to the first upcoming control box in a route.

```
void updateLockIndex(){
    while(lockIndex < NROUTELENGTH && !requiresLock[lockIndex]){
        lockIndex++;
    }
}
```

The complete initialize function can be seen below.

```
void initialize() {
    //Segments
    for(i : segRoute_i) {
        segments[i] = segRoutes[id][i];
        if(segments[i]>-1) {
            routeLength++;
        }
    }
    curSeg = segments[0];

    //Control boxes
    for(i : cBRoute_i) {
```

```
        boxes[i] = boxRoutes[id][i];
        if(boxes[i] > -1){
            requiresLock[i] = points[boxes[i]] > -1;
        }
    }

    //Locks and reservations
    resSegIndex = 1;
    updateLockIndex();
}
```

The remaining variables in a `Train`'s state space are `resBit`, `resCBIndex` and `locks`. These are already initialized in the declarations and not in the `initialize` function. `resBit` and `resCBIndex` are used for the reservation of segments and will be explained in details in section 6.4.2.1 and `locks` is used to keep track of the number of obtained locks as explained in details in section 6.4.2.2.

### 6.4.2 Operations

Once a `Train` is in its `SingleSegment` location, it is possible for it to execute one of the earlier described operations:

- Request a reservation of a segment
- Request the locking of a point
- Move from one segment to another by passing a control box

Once a `Train` has decided to start an operation, this operation must be completed before a new can begin. `Train` operations are therefore modelled as emissions on edges from `SingleSegment` to other locations. An operation is then complete when the `Train` returns to the `SingleSegment` location again from which a new operation can be started. To prevent a `Train` from starting new operations when the destination has been reached, a fourth operation is implemented for this: Arrive at destination.

The edge for this operation goes to a new location similarly to the other operations, but from this location, `Train` should not be able to return to `SingleSegment`.

#### 6.4.2.1 Reserving a Segment

To keep track of which reservations a `Train` currently has and which it needs, it stores two integers. `resSegIndex`, which was introduced earlier, is the index that

points to the segment in the segments array that the Train needs reservation of next and resCBIndex is the index of the boxes array that points to the control box that the reservation is needed at.

The reservation then starts with the emission on the channel:

```
reqSeg[boxes[resCBIndex]][id][segments[resSegIndex]]
```

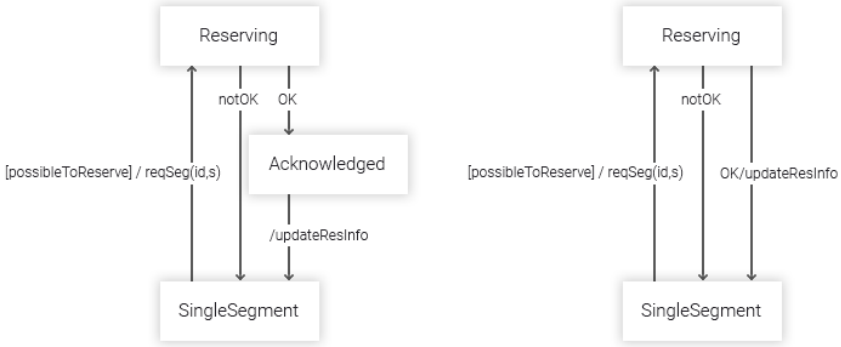
where segments[resSegIndex] is the segment that is requested while boxes[resCBIndex] is the CB that the request is directed towards and id is the ID of the Train.

After this emission, the Train enters the Reserving location from which it returns to SingleSegment when it either synchronizes on the acknowledgement channel OK[id] or the negative acknowledgement channel notOK[id].

Based on the protocol described in chapter 4, a Train updates its state space after having received an acknowledgement. Therefore, if a synchronization happens on OK[id], the Train should actually go to another location before it returns to SingleSegment and from there go to SingleSegment while updating its state space (see figure 6.3a). This would allow a CB to receive requests from other Trains even if the receiving Train has not yet updated its state space. However, since a Train's state space is only visible to the Train itself and since it will not be able to do anything else before it has updated its state space, the additional location is not necessary for the safety of the Train. This means that having Trains update their state spaces at the same time as they receive acknowledgement signals eliminates the need for an additional location without affecting the safety of the system (see figure 6.3b). This simplification has also been used for all other message exchanges in the model.

Since each segment must be reserved at each of its two surrounding control boxes before the segment is fully reserved and possible to enter, the integer resBit keeps track of whether resSegIndex or resCBIndex should be updated. At each successful reservation, Train and CB synchronize on OK[id], where the update function updateResInfo first flips resBit from 0 to 1 if it is currently 0, and from 1 to 0 otherwise.

If resBit is 0 after the flip, resSegIndex is incremented because it means that the reserved segment has been reserved at two control boxes; hence the next segment that the Train should reserve is the next segment in the route. Otherwise resCBIndex is incremented, since the reserved segment is currently only reserved at one control box. Since a Train's initial position has already been reserved for the Train during initialization, the Train is ready to reserve a whole new segment. The initial value of resBit is therefore 0. The value of




**(a)** Receive acknowledgement and update state space in two steps

**(b)** Receive acknowledgement and update state space at the same time

**Figure 6.3:** Simplified state machine part for an acknowledged reservation.

`resCBIndex` is initially 1 since this is the first control box that the upcoming segment needs to be reserved at first.

```
int[0,1] resBit = 0;
cBRoute_i resCBIndex = 1;
```

```
void updateResInfo(){
    resBit = resBit^1;
    resSegIndex = (resBit==0) ? resSegIndex + 1 : resSegIndex;
    resCBIndex = (resBit==1) ? resCBIndex + 1 : resCBIndex;
}
```

Since reservations (and locks) are made in the order of the route, `resSegIndex` and `resCBIndex` do not only indicate what need to be reserved next, but also what has already been reserved.

As seen so far, reservations are not stored as arrays of segment IDs, but the reserved segments are indicated by the reservation variables that have been introduced. This is explained in the following box.

**Reservations indicated by indices**
The reservations that a Train has obtained can be deduced from index, which is the index of the Train's current position, and resSegIndex, which is the index of the segment that needs to be reserved next. Since reservations for a Train at a CB are cleared when the Train passes the CB, all segments in segments that lie *between* index and resSegIndex are considered fully reserved by the Train.

Because every segment (except for the initial position) is reserved twice, it is possible that the segment at resSegIndex has been reserved once. This depends on the value of resCBIndex.

resCBIndex $>$ resSegIndex $\rightarrow$ the Train has one reservation of segments[resSegIndex]
resCBIndex $=$ resSegIndex $\rightarrow$ the Train has no reservations of segments[resSegIndex]

As for the segment at index, this is always only reserved at the upcoming control box.

As explained above, a Train's full reservations at any time are not everything in the segments array up until resSegIndex, but the segments between resSegIndex and index. This information is used in possibleToReserve, which is the function that guards the edge for segment reservation.

```
bool possibleToReserve() {
    return resSegIndex < routeLength && resSegIndex - 1 - index < resLimit;
}
```

The first part in the return statement ensures that a Train does not try to reserve segments beyond the route, and the last part of the statement ensures that there is room for more reservations such that resLimit is not exceeded after the reservation is complete.

In section 4.3.1, two of the train requirements for segment reservations stated that the control box and the segment in the request are parts of the train's route. This does not need to be checked in possibleToReserve since the segment ID and the control box ID in the synchronization are taken directly from the route arrays. As long as resSegIndex < routeLength and resCBIndex $\leq$ routeLength, the chosen segment and CB are indeed parts of the Train's route (routeLength does not need to be strictly larger than resCBIndex because there is always one control box more than there are segments on a route).

Another requirement from the Train's point of view stated that the segment

that is being reserved must not already have been reserved by the Train at the selected CB. This always holds because an acknowledgement for a reservation always result in an incrementation of either resSegIndex or resCBIndex while neither is ever decremented.

If instead a reservation could not be granted and a synchronization on notOK[id] therefore occurs, the Train returns to the SingleSegment location without updating its state space.

#### 6.4.2.2 Locking at a Switch Box

Requesting a lock at a control box is from a Train's point of view similar to the process of requesting a reservation.

The locking starts with the synchronization on

```
reqLock[boxes[lockIndex]][id][segments[lockIndex-1]][segments[lockIndex]]
```

where the Train's ID is passed to the control box along with the two segments that the Train wants to have connected. The guard that allows a locking request is the guard function possibleToLock.

```
bool possibleToLock() {
    return lockIndex < routeLength && locks < lockLimit && ((resBit == 0 &&
        resSegIndex > lockIndex) || (resBit == 1 && resSegIndex >= lockIndex
        ));
}
```

The first part of this return statement ensures that the Train only tries to request locks at control boxes on its route excluding the last one; and the second part of the statement ensures that the Train has room for more locks such that lockLimit is not exceeded after the locking is complete. Unlike reservations, the number of locks must be counted explicitly because it cannot be deduced based on indices since not all control boxes need to be locked. The variable locks is used for this.

```
    cB_id locks = 0;
```

Finally, the last part of the return statement ensures that the Train has the reservations for the segments that it wants to have connected.

Similar to possibleToReserve, it is enough to check that lockIndex < routeLength in order to ensure that the switch box is a part of the Train's route since this index will be used on the boxes array. Checking that the sent segments are adjacent segments in the route is not necessary either since these too are taken

directly from a route array (segments). For this, the indices lockIndex-1 and lockIndex are used since these two will point to the segments surrounding the point that needs locking. Based on these, it is clear that the segments are adjacent in the route.

If possibleToLock returns true and the synchronization is possible, the Train enters the location Locking where it waits for the CB to finish handling the request. The Train and the CB then again either synchronize on OK[id] or notOK[id] with id being the Train's ID. If they synchronize on notOK[id], there is no update of the Train's state space and if they synchronize on OK[id], locks and lockIndex are incremented and updateLockIndex is executed again to find the next switch box. All this is done in the function updateLockInfo.

```
void updateLockInfo(){
    locks++;
    lockIndex++;
    updateLockIndex();
}
```

#### 6.4.2.3 Passing a Control Box

The last possible interaction between a train and a control box is a train's passing of the control box. This starts with the synchronization on the channel

```
    pass[boxes[index+1]]
```

which is a synchronization with the upcoming CB in the Train's route. This synchronization models the sensor's detection of a train entering the critical section and it can take place if the guard function possibleToPass returns true.

```
bool possibleToPass() {
    return resSegIndex > index + 1 && lockIndex > index + 1 && index + 1 <
        routeLength;
}
```

This function checks all the stated TCC requirements from section 4.3.1: The segment that the train wants to enter has already been fully reserved for the train, the control box that it wants to pass has been locked for it and the control box is not the last one in the route. index + 1 is both the upcoming control box and the upcoming segment, so if resSegIndex is larger than index + 1, then the upcoming segment must have been fully reserved and if routeLength is larger than index + 1, then there are segments left on the route. If lockIndex is larger than index + 1 it must also mean that the upcoming control box has locked for the Train or that it does not need to be locked.

When a `Train` has successfully synchronized on `pass[boxes[index+1]]`, it enters the location `DoubleSegment`, which indicates that the `Train` is in a critical section between two segments - regardless of whether it is completely inside the critical section or not. `DoubleSegment` does not mean that the train is necessarily moving on two segments, but that it is possible, so the `Train` sets its variable `headSeg` to the ID of the upcoming segment in order to indicate that it is possible that it will be on this segment at any time now. This is done in the function `updateHeadInfo`.

```
void updateHeadInfo(){
    headSeg = nextSegment(boxes[index+1], curSeg);
}
```

From this location, it is only possible to leave the location and return to `SingleSegment` by synchronizing on

```
    passed[boxes[index+1]]
```

When this synchronization happens, the function `updateLocationInfo` updates `curSeg`, `headSeg` and `index` and if the passed control box is a switch box, it also decrements the lock counter `locks` since the lock must be cleared from its state space.

```
void updateLocationInfo(){
    curSeg = headSeg;
    headSeg = -1;
    if(requiresLock[index + 1]){
        locks--;
    }
    index++;
}
```

Compared to the other two described interactions between `Train`s and `CB`s, the operation here happens with two emissions from the `Train` rather than one emission from the `Train` and one emission from the `CB`. This corresponds to the registration of a train entering and leaving a control box' critical section.

#### 6.4.2.4 Arriving at Destination

The last outgoing edge from the `SingleSegment` location goes to the location `Arrived`. This edge is guarded by the function `hasArrived`, which simply checks whether the `Train` has reached the last segment in its route.

```
bool hasArrived() {
    return index == routeLength-1;
}
```

In the Arrived location, a single self-loop with no guard, update or synchronization is added to prevent a deadlock.

## 6.5 The CB Template

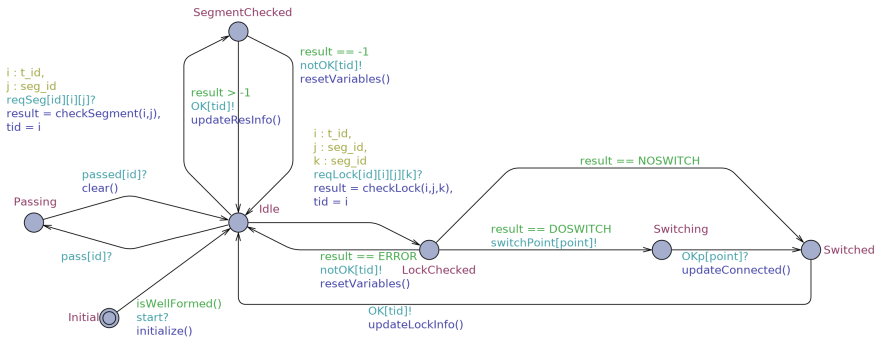



**Figure 6.4:** The CB state machine

Similar to the Train template section, this section first describes the state space and initialization of a CB (section 6.5.1) followed by the modelling of how a CB handles the different Train operations (section 6.5.2).

### 6.5.1 State Space and Initialization

The state machine for the CB template (see fig. 6.4) also starts in a location Initial from which it synchronizes on start while updating its state space with initial settings if its isWellFormed function returns true (this function is described in details in section 6.8). The initialization is performed in the function initialize.

**Associated Segments and Point**
The initialize function first initializes the array segments. This is an array of length three, which is simply a copy of the segments in the CB's definition. This information is hence retrieved from the global array cBs.

```
segV_id segments[3];
```

A CB instance that is also a switch box also needs to know the ID of its associated Point instance in order for them to synchronize when the CB needs to switch

the Point. This ID is saved in the point variable and retrieved from the earlier introduced global array points. By using this ID and the global pointInPlus array, the CB can also determine the initial connection connected.

```
segV_id connected =-1;
pV_id point = -1;
```

connected is a variable for the segment ID that a CB is currently connecting with the segment segments[0]. If the control box is placed at a segment's closed end, this value is -1 and it will never be used since Trains will never request locks at these kinds of CBs.

If the control box is placed at the meeting of exactly two segments, the connected segments will always be segments[0] and segments[1], which is why connected is first set to segments[1] in the initialize function. If instead a control box is placed at a point i.e. it is a switch box, then either segments[1] or segments[2] can be the segment initially connected to segments[0]. In this case, the value in pointInPlus for the associated Point is checked to see whether connected should be set to segments[2] instead.

**Reservations**
A last thing that must be done in the initialization step is the setting of the initial reservations, which are retrieved from the global array initialRes. A CB saves the reservations of its segments in its res array where res[i] = -1 if nobody has reserved the segment segments[i]. Otherwise res[i] has the value of the ID of the Train that has the reservation.

```
tV_id res[3] = {-1, -1, -1};
```

The initial reservations could also be set by synchronizing on the reqSeg channels like any other segment reservation is performed, but this would require an additional location between the Train's Initial location and SingleSegment location, since an edge can only carry a single synchronization. This would then result in more possible interleavings and an undesirably larger state space.

The complete initialize function can be seen below.

```
void initialize() {
    segments = cBs[id];
    point = points[id];
    connected = segments[1];

    if(point != -1 && !pointInPlus[point]){
        connected = segments[2];
    }

    for(i : t_id) {
        if (initialRes[i].cb == id) {
            seg_id s = initialRes[i].seg;
            if(s == segments[0]){
```

```
                res[0] = i;
            } else if (s == segments[1]){
                res[1] = i;
            } else {
                res[2] = i;
            }
        }
    }
}
```

The remaining variables in a CB's state space are result, tid, lockedBy, ERROR, NOSWITCH and DOSWITCH.

result and tid are used only during the handling of requests from a Train. ERROR, NOSWITCH and DOSWITCH are used as the values of result during the handling of a lock request. lockedBy is used to denote the Train that has obtained the lock at a CB. All of these are explained in more details in sections 6.5.2.1 and 6.5.2.2.

### 6.5.2 Handling Train Operations

Once a CB has initialized, it enters the location Idle. From this location, a CB is ready to synchronize with Trains so since a Train can initiate three kinds of operations that involve CBs, the CB template has three outgoing edges from Idle. These are used for:

- Handling a reservation request
- Handling a switch/lock request
- Handling the passing of a train

Whenever the handling of an operation is complete, CB returns to Idle again where it can participate in a new synchronization. This means that it is only possible to synchronize with a CB that is not already handling an operation.

#### 6.5.2.1 Handling a Reservation Request

For a CB to receive emissions from different Trains and with different data, select statements are used to allow CBs to receive emissions on all the channels that are associated with it.

The first described operation was the request of a segment reservation. A CB receives this request by synchronizing on reqSeg[id][i][j] where id is its own ID, and i and j are the Train ID and a segment ID respectively. The last two are obtained with select statements from the range of existing Train IDs and segment IDs. With the these two IDs, CB executes the function checkSegment while the Train ID is saved in the local variable tid, so the CB knows which Train to respond to from its next location.

checkSegment checks the segments array to see if it is possible for the CB to make a reservation of the requested segment. If the requested segment is found in segments and the corresponding value in res is equal to -1, it is possible to complete the reservation. In this case, the function will return the index at which it found the segment. If a reservation is not possible, -1 is returned.

```
int[-1,2] checkSegment(seg_id sid) {
    for(i:int[0,2]) {
        if(segments[i] == sid && res[i] == -1) {
            return i;
        }
    }
    return -1;
}
```

The returned value of the checkSegment function is saved in the variable result and used in the next location SegmentChecked. If result is -1, the CB emits on the notOK[tid] channel and resets the variables result and tid in resetVariables. If instead result is greater than -1, the CB emits on OK[tid] and both update its state space and resets result and tid. The update is done in the updateResInfo function.

```
void resetVariables(){
    tid = -1;
    result = -1;
}

void updateResInfo(){
    res[result]=tid;
    resetVariables();
}
```

#### 6.5.2.2 Handling a Switch/Lock Request

Similarly to the handling of a reservation request, when a CB receives an emission on one of its associated reqLock channels, it saves the emitting Train's ID in tid, executes the function checkLock and saves the returned result in result.

```
const int ERROR = 0;
const int NOSWITCH = 1;
```

```
const int DOSWITCH = 2;

...

int[0,2] checkLock(seg_id s1, seg_id s2){
  if(lockedBy == -1 && (segments[0] == s1 && exists(i:int[1,2]) segments[i]
      == s2) || (segments[0] == s2 && exists(i:int[1,2]) segments[i] == s1))
      {
    if ((s1 == segments[0] && s2 == connected) || (s2 == segments[0] && s1 ==
        connected)){
      return NOSWITCH;
    } else
      return DOSWITCH;
    }
    return ERROR;
}
```

checkLock takes the IDs of the two segments that the Train wants to have connected as arguments, and it returns one of three possible values: ERROR, NOSWITCH or DOSWITCH, which are simply integer constants for ease of reading.

For a lock request to succeed, the associated Point must first of all be unlocked. This is the case if the CB's lockedBy value is -1. Secondly, for a connection to be valid, the two segments sent in the request must be possible for the CB to connect, i.e. one of them must be the stem (segments[0]) while the other is either the plus segment (segments[1]) or the minus segment (segments[2]). If these requirements are *not* fulfilled, ERROR is returned.

If the lock is available, the CB has to determine whether or not a switch of its associated Point is necessary. This is done by determining whether the given segment that is *not* the stem is the same as the connected segment. If it is not, a switch is necessary, so DOSWITCH is returned and otherwise NOSWITCH is returned.

After the reqLock synchronization, CB enters the location LockChecked. If ERROR was returned from checkLock, CB sends a negative acknowledgement to the requesting Train while returning to Idle. Along this edge, it also resets its variables with resetVariables. This synchronization does not actually ever happen if the Train has checked its requirements properly before requesting a lock. This is because no other Train would have been able to obtain the lock at the CB if the first Train holds the reservations for two segments associated with the CB. It has nevertheless been modelled since it is a part of the real control system.

If checkLock returns DOSWITCH, CB will synchronize with its associated Point on

```
    switchPoint[point]
```

It then enters the location `Switching` from which it waits for the `Point` instance to emit on `OKp[point]`. When this synchronization takes place, the function `updateConnected` updates the `CB`'s state space and the `CB` enters the location `Switched`. `updateConnected` updates the value of `connected` by switching it from its current segment value to the the other possible segment.

```
void updateConnected(){
    if(connected == segments[1]){
        connected = segments[2];
    } else {
        connected = segments[1];
    }
}
```

Finally, if the result of `checkLock` is `NOSWITCH`, `CB` skips the `Switching` location and goes directly from the `LockChecked` location to the `Switched` location.

From the `Switched` location, the `CB` emits on the `OK[tid]` channel to inform the `Train` about the successful locking and it then updates its state space in `updateLockInfo`, which sets `lockedBy` to the `Train`'s ID and resets variables.

```
void updateLockInfo(){
    lockedBy = tid;
    resetVariables();
}
```

#### 6.5.2.3 Handling the Passing of a Train

When a train passes a control box, the control box simply needs to update its state space by clearing the reservations of the connected segments and by clearing the lock if it is a switch box.

When a `CB` receives an emission on its associated `pass` channel, it enters the `Passing` location. Once the `CB` receives another signal but on its `passed` channel, it updates its state space with the function `clear`. This function resets `lockedBy` to -1 and removes the reservation of the stem and the segment that the stem is currently connected to. The first synchronization corresponds to the control box' sensor becoming active while the second synchronization corresponds to the sensor returning to being passive. This means that while a `CB` is in its `Passing` location, a train is in the control box' critical section. Since a `CB` being in the `Passing` location indicates that its sensor is active, no explicit variable is needed for this information.

```
void clear(){
     lockedBy = -1;

     res[0] = -1;
     if(connected == segments[1]) {
```

```
        res[1] = -1;
    } else {
        res[2] = -1;
    }
}
```

It is here important that the CB does not just update its state space and return to Idle immediately but actually waits in Passing until the Train has finished passing. Otherwise, other Trains may be able to switch or lock the point before the first Train has left the critical section.

## 6.6 The Point Template

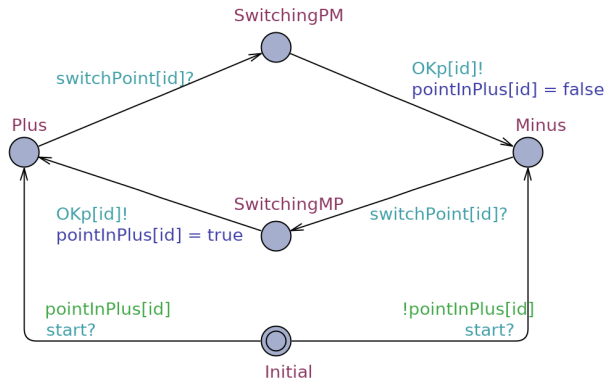



**Figure 6.5:** The Point state machine

The last template Point can be seen in figure 6.5. Again, this template starts in a location called Initial from which it enters its Plus location or its Minus location after a synchronization on start. The location that it goes to depends on its initial setting, which it retrieves from the global pointInPlus array.

From both the Plus and the Minus locations, the Point can synchronize with its associated CB on the channel switchPoint[id] where id is its own ID. It then enters an intermediate location (SwitchingMP or SwitchingPM), which represents the states in which it is in a position between its plus position and its minus position. From this location, it enters the opposite location - Plus if it started in Minus and Minus if it started in Plus. It can then signal the completion of the switching on the OKp[id] channel, which will be received by its associated CB.

Along the the last edges in the switching process, a `Point` also updates its state in the `pointInPlus` array. If it enters the `Plus` location, it sets its value `pointInPlus[id]` to true, and if it enters the `Minus` location, it sets the value to false. This should ensure that a train passing a switch box ends up on the correct segment.

## 6.7 System Declarations

As previously explained, bounded integer types are used for the ID parameters for the `Train` template, the `CB` template and the `Point` template. This means that it is sufficient to write the template names a single time in the `System Declarations` to create all the needed instances. Since the `Initializer` template takes no parameters, only one `Initializer` instance will be created.

```
system Initializer, Train, CB, Point;
```

## 6.8 Configuration Well-formedness

As seen in the previously shown state machines, the `CB` and the `Train` templates each had a guard function `isWellFormed` on their edges from their initial locations. The well-formedness checks have generally been implemented as functions that will block a `Train` or a `CB` that is not well-formed.

The well-formedness requirements could also have been defined as UPPAAL properties that can be verified in the verifier, but the specification language of UPPAAL properties is only a subset of CTL, so expressing well-formedness here is very limiting. Also, unlike the properties related to the development in the system, which is for *dynamic* data (presented in chapter 7), well-formedness is checked on the *static* configuration data. This means that it only needs to be checked once in the initial state, which is why it is more convenient to do this through functions.

By using these functions as guards on the first edges of the state machines, it is possible to see exactly which instances are not well-formed if there are any. If many instances exist, one can verify a single property that checks that all instances of a template are well-formed:

```
A<> forall(i:cB_id) not(CB(i).Initial)
```

```
A<> forall(i:t_id) not(Train(i).Initial)
```

The above properties check that all paths lead to a state in which all `Train` and `CB` instances are *not* in their *Initial* locations because this means that they could successfully initialize so their well-formedness functions must have yielded true [2]. If all instances really *are* well-formed, this verification should finish its execution almost immediately.

The well-formedness of `Train`s and `CBs` is actually the well-formedness of the constants `boxRoutes`, `segRoutes`, `cBs`, `points` and `initialRes`. These are described in the following sections.

The well-formedness functions are implemented in the global declarations and can be seen in appendix A.1.

### 6.8.1 Size

The four constants `NCB`, `NSEG`, `NTRAIN` and `NPOINT` were used to set the size of the network and number of trains. If the network has to be non-empty, the smallest network and configuration could simply contain a single segment, two control boxes and a single train positioned on the segment. As explained in section 6.2.6, `NPOINT` also has to be at least 1 because of its use in the bounded integer types `p_id` and `pV_id` and its use as an array length. That is, the size requirements are
$(\texttt{NSEG} \geq 1) \wedge (\texttt{NCB} \geq 2) \wedge (\texttt{NTRAIN} \geq 1) \wedge (\texttt{NROUTELENGTH} \geq 1) \wedge (\texttt{NPOINT} \geq 1)$.

These requirements could be written into functions or properties, but since the minimum size requirements are mainly required by the other definitions (bounded integers and array lengths), errors will already be caught at compilation time. The violation $\texttt{NCB} < 2$ will also be caught at compilation time because of a function, which will be explained later in this chapter.

### 6.8.2 Control Boxes

The well-formedness of the actual railway layout depends on how the control boxes and segments are placed in relation to each other, i.e. how the `cBs` array

[2] For these properties to be true, it is important that the `start` channel is urgent or committed, otherwise the system would be able to stay in its first state forever.

is defined (see section 6.2.5). Each `CB` instance with ID `i` should hence ensure that `cBs[i]` is well-formed.

A single control box definition is well-formed if

- its definition is valid according to table 6.1
- each segment that it contains is a part of exactly one other control box definition

Based on the valid definitions of control boxes in table 6.1, the *invalid* definitions are:

`{-1,x,y}`, `{-1,x,-1}`, `{-1,-1,x}`, `{x,-1,y}`, `{-1,-1,-1}`

and any definition that contains the same segment ID more than once.

The reason for having the second requirement is because a segment has exactly one control box placed at each of its ends. This means that if `otherBoxes(id, s)` is the number of control box definitions that contain segment `s` and with an ID different from `id`, then for each valid segment ID `x` in the control box definition of a control box with id `i`, it should hold that `otherBoxes(i, x) = 1`. This function is implemented as:

```
int otherBoxes(cB_id id, segV_id s){
    segV_id cB[3] = cBs[id];
    int found = 0;
    for(i:cB_id){
        if(id != i && (cBs[i][0] == s || cBs[i][1] == s || cBs[i][2] == s)){
            found++;
        }
    }
    return found;
}
```

Besides the two mentioned well-formedness requirements for `CB`s, there are two additional requirement for switch boxes:

- Two switch boxes may share at most one segment with the exception of the case where the switch boxes share the same plus segment and the same minus segment.
- Each switch box has an associated point, which is not associated with any other switch box

For two different switch boxes `sb1` and `sb2`, the first requirement can be expressed formally as:

- (sharedSegments(sb1,sb2) $\leq$ 1) $\vee$
  (sb1[0] $\neq$ sb2[0] $\wedge$ sb1[1] = sb2[1] $\wedge$ sb1[2] = sb2[2])

where sharedSegments(sb1,sb2) is the number of segments that sb1 and sb2 share:

```
int sharedSegments(cB_id i, cB_id j){
    int count = 0;
    if (cBs[i][0] != -1 && (cBs[i][0] == cBs[j][0]
        || cBs[i][0] == cBs[j][1] || cBs[i][0] == cBs[j][2])){
        count++;
    }
    if (cBs[i][1] != -1 && (cBs[i][1] == cBs[j][0]
        || cBs[i][1] == cBs[j][1] || cBs[i][1] == cBs[j][2])){
        count++;
    }
    if (cBs[i][2] != -1 && (cBs[i][2] == cBs[j][0]
        || cBs[i][2] == cBs[j][1] || cBs[i][2] == cBs[j][2])){
         count++;
    }
    return count;
}
```

The full implementation of these requirements in the function cBIsWellFormed can be seen in appendix A.1.

If id is the ID of a CB, the point requirements can be formulated as:

- cBs[id][2] $\neq$ -1 $\leftrightarrow$ points[id] $\neq$ -1
- $\forall$ i$\in$cB_id (i $\neq$ id $\wedge$ points[id] $\neq$ -1
  $\rightarrow$ points[id] $\neq$ points[i])

The first requirement states that if the CB has a minus segment, i.e. is a switch box, then it also has an associated Point and vice versa. The second requirement states that if the CB is a switch box, then all CBs besides the one that is being checked are either regular CBs or they have associated Points different from the CB that is being checked.

The point requirements are implemented in the function pointIsWellFormed:

```
int pointIsWellFormed(cBV_id id){
    if(points[id] != -1){
        for(i : cB_id){
            if(i != id && points[i] == points[id]){
                return false;
            }
        }
    }
    return (points[id] == -1) == (cBs[id][2] == -1);
}
```

Both `cBIsWellFormed` and `pointIsWellFormed` can finally be used in the `CB` template's `isWellFormed` function.

```
bool isWellFormed(){
    return cBIsWellFormed(id) && pointIsWellFormed(id);
}
```

### 6.8.3 Routes

The remaining configuration constants are related to train routes and the corresponding initial reservations.[3]

For a segment route `sr` in `segRoutes` to be well-formed, the route must satisfy the following requirements:

- $\forall$ `i,j` $\in$ `[0,NROUTELENGTH-1] (i` $\neq$ `j` $\rightarrow$ `sr[i]` $\neq$ `sr[j])`
- $\forall$ `i` $\in$ `[0,NROUTELENGTH-2] (sr[i] = -1` $\rightarrow$ `sr[i+1] = -1)`
- `sr[0]` $\neq$ `-1`
- $\forall$ `i` $\in$ `[0,NROUTELENGTH-2] (sr[i+1]` $\neq$ `-1` $\rightarrow$ `canConnect(sr[i],sr[i+1]))`

The first requirement states that all segments in a segment route are unique since a `Train` may never pass the same segment twice.

The second requirement states that once -1 appears, it may also only be followed by a -1. This ensures that no -1 appears in the middle of a route.

The third requirement states that there must be at least one valid segment in a `Train`'s route.

Finally, the fourth requirement states that all adjacent pairs of valid segment IDs can be connected, otherwise it would not be possible for a train to move from one segment to the other.

For `canConnect` to return true, the first value in a `CB` definition must be the same as the segment ID `sr[i]` or `sr[i+1]` while its second or third value must be the other:

[3] In reality, it is of course possible to install invalid routes in TCCs, but in such cases, the trains would and should never be able to reach their destinations.

```
bool canConnect(seg_id s1, seg_id s2){
    for(i:cB_id){
        if(cBs[i][0] == s1 && (cBs[i][1] == s2 || cBs[i][2] == s2)){
            return true;
        }
        if (cBs[i][0] == s2 && (cBs[i][1] == s1 || cBs[i][2] == s1)){
            return true;
        }
    }
    return false;
}
```

The function checking the well-formedness of `segRoutes` can then be implemented as:

```
bool segRouteIsWellFormed(segV_id route[NROUTELENGTH]){
    int i = 0;
    if(route[0] == -1){
        return false;
    }

    for(i:segRoute_i){
        for(j:segRoute_i){
            if(j != i && route[i] == route[j] && route[i] != -1){
                return false;
            }
        }
    }

    while(i <= NROUTELENGTH - 2){
        if(route[i] == -1 && route[i+1] != -1){
            return false;
        }
        if(route[i+1] != -1 && !canConnect(route[i], route[i+1])){
            return false;
        }
        i++;
    }
    return true;
}
```

Using a while loop rather than a for loop for the last check is here crucial since looping from 0 to `NROUTELENGTH-2` in a for loop would give an error if `NROUTELENGTH = 1`, in which case the `canConnect` check is not relevant since there is only one segment.

Besides `segRoutes`, routes are also defined in terms of control boxes. The requirements for a control box route `br` in `boxRoutes` are:

- $\forall$ `i` $\in$ `[0,NCB-2]` `(br[i] = -1` $\rightarrow$ `br[i+1] = -1)`
- `br[0]` $\neq$ `-1` $\wedge$ `br[1]` $\neq$ `-1`
- $\forall$ `i` $\in$ `[0,NCB-2]` `(br[i+1]` $\neq$ `-1` $\rightarrow$ `sharesSegment(br[i], br[i+1]))`

The first two requirements correspond to the second requirement and the third requirement for a route in `segRoutes`, the second requirement here just states that the second control box in the array may not be -1 either. This is because every segment in a segment route should have its two associated control boxes in the corresponding control box route.

The third requirement states that for every adjacent pair of valid control box IDs in `br`, the two control boxes in question share a segment. This ensures that a train only moves between control boxes that actually share segments. The implementation of the function that checks the well-formedness of a route in `boxRoutes` and the auxiliary function `sharesSegment` can be seen below.

```
bool boxRouteIsWellFormed(cBV_id route[NROUTELENGTH+1]){
    for(i:int[0,NROUTELENGTH-1]){
        if(route[i] == -1 && route[i+1] != -1){
            return false;
        }
        if(route[i+1] != -1 && !sharesSegment(route[i], route[i+1])){
            return false;
        }

    }
    return true;
}
```

```
bool sharesSegment(cB_id i, cB_id j){
    return (i != j) &&
            ((cBs[i][0] != -1 && (cBs[i][0] == cBs[j][0] || cBs[i][0] == cBs[
                j][1] || cBs[i][0] == cBs[j][2])) ||
            (cBs[i][1] != -1 && (cBs[i][1] == cBs[j][0] || cBs[i][1] == cBs[j
                ][1] || cBs[i][1] == cBs[j][2])) ||
            (cBs[i][2] != -1 && (cBs[i][2] == cBs[j][0] || cBs[i][2] == cBs[j
                ][1] || cBs[i][2] == cBs[j][2])));
}
```

After ensuring that routes in `segRoutes` and `boxRoutes` are all well-formed, the consistency between the two must also be ensured. The routes `segRoutes[i]` and `boxRoutes[i]` may each be well-formed, but they could define completely different routes. For two routes of the same train to be consistent, they must satisfy:

- $\forall$ `i` $\in$ `[0,NCB-2] (br[i+1] = -1` $\leftrightarrow$ `sr[i] = -1)`
- $\forall$ `i` $\in$ `[0,NCB-2] (br[i+1]` $\neq$ `-1` $\rightarrow$ `sharesSegmentS(br[i], br[i+1], sr[i]))`

The first requirement states that there is always one control box ID more in `br` than there are segment IDs in `sr`. The second requirement uses a variation of `sharesSegment`, which takes a segment ID as an additional parameter. This function should return true if the two different control boxes share exactly the segment with the passed segment ID:

```
bool sharesSegmentS(cB_id i, cB_id j, seg_id s){
    return  (i != j) &&
            (cBs[i][0] == s || cBs[i][1] == s || cBs[i][2] == s) &&
            (cBs[j][0] == s || cBs[j][1] == s || cBs[j][2] == s);
}
```

The consistency function is then:

```
bool routesAreConsistent(t_id id){
    cBV_id bRoute[NROUTELENGTH+1] = boxRoutes[id];
    segV_id sRoute[NROUTELENGTH] = segRoutes[id];

    for(i:int[0,NCB-2]){
        if((bRoute[i+1] != -1) == (sRoute[i] == -1)){
            return false;
        }
        if(!sharesSegmentS(bRoute[i], bRoute[i+1], sRoute[i])){
            return false;
        }
    }
    return true;
}
```

The last constant that must be checked for well-formedness is the `initialRes` array. A reservation in this array is well-formed if the reserved segment *can* be reserved at the chosen control box, i.e. if the segment is associated with the control box:

```
bool reservationIsWellFormed(reservation res){
    return cBs[res.cb][0] == res.seg || cBs[res.cb][1] == res.seg || cBs[res.
        cb][2] == res.seg;
}
```

Similar to routes, reservations can also be well-formed without being consistent with the rest of the configuration data. For a `Train`'s initial reservation to be consistent with the rest of the `Train`'s data, the reservation must be for the `Train`'s initial position at the first upcoming `CB` in its route:

```
bool initialResIsConsistent(t_id id){
    return initialRes[id].cb == boxRoutes[id][1] && initialRes[id].seg ==
        segRoutes[id][0];
}
```

Some of the defined well-formedness requirements that were introduced in this chapter are not necessarily requirements in other railway systems and similarly, some real-life systems may have other requirements that were not modelled here. For example, in this model, it is possible to have loops without violating any rules while this may not be the case in other railway systems. This chapter was meant to illustrate how well-formedness checks can be implemented in UPPAAL and not how railway systems should be configured.

Chapter 7

# Formulating Properties for the UPPAAL Model

With the first model created, desired properties for a railway network should be formulated so they can be used to verify the implemented railway control system for specific configurations.

The properties that are presented in this chapter are generally divided into four categories: Safety properties, properties related to operation requirements, consistency properties and liveness properties. These are presented in sections 7.3 to 7.6. Section 7.7 also presents some other properties that do not fit under the stated four categories.

## 7.1 General Property Specification

As mentioned in chapter 6, there were some advantages in limiting the parameters that the railway component templates take to only IDs defined by bounded integers. Besides being able to automatically generate instances in the system declarations, all UPPAAL queries can be written on a general form that can be used with any configuration data.

For example, instead of explicitly stating that `Trains t0` and `t1` may not be on the same segment, and `Trains t1` and `t2` may not be on the same segment and so on, one can get all possible pairs of `Train` instances with two `forall` expressions and the type for train IDs:

```
forall(i:t_id) forall(j:t_id) p(i,j)
```

`p(i,j)` is here a boolean expression that uses the two `Train` IDs `i` and `j`. In this expression, the two `Trains` are referred to as `Train(i)` and `Train(j)`. That is, by iterating over instances, the queries can be used for any configuration.

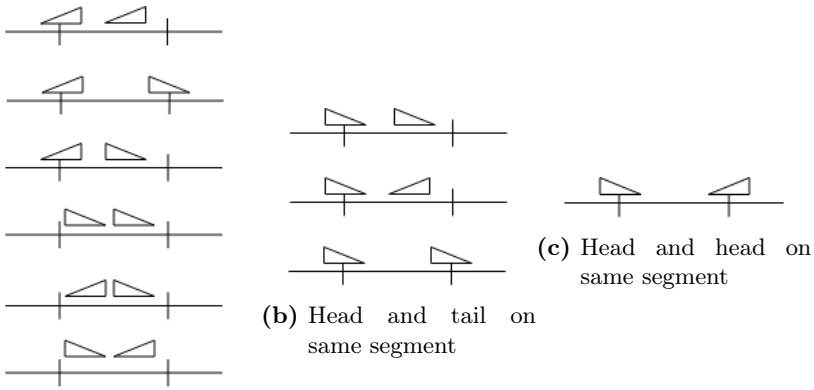
## 7.2 Assumptions

During the modelling of the railway control system, several assumptions were made. This means that any property that holds, holds under these assumptions. A summary of the assumptions made in chapter 6 and the assumptions stated in the case study in chapter 4 can be seen below.

- The railway networks are small and local.
- All trains are shorter than the shortest segment.
- Trains never move without permission.
- No route contains the same segment multiple times.
- Messages are never lost.
- There is no delay in the communication between instances.
- TCCs and CBs can update their state spaces at the same time as they exchange messages.
- CBs handle requests at the same time as they receive them.

## 7.3 Safety Properties

The most important task of the railway control system is to ensure safety in terms of no collision between trains and no derailment at points.



**(a)** Tail and tail on same segment

**(b)** Head and tail on same segment

**(c)** Head and head on same segment

**Figure 7.1:** Collision scenarios

### 7.3.1 No Collision

*At most one train may be on a segment at a time.*

As explained before, collision is defined as two or more trains being on the same segment.

A train's position is either a single segment or if it has a double position, its head is on one segment and its tail is on another. The different scenarios in which two different trains can occupy (parts of) the same segment can be seen in figure 7.1, where the obvious cases of two trains being in the same critical section have been omitted. To ensure the safety of trains, these are the situations that should never occur.

Because `curSeg` is used for both whole single positions and for tail positions, the number of undesired position combinations can be reduced to three different types as seen in figure 7.1.

Figure 7.1a shows the scenario in which two trains have the same `curSeg` value, so either a train is completely on the shared segment, or a train's tail is on that segment. This case also covers all the cases where two trains with the same direction are in the same critical section.

Figure 7.1b shows the scenarios in which one train's `headSeg` is the same as another train's `curSeg`, i.e. the second train is either completely on the segment,

or its tail is on the segment. This combination also covers all the cases where the two trains with different directions are in the same critical section.

The last case (see figure 7.1c) is a unique case where both trains are in double positions and share the segment that their heads are on.

All these situations must be avoided for every pair of different `Train`s and because all `Train` instances have the same `curSeg` value until the initialization step, this property should be checked after the initialization. This means that the following formula must hold for all pairs of different `Train` instances `t1` and `t2`.

$$\texttt{AG Initializer.Initialized} \rightarrow \texttt{t1.curSeg} \neq \texttt{t2.curSeg} \wedge \texttt{t1.DoubleSegment} \rightarrow \texttt{t1.headSeg} \neq \texttt{t2.curSeg} \wedge \texttt{t1.DoubleSegment} \wedge \texttt{t2.DoubleSegment} \rightarrow \texttt{t1.headSeg} \neq \texttt{t2.headSeg}$$

Expressed in UPPAAL for all relevant instances:

```
A[] forall(i:t_id) forall(j:t_id)
    Initializer.Initialized && i != j imply
    (Train(i).curSeg != Train(j).curSeg) &&
    (Train(i).DoubleSegment imply Train(i).headSeg != Train(j).curSeg) &&
    (Train(i).DoubleSegment && Train(j).DoubleSegment imply Train(i).headSeg
        != Train(j).headSeg)
```

### 7.3.2 No Derailment

There are two types of derailments and therefore two properties must be expressed for this.

- *If a train is in a critical section, the point in that section is not in the middle of switching.*

  A train being in the critical section of a switch box `cb` can be expressed as a `Train` being in its `DoubleSegment` with `points[cb.id]` $\neq$ `-1`. The associated `Point` may then not be in either of its two switching locations `SwitchingPM` and `SwitchingMP`.

  If `cbp` is the `CB` that is being passed by a `Train t` in its `DoubleSegment` location, then this property can be formulated as:

```
AG t.DoubleSegment ∧ points[cbp.id]≠-1 →
                   ¬Point(points[cbp.id]).SwitchingPM ∧
                   ¬Point(points[cbp.id]).SwitchingMP
```

Expressed in UPPAAL for all relevant instances:

```
A[] forall(i:t_id) Train(i).DoubleSegment &&
    points[Train(i).boxes[Train(i).index+1]] != -1 imply
    !Point(points[Train(i).boxes[Train(i).index+1]]).SwitchingPM &&
    !Point(points[Train(i).boxes[Train(i).index+1]]).SwitchingMP
```

- *If a train is in a critical section, then the segments that it is moving on are connected.*

  While a `Train` is in its `DoubleSegment` location, its `headSeg` variable denotes the ID of next segment that it will move to. If the train is not derailing, this segment should be connected to the segment that the train came from, i.e. the position of its tail `curSeg`. These two values should hence be matched with the stem segment of the control box that is being passed and the segment connected to this. The former can be found in the `CB`'s `segments` array at index 0 while the latter is saved in its `connected` variable.

```
AG t.DoubleSegment →
   (t.headSeg = cbp.segments[0] ∧ t.curSeg = cbp.connected) ∨
   (t.curSeg = cbp.segments[0] ∧ t.headSeg = cbp.connected)
```

Expressed in UPPAAL for all relevant instances:

```
A[] forall(i:t_id)
    Train(i).DoubleSegment imply
        (Train(i).headSeg == CB(Train(i).boxes[Train(i).index+1]).
             segments[0] && Train(i).curSeg == CB(Train(i).boxes[Train(i
             ).index+1]).connected) ||
        (Train(i).curSeg == CB(Train(i).boxes[Train(i).index+1]).
             segments[0] && Train(i).headSeg == CB(Train(i).boxes[Train(
             i).index+1]).connected)
```

## 7.4 Operation Requirements

Properties can also be specified and verified for the operation requirements introduced in section 4.3.1 to ensure that all the requirements have been implemented correctly.

Although the safety properties are the most important properties to verify, verifying properties for the different operations can be useful when figuring out the cause(s) when the safety properties do *not* hold.

Many of the operation requirements are requirements that a train has to meet in order to even initiate an operation, but since it is not possible to formulate queries on synchronizations, these properties have been formulated in a slightly different way. Instead of ensuring that an operation is only initiated if certain train requirements are met, the properties state that the operations are only *successful* if the train requirements are met. This means that if one of these properties fail, the cause could be from either the train side or the control box side.

### 7.4.1 Requesting a Reservation of a Segment

When a TCC requests a reservation of a segment at a control box, the following properties must hold:

- *A train never has more reservations than the reservation limit.*

  This property can be validated by calculating the difference between the index of a `Train`'s latest reservation and the index of its current position. This difference should never exceed the reservation limit since it is exactly the number of fully reserved segments. Finding the index of the latest reservation is simply done by subtracting one from `resSegIndex` since this is the index of the next segment that the train wishes to reserve.

  `AG t.resSegIndex-1-t.index` $\leq$ `resLimit`

  Expressed in UPPAAL for all relevant instances:

  ```
  A[] forall(i:t_id)
  Train(i).resSegIndex - 1 - Train(i).index <= resLimit
  ```

- *A reservation is only successful if the requested segment is a part of the requesting train's route.*

  This property means that the ID of the segment that a `Train` tries to reserve exists in its `segments` array. Finding the successful reservations can be done by looking through all `CB`s `res` arrays. The query can then be stated as: For each saved reservation, the `Train` that owns it must have it in its route array `segments`. The `Train` that owns the reservation is the

value of `res[x]` when it is not -1, and the reserved segment can be found at the same index `x` in the `CB`'s `segments` array.

For a `CB cb`, this can be formulated as:

`AG` $\forall_{j \in [0,2]}$ `(cb.res[j]≠-1 →` $\exists_{k \in segRoute\_i}$ `Train(cb.res[j]).segments[k] = cb.segments[j])`

Expressed in UPPAAL for all relevant instances:

```
A[] forall(i:cB_id) forall(j:int[0,2])
    CB(i).res[j] != -1 imply
        exists(k:segRoute_i) Train(CB(i).res[j]).segments[k] == CB(i).
            segments[j]
```

- *A reservation is only successful if the control box that a train contacts is a part of the train's route.*

  This property is slightly similar to the previous one. The antecedent is now that the ID of the `CB` with the reservation must exist in the `Train`'s `boxes` array.

  `AG` $\forall_{j \in [0,2]}$ `(cb.res[j]≠-1 →` $\exists_{k \in cBRoute\_i}$ `Train(cb.res[j]).boxes[k] = cb.id)`

  Expressed in UPPAAL for all relevant instances:

  ```
  A[] forall(i:cB_id) forall(j:int[0,2])
      CB(i).res[j] != -1 imply
          exists(k:cBRoute_i) Train(CB(i).res[j]).boxes[k] == i
  ```

- *A reservation is only successful if the requested segment is associated with the control box that receives the request.*

  The antecedent in this query must be written a bit differently because a `CB` always saves reservations with the belief that it is associated with the requested segment: `res[x]` is the ID of the train that has the reservation of `segments[x]` (the `CB`'s array) and `segments` always contains the `CB`'s associated segments. Instead of looking at the case where reservations can already be found, the situation where the reservation has not yet been saved but is about to be saved can be checked. This is when the `Train` is in its `Reserving` location, while the `CB` in its `SegmentChecked` location has a `tid` value that matches the ID of the `Train` and a `result` value greater than -1 indicating a possible reservation.

Whenever the above holds, the ID of the segment that the `Train` sends must exist in the `CB`'s `segments` array.

If `tsc` is the `Train` that `CB cb` is communicating with when it is in its `SegmentChecked` location, then this property can be formulated as:

$$\texttt{AG cb.SegmentChecked} \wedge \texttt{cb.result > -1} \rightarrow (\exists_{k \in [0,2]}\ \texttt{cb.segments[k] = tsc.segments[tsc.resSegIndex]})$$

Expressed in UPPAAL for all relevant instances:

```
A[] forall(i:t_id) forall(j:cB_id)
    (Train(i).Reserving && CB(j).SegmentChecked && CB(j).tid == i
    && CB(j).result > -1) imply
        (exists(k:int[0,2]) CB(j).segments[k] ==
            Train(i).segments[Train(i).resSegIndex])
```

- *A reservation is only successful if the requested segment is not already reserved.*

  This property must also be checked before the new reservation is actually saved. Otherwise it will not be possible to see whether the segment was reserved before or not. The query must ensure that in the states where a reservation save is about to happen, `res[result]` is equal to -1. This can be checked in these states because the value is only updated during the acknowledgement synchronization on the outgoing edge of `SegmentChecked`.

$$\texttt{AG cb.SegmentChecked} \wedge \texttt{cb.result > -1} \rightarrow \texttt{(cb.res[cb.result] = -1)}$$

Expressed in UPPAAL for all relevant instances:

```
A[] forall(i:t_id) forall(j:cB_id)
    (Train(i).Reserving && CB(j).SegmentChecked
    && CB(j).tid == i && CB(j).result > -1) imply
        CB(j).res[CB(j).result] == -1
```

### 7.4.2 Requesting a Switch and a Lock

Also the operations related to the switching and locking of points have requirements that can be formulated as properties and verified.

- *A train never has more locks than the lock limit.*

Similar to reservations, there is a maximum number of locks that a `Train t` may have at a time. Because every `Train` explicitly counts the number of obtained locks as `locks`, this number can simply be compared to the lock limit.

$$\texttt{AG t.locks} \leq \texttt{lockLimit}$$

Expressed in UPPAAL for all relevant instances:

```
A[] forall(i:t_id) Train(i).locks <= lockLimit
```

- *A lock is only successful if the involved switch box is in the route of the requesting train.*

  To find the granted locks, one can go through all control boxes and look for the ones with `lockedBy` $\neq$ `-1`. Every time such a case is found, the value of `lockedBy` is the ID of the `Train` with the lock and this `Train` must have the `CB`'s ID in its `boxes` array.

  If `tl` is the `Train` that has the lock at a `CB cb` whenever `cb.lockedBy` $\neq$ `-1` and `CB(t,i)` is the `CB` with ID `t.boxes[i]`, this property can be formulated as:

$$\texttt{AG cb.lockedBy} \neq \texttt{-1} \rightarrow \exists_{j \in cBRoute\_i}(\texttt{CB(tl,j) = cb})$$

  Expressed in UPPAAL for all relevant instances:

```
A[] forall(i:cB_id) (CB(i).lockedBy != -1) imply
    (exists(j:cBRoute_i) Train(CB(i).lockedBy).boxes[j] == i)
```

- *A lock is only successful if the requesting train has the reservation for the stem segment at the switch box and one other segment.*

  Because a `CB` stores information about both who it has locked for and who it has reserved for, the `CB` can simply check that the one who has locked it, has the reservation for its stem segment and the connected segment since the switch box will have switched before locking.

$$\begin{aligned}&\texttt{AG cb.lockedBy} \neq \texttt{-1} \rightarrow \\ &\quad \texttt{cb.res[0] = tl.id} \wedge \exists_{j \in [0,2]}(\texttt{cb.segments[j] = cb.connected} \\ &\qquad\qquad\qquad\qquad\qquad\qquad \wedge\ \texttt{cb.res[j] = tl.id})\end{aligned}$$

  Expressed in UPPAAL for all relevant instances:

```
A[] forall(i:cB_id) CB(i).lockedBy != -1 imply
    CB(i).res[0] == CB(i).lockedBy &&
    exists(j:int[0,2]) CB(i).segments[j] == CB(i).connected && CB(i).
        res[j] == CB(i).lockedBy
```

- *A lock is only successful if the point involved in the request was unlocked prior to the request.*

  This property also requires the query to examine the situation right before a locking is actually saved. The situation is found when a `CB` is in its `Switched` state meaning that it has already accepted the request and has switched but not yet locked. When a `CB` is about to lock its `Point`, its `lockedBy` variable should be equal to -1, which signifies an unlocked `Point`.

  `AG cb.Switched` $\rightarrow$ `cb.lockedBy = -1`

  Expressed in UPPAAL for all relevant instances:

```
A[] forall(i:cB_id)
    CB(i).Switched imply CB(i).lockedBy == -1
```

- *A switch is only successful if the requested connection is of segments that are adjacent in the train's route and the stem segment and plus or minus segment of the switch box.*

  This property can be checked by comparing the segments that a `Train` sends during a switch/lock request and the `CB`'s associated segments. This should be done in the states where a `CB` is in its `Switched` location and the `Train` that it communicates with is in its `Locking` location. In such cases, one of the segments that the `Train` sends must match the stem at the switch box and one other associated segment.

  If `ts` is the `Train` that `CB` `cb` communicates with when it is in its `Switched` location, then this property can be formulated as:

  `AG ts.Locking` $\wedge$ `cb.Switched` $\rightarrow$ $\exists_{k\in[1,2]}$
  `(cb.segments[0] = ts.segments[ts.lockIndex-1]` $\wedge$
  `cb.segments[k] = ts.segments[ts.lockIndex])` $\vee$
  `(cb.segments[0] = ts.segments[ts.lockIndex]` $\wedge$
  `cb.segments[k] = ts.segments[ts.lockIndex-1])`

  Expressed in UPPAAL for all relevant instances:

```
A[] forall(i:t_id) forall(j:cB_id) (Train(i).Locking &&
CB(j).Switched && CB(j).tid == i) imply
    (exists(k:int[1,2]) (CB(j).segments[0] == Train(i).segments[Train(i
        ).lockIndex-1]
```

```
        && CB(j).segments[k] == Train(i).segments[Train(i).lockIndex])
             ||
    (CB(j).segments[0] == Train(i).segments[Train(i).lockIndex] &&
        CB(j).segments[k] == Train(i).segments[Train(i).lockIndex-1]))
```

- *A switch box only switches and locks its point if no train is in its critical section.*

  The CB locations that correspond to the switching and locking of a Point are the CBs LockChecked, Switching and Switched locations. In these cases, no Train should be in its DoubleSegment location while the CB that it passes has the same ID as the one that is switching/locking.

  $$\texttt{AG cb.Switched} \lor \texttt{cb.Switching} \rightarrow \neg\exists_{t \in Trains}\ \texttt{(t.DoubleSegment} \land \texttt{CB(t,index+1) = cb)}$$

  Expressed in UPPAAL for all relevant instances:

```
A[] forall(i:cB_id)
    (CB(i).Switched || CB(i).Switching) imply
        !(exists(j:t_id) (Train(j).DoubleSegment &&
            Train(j).boxes[Train(j).index+1] == i))
```

  Although a query is written for this property, it is clear that no Train can ever pass a CB while it is switching or locking since these are modelled as completely unrelated locations. This makes this property correct by construction.

### 7.4.3 Passing a Control Box

The final train interaction with control boxes is the passing of the control boxes. The requirements for passing a control box are as follows:

- *A train only passes a switch box if it has been locked for the train.*

  If a Train is in its DoubleSegment location and the CB that it passes is a switch box, then the switch box in question is locked for the Train.

  $$\texttt{AG t.DoubleSegment} \land \texttt{points[cbp.id]} \neq \texttt{-1} \rightarrow \texttt{cbp.lockedBy = t.id}$$

  Expressed in UPPAAL for all relevant instances:

```
A[] forall(i:t_id) Train(i).DoubleSegment &&
    points[Train(i).boxes[Train(i).index+1]] != -1 imply
        CB(Train(i).boxes[Train(i).index+1]).lockedBy == i
```

- *A train never passes the last control box in its route.*

  As described earlier, a train must halt at its final segment and not pass the last control box in its route. This can be checked by ensuring that whenever a `Train` is passing a `CB`, then that `CB` is not the last in the route.

  `AG t.DoubleSegment` $\rightarrow$ `cbp.id` $\neq$ `t.boxes[t.routeLength]`

  Expressed in UPPAAL for all relevant instances:

```
A[] forall(i:t_id) Train(i).DoubleSegment imply
    Train(i).boxes[Train(i).index+1] !=
        Train(i).boxes[Train(i).routeLength]
```

- *A train only enters a segment that it has the full reservation of.*

  This property means that a train that enters a new segment, has the reservation of that segment at both of the two associated control boxes. The states in which a `Train` is in its `DoubleSegment` are again examined. This time `index+1` - which is the index of the `Train`'s next segment and upcoming control box - can be used in the query. This value must be less than the `Train`'s `resSegIndex`, which is the index of the next segment that it does not have the full reservation of yet. In other words, the index of the segment that the `Train` wants to enter must be less than the index of the next segment that needs to be reserved since this means that the segment that it wants to enter has already been reserved.

  `AG t.DoubleSegment` $\rightarrow$ `t.index+1 < t.resSegIndex`

  Expressed in UPPAAL for all relevant instances:

```
A[] forall(i:t_id) Train(i).DoubleSegment imply
    Train(i).index+1 < Train(i).resSegIndex
```

## 7.5 State Space Consistency

When evaluating the previous requirements, it is naturally desired to have consistent state spaces. For example, when checking reservations in the previous

section, the values in a CB's res variable were checked, but for these reservations to actually be true reservations, the involved Trains should have these reservations saved in their state spaces as well.

The consistency that must be ensured between Trains and CBs are related to their reservation information and their lock information.[1]

- *The reservations saved in the state space of a Train are also saved in the state spaces of the involved CBs.*

  A reserved segment at index j of a Train's segments array should be reserved for that Train at the CB boxes[j]. If it is also fully reserved, then it should also be reserved at the CB at index j+1. That is, when $j \geq$ index $\wedge$ j < resSegIndex, it is safe to say that segments[j] has been reserved at boxes[j+1]. Similarly, if j > index $\wedge$ j < resCBIndex, then segments[j] has been reserved at boxes[j].

  AG $\forall_{j \in segRoute\_i}$ j$\geq$t.index $\wedge$ j<t.resSegIndex $\rightarrow$
  $\exists_{i \in [0,2]}$ (CB(t,j+1).segments[i] = t.segments[j] $\wedge$
  CB(t,j+1).res[i] = t.id) $\wedge$
  (j>t.index $\wedge$ j<t.resCBIndex $\rightarrow$
  $\exists_{i \in [0,2]}$ (CB(t,j).segments[i] = t.segments[j] $\wedge$
  CB(t,j).res[i] = t.id)

  Expressed in UPPAAL for all relevant instances:

```
A[]  forall(i:t_id) forall(j:segRoute_i)
    (j >= Train(i).index && j < Train(i).resSegIndex imply
        exists(l:int[0,2]) CB(Train(i).boxes[j+1]).segments[l] == Train
            (i).segments[j] && CB(Train(i).boxes[j+1]).res[l] == i) &&
    (j > Train(i).index && j < Train(i).resCBIndex imply
        exists(l:int[0,2]) CB(Train(i).boxes[j]).segments[l] == Train(i
            ).segments[j] && CB(Train(i).boxes[j]).res[l] == i)
```

- *The locks saved in the state space of a Train are also saved in the state space of the involved CBs.*

  The query for this property must go through the switch boxes that a Train considers locked and ensure that these CBs' lockedBy variables are equal to the Train's ID. requiresLock is used to see whether the Train sees a CB as a switch box, which then needs locking.

[1] For this implementation of the railway control system, the reverse of the two first properties could also be formulated, but these do not necessarily hold in a real system and they have therefore not been included here.

AG $\forall_{j \in cbRoute\_i}$(j>t.index $\wedge$ j<t.lockIndex $\wedge$ t.requiresLock[j] $\rightarrow$
CB(t,j).lockedBy = t.id)

Expressed in UPPAAL for all relevant instances:

```
A[] forall(i:t_id) forall(j:cBRoute_i)
    (j > Train(i).index && j < Train(i).lockIndex && Train(i).
        requiresLock[j] imply
       CB(Train(i).boxes[j]).lockedBy == i)
```

- *The number of saved locks in the state space of a* ***Train*** *is the same number of locks that it believes that it has.*

  The index `lockIndex` for the `CB` that a `Train` wishes to lock at next cannot be used directly to compute a `Train`'s number of locks, which is why the `locks` variable was introduced. This variable must be consistent with the locks derived from `lockIndex` since this variable is used when a `Train` checks which locks it has currently obtained. By using UPPAAL's `sum` function, it is possible to count the number of locks manually to compare with the value of `locks`. The function should go through all indices in `boxes` and count the switch boxes found between `index` and `lockIndex`.

  If $\sum_{j \in cB\_id} p(j)$ is the number of `j`'s that satisfy the property `p(j)`, then this can be formulated as:

  AG t.locks = $\sum_{j \in cBRoute\_i}$(j>t.index $\wedge$ j<t.lockIndex $\wedge$
  points[t.boxes[j]]>-1)

  Expressed in UPPAAL for all relevant instances:

```
A[] forall(i:t_id) Train(i).locks == (sum(j:cBRoute_i) (j > Train(i).
     index && j < Train(i).lockIndex && points[Train(i).boxes[j]] > -1))
```

- *The position of a* ***Point*** *in the global data is consistent with the actual position of the* ***Point***

  As the only global array, the `pointInPlus` array is not a constant because it is supposed to reflect the current position of a `Point` since this information is used to update a `Train`'s position.

  While a `Point` is switching, the information about its position in the array has not yet been updated. However, since this information is only used when a `Train` passes the associated `CB` - which it cannot while the `Point` is switching (verified by another property) - this moment of inconsistency is not a problem by itself, but the property still has to be checked in the non-switching locations.

$$\texttt{AG } \neg \texttt{p.SwitchingPM} \wedge \neg \texttt{p.SwitchingMP} \rightarrow \texttt{pointInPlus[p.id]} \rightarrow \texttt{p.Plus} \wedge \neg \texttt{pointInPlus[p.id]} \rightarrow \texttt{p.Minus}$$

Expressed in UPPAAL for all relevant instances:

```
A[] forall(i:p_id) Initializer.Initialized &&
!Point(i).SwitchingPM && !Point(i).SwitchingMP imply
    (pointInPlus[i] imply Point(i).Plus) &&
    (!pointInPlus[i] imply Point(i).Minus)
```

- *A `CB`'s information about its associated `Point`'s position is consistent with the `Point`'s actual position.*

  When a `CB` receives a switch/lock request, it uses its `connected` variable to determine whether it needs to issue a switch operation before locking. This variable must hence be consistent with the actual position of the associated `Point` to ensure that switch operations happen correctly.

  While a `Point` is switching, it has not yet informed the associated `CB` about the completion of the switch. Similar to the previous property, this means that there will be inconsistency between the `Point`'s position and the `CB`'s `connected` variable in some states. This is again not a problem since the `CB` will only interact with a `Train` after the switch is completed. Therefore, the property only needs to hold in the states where the `CB` is not in its `Switching` location.

  If `cbpo` is the `CB` associated with a `Point p`, then the property can be formulated as:

$$\texttt{AG points[cbpo.index] > -1} \wedge \neg \texttt{cbpo.Switching} \rightarrow (\texttt{cbpo.connected = cbpo.segments[1]} \rightarrow \texttt{p.Plus} \wedge \texttt{cbpo.connected = cbpo.segments[2]} \rightarrow \texttt{p.Minus})$$

  Expressed in UPPAAL for all relevant instances:

```
A[] forall(i:cB_id) Initializer.Initialized && points[i] > -1 && !CB(i)
    .Switching imply
    ((CB(i).connected == CB(i).segments[1]) imply Point(points[i]).Plus
        &&
    (CB(i).connected == CB(i).segments[2]) imply Point(points[i]).Minus
        )
```

## 7.6 Liveness Properties

Although it is not a requirement in this project, it is a natural desire that a train should be able to reach its final destination. This means that there should be an

execution path that will allow all trains to reach their destinations. However, not all possible execution paths will or need to lead to the state where all trains are at their destinations since it is possible for livelocks to occur as mentioned in section 5.2.1.

The possibility of livelocks means that the desired progress property will not be required for all paths, but instead the query should use the existential quantifier and the 'finally' operator to find just any execution path that will lead to the state where all `Train`s have arrived at their destinations. That is, *there exists a path that will eventually lead to a state in which all trains are in their* ***`Arrived`*** *locations.*

For a single `Train t`, this can formally be formulated as:
`EF t.Arrived`

Expressed in UPPAAL for all relevant instances:

```
    E<> forall(i:t_id) Train(i).Arrived
```

## 7.7 Other Properties

### 7.7.1 Deadlock

With UPPAAL's verifier, it is also easy to formulate a query that checks whether there are any deadlocks. For this system, it is desired that deadlocks should never occur. This can be checked with the following query:

```
A[] !deadlock
```

### 7.7.2 Well-Formedness

The topic of well-formedness was also discussed in section 6.8. The requirement of well-formedness of instances could be checked with the queries:

```
A<> forall(i:cB_id) not(CB(i).Initial)
```

```
A<> forall(i:t_id) not(Train(i).Initial)
```

Chapter 8

# Verifying Safety Properties for the UPPAAL Model

As explained in section 4.2, the most important task of the control strategy is to ensure the safety of trains by preventing collisions and derailments. When checking the implementation of the strategy, one quickly realizes that it is impossible to check it for all possible configurations. Hence, with the inspiration from a compositional model checking approach as seen in [MFH17], the model is instead checked for the possible scenarios expressed in small networks. The idea is that if the model is safe for small configurations, then it is also safe for larger configurations.

All checks have been done with reservation limit 1 and lock limit 1 for the fastest execution, but these limits do not have any influence on the safety because the number of reservations and locks does not affect a train's position or movement. The queries that are used for the checked properties are the ones described in chapter 7. The configuration data used for the checks can be found in appendix B and the complete models and results can be found in *s144449-s144456_FVDRCS.zip > Models > Check Configurations and Results.*

## 8.1 No Collision

The smallest possible networks in which collisions can occur include two trains. The networks where the 'no collision' property is initially true must therefore also include at least two segments and at least one segment that is in both of the trains' routes. The property that is checked in this section is mainly the 'no collision' property, but also the liveness property will be checked.

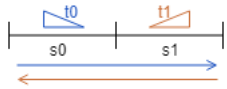
### 8.1.1 Check 1


The first check shows a train `t0`, with the route `{s0, s1}` and a train `t1` with the route `{s1}`. This example simply checks that a train will never enter an occupied segment. Since `t1` is already at its destination, it will not be moving so it should never be possible for `t0` to complete its route.

| Property | Expected Result | Actual Result |
|---|---|---|
| No collision | Satisfied | Satisfied |
| Liveness | Not satisfied | Not satisfied |

**Table 8.1:** Check 1 results

### 8.1.2 Check 2

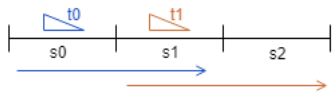



In this check, train `t1` has the route `{s1, s0}`. This means that the two trains will attempt to switch places, which should not be possible.

| Property | Expected Result | Actual Result |
|---|---|---|
| No collision | Satisfied | Satisfied |
| Liveness | Not satisfied | Not satisfied |

**Table 8.2:** Check 2 results

### 8.1.3 Check 3




In this final check to show no collision, another segment is added after segment `s1` and train `t1` then has the route `{s1,s2}`. This checks that it is possible for `t0` to eventually enter `s1`, because `t1` will leave it. This shows that check 1 and check 2 did not just succeed because nothing ever happens.

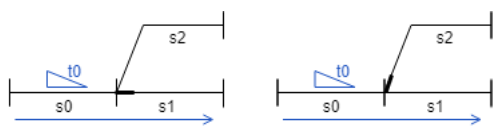
| Property | Expected Result | Actual Result |
|---|---|---|
| No collision | Satisfied | Satisfied |
| Liveness | Satisfied | Satisfied |

**Table 8.3:** Check 3 results

## 8.2 No Derailment

In order for derailments to be possible at all, the network must contain at least one point. At each point, a train can come from three different directions: From the stem segment, the plus segment or the minus segment. When a train passes a point coming from the stem segment, it can enter either the plus segment or the minus segment. If a train instead comes from the plus segment or the minus segment, it can only enter the stem segment. Furthermore, the point can be in two different positions: Plus or minus. These cases give rise to eight different scenarios. The properties that are mainly checked in this section are the two 'no derailment' properties, but also the liveness property is checked again.

### 8.2.1 Check 4 + 5




In these two checks, the train comes from the stem direction and wishes to continue onto the plus segment. The point in the network on the left-hand side

is in its plus position while the point in the network on the right-hand side is in its minus position. In the network on the left, the train should be able to move to `s1` without a need for a switching of the point, while this is needed in the network on the right.

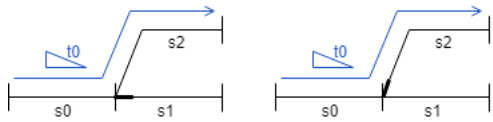
| Property | Expected Result | Actual Result |
|---|---|---|
| No derailment (no switching) | Satisfied | Satisfied |
| No derailment (segments connected) | Satisfied | Satisfied |
| Liveness | Satisfied | Satisfied |

**Table 8.4:** Check 4 results

| Property | Expected Result | Actual Result |
|---|---|---|
| No derailment (no switching) | Satisfied | Satisfied |
| No derailment (segments connected) | Satisfied | Satisfied |
| Liveness | Satisfied | Satisfied |

**Table 8.5:** Check 5 results

### 8.2.2 Check 6 + 7




The two scenarios here are similar to the previous two except for the fact that `t0` wants to move to the minus segment instead. This means that the point on the left-hand side network needs switching this time while the point on the right-hand side network does not.

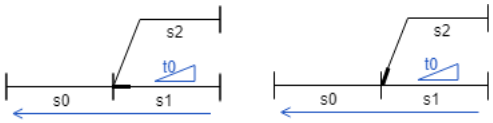
| Property | Expected Result | Actual Result |
|---|---|---|
| No derailment (no switching) | Satisfied | Satisfied |
| No derailment (segments connected) | Satisfied | Satisfied |
| Liveness | Satisfied | Satisfied |

**Table 8.6:** Check 6 results

| Property | Expected Result | Actual Result |
|---|---|---|
| No derailment (no switching) | Satisfied | Satisfied |
| No derailment (segments connected) | Satisfied | Satisfied |
| Liveness | Satisfied | Satisfied |

**Table 8.7:** Check 7 results

### 8.2.3 Check 8 + 9




The two scenarios here are for the cases where the train comes from the plus segment. Again, there is one scenario that requires switching, while the other one does not.

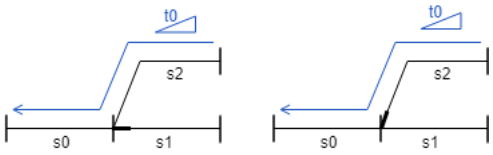
| Property | Expected Result | Actual Result |
|---|---|---|
| No derailment (no switching) | Satisfied | Satisfied |
| No derailment (segments connected) | Satisfied | Satisfied |
| Liveness | Satisfied | Satisfied |

**Table 8.8:** Check 8 results

| Property | Expected Result | Actual Result |
|---|---|---|
| No derailment (no switching) | Satisfied | Satisfied |
| No derailment (segments connected) | Satisfied | Satisfied |
| Liveness | Satisfied | Satisfied |

**Table 8.9:** Check 9 results

### 8.2.4 Check 10 + 11



The last two checks in this series of checks are again similar to the previous two checks, but the train now comes from the minus segment.

| Property | Expected Result | Actual Result |
|---|---|---|
| No derailment (no switching) | Satisfied | Satisfied |
| No derailment (segments connected) | Satisfied | Satisfied |
| Liveness | Satisfied | Satisfied |

**Table 8.10:** Check 10 results

| Property | Expected Result | Actual Result |
|---|---|---|
| No derailment (no switching) | Satisfied | Satisfied |
| No derailment (segments connected) | Satisfied | Satisfied |
| Liveness | Satisfied | Satisfied |

**Table 8.11:** Check 11 results

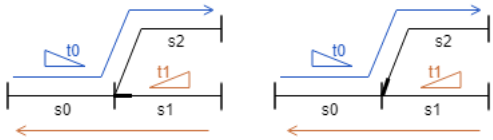
## 8.3 No Collision and No Derailment

The checks so far show that no trains collide with each other when there are no points and that trains never derail when no other trains interfere. To ensure that there are also no collisions and no derailments when two trains want to pass the same point, the previous scenarios must be combined.

If two trains move towards a point and both are coming from the same direction (stem, plus or minus), one train will be on a segment behind the other. These cases would therefore just be a combination of two 'no derailment' scenarios where the trains take turns to interact with and pass the switch box.

If one train comes from the stem segment and the other train comes from the plus or minus segment and they want to switch places, then this would be the same as a 'no collision' case since the trains would not even be able to reserve the segments that they need for obtaining a lock.

Eliminating these cases leaves behind six scenarios.




### 8.3.1 Check 12 + 13

In these two scenarios, one train comes from the stem and wants to move to the minus segment while the other train comes from the plus segment and wants to move to the stem. On the left-hand side network, the point is in its plus position and on the right-hand side network, the point is in its minus position.

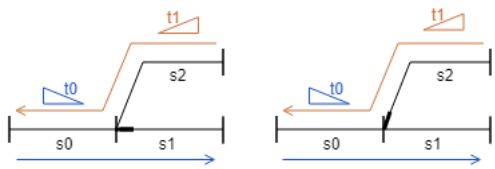
| Property | Expected Result | Actual Result |
|---|---|---|
| No collision | Satisfied | Satisfied |
| No derailment (no switching) | Satisfied | Satisfied |
| No derailment (segments connected) | Satisfied | Satisfied |
| Liveness | Satisfied | Satisfied |

**Table 8.12:** Check 12 results

| Property | Expected Result | Actual Result |
|---|---|---|
| No collision | Satisfied | Satisfied |
| No derailment (no switching) | Satisfied | Satisfied |
| No derailment (segments connected) | Satisfied | Satisfied |
| Liveness | Satisfied | Satisfied |

**Table 8.13:** Check 13 results

### 8.3.2 Check 14 + 15




These cases are similar to the previous two, but train `t1` now comes from the minus segment while train `t0` wants to move to the plus segment.

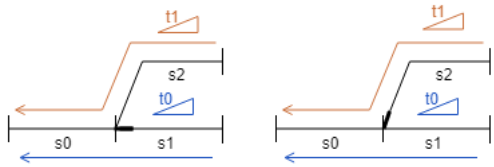
| Property | Expected Result | Actual Result |
|---|---|---|
| No collision | Satisfied | Satisfied |
| No derailment (no switching) | Satisfied | Satisfied |
| No derailment (segments connected) | Satisfied | Satisfied |
| Liveness | Satisfied | Satisfied |

**Table 8.14:** Check 14 results

| Property | Expected Result | Actual Result |
|---|---|---|
| No collision | Satisfied | Satisfied |
| No derailment (no switching) | Satisfied | Satisfied |
| No derailment (segments connected) | Satisfied | Satisfied |
| Liveness | Satisfied | Satisfied |

**Table 8.15:** Check 15 results

### 8.3.3 Check 16 + 17




In the last two cases, one train comes from the plus segment while the other train comes from the minus segment. Since only one train may be on `s0` at any time, it is not possible for all trains to complete their routes.

| Property | Expected Result | Actual Result |
|---|---|---|
| No collision | Satisfied | Satisfied |
| No derailment (no switching) | Satisfied | Satisfied |
| No derailment (segments connected) | Satisfied | Satisfied |
| Liveness | Not satisfied | Not satisfied |

**Table 8.16:** Check 16 results

| Property | Expected Result | Actual Result |
|---|---|---|
| No collision | Satisfied | Satisfied |
| No derailment (no switching) | Satisfied | Satisfied |
| No derailment (segments connected) | Satisfied | Satisfied |
| Liveness | Not satisfied | Not satisfied |

**Table 8.17:** Check 17 results

Since the safety properties were successfully verified for these small cases, they should also hold for networks that are created by mirroring and/or combining them. Some examples of this can be seen in chapter 13 where the efficiency of the model will be examined as well.

Chapter 9

# UPPAAL Models for Other Control System Variants

In this chapter, some alternative UPPAAL models will be presented. In section 9.1, a variant that enforces a stricter operation sequence is introduced while section 9.2 introduces a variant with an additional 'cancel' operation, which will allow trains to cancel obtained reservations and locks. Finally, section 9.3 introduces a variant in which instances read from the global declarations every time certain data is needed instead of reading the data from a copy of them in their local declarations.

## 9.1 Restricted Model

The model described in chapter 6 was created to be somewhat flexible, since `Train` instances could start operations from `SingleSegment` in any order as long as their guards yielded true.

For example, if a `Train` is allowed to have at least one full reservation at a time and the upcoming control box is a switch box, then it can choose to reserve the upcoming segment at just the upcoming control box before it requests a lock or it can reserve the upcoming segment at both associated control boxes before

requesting the lock. Similarly, if many reservations and locks are allowed, trains can obtain the rights to large parts of their routes before they move or they can choose to move immediately when possible.

The larger `resLimit` and `lockLimit` are, the more possible sequences of operations exist. This means that the state space can become quite large very quickly so to reduce the number of possible interleavings and thereby the resulting state space, the model can be modified to enforce a more specific sequence of operations.

### 9.1.1 Design

The idea behind the new model is that at any time in the system, a `Train` instance should have at most one operation that it can perform, i.e. one enabled edge at a time.

There are different ways for a `Train` to decide which action it should take. The most straightforward sequence would be to

1. make one full segment reservation of the upcoming segment
2. obtain one lock (if the upcoming control box is a switch box)
3. pass the upcoming control box

Once the above sequence of actions has been carried out successfully, the `Train` will have no reservations or locks left so even though this sequence is very straightforward, this defeats the purpose of being allowed to have multiple reservations and locks at the same time. Keeping the flexibility with `resLimit` and `lockLimit` will instead emphasize the difference between the models, since the increase of limits causes the increase in the number of operation options.

A way to enforce a specific sequence while still allowing multiple reservations and locks per `Train` is to force a `Train` to take advantage of the limits to the fullest. This is the strategy that has been used here. The order of operations is still the same as before, but once a `Train` has started on a new type of operation, it should repeat that operation as many times as possible before it may move on to the next operation. This means that a `Train` should

1. reserve segments until it is no longer possible

2. obtain locks until it is no longer possible
3. pass control boxes until it is no longer possible

With this strategy, the strategy described earlier with only a single full segment reservation and lock at a time could then simply be modelled by setting the values of the two limit variables to 1.

### 9.1.2 Implementation

To implement this stricter design, the `SingleSegment` location is divided into several locations to represent the state after the performance of a specific operation. One must then ensure that the `Train` template only has one enabled edge in each location. This means that there should be a location from which `possibleToReserve` is true, a location from which `possibleToLock` is true and a location from which `possibleToPass` is true.

Since reserving a segment should come first in the sequence of operations, the edge going to `Reserving` should be the first one coming after the initialization edge. The new `Train` state machine can be seen in figure 9.1.

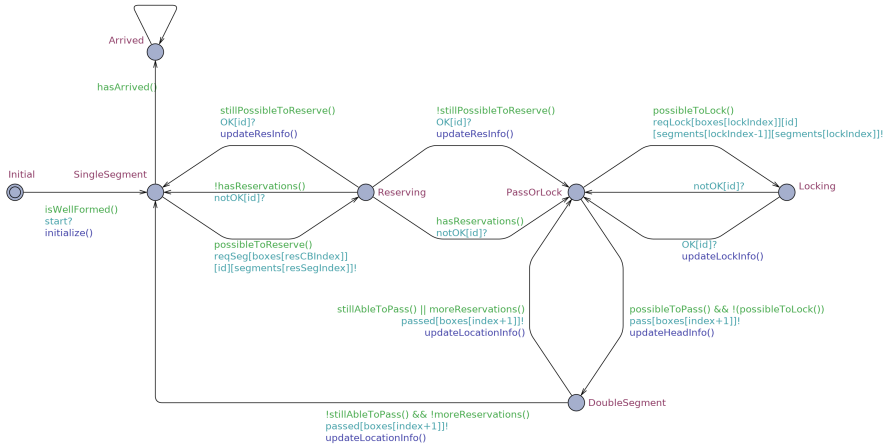



**Figure 9.1:** Restricted version of `Train`

A `Train` starts in its `Initial` location and moves to the `SingleSegment` location where it can move to `Arrived` if it has completed its route. The only other option from the `SingleSegment` location is to reserve a segment.

#### 9.1.2.1 Segment Reservation

When a `Train` attempts to reserve a segment, it moves to the location `Reserving`. From this location, there are four outgoing edges. The edges that go back to the `SingleSegment` location are for the following cases:

- There are more reservations that can be made and the last reservation was successful (synchronize on `OK`). In order to ensure that there are more reservations to be made when the `Train` is back in `SingleSegment`, a new function `stillPossibleToReserve` is introduced.

```
bool stillPossibleToReserve() {
    int[0,1] tempBit = resBit^1;
    int[0,NSEG]  tempSegIndex = (tempBit == 0) ? resSegIndex + 1 :
        resSegIndex;
    return tempSegIndex < routeLength && tempSegIndex - 1 -index <
        resLimit;
}
```

  The functionality of this predicate is the same as `possibleToReserve`, but instead of assessing the current value of `resSegIndex`, it looks at its *future* value after an update.

- The reservation failed (synchronize on `notOK`) and the `Train` currently has no reservations that will allow it to move to the next segment or request locks. In this case, the `Train` has to return to `SingleSegment` so it can make another reservation attempt later. This guard is implemented with a new function `hasReservations`.

```
bool hasReservations() {
    return resSegIndex > index + 1;
}
```

The remaining edges go to the location `PassOrLock`. From here it is possible to request locks or pass control boxes.

#### 9.1.2.2 Locking

The structure of the locking process is identical to the first model. A `Train` moves from the `PassOrLock` location to the `Locking` location if `possibleToLock` is true and from here it can synchronize on `OK[id]` or `notOK[id]`. It then returns to `PassOrLock` and continues to request locks until `possibleToLock` returns false.

When it is no longer possible to request locks and it is possible to pass the upcoming `CB`, the `Train` will instead move to `DoubleSegment`.

#### 9.1.2.3 Passing

Passing a `CB` is not much different from the first model's passing process either, but instead of returning to the `SingleSegment` location after having passed a `CB`, a `Train` must check whether it can pass more control boxes or whether it has more pending reservations that need to be handled.

That is, if `stillAbleToPass` or `moreReservations` is true, the `Train` should go back to `PassOrLock` so it can continue passing more `CB`s or get locks for its reservations. The `stillAbleToPass` function is as follows:

```
bool stillAbleToPass() {
    return resSegIndex > index + 2 && lockIndex > index + 2 && index + 2 <
        routeLength;
}
```

This function is similar to `possibleToPass` with the only difference being that it considers the value of `index` *after* an update instead of the current value. The same applies to the `moreReservations` function.

```
bool moreReservations() {
    return resSegIndex > index + 2;
}
```

If both of these functions evaluate to false, the `Train` will instead move from the `DoubleSegment` location to the `SingleSegment` location. It then either has completed its route or has more reservations to make.

The new `Train` model's local declarations can be seen in appendix C.1. The remaining part of the UPPAAL model is unchanged.

#### 9.1.2.4 Comparison Between First and Restricted Model

As explained before, the new model is supposed to be a restricted version of the first model. This means that the states that the new model generates should be generated by the first model as well. The locations cannot be mapped directly between the two models because the new `Train` has more locations than the original, but it is possible to execute the same actions from the restricted model in the first model.

The possible sequences of locations that can be found in the new model and the corresponding sequences in the first model can be seen in table 9.1.

The table shows that the possible action sequences in the restricted model can be

| Restricted Model | First Model |
|---|---|
| Initial → SingleSegment | Initial → SingleSegment |
| SingleSegment → Reserving $\xrightarrow{\texttt{OK}}$ SingleSegment | SingleSegment → Reserving $\xrightarrow{\texttt{OK}}$ SingleSegment |
| SingleSegment → Reserving $\xrightarrow{\texttt{OK}}$ PassOrLock | SingleSegment → Reserving $\xrightarrow{\texttt{OK}}$ SingleSegment |
| SingleSegment → Reserving $\xrightarrow{\texttt{notOK}}$ SingleSegment | SingleSegment → Reserving $\xrightarrow{\texttt{notOK}}$ SingleSegment |
| SingleSegment → Reserving $\xrightarrow{\texttt{notOK}}$ PassOrLock | SingleSegment → Reserving $\xrightarrow{\texttt{notOK}}$ SingleSegment |
| PassOrLock → Locking $\xrightarrow{\texttt{OK}}$ PassOrLock | SingleSegment → Locking $\xrightarrow{\texttt{OK}}$ SingleSegment |
| PassOrLock → Locking $\xrightarrow{\texttt{notOK}}$ PassOrLock | SingleSegment → Locking $\xrightarrow{\texttt{notOK}}$ SingleSegment |
| PassOrLock → DoubleSegment → PassOrLock | SingleSegment → DoubleSegment → SingleSegment |
| PassOrLock → DoubleSegment → SingleSegment | SingleSegment → DoubleSegment → SingleSegment |
| SingleSegment → Arrived | SingleSegment → Arrived |

**Table 9.1:** Location sequences in restricted and first model

found in the first model as well. While the locations in the `Train` in the restricted model only have one outgoing edge enabled at a time each, the `SingleSegment` location of the first model's `Train` can have multiple edges enabled at the same time. This means that if the railway control system is safe for a configuration in the first model, it will also be safe for the configuration in this model.

## 9.2 Extended Model

As mentioned in section 5.2.1, trains can end up in livelock situations if two trains need reservations or locks that are taken by the other. Therefore, a desired feature would be to be able to cancel reservations and locks. This will allow trains to give up obtained reservations and locks that may allow other trains to proceed. In a real-life situation, it would also be useful for a train to be able to cancel reservations and locks if it is unable to continue its route due to an error.

### 9.2.1 Design

Like the other train operations, cancelling should only be possible if a train is not already in the middle of executing another operation, i.e. one could introduce a 'cancel reservation' operation and a 'cancel lock' operation that are available if a train is idle. The cancelling of reservations and the cancelling of locks are not completely unrelated though and to keep the system consistent with the design of the first model, a train should not be completely free to cancel just anything.

The first model was designed such that reservations and locks are made in the order that they are needed. This means that cancelling them should be done in

the reverse order, otherwise a TCC will no longer know which reservations and locks it has since this knowledge is based on indices and the order of segments in the route. Similar to how reserving and locking segments far away from a `Train`'s current position do not make sense in practice, it does also not make sense to cancel reservations and locks close to the `Train` if it has reservations and locks further away.

Another matter to decide is whether it actually makes sense for a TCC to choose whether it wants to cancel a reservation or cancel a lock. One of the requirements for locking was that the segments involved in the requested connection should already have been reserved by the requesting TCC at the control box in question. This means that if a TCC decides to cancel a lock, but still has two reservations at the control box, then it is impossible for other `Train`s to obtain the released lock anyway. It is hence useless to cancel a lock if not also reservations are cancelled.

Conversely, if a reservation is cancelled, a TCC may be forced to cancel a lock as well since it may only have a lock at control boxes at which it has the reservations of the connected segments.

Because of the close relation between the two types of cancellations, they are merged into one operation such that a TCC primarily cancels its most recent segment reservation and a lock is then automatically cancelled if it required the cancelled reservation.

The requirements for the cancel operation have been summarized in table 9.2.

**Cancel reservation of segment (and lock)**

| Sender | TCC |
|---|---|
| Receiver | Control box |
| TCC Requirements | • The TCC has the reservation for the segment at the control box<br>• The TCC's current position does not require the reservation<br>• The segment is the most recent segment that the TCC reserved |

**Table 9.2:** Requirements for cancelling reservations of segments

### 9.2.2 Implementation

To implement the cancel operation, only the global declarations, the `Train` template and the `CB` template need to be updated. A `Train` cancels a segment `s` at a control box `cb` by emitting on the channel `cancel[cb][s]`, which must then be introduced in the global declarations. It is here assumed that `Train`s do not send segment IDs that they do not have the reservation for.

```
chan cancel[NCB][NSEG];
```

#### 9.2.2.1 The `Train` Template

The state machine of `Train` now needs an edge from the `SingleSegment` location to itself in order to use the cancel operation. The `Train` does not need an acknowledgement since there are no `CB` requirements for cancellations. The instances will therefore synchronize at the same time as they update their state spaces.

The updated state machine for `Train` can be seen in figure 9.2 where two new edges are added from `SingleSegment`. Both edges use the new guard function `possibleToCancel`:

```
bool possibleToCancel(){
    return resCBIndex > index + 1;
}
```

This function checks whether the `CB` that the `Train` needs to reserve at next is ahead of the upcoming `CB`. If it is, it means that the `Train` has already at least obtained the reservation of the segment at the upcoming `CB` so there is at least one reservation that is possible to cancel.

The part that deviates in the two cancel edges' guards is the value check of `resBit`, which is used to determine which channel the `Train` should cancel on.

If `resBit = 0`, it means that the most recent reservation completed a full reservation of a segment. Cancelling it must therefore be done at the same `CB` that the next reservation is needed at, while the segment that should be cancelled is the previous segment.

```
cancel[boxes[resCBIndex]][segments[resSegIndex - 1]]
```

If instead `resBit = 1`, it means that the most recently performed reservation was for the segment at `resSegIndex`, but at the previous `CB`. The channel that must be cancelled on is hence:

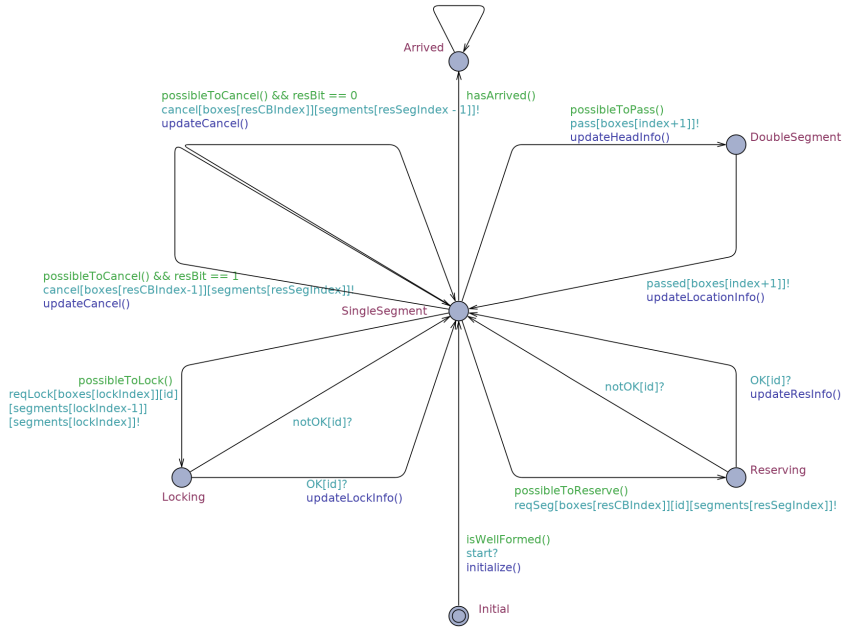



**Figure 9.2:** `Train` with cancel operation

```
cancel[boxes[resCBIndex-1]][segments[resSegIndex]]
```

Regardless of which reservation is cancelled, the TCC's state space is updated with `updateCancel`.

```
bool void updateCancel(){
    resSegIndex = (resBit==0) ? resSegIndex - 1 : resSegIndex;
    resCBIndex = (resBit==1) ? resCBIndex - 1 : resCBIndex;
    resBit = resBit^1;

    if(requiresLock[resCBIndex] && lockIndex > resCBIndex){
        locks--;
        lockIndex = resCBIndex;
    }
}
```

This function first reverses the update made by `updateResInfo` and if the reservation was needed for an obtained lock, the lock is removed and `lockIndex` is set to the index of the control box in the cancel operation. This is done to indicate that this lock must once again be requested.

#### 9.2.2.2 The CB Template

The state machine for the CB template now also needs a new edge that can be used for the synchronization (see figure 9.3). On this edge, the received segment ID is used in the the update function `updateCancel`.

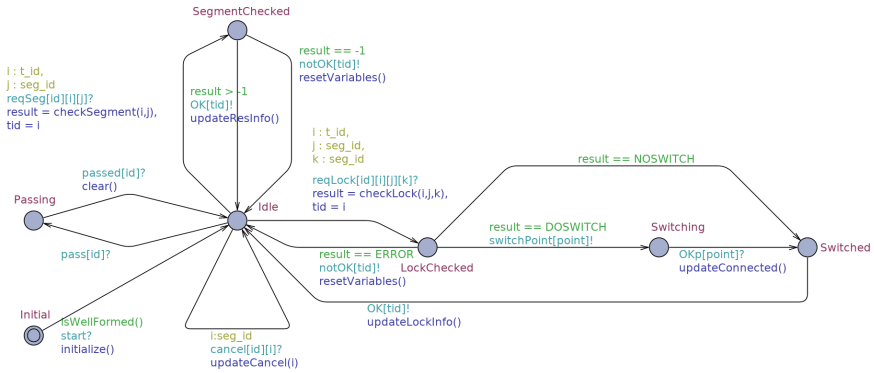



**Figure 9.3:** CB with cancel operation

```
bool void void updateCancel(seg_id s){
    for(i:int[0,2]){
        if(segments[i] == s){
            if(res[i] == lockedBy){
                lockedBy = -1;
            }
            res[i] = -1;
        }
    }
}
```

Similarly to `Train`'s `updateCancel` function, this function updates the state space, but of the `CB`. Given a segment ID `s`, the function looks for it in its `segments` array and uses its index to clear the reservation in the `res` array. It then also resets its 'locked by' value if the `Train` ID of the one with the lock is the same as the one with the reservation.

It can be here be seen that the `CB` blindly cancels the reservation without making sure that the sender is actually the one with the reservation. The ID of the `Train` could be sent as well, but the properties that check the consistency between the state spaces of `CB`s and `Train`s actually also implicitly checks that `Train`s only cancel their own reservations and locks.

The complete list of local declarations for the new `Train` template and the new

CB template can be seen in appendix C.2.

### 9.2.3 Comparison Between First and Extended Model

Since the system described in this section is an extension of the first model, it naturally has different states in its state space. So, unlike the restricted model, this model cannot rely on the verification of properties of the first model, but it must have the properties verified from scratch again.

## 9.3 Global Model

The *Global Model* is a variant of the first model in which Trains and CBs do not copy their data from the global declarations during the initialization step. Instead they access the data directly from the global declarations every time this is needed.

The purpose of this variant is to inspect whether accessing global constants rather than local variables has an influence on the efficiency of verification.

### 9.3.1 Implementation

In the Train template, the functions that are changed are initialize and updateHeadInfo.

In the initialize function, the copying of data into segments and boxes has been removed and the use of them has been replaced with segRoutes[id] and boxRoutes[id] respectively (their uses can be found in updateHeadInfo, edges of the Train's state machine and in queries).

Similarly, the copying of data into the segments array in CB has been removed from its initialize function and whenever this array was used, cBs[id] is used instead.

The complete list of local declarations for the new Train template and the new CB template can be seen in appendix C.3.

### 9.3.2 Comparison Between First and Global Model

The only difference between the first model and the global model is where data is stored and accessed so verification of the first model should imply verification of the global model as well.

## 9.4 Checking the Variants

As explained earlier, the restricted model and the global model should not actually need to be verified if the first model is verified while the extended model must be verified by itself because it introduces new states.

All new variants have nevertheless been checked with the configurations presented in chapter 8 to ensure that they have been implemented correctly and as expected, the results are identical to the results of the first model. All results can be seen in *s144449-s144456_FVDRCS.zip > Models > Check Configurations and Results.*

Chapter 10

# UMC Model of First Control System Variant

This chapter describes the implementation of a UMC variant of the first UPPAAL model described in chapter 6. The purpose of creating this is to compare the UPPAAL and UMC model checkers with respect to modelling the first control system variant and verifying the properties specified in chapter 7.

## 10.1 Translating the First UPPAAL Model to a UMC Model

In order to compare the two tools, the UMC implementation of the first control system variant has the same general structure as the UPPAAL model. Since UMC models simply consist of textual representations of UML state machines, the UPPAAL templates' state machines can be translated almost directly to UMC classes. UPPAAL locations are translated to UMC states, and UPPAAL edges become UMC transitions. Despite having similarities, there are also certain differences between the expressive power of the two tools, which means that there will be some significant differences as well.

### 10.1.1 Initialization and Instantiation

In the UPPAAL model, an `Initializer` instance (described in section 6.3) was used to initialize all the other component instances, i.e. all the `Train`s, `CB`s and `Point`s. For this, UPPAAL's broadcast semantics were used as explained in section 6.2.7.

For the UMC model, the same technique that initializes all other objects in one step cannot be used since UMC does not support broadcast semantics. On the other hand, the instantiation of UMC objects of the same class do not have to include the same number of arguments and array arguments do also not need to be of the same size or known beforehand. This means that instead of initializing objects in a transition, variables can be initialized through the instantiation of the objects - which has to be done manually in UMC anyway since automatic instantiation is not supported either.

Due to the absence of an initialization step, the well-formedness check that was included in the initialization step in the first UPPAAL model has also been omitted here. Instead, this check will be included in the model generator tool introduced in chapter 12.

A quite crucial part of the initialization before was also to determine whether a `Point` should start in its `Plus` location or its `Minus` location. Without the additional step, one must then already know which state a `Point` should be in from the beginning. To ensure that the correct state is always chosen, a new `Still` state is introduced. This replaces both the `Plus` and the `Minus` locations and as its name indicates, this state represents a `Point` when it is not in the middle of switching, but is still. To distinguish between a `Point`'s plus position and minus position, a boolean variable `inPlus` is instead used, since this variable can be set during instantiation. A single `Switching` state will then replace the `SwitchingMP` and `SwitchingPM` locations. The transition to it will be triggered by a `switchPoint` signal and a transition back to `Still` will then flip the value of `inPlus`.

### 10.1.2 Communication

Another major difference between UPPAAL and UMC are their communication options. UPPAAL uses channel synchronization, which synchronizes two state machines in a single step while UMC uses asynchronous signals and synchronous operations, which are both multiple-transition processes of sending/calling and receiving. Although the two UMC solutions can be used to create an overall

similar effect to UPPAAL's channels, they will result in more evolutions because only a single transition can happen in each evolution.

In this model, channels have been replaced with signals due to them being functionally similar to the real-world asynchronous signal system. This means that once a signal has been sent, the sender does not have to wait for the receiver to receive it before it can continue itself. This also means that operations are actually more similar to UPPAAL channels since the sender here would not be able to do anything before the message has been received. However, the reason for using channels in UPPAAL was that it efficiently allows for synchronization in a single step. On the other hand, the synchronous UMC operations actually result in more evolutions than signals since the sender is also notified about the receipt of a message in a separate evolution. That is, there are no advantages of operations compared to signals in this case.

### 10.1.3 Translating Transitions

With a chosen method of communication, the UPPAAL edges can now be translated to UMC transitions. The general approach is depicted in figure 10.1.

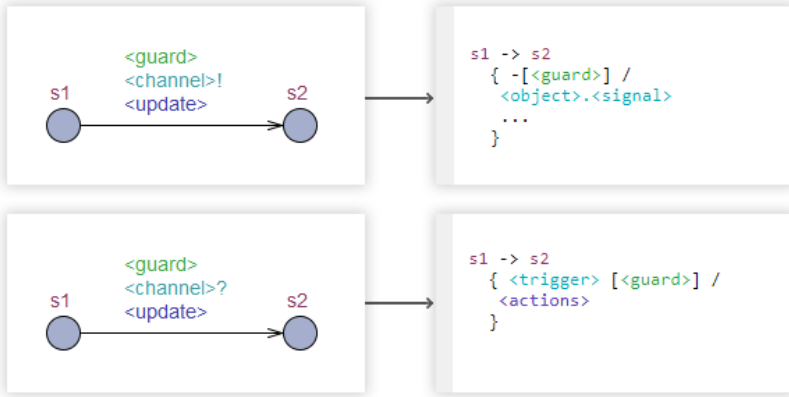



**Figure 10.1:** Translation of UPPAAL edges to UMC transitions

As figure 10.1 shows, a UPPAAL emission on a channel becomes a UMC action while a UPPAAL reception on a channel becomes a UMC trigger.

Another subtle difference that must be taken into consideration is that in UPPAAL, the synchronization happens on a specific channel retrieved through an

ID. Sending a signal in UMC, however, is done through a reference to the receiving object, i.e. instead of sending one's ID, a UMC `Train` sends a reference to itself when making requests. So, instead of having control box route arrays of `CB` IDs, `Trains` have control box route arrays of the actual `CB` objects.

There is also a minor difference in the implementation of UMC actions compared to UPPAAL updates. For improved readability, functions were used to execute the updates on an edge, but since functions are not supported in UMC, the updates are written directly into the set of actions in a transition. This means that there may be duplicate code if multiple edges need to make similar updates, e.g. when a `CB` resets variables after having handled a request.

Besides functions, UMC also does not support the use of global declarations.

### 10.1.4 Updating a Train's Position

In the first model, global declarations were used for both configuration data and well-formedness functions, which have both been left out of this model as explained earlier. Additionally, they were also used for updating a `Train`'s position according to the network's current state.

As explained in section 6.2.8, the global `nextSegment` function in the first model is used to update a `Train`'s position when it passes a `CB`. Since there are no global declarations in UMC, this way of implementing an update of a `Train` position is no longer possible. Instead, a `Train` must rely on its position according to its TCC, i.e. the position is updated based on where it will be if it always follows its route correctly. Whether this position is its actual position can then be formulated as a property and verified along with the other properties.

### 10.1.5 Reservation Limit and Lock Limit

The absence of global declarations also means that it is not possible to set global values for the reservation limit and the lock limit. Since these values are only used by `Train`s, they can instead be implemented as local variables in the `Train` class. To ensure that all `Train`s use the same limits, the limits should be set in the declarations of them in the `Train` class rather than in the arguments during instantiation of `Train` objects.

The resulting model can be seen in appendix D and an example of configuration data can be seen in appendix I. The graphical state machines generated by UMC

from the textual state machines can be seen in figures 10.2, 10.3 and 10.4.

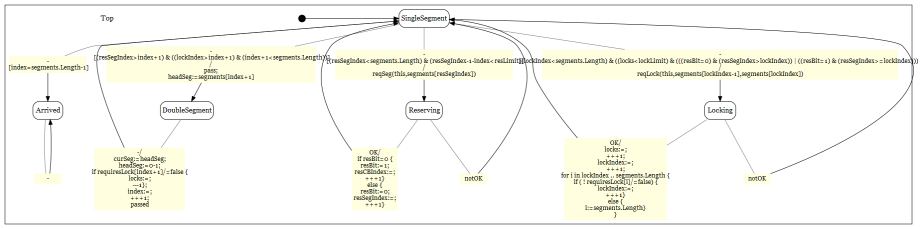



**Figure 10.2:** State machine for `Train`

## 10.2 Formulating Properties for the UMC Model

While the state machines in UMC and UPPAAL are similar in many aspects, properties formulated in UMC are very different from properties formulated in UPPAAL. For example, with UPPAAL it is possible to use the 'forall' and 'exists' functions to iterate over all instances that were given an ID while this is not possible in UMC where objects must be referred to specifically by their names.

This section describes the UMC formulation of the properties from chapter 7 and the abstractions needed for them. To distinguish between the fixed part and the references that depend on the configuration in question, the references are indicated by an underscore (_). These references should be replaced with actual object names or values when formulating the abstractions and properties for specific configuration data.

Section 10.2.1 describes the formulation of the safety properties, section 10.2.2 describes the formulation of the operation properties, section 10.2.3 describes the formulation of the consistency properties and the final liveness property is formulated in section 10.2.4. Like the other models, the UMC model has been checked with the configurations from chapter 8 and the checks show that the properties are also satisfied for this model. The checked models and results can be found in *s144449-s144456_FVDRCS.zip > Models > Check Configurations and Results.*

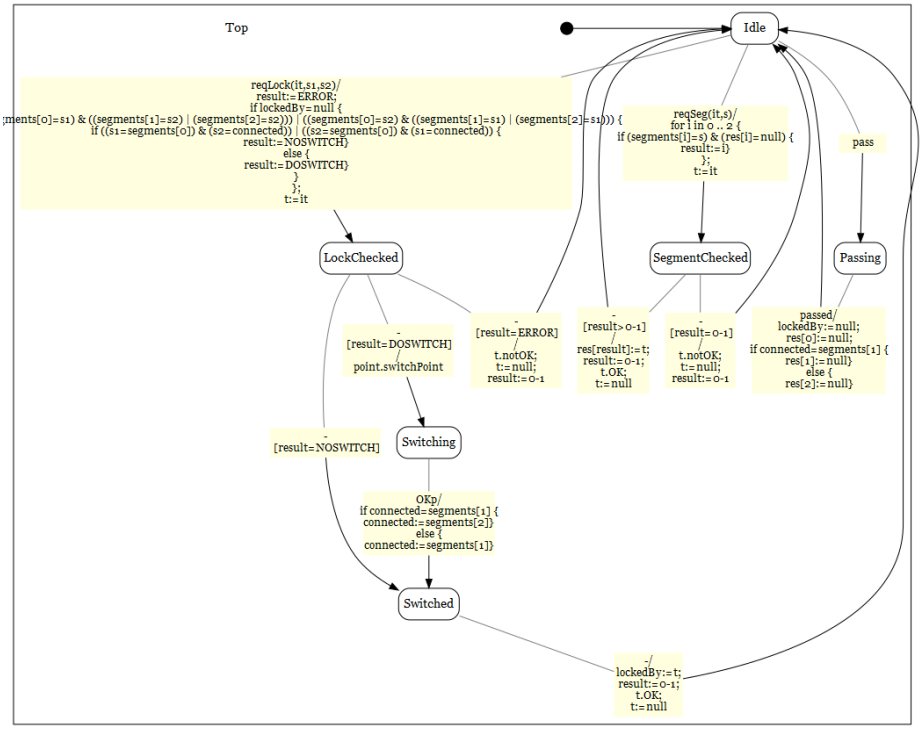



**Figure 10.3:** State machine for `CB`

## 10.2.1 Safety Properties

### 10.2.1.1 No Collision

*At most one train may be on a segment at a time.*

The three cases of collisions seen in figure 7.1 can be formulated in terms of abstractions in a similar way to the corresponding UPPAAL property, but since references to `Train` objects in UMC must be to specific `Train` objects, abstractions for every combination of two distinct `Trains` must be created.

For two different `Trains` `_t1` and `_t2`, the abstractions used for the 'no collision' property are:

```
State: _t1.curSeg = _t2.curSeg -> ccCol(_t1,_t2)
State: inState(_t1.DoubleSegment)
  and _t1.headSeg = _t2.curSeg -> hcCol(_t1,_t2)
State: inState(_t1.DoubleSegment)
```

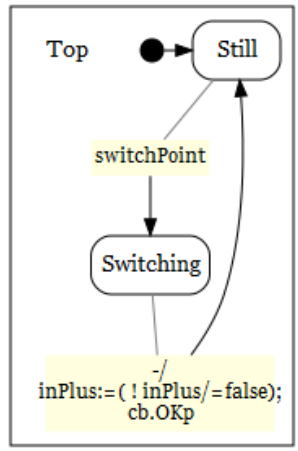



**Figure 10.4:** State machine for `Point`

```
    and inState(_t2.DoubleSegment)
    and _t1.headSeg = _t2.headSeg -> hhCol(_t1,_t2)

  State: _t2.curSeg = _t1.curSeg -> ccCol(_t2,_t1)
  State: inState(_t2.DoubleSegment)
    and _t2.headSeg = _t1.curSeg -> hcCol(_t2,_t1)
  State: inState(_t2.DoubleSegment)
    and inState(_t1.DoubleSegment)
    and _t2.headSeg = _t1.headSeg -> hhCol(_t2,_t1)
```

The first and fourth abstraction and the third and sixth abstraction actually label the exact same configurations. It is, however, convenient to include them all, because it will make the property formulae independent of the order of the `Train` names in the `ccCol` and the `hhCol` labels. Hence, for $N$ trains, there will be $N \cdot (N-1) \cdot 3$ abstractions.

With these abstractions, the UMC property for the 'no collision' property can be formulated as:

```
AG((~ccCol(_t1,_t2) & ~hcCol(_t1,_t2) & ~hhCol(_t1,_t2)) & (~ccCol(_t2,_t1)
      & ~hcCol(_t2,_t1) & ~hhCol(_t2,_t1)) & ...)
```

#### 10.2.1.2 No Derailment

- *If a train is in a critical section, the point in that section is not in the middle of switching.*

  With the use of UMC's action abstractions, this property can be formulated more intuitively than it was in UPPAAL.

  Another reason for using action abstractions for this is that the upcoming `CB` in a route cannot be found with `boxes[index+1]` as it was in UPPAAL. This is because it is not possible to use arithmetic expressions in UMC abstractions. The upcoming `CB` is instead found through an action abstraction for the `passing` signal, which takes the initiating `Train` and receiving `CB` (the upcoming `CB`) as arguments.

  To find the evolutions for a `Train` passing a `CB`, a general abstraction can be used:

  ```
  Action: $t:$cb.pass -> passing($t,$cb)
  ```

  Whether the associated `Point` is switching or not can simply be checked by looking at the state it is in. For a `Point _p`, this abstraction is formulated as the object being in its `Switching` state:

  ```
  State: inState(_p.Switching) -> inSwitching(_p)
  ```

  The formula for the property should then state that whenever a `passing` evolution is found for a switch box `_cb` with associated `Point _p`, then the next configuration should *not* be labelled `inSwitching(_p)`.

  ```
  AG(([passing(_t, _cb)] ~inSwitching(_p)) & ...)
  ```

  This property actually checks that the `Point` is not switching in the *next* configuration whereas the original property states that a passing of a control box does not occur *while* the point is switching. Although the formulations are slightly different, the above property also ensures that a train does not pass a point while the point is switching since only one state change can happen in an evolution. That is, if a `Point` is not in its `Switching` state after a `passing` evolution, it was also not in this state in the previous configuration.

- *If a train is in a critical section, then the segments that it is moving on are connected.*

  For this property, the `passing` abstraction can again be used. When an evolution with a label of this form is found, the `Train`'s `headSeg` and `curSeg` must be connected by the upcoming `CB`.

  For a `CB _cb`, the connected segments can be checked in a configuration using the abstractions:

```
State: _cb.segments[0] = $s1 and _cb.connected = $s2 -> connects(_cb,
    $s1,$s2)
State: _cb.segments[0] = $s1 and _cb.connected = $s2 -> connects(_cb,
    $s2,$s1)
```

Two abstractions are again created to make the property check independently of the segment order in the `connects` label.

With the introduced abstractions, one can now create a formula that expresses that if a `Train` passes a `CB`, the segments that it moves on are connected. A `Train` being on two segments can be found with the abstraction:

```
State: inState(t.DoubleSegment)
  and _t.curSeg = $s1
  and _t.headSeg = $s2 -> doublePos(_t,$s1,$s2)
```

With this, the property can be formulated as:

```
  AG((doublePos(_t,_s1,_s2) -> connects(_cb,_s1,_s2)) & ...)
```

This formula states that whenever a `doublePos(t,s1,s2)` label is found in a configuration, `cb` connects `s1` and `s2`.

Since the property must refer to specific segments, all combinations of two different segments in the network must actually be checked. It is also possible to only check for the segments that are relevant for a `Train` - the adjacent segments in its route - if it can be shown that `doublePos(t,s1,s2)` never exists for two segments that are not adjacent in that order in `Train` `t`'s route. This can be ensured by the auxiliary formula:

```
  ~EF(doublePos(_t,_s1,_s2) | ...)
```

where `{_s1,_s2}` never occurs in `t`'s route.

This can turn out to be a rather long property, but since the `doublePos` label will be used as the antecedent in several properties, it is more convenient to use this auxiliary property. By ensuring that only desirable `doublePos` labels exist, the other properties only need to concern themselves with these.

The formula for the property here can actually also be used to verify the correctness of `curSeg` (see section 10.1.4). By checking the formula for all segments and `CB`s in a `Train`'s route, one can ensure that `Train`s always follow their routes correctly.

### 10.2.2 Operation Requirements

As explained in section 7.4, UPPAAL queries cannot express the occurrence of communication, but since UMC allows for abstractions on actions, this can be

done here. On the other hand, there are other properties that could be expressed in UPPAAL, which cannot be expressed here at all.

#### 10.2.2.1 Requesting a Reservation of a Segment

- *A train never has more reservations than the reservation limit.*

  With the way of implementing saved reservations as all segments between two indices in an array, the number of reservations can only be obtained by calculating the difference between the two indices or by counting the segments between them. As mentioned earlier, UMC abstractions cannot contain arithmetic expressions so it is therefore not possible to formulate this property in UMC.

- *A reservation is only requested if the requested segment is a part of the requesting train's route.*

  This property can be checked by labelling all `reqSeg` evolutions with the initiating `Train` and requested segment. It then needs to ensure that all existing labels of this type are only for `Train`s and the segments in their routes.

  The required action abstraction is:

  ```
  Action: $t:*.reqSeg($t,$s) -> reqSegS($t,$s)
  ```

  For a `Train t`, the label `reqSegS(t,s)` may then *not* exist for all segments `s` that are not in `t`'s route. This can be formulated as:

  ```
  ~EF{reqSegS(_t,_s) | ... }
  ```

- *A reservation is only requested if the control box that a train contacts is a part of the train's route.*

  This property can be formulated similarly to the previous property, but this time, the `CB` that receives the request is checked instead of the requested segment.

  ```
  Action: $t:$cb.reqSeg($t,*) -> reqSegAt($t,$cb)
  ```

  The formula for the property must then ensure that there are never any labels of type `reqSegAt` with a `Train _t` and a `CB _cb` that is not in `_t`'s route.

  ```
  ~EF{reqSegAt(_t,_cb) | ... }
  ```

- *A reservation is only successful if the requested segment is associated with the control box that receives the request and if it is not already reserved.*

  An action abstraction cannot be used here in the same way as seen with the two previous properties. The reason for this is that the indicating action for a successful segment reservation (`OK`) is also used to indicate successful lock requests.

  The two signals could be renamed to have different names, but it is also possible to find the configurations in which a reservation is possible in the same way as done in UPPAAL. These configurations are the configurations in which a `CB` is in its `SegmentChecked` state with `result` $\geq$ `0`:

```
State: inState(_cb.SegmentChecked)
  and _cb.result >= 0
  and _cb.result = $i
  and _cb.segments[$i] = $s -> resOK(_cb,$s)
```

  In line four in the above abstraction, the value of `result` is used with the `segments` array in order to find the segment that was requested. Whenever a configuration is labelled `resOK(cb,s)`, the segment `s` must then be available in the same configuration - i.e. not reserved by any `Train` (if the segment is available in the same configuration, then it was also available in the previous configuration).

  Available segments can be found by labelling all configurations with the segments that are available in them:

```
State: _cb.segments[_i] /= null
  and _cb.segments[_i] = $s
  and _cb.res[_i] = null -> segFree(_cb,$s)
```

  This formula states that whenever `cb.segments[i]` is *not* null and `cb.res[i]` *is* null, then there is an unreserved segment at index `i` at `CB cb`. This abstraction should hence be formulated for `i` $\in [0, 2]$ and for all `CB`s in a network. The 'not null' check for index 0 is not necessary since all `CB`s have at least one associated segment, but it has nevertheless been included in order to have a similar structure for all `segFree` abstractions.

  With these two state abstractions, the property can be formulated as:

```
  AG((resOK(_cb,_s) -> segFree(_cb,_s) & ...)
```

  Besides ensuring that a segment reservation is only successful if the segment is available, the formula actually also ensures that the segment is associated with the `CB`. This is because `segFree` is only created for `CB`s and their associated segments.

#### 10.2.2.2 Requesting a Switch and a Lock

- *A train never has more locks than the lock limit.*

  Configurations with `Train`s that have more locks than the lock limit can be found with the following abstraction for every `Train _t`:

  ```
  State: _t.locks > _t.lockLimit -> lockLimitExceeded(_t)
  ```

  Ensuring that the lock limit is never exceeded is then simply done by ensuring that no configuration has the above kind of label:

  ```
  ~EF(lockLimitExceeded(_t) | ...)
  ```

- *A lock is only requested if the involved switch box is in the route of the requesting train.*

  This property can be formulated in a similar manner to how reservation requests were checked.

  Lock request evolutions can be found with:

  ```
  Action: $t:$cb.reqLock($t,*,*) -> reqLockingAt($t,$cb)
  ```

  With this abstraction, the property must state that the label `reqLockingAt(t,cb)` may never be found for `CB`s `cb` that are not in the route of a `Train t`.

  ```
  ~EF{reqLockingAt(_t,_cb) | ...}
  ```

- *A lock is only requested if the requesting train has the reservation for the stem segment at the switch box and one other segment.*

  For this property, an action abstraction similar to the one from the previous property can be used. The segments sent in the signal are here included in the label to make it clear which segment reservations are checked.

  ```
  Action: $t:$cb.reqLock($t,$s1,$s2) -> reqLocking($t,$cb,$s1,$s2)
  ```

  An evolution is labelled `reqLocking(t,cb,s1,s2)` if a `Train t` is requesting the switching/locking for segments `s1` and `s2` at `CB cb`. Whenever such a label is found, the two segments should be reserved in the next configuration. This will also mean that they were reserved in the current configuration since reservations cannot be cleared in the same step as a switch/lock request is sent.

  Obtained reservations in a configuration can be found with the following abstractions:

  ```
  State: _t.index <= _i
    and _t.resSegIndex > _i
    and _t.segments[_i] = $s
  ```

```
    and _t.boxes[_i+1] = $cb -> reserved(_t,$s,$cb)

  State: _t.index < _i
    and _t.resCBIndex > _i
    and _t.segments[_i] = $s
    and _t.boxes[_i] = $cb -> reserved(_t,$s,$cb)
```

These abstractions label configurations with `reserved(t,s,cb)` if `Train t` has the reservation of segment `s` at `CB cb`. The segment ID is found at `segments[i]`, so the above abstractions should be created for every `Train t` and every index `i` ∈ `[0,segments.length-1]` where `segments` is the segment array of `t`.

With these abstractions, the property can be formulated as:

```
    AG(([reqLocking(_t,_cb,_s1,_s2)]
        (reserved(_t,_s1,_cb) & reserved(_t,_s2,_cb))) & ...)
```

- *A lock is only successful if the point involved in the request was unlocked prior to the request.*

  When a `CB` has received a lock request that is possible to grant, it enters its `switched` state. These situations can be captured by the abstraction:

```
    State: inState(_cb.Switched) -> inSwitched(_cb)
```

  In the configurations with this label, the `CB` in question must be unlocked. This can be checked by labelling configurations with `CB`s and the `Train`s that they have locked for.

```
    State: _cb.lockedBy = $t -> lockedBy(_cb,$t)
```

  With this abstraction, an unlocked `CB cb` is found with `lockedBy(cb,null)`, so the property can be formulated as:

```
    AG((inSwitched(_cb) -> lockedBy(_cb,null)) & ...)
```

- *A switch is only requested if the requested connection is of segments that are adjacent in the train's route.*

  Although it is somewhat cumbersome, it is possible check that switch/lock requests are only made for adjacent segments in a `Train`'s route. This can be done by ensuring that no `Train` sends a `reqLock` signal for segments that are *not* adjacent in its route. That is, for each `Train t` and any `CB cb` and all pairs of segments `s1` and `s2` that are not adjacent in the `Train`'s route, `reqLock(t,cb,s1,s2)` may never be called.

  The action abstraction can be formulated as:

```
    Action: $t:*.reqLock($t,$s1,$s2) -> reqLockingS($t,$s1,$s2)
```

The property can then be formulated as:

```
~EF{reqLockingS(_t,_s1,_s2) | ...}
```

where {_s1,_s2} never occurs in t's route.

- *A switch is only successful if the requested connection is of the stem segment and plus or minus segment of the switch box.*

  Configurations with possible locks were previously found where a CB is in its inSwitched state. This approach cannot be used here because it does not mention the requested segments for the connection. This means that two different connection requests will both be labelled with the same label.

  Instead, the action abstraction with the label of type reqLocking can again be used. This label indicates a request of a switch/lock regardless of whether that request can be granted or not. This means that if a formula with this label holds, it will also hold if the switch/lock is indeed possible.

  Possible segment connections can be indicated by:

```
State: _cb.segments[0] = $s1
    and _cb.segments[1] = $s2 -> canConnect(_cb,$s1,$s2)
State: _cb.segments[0] = $s1
    and _cb.segments[1] = $s2 -> canConnect(_cb,$s2,$s1)

//Additional abstractions if _cb is a switch box:
State: _cb.segments[0] = $s1
    and _cb.segments[2] = $s2 -> canConnect(_cb,$s1,$s2)
State: _cb.segments[0] = $s1
    and _cb.segments[2] = $s2 -> canConnect(_cb,$s2,$s1)
```

  The property can then finally be formulated as:

```
AG(([reqLocking(_t,_cb,_s1,_s2)] canConnect(_cb,_s1,_s2)) & ...)
```

- *A switch box only switches and locks its point if no train is in its critical section.*
  A switch signal can be found with the action abstraction:

```
Action: $cb:*.switchPoint -> switching($cb)
```

  This label indicates that a switch signal has been sent by a CB, and if there are no Trains in the CB's critical section in the following configuration, then there would also be no Trains in the critical section in the previous configuration.

  Trains being in critical sections can be found by:

```
State: _t.curSeg = _s1 and _t.headSeg = _s2 -> inCrit(_cb,_t)
```

  where _s1 and _s2 are segments that can be connected by _cb.

The final property can then be formulated as:

```
AG(([switching(_cb)] (~inCrit(_cb,_t) & ...)) & ...)
```

The innermost parentheses should contain conjunctions of ∼`inCrit(_cb,_t)` with a `_cb` combined with all `Train`s in a network while the outer parentheses are for the remaining `CB`s.

#### 10.2.2.3 Passing a Control Box

- *A train only passes a switch box if it has been locked for the train.*

  A passing `Train` can again be found with the `doublePos` label. If the segments used in this label are connected by a `CB _cb`, determining if this `CB` is a switch box can be done by checking the value of its `point` variable.

  ```
  State: _cb.point /= null -> isSwitchBox(_cb)
  ```

  Whether this `CB` is considered locked by the `Train` can be done in a similar manner to how the `reserved` label was implemented.

  ```
  State: _t.index < _i
      and _t.lockIndex > _i
      and _t.requiresLock[_i] = true
      and _t.boxes[_i] = $cb -> locked(t0,$cb)
  ```

  With these three types of labels, one can express that if a `Train` is in a double position at a switch box `cb`, then the `Train` believes that it has obtained the lock at that switch box.

  ```
  AG(((doublePos(_t,_s1,_s2) & isSwitchBox(_cb)) -> locked(_t,_cb)) &
       ...)
  ```

- *A train never passes the last control box in its route.*

  For this property, the previously introduced label `passing` can again be used. No evolution should ever be labelled `passing(t,cb)` if `cb` is the last `CB` in `t`'s route.

  ```
  ~EF{passing(_t,_cb) | ...}
  ```

- *A train only enters a segment that it has the full reservation of.*

  Previously introduced abstractions can also be used for this property. If a `Train` is in a double position, the segment indicated by `headSeg` is reserved at the two upcoming `CB`s.

  ```
  AG((doublePos(_t,_s1,_s2) -> (reserved(_t,_s2,_cb1) & reserved(_t,
       _s2,_cb2))) & ...)
  ```

### 10.2.3 State Space Consistency

- *The reservations saved in the state space of a* ***Train*** *are also saved in the state spaces of the involved* ***CBs****.*

  The previously formulated `reserved` abstraction was used for the reservations saved in the state spaces of `Train`s. A similar abstraction can be made for the reservations saved in the state spaces of `CB`s:

```
State: _cb.res[_i] /= null
    and _cb.segments[_i] = $s
    and _cb.res[_i] = $t -> reservedBy(_cb,$s,$t)
```

  Checking the consistency between these two can then be done with a simple implication.

```
AG((reserved(_t,_s,_cb) -> reservedBy(_cb,_s,_t)) & ...)
```

- *The locks saved in the state space of a* ***Train*** *are also saved in the state spaces of the involved* ***CBs****.*

  This property can be formulated similarly to the previous property about reservations, but this time using the labels `locked` and `lockedBy`.

```
AG((locked(_t,_cb) -> lockedBy(_cb,_t)) & ...)
```

- *The number of saved locks in the state space of a* ***Train*** *is the same number of locks that it believes that it has.*

  Like the property for the number of reservations, this property cannot be formulated in UMC since the actual number of saved locks can only be obtained by counting them based on the indices `index` and `lockIndex`.

- *A* ***CB****'s information about its associated* ***Point****'s position is consistent with the* ***Point****'s actual position.*

  With the use of the abstractions for `inSwitching` and `connects`, this property can be formulated in a similar way to the UPPAAL query, but instead of checking a `Point`'s state, the value of its `inPlus` variable is checked.

```
State: _p.inPlus = true -> inPlus(_p)
```

For switch boxes `_cb` with associated `Points` `_p`, stem segments `_s`, plus segment `_s1` and minus segment `_s2`, the property can be formulated as:

```
AG(((connects(_cb,_s,_s1) & ~inSwitching(_cb)) ->
        inPlus(_p)) &
    ((connects(_cb,_s,_s2) & ~inSwitching(_cb)) ->
        ~inPlus(_p)) & ...)
```

### 10.2.4 Liveness Properties

The last property that is verified for this model is the liveness property. This property states that it is possible for all `Trains` to arrive at their destinations.

Since this property is rather simple, a single state abstraction that encompasses the arrival of all `Trains` has been used.

```
State: inState(_t.Arrived) and ... -> All_Trains_Arrive
```

The formula for the property is then simply:

```
EF All_Trains_Arrive
```

Chapter 11

# Revised UPPAAL Model: UPPAAL × UMC

After having implemented the first model with UMC, it became clear that the design was made to the advantage of UPPAAL. For example, with UPPAAL it was possible to assume that messages were received instantly. That is, a sent request is received at the moment it is sent. This was not possible to implement with UMC, where a message is sent in one step and received in another.

Because of these differences, another UPPAAL model has been created in order to make a more fair comparison between the two tools. This chapter describes the implementation of a *revised* variant of the first model, which attempts to imitate the UMC implementation. The model was checked with the configurations presented in chapter 8 and the results show that the safety properties are satisfied here as well. The checked models and their results can be found in *s144449-s144456_FVDRCS.zip > Models > Check Configurations and Results.*

## 11.1 Implementation

The general idea behind the revised variant is to reduce the differences between the first model and the UMC model. The main differences that must be taken

into consideration are:

- Message sending in two steps instead of one step.
- Update of a `Train`'s position using its route rather than the network data.
- `Point` uses the same location/state for both its plus position and its minus position.

Additionally, the well-formedness functions have been removed from the revised model since these were omitted from the UMC model.

### 11.1.1 Message Sending and Receiving

To imitate the two-step communication in UMC, additional locations are added such that a single synchronization edge is divided into two edges. From a sender's point of view, the message sending is still only done in one step. From the receiver's point of view, it first synchronizes on one edge and finishes the communication by moving along another edge, which could be used to update its state space if that is needed.

According to the UMC model, the synchronization should actually happen on the first edge while the data should be received (and used) on the second edge, but this is not possible since data is sent through the synchronization. However, as long as the update happens on the second edge, it does not matter whether the data is received on the first or the second edge.

Figures 11.1 and 11.2 show the new state machines for the `Train` template and the `CB` template respectively. The state machines show that whenever they synchronize, they first move to an auxiliary location, and only through another edge do they move to the locations that they were originally supposed to move to.

The design of the 'pass' operation is a bit different compared to the other operations since this operation has two messages sent from a `Train` rather than one message from the `Train` and one from the `CB`. In `CB`, the two synchronizations must come before the two extra edges rather than having one extra edge right after `pass` and another right after `passed`. This is because it is up to the `Train`s to "decide" when they have passed a critical section. If the `pass` and `passed` edges in `CB` were separated by an additional edge, a `CB` would be able to block a `Train` from returning to `SingleSegment` since the `Train` has to wait for the

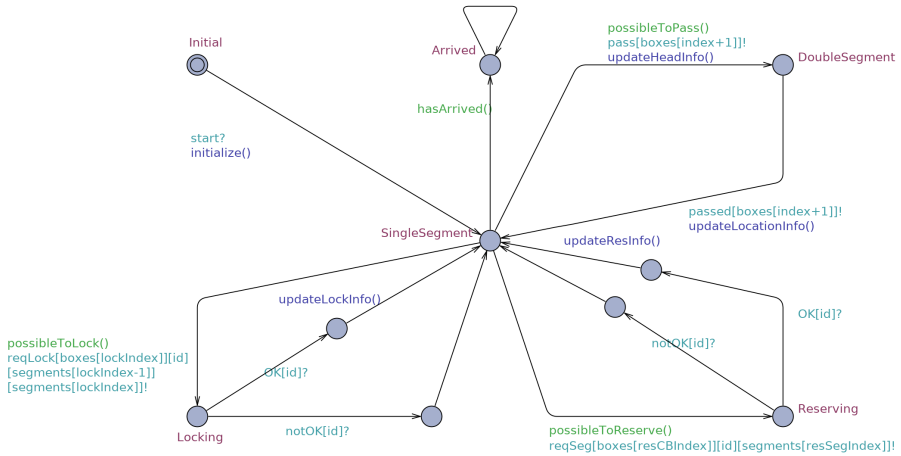



**Figure 11.1:** Train implementation for the revised model

`CB` to move along the empty edge first. By having both synchronizations first, a `Train` can emit on the `passed` channel whenever it wants to while the `CB` can move along the two empty edges whenever it wants to.

### 11.1.2 Update of `Train` Position

Similarly to how a `Train`'s position was updated in the UMC model, `updateHeadInfo` has been changed so `headSeg` is updated according to a `Train`'s route while the global `nextSegment` has been removed.

```
void updateHeadInfo(){
    headSeg = segments[index+1];
}
```

### 11.1.3 Design of `Point`

The `Point` template has been changed such that `Plus` and `Minus` are merged into a single location `Idle` and the locations `SwitchingMP` and `SwitchingPM` have been merged into the single location `Switching`. The template now also has a boolean variable `inPlus`, which is first set to the value of `pointInPlus[id]` and then switched every time a synchronization on `switchPoint[id]` takes place.

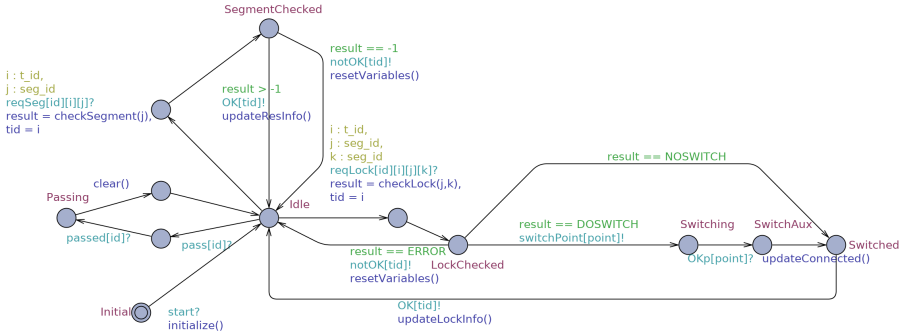



**Figure 11.2:** CB implementation for the revised model

```
int inPlus;
```

For this synchronization, an additional location has also been placed between `Idle` and `Switching` to model the two-step communication as described earlier in this chapter. The `Point` state machine can be seen in figure 11.3.

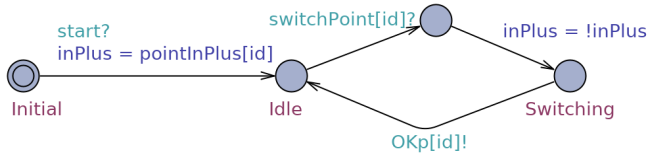



**Figure 11.3:** Point implementation for the revised model

Chapter 12

# Eclipse Plugin for Generating Models

With the various well-formedness requirements for the configuration data that is used when verifying the railway control system for specific networks, it is easy to make mistakes, but it might be less easy to detect them. The UPPAAL models have built-in well-formedness checks, but they only show whether all data *is* or is *not* well-formed, not *where* errors are located if there are any.

To ease the process of correctly formulating configuration data for any of the implemented models, an **Eclipse plugin** is created. This plugin should be able to turn a user's more readable configuration data into either UPPAAL code or UMC code.

This chapter describes how the tool has been created with regard to the used method (section 12.1) and the overall design (section 12.2). Appendix G describes how the final product is installed and used.

## 12.1 Method

To create the tool, a model-based software engineering approach is used, since the model in this case is already very straightforward to design. By using the **Eclipse Modelling Framework (EMF)**, it is possible to turn an abstract syntax into a concrete syntax in a fast and partially automatic way since the framework can generate code automatically from an *Ecore model* [Ecla].

The Ecore model in this case is a domain model that describes a general railway network and trains. This domain model can be designed in a UML class diagram, which EMF can automatically generate *model code*, *editor code* and *edit code* from. These are are all used for the editor that the user can use to translate his configuration data into UPPAAL or UMC models.

The editor created with EMF is a tree-structured editor, which - although being visual - is still not very user-friendly since the representation of a railway network as a tree is not completely intuitive. Therefore, in addition to this framework, Eclipse's **Graphical Modelling Framework (GMF)** is used to create a graphical user interface that is easier to use [Eclb]. The graphical user interface created with GMF will allow the user to draw a network with tools for control boxes, segments and trains (see figure 12.1).

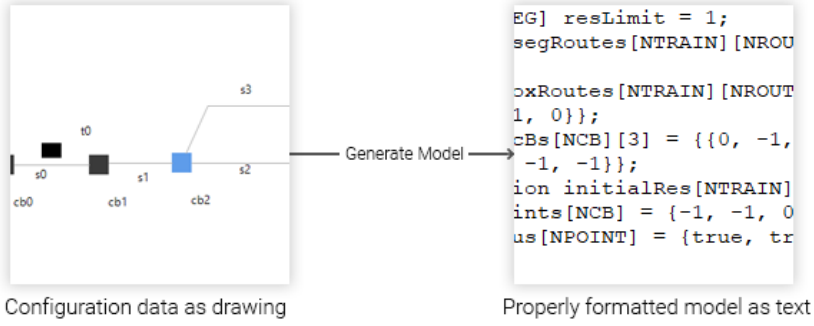



**Figure 12.1:** From graphical configuration data to textual model

The general way in which these frameworks can be used to create a plugin is to first create an Ecore model of the domain (see figure 12.2). From the domain model, a *domain gen model* can be derived. This model is used to generate the model code, the editor code and the edit code. With GMF, the domain model can also be used to derive a *graphical def model* and a *tooling def model*. The former is used for the graphics of the graph while the latter is used for the

graphics of the graph tools.

By combining the domain model, the graphical def model and the tooling def model, a *mapping model* is obtained. This model is the one that is used to map the tools in tooling def model with the graphics defined in graphical def model, which are again mapped to the railway components in the domain model. This mapping can then finally be transformed into a *diagram editor gen model*, which is the generator model that generates the diagram editor for the plugin. By using GMF, one then naturally gets a model-view-controller pattern since the framework generates a view part and a controller part from a model.

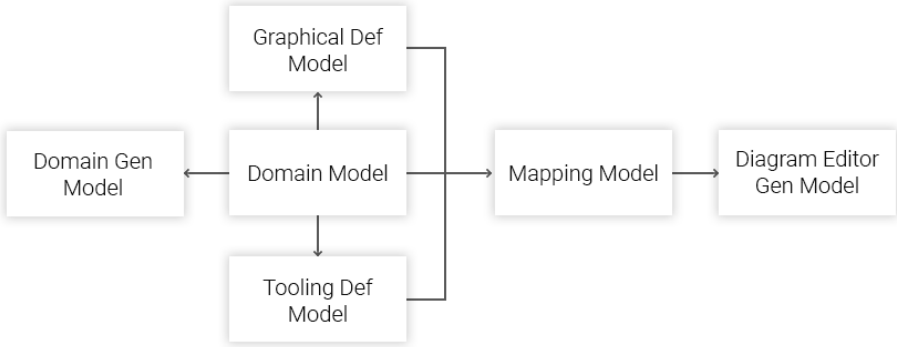



**Figure 12.2:** Models for creating graphical plugin

## 12.2 Design

The drawing tool that can be created with GMF is a graph tool, which means that a user of the tool will be able to draw nodes connected by edges. The railway components are mapped to graph components as:

- Control box → Node
- Segment → Edge
- Train → Node

Additional information (e.g. reservation and lock limits) will then be given in textual form as graph, node or edge attributes. While control box nodes can

be connected by segment edges, trains are simply stand-alone nodes. Based on this mapping, the Ecore model can be designed.

### 12.2.1 Ecore Model

In this tool, `Network` will be used as the name for the root of the Ecore model so this class will include both railway network details and train details. The class diagram for the model can be seen in figure 12.3.

`Network`
The user of the tool has to provide information about his network's control boxes and segments and how they are and can be connected and finally which trains are included in the system. This means that `Network` contains sets of

- `ControlBox`'
- `Segment`s
- `Train`s

Based on the well-formedness requirements from section 6.8, the smallest network requires that the cardinalities of `segments` and `trains` are `1..*` and the cardinality of `controlBoxes` is `2..*`. The `Network` attributes that can be set through text input are the reservation and lock limits and a name of the network, which will be used for the file name of the output file.

`ControlBox` **and** `Segment`
As seen in figure 12.3, the `ControlBox` class has references to the `Segment`s that it is associated with. It would be straightforward to have a list of one to three `Segment` references, but because the created graph needs to be able to distinguish between the source node and target node of an edge, a `ControlBox`' `Segment` references are divided into two types: `outgoing` `Segment`s and `ingoing` `Segment`s.

If a `Segment` is a `ControlBox`' `ingoing` `Segment`, the `ControlBox` is the `Segment`'s `end` `ControlBox`. If instead a `Segment` is a `ControlBox`' `outgoing` `Segment`, then the `ControlBox` is the `Segment`'s `start` `ControlBox`.

Since there is no specific direction linked to a segment in the system, it does not actually matter what a user chooses to be the source and the target. The cardinalities of `ingoing` and `outgoing` are therefore both `[0..3]`, because all

three segments associated with a control box could be of the same kind. This also means that it is actually possible to draw more than three segments from or to a single control box so the limit of three associated segments in total must be ensured during the well-formedness check.

`SwitchBox` **and** `PointSetting`
Because control boxes can also be switch boxes, a `SwitchBox` class is implemented as a subclass of `ControlBox`, but in addition to the position attributes, `SwitchBox` has three segment references each used to refer to the stem, the plus and the minus segments of a switch box. These must hence be chosen from the `outgoing` and `ingoing` lists. The user also has to set the initial position `connected` of a switch box' point. This attribute either takes the value `PLUS` or the value `MINUS`, which are both values from the enumeration class `PointSetting`.

`Train`
Similarly to `ControlBox`, the `Train` class also has a reference to the `Segment` class. This reference `route` is used to store an ordered list of segments that represent a `Train`'s route. Because a `Train` must always be positioned on a `Segment`, the minimum length of `route` is 1, so the cardinality of the reference is `1..*`.

`Component`
The last class that can be seen in the class diagram is the abstract class `Component`, which is a superclass of `Segment`, `ControlBox` and `Train`. This class is used to simply have all `Network` classes have an ID attribute `id`. IDs are not strictly necessary, but they are used to more easily be able to refer to specific nodes or edges in a graph.

Once the plugin code has been generated from the domain model, the translation commands can be implemented. These are the commands that will enable the user to translate his `Network` graph into the UPPAAL and UMC models introduced in chapter 3.

### 12.2.2 Translators

The railway control system models that the tool should be able to translate `Networks` into are

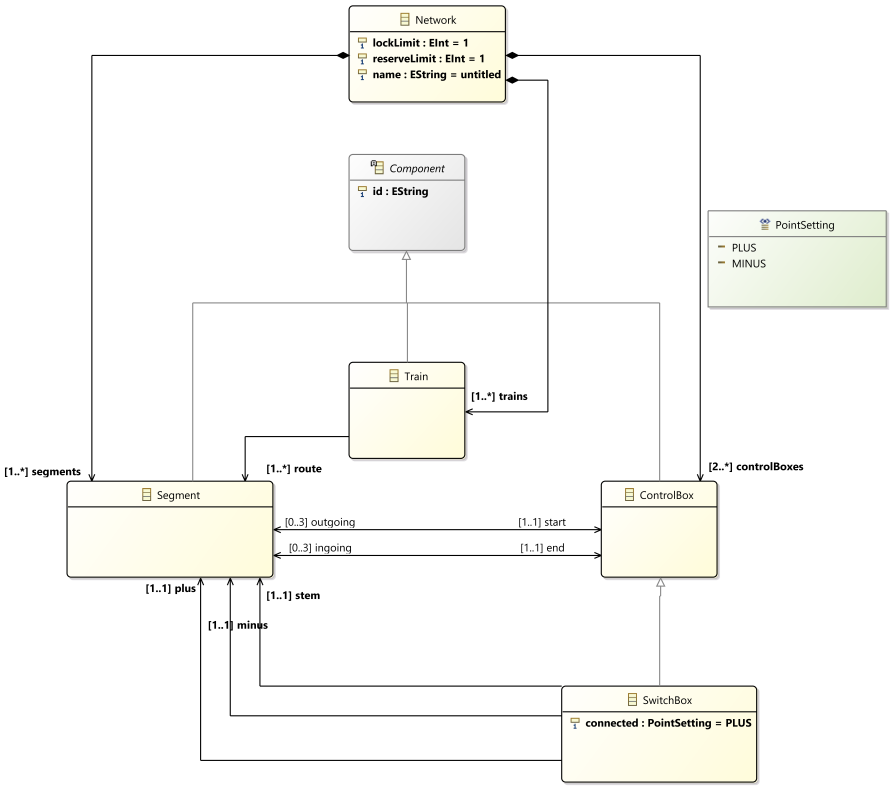



**Figure 12.3:** Class diagram of domain model

- UPPAAL First Model
- UPPAAL Restricted Model
- UPPAAL Extended Model (with cancel operation)
- UPPAAL Global Model
- UMC Model
- UPPAAL Revised Model (UPPAAL x UMC)

Since all of the implemented railway control systems are based on the same algorithm and with similar data structures, a general `Translator` class will be used derive data from a `Network` graph and store them in an appropriate way that can ease the implementation of any of the translation options. Each railway

control system model will then have its own subclass of `Translator`, which will be used to generate the specific code from the derived data.

Before code is generated, the `Network` should also be checked for well-formedness, which the `Translator` class will be responsible for as well.

Because all the additional UPPAAL model variants are built from the first model a `UPPAALTranslator` class will be used to implement the translation of the first model while a subclass of this will be created for each of the other versions. These subclasses then only have to replace the parts with which they differ from the first model. The class diagram for all the translator classes can be seen in figure 12.4.

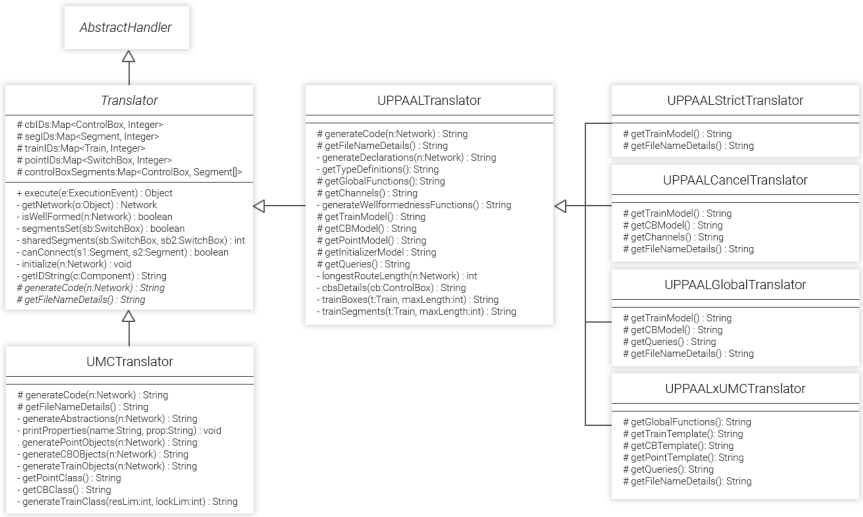



**Figure 12.4:** Class diagram for translator classes

When a translation command has been selected by the user, the `execute` method in the `Translator` class is called. This method checks the well-formedness of the created `Network` with regard to the same requirements presented in section 6.8. If the `Network` is not well-formed, the errors are stated for the user in a dialog window, otherwise the `Network` data is prepared for the subclasses in the `initialize` method.

The initialization step assigns integer IDs to the railway objects from the user's `Network` since the IDs assigned by the user are not necessarily ordered integers. These mappings are stored in maps that can then be used by the subclasses

during translation.

After the initialization step, the selected model can finally be generated in the abstract method `generateCode`, which the subclasses have to implement. The code is generally generated by using extracted code from the UPPAAL and UMC models, and replacing the configuration data parts with the data that was extracted from the `Network`. The full implementation of the translator classes can be found in *s144449-s144456_FVDRCS.zip > Railway Model Generator Tool*.

Chapter 13

# Verification Experiments

While chapter 8 presented some small configurations used for model checking the correctness of the different models, these configurations are not ideal for testing the efficiency of verifying them.

This chapter describes the experimentation setup and results for all the previously described railway control system variants from chapters 6, 9 and 10. Section 13.1 describes the general setup for all of the experiments and the specific configurations used for the model checkers while sections 13.2 to 13.7 show the results. These results are then compared in section 13.8.

## 13.1 Experiment Setup

All experiments are performed on a machine running on an AMD Ryzen 7 2700X CPU clocked at 4GHz and 64 GB DDR4 RAM clocked at 2666MHz. Each experiment configuration is executed three times for each property and the average of the three results will then be calculated. The following sections describes the configuration of the specific model checkers and the railway system configurations that will be used in the experiments.

### 13.1.1 Limit Configuration

There are two specific configuration parameters that control which operations a `Train` can execute: A reservation limit and a lock limit. The configurations of these that have been used in the experiments are

- Both the reservation limit and the lock limit are set to 1
- Both the reservation limit and the lock limit are set to 2

The first configuration enforces a more specific operation sequence since a `Train` is forced to use its obtained reservations and locks more often, while the second configuration allows for more flexibility. It is therefore expected that the first configuration will generally have a faster evaluation time compared to the second configuration. For both configurations, the safety properties formulated in sections 7.3 and 10.2.1 should hold.

### 13.1.2 UPPAAL Setup and Measurements

There are two different types of measurements that can be extracted from the UPPAAL Model Checker: Time and memory usage. The time measurements that are extracted from UPPAAL's model checker are as follows:

- Verification time: The time it takes UPPAAL to process the particular query.
- Kernel time: The time in which UPPAAL interacts with the underlying system for resources.
- Elapsed time: The total time which includes verification time, kernel time and the time to parse the model into an internal representation.

The measurements related to memory usage are the memory usage peak and the virtual memory usage. However, for these experiments, only the elapsed time and the memory peak usage are considered since UPPAAL may round the verification time and kernel time such that the registered value is greater than the elapsed time. This is clearly not possible in the real world and these values are therefore discarded. The virtual memory usage only indicates how much memory the UPPAAL server has allocated to fit the entire state space, which

is useful for debugging UPPAAL, but it is less relevant for the purpose of this project's experiments.

UPPAAL has been configured with the following presets, which are set through the *Options* menu:

- Search order: **Breadth First Search**
  This is done to ensure that every possible symbolic state will be visited during the evaluation of a specific query. For some properties like the liveness property, UPPAAL may find a valid solution faster by using the alternative *Depth First Search* option, but for the sake of comparison, *Breadth First Search* is always used.

- State space reduction: **Conservative**
  This is the default reduction used by UPPAAL. This option results in a balance between the used verification time and the used amount of memory. The *None* option gave no real improvement to the verification time. But the memory usage was higher, since all states are always stored. The other two options *Aggressive* and *Extreme* both gave a significant increases in verification time.

- State space representation: **DBM**
  This is the default setting for UPPAAL and the other options have not been tested. The impact of the actual representation is therefore unknown.

- Reuse: **Disabled**
  *Reuse* has been disabled in order to force UPPAAL to always re-create the entire state space. This ensures that memory measurements are more accurate and that UPPAAL does not return a cached result.

- All other options are left with their default values.

### 13.1.3 UMC Setup and Measurements

The UMC model checker is configured to use the default parameters for execution. Regarding measurements in UMC, it is only possible to get the total elapsed time for a property verification. That is, the actual memory consumption is not registered. Additional processes could be used to measure this, but that would worsen the results, which would make UMC seem slower than it actually is.

### 13.1.4 Properties

The properties that are checked in the experiments are the safety properties and the liveness property:

- **No collision**: Two trains will never occupy the same segment at the same time.
- **No derailment 1**: A train always moves on connected segments when passing a point.
- **No derailment 2**: A point is never switching while a train is passing it.
- **Will arrive**: All trains will eventually arrive at their destinations.

### 13.1.5 Railway System Configurations

The different configurations that are used for the experiments will be described in this section. All experiment configurations can be found in a runnable format in *s144449-s144456_FVDRCS.zip > Models > Experiment Configurations*.

**Configuration 1: Example with One Train**
The first configuration is called *Example with One Train* and it is shown in figure 13.1. As its name suggests, there is exactly one train with the route `{s0, s1, s3}`. This configuration originates from [Gei18, p. 103].

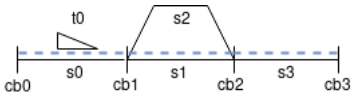



**Figure 13.1:** Example with One Train

**Configuration 2: Example with Two Trains**
The second configuration *Example with Two Trains* is an extension of the previous one where an additional train `t1` has been added with the route `{s3, s2, s0}`. The purpose of this configuration is to see how well the models handle having more than one train. The configuration can be seen in figure 13.2 and it originates from [Gei18, p. 120].

**Configuration 3: Shared Segment**
The third configuration is called *Shared Segment*. It contains exactly two trains

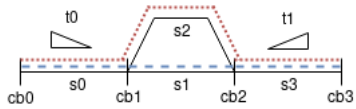



**Figure 13.2:** Example with Two Trains

`t0` and `t1`. Train `t0` has the route `{s1, s2, s3}` while train `t1` has the route `{s3, s2, s0}`. The purpose of this experiment is to see how well the models handle multiple trains with overlapping routes. The configuration can be seen in figure 13.3 and it originates from [Gei18, p. 120].

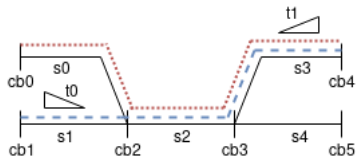



**Figure 13.3:** Shared Segment

**Configuration 4: Large Network**
The fourth configuration is called *Large Network* and it is an extension of *Shared Segment*, where additional control boxes and points have been added to each end of the original configuration. The trains and their routes are the same as in *Shared Segment*. The configuration can be seen in figure 13.4 and it originates from [Gei18, p. 120].

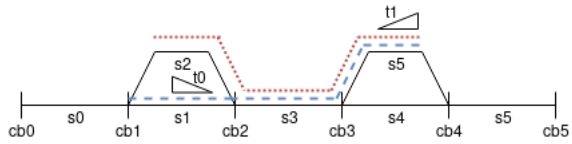



**Figure 13.4:** Large Network

**Configurations 5-11: Network with $n$ stations**
The fifth to eleventh configurations are called:

- *Station One*
- *Station Two*
- *Station Four*

- *Station Six*
- *Station Eight*
- *Station Ten*
- *Station Twelve*

and the general format of them can be seen in figure 13.5. As with most of the previous configurations, there are two trains `t0` and `t1`, and their routes depend on the specific number of stations. Train `t0`'s route is shown as the dashed line in figure 13.5 while `t1`'s route is shown as the dotted line. The figure shows that `t0` generally moves on points' plus segments while `t1` moves on their minus segments. The general idea behind these configurations originate from [Nie16, p. 98-101]. The purpose of these configurations is to see how well the models handle an increasing network size and increasing route sizes.

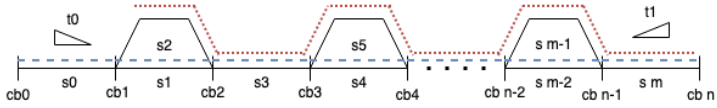



**Figure 13.5:** Network with $n$ stations.

**Configuration 12: Two Trains Same Direction**
Similarly to most of the configurations presented so far, this configuration has two trains, but instead of focusing on the network size, it focuses on the number of trains. Its form as shown in figure 13.6 is similar to *Shared Segment* with the only difference that the trains are moving in the same direction. Train `t0` has the route `{s0, s2, s4}` while train `t1`'s route is `{s1, s2, s3}`. The general idea behind this configuration originates from [Nie16, p. 102-103].

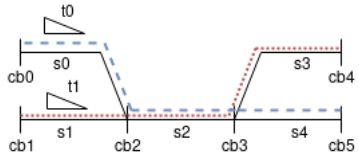



**Figure 13.6:** Two Trains Same Direction

**Configuration 13: Four Trains Same Direction**
In the thirteenth configuration, the number of trains has been increased to four and to accommodate this change, the number of segments and control boxes have been increased as well as seen in figure 13.7. This configuration too originates from [Nie16, p. 102-103].

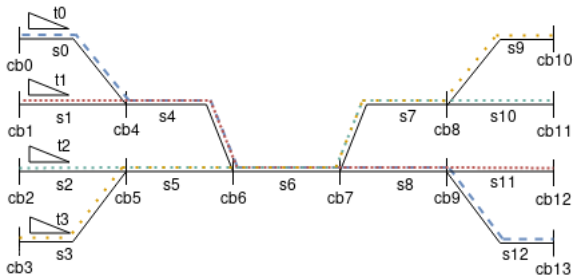



**Figure 13.7:** Four Trains Same Direction

The specific routes of the four trains are:

- `t0`: `{s0, s4, s6, s8, s12}`
- `t1`: `{s1, s4, s6, s8, s11}`
- `t2`: `{s2, s5, s6, s7, s10}`
- `t3`: `{s3, s5, s6, s7, s9}`

**Configurations 14 and 15: Nærumbanen**
The fourteenth and fifteenth configurations are based on the real-life railway network *Nærumbanen*[1]. There are two different variants of this network.

- **Configuration 14** models the regular traffic, where two trains start in each end of the network. Their destinations are in the opposite ends as seen in figure 13.8.
- **Configuration 15** models the rush hour traffic where a third train `t2` is also present. `t0` starts at segment `s0` with destination at `s18`, `t1` starts at `s18` with destination at `s10` and `t2` starts at `s10` with destination at `s0`.

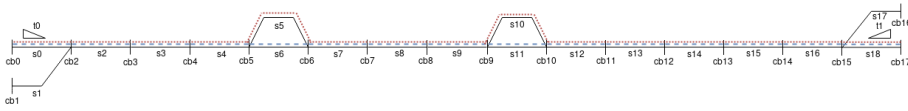


**Figure 13.8:** Nærumbanen

[1]The network here is a slight simplification where the segments that are used for the engine shed and the segment that connects the network to *S-Togsnettet* have been omitted.

## 13.2 First Model Results

When checking the different safety properties for the first model and different limit configurations, the experiments show that they were met for all of the railway system configurations described in section 13.1.5.

The complete results for all the safety properties for the first model can be seen in tables E.1 - E.16 in appendix E.

Since the resulting values are approximately identical for all tested properties, only the results for the 'No Collision' property are shown here in table 13.1 for elapsed time and in table 13.2 for memory usage - both with reservation limit and lock limit 1.

When looking at the results for *Example with One Train*, *Example with Two Trains*, *Shared Segment* and *Large Network* in terms of elapsed time, the values are approximately equivalent to each other although the actual composition of the networks vary from each other. For all configurations, the memory usage generally increases when the network size or number of trains increases.

In the *Station n* experiments in particular, the time and memory usage increase exponentially as the number of stations is increased. The actual growth rate is determined in section 13.8.

The time and memory usage also increase exponentially for the remaining four networks (*Two Trains Same Direction*, *Four Trains Same Direction* and *Nærumbanen* with two and three trains) - and at a greater rate than the *Station n* experiments. The reason for this is that trains are the components that initiate communication with control boxes. Since there are more trains, there are going to more interleavings of operations, which results in a larger state space and thereby a higher time and memory usage. Besides having more trains, the *Four Trains Same Direction* configuration also has more points and segments than *Two Trains Same Direction*, while the network of *Nærumbanen (3T)* is the same as *Nærumbanen (2T)*. The growth rate is therefore higher in the first case compared to the second case.

The results for the 'No Collision' property when the reservation and lock limits are both set to 2 has a similar pattern in all of the experiment scenarios. The results are shown in table 13.3 for elapsed time and in table 13.4 for memory usage. As expected, the growth rates are higher in both time and memory usage since a train has more options. This results in a larger state space and it is the primary reason why there are no results for both *Station Twelve* and *Four Trains Same Direction* - the growth rate in the state space was quite significant so there

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.002 | 0.003 | 0.004 | 0.003 |
| Example with Two Trains | 0.022 | 0.024 | 0.015 | 0.020 |
| Shared Segment | 0.014 | 0.014 | 0.014 | 0.014 |
| Large Network | 0.011 | 0.015 | 0.021 | 0.016 |
| Station One | 0.007 | 0.01 | 0.008 | 0.008 |
| Station Two | 0.082 | 0.093 | 0.081 | 0.085 |
| Station Four | 1.702 | 1.911 | 1.738 | 1.784 |
| Station Six | 29.678 | 29.678 | 29.803 | 29.720 |
| Station Eight | 164.477 | 165.993 | 165.677 | 165.382 |
| Station Ten | 599.563 | 599.868 | 597.934 | 599.122 |
| Station Twelve | 1741.865 | 1741.334 | 1747.987 | 1743.729 |
| Two Trains Same Direction | 0.016 | 0.019 | 0.019 | 0.018 |
| Four Trains Same Direction | 1142.559 | 1141.663 | 1139.661 | 1141.294 |
| Nærumbanen (2T) | 24.081 | 24.345 | 24.339 | 24.255 |
| Nærumbanen (3T) | 671.008 | 653.381 | 663.635 | 662.675 |

**Table 13.1:** First Model: 'No Collision', limits 1, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with one train | 6000 | 6016 | 6020 | 6012 |
| Example with two trains | 6528 | 6552 | 6572 | 6551 |
| Shared Segment | 7468 | 7520 | 7532 | 7507 |
| Large Network | 8932 | 8972 | 8988 | 8964 |
| Station One | 6484 | 6524 | 6536 | 6515 |
| Station Two | 9268 | 9308 | 9324 | 9300 |
| Station Four | 25792 | 25880 | 25896 | 25856 |
| Station Six | 69916 | 70200 | 70220 | 70112 |
| Station Eight | 158424 | 158440 | 158628 | 158497 |
| Station Ten | 312972 | 312984 | 313116 | 313024 |
| Station Twelfth | 723464 | 1051300 | 887412 | 887392 |
| Two trains same direction | 7496 | 7548 | 7560 | 7535 |
| Four trains same direction | 726716 | 726880 | 726904 | 726833 |
| Nærumbanen (2T) | 78436 | 78828 | 78888 | 78717 |
| Nærumbanen (3T) | 382548 | 383224 | 383544 | 383105 |

**Table 13.2:** First model: 'No Collision', limits 1, Memory Usage Peak in KB

were no assurance that UPPAAL would ever terminate within a reasonable time frame.

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.004 | 0.004 | 0.004 | 0.004 |
| Example with Two Trains | 0.018 | 0.018 | 0.024 | 0.020 |
| Shared Segment | 0.015 | 0.02 | 0.023 | 0.019 |
| Large Network | 0.023 | 0.032 | 0.027 | 0.027 |
| Station One | 0.011 | 0.017 | 0.012 | 0.013 |
| Station Two | 0.183 | 0.215 | 0.187 | 0.195 |
| Station Four | 8.090 | 8.115 | 8.241 | 8.149 |
| Station Six | 176.156 | 175.635 | 175.459 | 175.750 |
| Station Eight | 1144.312 | 1146.043 | 1144.102 | 1144.819 |
| Station Ten | 4327.887 | 4350.655 | 4352.747 | 4343.763 |
| Station Twelve | | | | |
| Two Trains Same Direction | 0.024 | 0.029 | 0.022 | 0.025 |
| Four Trains Same Direction | | | | |
| Nærumbanen (2T) | 111.647 | 111.867 | 111.601 | 111.705 |
| Nærumbanen (3T) | 4707.002 | 4711.929 | 4683.026 | 4700.652 |

**Table 13.3:** First model: 'No Collision', limits 2, Elapsed time in seconds

## 13.3 Restricted Model Results

For this model, it is expected that the experiments will show that the verification time and memory usage are lower than those of the first model since there is always a deterministic choice of operation as described in section 9.1. The safety properties must of course also still hold.

When looking at the time usage for the restricted variant with both the reservation limit and the lock limit set to 1 (see table 13.5), it is indeed the case for the larger configurations that the verification time is lower than the equivalent results for the first model (see table 13.1 in section 13.2). The same generally does not apply when looking at the corresponding memory usage in table 13.6. The memory usage is higher in most cases, due to the additional locations for `Train`. This shows that by reducing the number of choices that a `Train` has, a lower growth rate is achieved. The benefit of this is that it becomes easier to verify larger networks, which would otherwise be harder to verify due to the larger state spaces.

When increasing the reservation limit and the lock limit to 2, verification with

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with one train | 5996 | 5992 | 5996 | 5995 |
| Example with two trains | 6548 | 6552 | 6576 | 6559 |
| Shared Segment | 7480 | 7532 | 7554 | 7522 |
| Large Network | 8960 | 9000 | 9016 | 8992 |
| Station One | 6492 | 6520 | 6532 | 6515 |
| Station Two | 9912 | 9952 | 9968 | 9944 |
| Station Four | 41416 | 41548 | 41560 | 41508 |
| Station Six | 152804 | 152904 | 152916 | 152875 |
| Station Eight | 422196 | 422388 | 422396 | 422327 |
| Station Ten | 955772 | 954604 | 954952 | 955109 |
| Station Twelfth | | | | |
| Two trains same direction | 7536 | 7588 | 7600 | 7575 |
| Four trains same direction | | | | |
| Nærumbanen (2T) | 122040 | 122144 | 122404 | 122196 |
| Nærumbanen (3T) | 1833096 | 1833144 | 1833148 | 1833129 |

**Table 13.4:** First model: 'No Collision', limits 2, Memory Usage Peak in KB

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with one train | 0.003 | 0.004 | 0.003 | 0.003 |
| Example with Two Trains | 0.026 | 0.017 | 0.030 | 0.024 |
| Shared Segment | 0.014 | 0.016 | 0.017 | 0.016 |
| Large Network | 0.012 | 0.019 | 0.019 | 0.017 |
| Station One | 0.012 | 0.012 | 0.019 | 0.014 |
| Station Two | 0.065 | 0.111 | 0.114 | 0.097 |
| Station Four | 1.286 | 1.315 | 1.266 | 1.289 |
| Station Six | 23.164 | 23.206 | 23.325 | 23.232 |
| Station Eight | 127.860 | 124.623 | 124.672 | 125.718 |
| Station Ten | 474.403 | 459.789 | 444.659 | 459.617 |
| Station Twelve | 1337.102 | 1326.536 | 1320.822 | 1328.153 |
| Two Trains Same Direction | 0.024 | 0.020 | 0.025 | 0.023 |
| Four Trains Same Direction | 828.480 | 839.239 | 839.249 | 835.656 |
| Nærumbanen (2T) | 19.309 | 18.324 | 18.342 | 18.658 |
| Nærumbanen (3T) | 323.919 | 321.851 | 320.536 | 322.102 |

**Table 13.5:** Restricted model: 'No Collision', limits 1, Elapsed time in seconds

the restricted model was also faster in terms of time (see table 13.7) and memory usage (see table 13.8) compared to the equivalent setup for the first model. As seen in the tables, the improvement made it possible to verify both *Station Twelve* and *Four Trains Same Direction* in a relatively efficient manner, and many of the results are approximately the same as the results for limits 1. The reason for this is that the operations that a train will perform remain the same with only the order being changed. For example, when a train wants to move across two segments

- if the limits are set to 1, the train will make one full segment reservation, obtain the lock at the upcoming control box if it is a switch box and then pass the control box in order to enter the reserved segment. This is repeated for the next segment.
- if the limits are set to 2, the train will make two full segment reservations, obtain the lock(s) at the upcoming control box(es) if it/they are switch boxes and then pass the two control boxes in order to enter the last reserved segment.

The example shows that the number of steps remain the same: Two full reservations, at most two locks and two passings. This means that the state spaces are equivalent to each other in terms of size.

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with one train | 7592 | 7608 | 7612 | 7604 |
| Example with two trains | 8544 | 8552 | 8560 | 8552 |
| Shared Segment | 9772 | 9808 | 9820 | 9800 |
| Large Network | 11980 | 12044 | 12506 | 12177 |
| Station One | 8516 | 8524 | 8532 | 8524 |
| Station Two | 12384 | 12436 | 12444 | 12421 |
| Station Four | 37948 | 38172 | 38184 | 38101 |
| Station Six | 101400 | 102152 | 102160 | 101904 |
| Station Eight | 223784 | 224888 | 224888 | 224520 |
| Station Ten | 421316 | 426156 | 426156 | 424543 |
| Station Twelfth | 725644 | 727908 | 732620 | 728724 |
| Two trains same direction | 9916 | 9948 | 9960 | 9941 |
| Four trains same direction | 688684 | 688700 | 688708 | 688697 |
| Nærumbanen (2T) | 118312 | 119176 | 119984 | 119157 |
| Nærumbanen (3T) | 287632 | 288220 | 288584 | 288145 |

**Table 13.6:** Restricted model: 'No Collision', limits 1, Memory Usage Peak in KB

| Configuration | Second | First | Third | Average |
|---|---|---|---|---|
| Example with one train | 0.004 | 0.004 | 0.004 | 0.004 |
| Example with Two Trains | 0.034 | 0.036 | 0.033 | 0.034 |
| Shared Segment | 0.015 | 0.014 | 0.015 | 0.015 |
| Large Network | 0.029 | 0.011 | 0.021 | 0.020 |
| Station One | 0.019 | 0.022 | 0.022 | 0.021 |
| Station Two | 0.142 | 0.102 | 0.136 | 0.127 |
| Station Four | 2.399 | 2.094 | 2.331 | 2.275 |
| Station Six | 47.752 | 47.224 | 47.632 | 47.536 |
| Station Eight | 271.339 | 290.211 | 269.255 | 276.935 |
| Station Ten | 1072.370 | 1069.352 | 1051.182 | 1064.301 |
| Station Twelve | 3154.039 | 3032.947 | 3265.269 | 3150.752 |
| Two Trains Same Direction | 0.011 | 0.015 | 0.014 | 0.013 |
| Four Trains Same Direction | 1513.903 | 1519.292 | 1466.833 | 1500.009 |
| Nærumbanen (2T) | 20.008 | 19.764 | 19.959 | 19.910 |
| Nærumbanen (3T) | 395.981 | 393.703 | 393.868 | 394.517 |

**Table 13.7:** Restricted model: 'No Collision', limits 2, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with one train | 7560 | 7576 | 7576 | 7571 |
| Example with two trains | 8548 | 8560 | 8568 | 8559 |
| Shared Segment | 9744 | 9776 | 9784 | 9768 |
| Large Network | 11944 | 11996 | 12016 | 11985 |
| Station One | 8500 | 8508 | 8516 | 8508 |
| Station Two | 12540 | 12588 | 12608 | 12579 |
| Station Four | 39228 | 39448 | 39476 | 39384 |
| Station Six | 122276 | 123040 | 123044 | 122787 |
| Station Eight | 276256 | 277364 | 277368 | 276996 |
| Station Ten | 560320 | 560176 | 561456 | 560651 |
| Station Twelfth | 1266208 | 1796796 | 1019336 | 1360780 |
| Two trains same direction | 9744 | 9780 | 9792 | 9772 |
| Four trains same direction | 903736 | 903792 | 904192 | 903907 |
| Nærumbanen (2T) | 112928 | 113756 | 113976 | 113553 |
| Nærumbanen (3T) | 313656 | 313988 | 314000 | 313881 |

**Table 13.8:** Restricted model: 'No Collision', limits 2, Memory Usage Peak in KB

The complete set of results for all the safety properties for the restricted model can be seen in tables E.17 - E.32 in appendix E.

## 13.4 Global Model Results

The purpose creating the global variant of the first model was to see if there was a difference in verification efficiency between using local variables for each template instead of solely using and accessing global constants. Hence for this variant, only some of the presented configurations have been used for the experiments since it is only the scalability of the number of segments, points and trains that is of interest.

The results for elapsed time (see table 13.9) and memory usage (see table 13.10) are generally higher for the global variant than the corresponding results for the first model. The higher time and memory usage may indicate that UPPAAL makes some optimizations related to local variables, which makes the generated state space smaller than if global constants are used. In theory, these constants could have been excluded from the state space generation, which in turn would potentially allow for a more compact state space. However, as the experiments show, this is not the case.

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with one train | 0.004 | 0.004 | 0.004 | 0.004 |
| Example with Two Trains | 0.036 | 0.034 | 0.042 | 0.037 |
| Shared Segment | 0.020 | 0.019 | 0.020 | 0.020 |
| Large Network | 0.019 | 0.027 | 0.025 | 0.024 |
| Station One | 0.025 | 0.015 | 0.025 | 0.022 |
| Station Two | 0.125 | 0.129 | 0.128 | 0.127 |
| Station Four | 2.685 | 2.651 | 2.694 | 2.677 |
| Station Six | 44.791 | 45.379 | 44.759 | 44.976 |
| Station Eight | 286.995 | 287.993 | 285.826 | 286.938 |
| Station Ten | 1022.700 | 1029.362 | 1027.356 | 1026.473 |
| Nærumbanen (2T) | 27.643 | 28.318 | 28.409 | 28.123 |
| Nærumbanen (3T) | 589.217 | 596.547 | 592.764 | 592.843 |

**Table 13.9:** Global model: 'No Collision', limits 1, Elapsed time in seconds

When the reservation and lock limits are increased from 1 to 2, a similar result is seen - both in terms of verification time (see table 13.11) and in terms of memory usage (see table 13.12). This supports the idea that UPPAAL has

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with one train | 7632 | 7652 | 7656 | 7647 |
| Example with two trains | 8392 | 8400 | 8408 | 8400 |
| Shared Segment | 9900 | 9936 | 10192 | 10009 |
| Large Network | 11920 | 11972 | 11980 | 11957 |
| Station One | 8344 | 8352 | 8360 | 8352 |
| Station Two | 12504 | 12560 | 12572 | 12545 |
| Station Four | 37696 | 37892 | 37948 | 37845 |
| Station Six | 103528 | 104260 | 104352 | 104047 |
| Station Eight | 231500 | 233320 | 233340 | 232720 |
| Station Ten | 451224 | 454576 | 454680 | 453493 |
| Nærumbanen (2T) | 111960 | 112988 | 112996 | 112648 |
| Nærumbanen (3T) | 309820 | 310408 | 310736 | 310321 |

**Table 13.10:** Global model: 'No Collision', limits 1, Memory Usage Peak in KB

some optimization mechanism related to local variables, which makes it more efficient to use local declarations instead of global.

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with one train | 0.006 | 0.006 | 0.007 | 0.006 |
| Example with Two Trains | 0.034 | 0.035 | 0.063 | 0.044 |
| Shared Segment | 0.018 | 0.037 | 0.033 | 0.029 |
| Large Network | 0.024 | 0.029 | 0.040 | 0.031 |
| Station One | 0.018 | 0.017 | 0.019 | 0.018 |
| Station Two | 0.306 | 0.398 | 0.349 | 0.351 |
| Station Four | 12.486 | 10.081 | 12.083 | 11.550 |
| Station Six | 252.827 | 246.848 | 249.513 | 249.729 |
| Station Eight | 1625.016 | 1620.067 | 1548.218 | 1597.767 |
| Station Ten | 6219.558 | 5960.382 | 6024.153 | 6068.031 |
| Nærumbanen (2T) | 133.816 | 133.661 | 133.648 | 133.708 |
| Nærumbanen (3T) | 4380.034 | 4368.982 | 4373.205 | 4374.074 |

**Table 13.11:** Global model: 'No Collision', limits 2, Elapsed time in seconds

The complete set of results for all the safety properties for the global model can be seen in tables E.33 - E.48 in appendix E.

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 7648 | 7660 | 7660 | 7656 |
| Example with Two Trains | 8420 | 8428 | 8436 | 8428 |
| Shared Segment | 9924 | 9956 | 10216 | 10032 |
| Large Network | 11948 | 12000 | 12008 | 11985 |
| Station One | 8404 | 8412 | 8420 | 8412 |
| Station Two | 14920 | 14976 | 14984 | 14960 |
| Station Four | 58036 | 58280 | 58288 | 58201 |
| Station Six | 209552 | 210308 | 210312 | 210057 |
| Station Eight | 543948 | 544996 | 545256 | 544733 |
| Station Ten | 1176496 | 1177572 | 1178464 | 1177511 |
| Nærumbanen (2T) | 151368 | 152404 | 152424 | 152065 |
| Nærumbanen (3T) | 1303336 | 1303388 | 1303408 | 1303377 |

**Table 13.12:** Global model: 'No Collision', limits 2, Memory Usage Peak in KB

## 13.5 Extended Model Results

The purpose of the extended model was mainly to see how the first model could be modified if new operations were to be introduced. It is hence not expected that verification of this model will be very efficient, but it is still interesting to see the impact on the efficiency when a new operation is introduced.

The results for the 'No Collision' property for the extended model can be seen in table 13.13 for the time usage and in table 13.14 for the memory usage. The growth rate in terms of both time and memory usage is unsurprisingly significantly higher than for the first model. The additional operation gives a train an additional edge that it can non-deterministically choose. This in turn results in a larger number of successor states for each state, which means that the complete state space becomes larger as well. Since the verification time is approximately four times higher for *Station Eight* for the extended model compared to the first model, *Station Ten* and *Station Twelve* were not included in this series of experiments.

Again, when the reservation and lock limits are increased to 2, both the time usage and memory usage are increased (table 13.15) and memory usage (table 13.16).

The complete set of results for all the safety properties for the extended model can be seen in tables E.49 - E.64 in appendix E.

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.004 | 0.004 | 0.004 | 0.004 |
| Example with Two Trains | 0.004 | 0.007 | 0.008 | 0.006 |
| Shared Segment | 0.026 | 0.019 | 0.031 | 0.025 |
| Large Network | 0.023 | 0.045 | 0.031 | 0.033 |
| Station One | 0.025 | 0.034 | 0.036 | 0.032 |
| Station Two | 0.266 | 0.244 | 0.250 | 0.253 |
| Station Four | 5.340 | 5.289 | 5.214 | 5.281 |
| Station Six | 107.415 | 108.704 | 106.545 | 107.555 |
| Station Eight | 633.211 | 636.107 | 642.983 | 637.434 |
| Nærumbanen (2T) | 42.576 | 43.328 | 43.498 | 43.134 |
| Nærumbanen (3T) | 1121.852 | 1129.423 | 1129.642 | 1126.972 |

**Table 13.13:** Extended model: 'No Collision', limits 1, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 7784 | 7800 | 7804 | 7796 |
| Example with Two Trains | 7788 | 7800 | 7804 | 7797 |
| Shared segment | 10176 | 10180 | 10448 | 10268 |
| Large network | 12256 | 12304 | 12324 | 12295 |
| Station One | 8680 | 8708 | 8716 | 8701 |
| Station Two | 13528 | 13588 | 13596 | 13571 |
| Station Four | 48280 | 48504 | 48540 | 48441 |
| Station Six | 150972 | 151848 | 151860 | 151560 |
| Station Eight | 370944 | 372896 | 372908 | 372249 |
| Nærumbanen (2T) | 125832 | 126796 | 127016 | 126548 |
| Nærumbanen (3T) | 550268 | 550300 | 550272 | 550280 |

**Table 13.14:** Extended model: 'No Collision', limits 1, Memory Usage Peak in KB

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.006 | 0.006 | 0.006 | 0.006 |
| Example with Two Trains | 0.068 | 0.064 | 0.075 | 0.069 |
| Shared Segment | 0.033 | 0.055 | 0.052 | 0.047 |
| Large Network | 0.038 | 0.063 | 0.040 | 0.047 |
| Station One | 0.048 | 0.044 | 0.047 | 0.046 |
| Station Two | 0.915 | 0.932 | 0.835 | 0.894 |
| Station Four | 33.160 | 32.814 | 35.634 | 33.869 |
| Station Six | 946.866 | 952.055 | 954.846 | 951.256 |
| Station Eight | 6550.005 | 6608.182 | 6603.568 | 6587.252 |
| Nærumbanen (2T) | 254.086 | 251.442 | 250.039 | 251.856 |
| Nærumbanen (3T) | 9754.725 | 9692.086 | 9644.453 | 9697.088 |

**Table 13.15:** Extended model: 'No Collision', limits 2, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 7780 | 7792 | 7796 | 7789 |
| Example with Two Trains | 8980 | 9004 | 9016 | 9000 |
| Shared segment | 10252 | 10256 | 10524 | 10344 |
| Large network | 12336 | 12384 | 12400 | 12373 |
| Station One | 8708 | 8732 | 8740 | 8727 |
| Station Two | 16604 | 16656 | 16664 | 16641 |
| Station Four | 139356 | 139608 | 139616 | 139527 |
| Station Six | 680100 | 680448 | 680452 | 680333 |
| Station Eight | 2103804 | 2104608 | 2104624 | 2104345 |
| Nærumbanen (2T) | 238384 | 239212 | 239240 | 238945 |
| Nærumbanen (3T) | 3638672 | 3638708 | 3638728 | 3638703 |

**Table 13.16:** Extended model: 'No Collision', limits 2, Memory Usage Peak in KB

## 13.6 UMC Model Results

This section presents the results for the experiments with the UMC implementation of the first control system variant as described in chapter 10. As mentioned in section 13.1.3, only the elapsed time is recorded. As with the experiments for the UPPAAL models, only the results for the 'No Collision' property will be shown. The complete set of results can instead be seen in tables E.65 - E.72 in appendix E.

When looking at the results for the 'No Collision' property (see table 13.17) with both the reservation and lock limits set to 1, it is evident that the verification time is higher than when using UPPAAL and the general growth rate is higher as well.

During the experimentation with *Station Four*, the number of computations (states) rose to over 100,000,000 and the test machine went into a halt due to exhausted resources. Naturally, the larger configurations would then result in state explosion as well so no results were obtained for *Station Four - Nærumbanen (3T)*.

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.063 | 0.063 | 0.064 | 0.063 |
| Example with Two Trains | 0.752 | 0.748 | 0.758 | 0.753 |
| Shared Segment | 0.342 | 0.341 | 0.341 | 0.341 |
| Large Network | 0.346 | 0.343 | 0.345 | 0.345 |
| Station One | 0.368 | 0.379 | 0.378 | 0.375 |
| Station Two | 12.400 | 12.355 | 12.352 | 12.369 |
| Station Four | | | | |
| Station Six | | | | |
| Station Eight | | | | |
| Nærumbanen (2T) | | | | |
| Nærumbanen (3T) | | | | |

**Table 13.17:** UMC Model: 'No Collision', limits 1, Elapsed time in seconds

When the reservation limit and the lock limits were increased to 2, the same observations again arose. It took more time to verify the configurations and the growth rate increased at a higher rate. Also, during the experimentation with different properties, it was apparent that the verification times for the various formulae were significantly different from each other, while the verification time among the different queries was relatively consistent in UPPAAL. The reason for this could be that each property formula depends on some abstractions, which may vary in complexity depending on the action or state properties that need

to be assessed.

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.066 | 0.066 | 0.066 | 0.066 |
| Example with Two Trains | 0.886 | 0.886 | 0.894 | 0.889 |
| Shared segment | 0.417 | 0.415 | 0.416 | 0.416 |
| Large network | 0.420 | 0.419 | 0.420 | 0.420 |
| Station One | 0.429 | 0.430 | 0.429 | 0.429 |
| Station Two | 20.210 | 20.193 | 20.154 | 20.186 |
| Station Four | | | | |
| Station Six | | | | |
| Station Eight | | | | |
| Nærumbanen (2T) | | | | |
| Nærumbanen (3T) | | | | |

**Table 13.18:** UMC model: 'No Collision', limits 2, Elapsed time in seconds

## 13.7 Revised Model Results

The purpose of these final experiments is to compare the efficiency between UPPAAL and UMC with more similar models.

When examining the results for the 'No Collision' property in terms of time usage (see table 13.19) with both limits set to 1, it is still clear that the evaluation time is generally lower than for UMC, but the growth rate is significantly worse than for the extended model (see section 13.5).

The growth rate causes a state explosion when experimenting with *Station Six*, which is why there are no results for *Station Six - Nærumbanen (3T)*. *Nærumbanen* with two trains was checked, but it also resulted in a state space explosion.

The additional interleavings also result in a larger memory usage as shown in table 13.20 compared to all the other variants.

The growth rate again becomes significantly higher both in terms of time and memory usage when the reservation limit and the lock limit are set to 2 (see tables 13.22 and 13.21).

The complete set of results for the revised model can be seen in tables E.73 - E.88 in appendix E.

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.010 | 0.010 | 0.012 | 0.011 |
| Example with Two Trains | 0.016 | 0.017 | 0.017 | 0.017 |
| Shared segment | 0.062 | 0.066 | 0.072 | 0.067 |
| Large network | 0.066 | 0.092 | 0.083 | 0.080 |
| Station One | 0.062 | 0.056 | 0.079 | 0.066 |
| Station Two | 1.713 | 1.782 | 1.785 | 1.760 |
| Station Four | 2101.282 | 2120.291 | 2110.154 | 2110.576 |
| Station Six | | | | |
| Station Eight | | | | |
| Nærumbanen (2T) | | | | |
| Nærumbanen (3T) | | | | |

**Table 13.19:** Revised model: 'No Collision', limits 1, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 7636 | 7652 | 7652 | 7647 |
| Example with Two Trains | 7648 | 7664 | 7664 | 7659 |
| Shared segment | 10184 | 10216 | 10224 | 10208 |
| Large network | 12180 | 12228 | 12244 | 12217 |
| Station One | 8516 | 8532 | 8552 | 8533 |
| Station Two | 20620 | 20672 | 20680 | 20657 |
| Station Four | 2347104 | 2347108 | 2347108 | 2347107 |
| Station Six | | | | |
| Station Eight | | | | |
| Nærumbanen (2T) | | | | |
| Nærumbanen (3T) | | | | |

**Table 13.20:** Revised model: 'No Collision', limits 1, Memory Usage Peak in KB

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.014 | 0.020 | 0.015 | 0.016 |
| Example with Two Trains | 0.170 | 0.194 | 0.174 | 0.179 |
| Shared segment | 0.085 | 0.033 | 0.032 | 0.050 |
| Large network | 0.090 | 0.112 | 0.101 | 0.101 |
| Station One | 0.082 | 0.069 | 0.095 | 0.082 |
| Station Two | 2.607 | 2.726 | 2.557 | 2.630 |
| Station Four | 4102.210 | 4170.611 | 4313.341 | 4195.387 |
| Station Six | | | | |
| Station Eight | | | | |
| Nærumbanen (2T) | | | | |
| Nærumbanen (3T) | | | | |

**Table 13.21:** Revised model: 'No Collision', limits 2, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 7636 | 7652 | 7652 | 7647 |
| Example with Two Trains | 8916 | 8936 | 8964 | 8939 |
| Shared Segment | 10248 | 10288 | 10292 | 10276 |
| Large Network | 12248 | 12296 | 12312 | 12285 |
| Station One | 8536 | 8560 | 8588 | 8561 |
| Station Two | 26932 | 26984 | 26992 | 26969 |
| Station Four | 4533576 | 4533580 | 4533572 | 4533576 |
| Station Six | | | | |
| Station Eight | | | | |
| Nærumbanen (2T) | | | | |
| Nærumbanen (3T) | | | | |

**Table 13.22:** Revised model: 'No Collision', limits 2, Memory Usage Peak in KB

## 13.8 Comparison of the Verification Results

As mentioned in the previous section, the growth rates for all the different models are calculated for the time usage and memory usage where the memory usage values have been converted from KB to MB. Figure 13.9 shows the graphs for the time usage where the number of stations vary along the x-axis. These

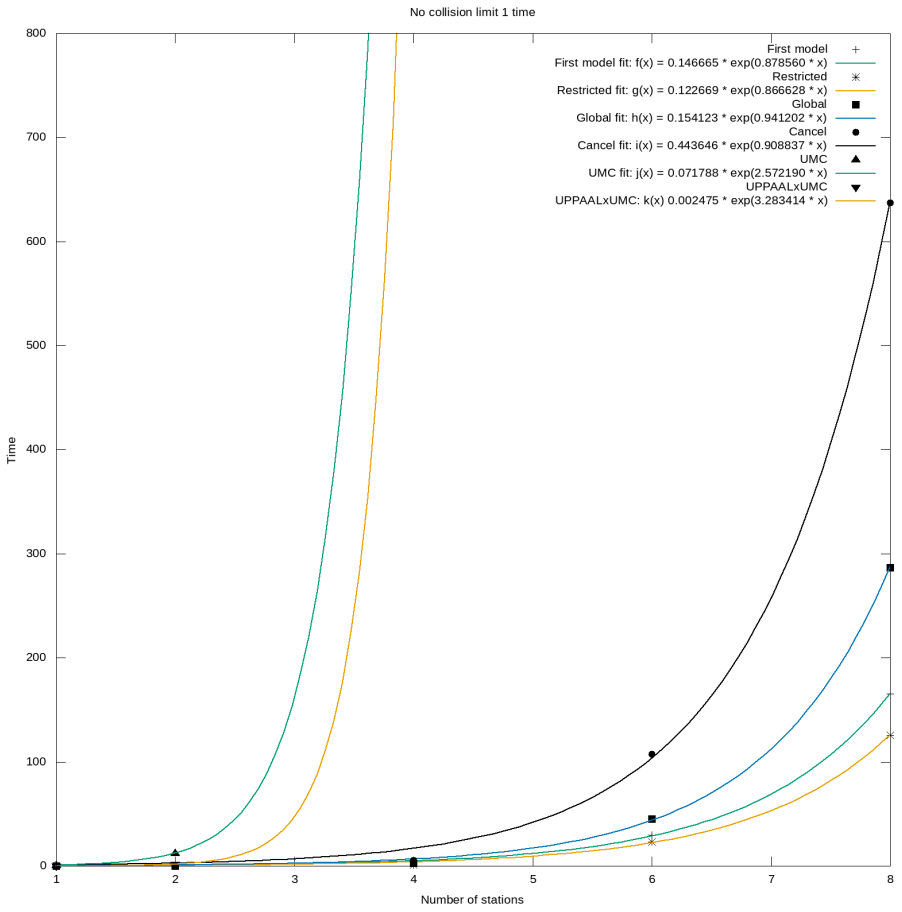



**Figure 13.9:** Elapsed time in seconds for 'No Collision' with reservation limit and lock limit set to 1 for varying number of stations, for each of the model variants.

are defined using the following functions:

- First model: $f(x) = 0.146673e^{0.878554x}$
- Restricted model: $g(x) = 0.122670e^{0.866628x}$
- Global model: $h(x) = 0.154124e^{0.941201x}$
- Extended model: $i(x) = 0.443645e^{0.908837x}$
- UMC model: $j(x) = 0.071812e^{2.572106x}$
- Revised model: $k(x) = 0.002475e^{3.283414x}$

Based on these functions, it can be seen that the restricted model and the first model have approximately the same growth rate, but the constant factor is lower for the restricted model and the verification of this model is therefore more efficient. The global variant of the first model is generally worse than the first model in terms of growth rate and the constant factor is also higher.

Unsurprisingly, the extended model have a significantly higher growth rate and constant factor than the previous variants. The variants with the lowest constant factors and the highest growth rates are the UMC model and the revised model. However, the UMC function could only be derived from two data points (*Station One* and *Station Two*), which means that it is possible that different functions fit these points. Based on the actual experiment results, it is known that the growth rate is higher than the revised model, so the chosen fit is acceptable on this basis, but it is not a very precise fit in terms of the actual verification time.

Figure 13.10 shows the memory usage graphs for all models except for the UMC model which had no memory measurements.

The functions for the fitted curves related to the memory usage are as follows:

- First model: $f(x) = 4.894418e^{0.435364x}$
- Restricted model: $g(x) = 7.332739e^{0.428563x}$
- Global model: $h(x) = 7.163201e^{0.435932x}$
- Extended model: $i(x) = 7.897601e^{0.482296x}$
- Revised model: $k(x) = 0.198516e^{2.344456x}$

Based on these functions, it can be seen that the growth rates for the first model and the global model are approximately the same, which makes sense since the number of generated states is about the same even though UPPAAL

can optimize the time taken to generate them. In terms of memory usage, the first model generally requires less memory except for smaller configurations, which may be due to the more rigid structure of the state machine. The memory usage for the extended model is generally higher than those of the other models with the only exception being the revised model. The revised model has the highest growth rate, but the number of data points used for the fit was also only three (*Station One*, *Station Two* and *Station Four*), which means that the fit may be less precise than the ones using more data points.

For reservation limit and lock limit set to 2, the general tendencies in terms of verification time and memory usage are the same as for the limits set to 1 (see figure 13.11).

The functions for the verification time usage with limits set to 2 are as follows:

- First model: $f(x) = 0.560786e^{0.952722x}$
- Restricted model: $g(x) = 0.204902e^{0.901196x}$
- Global variant of first model: $h(x) = 0.836408e^{0.9444423x}$
- Extended first model: $i(x) = 2.540509e^{0.982604x}$
- UMC variant of first model: $j(x) = 0.096373e^{2.670318x}$
- Revised first model variant of first model $k(x) = 0.002557e^{3.468020x}$

The functions for the memory usage with limits set to 2 are as follows:

- First model: $f(x) = 5.715752e^{0.538299x}$
- Restricted model: $g(x) = 7.556901e^{0.451200x}$
- Global variant of first model: $h(x) = 8.970483e^{0.513957x}$
- Extended first model: $i(x) = 17.218585e^{0.601176x}$
- Revised first model variant of first model $k(x) = 0.148699e^{2.581208x}$

Additional graphs for the other safety properties can be seen in appendix F.

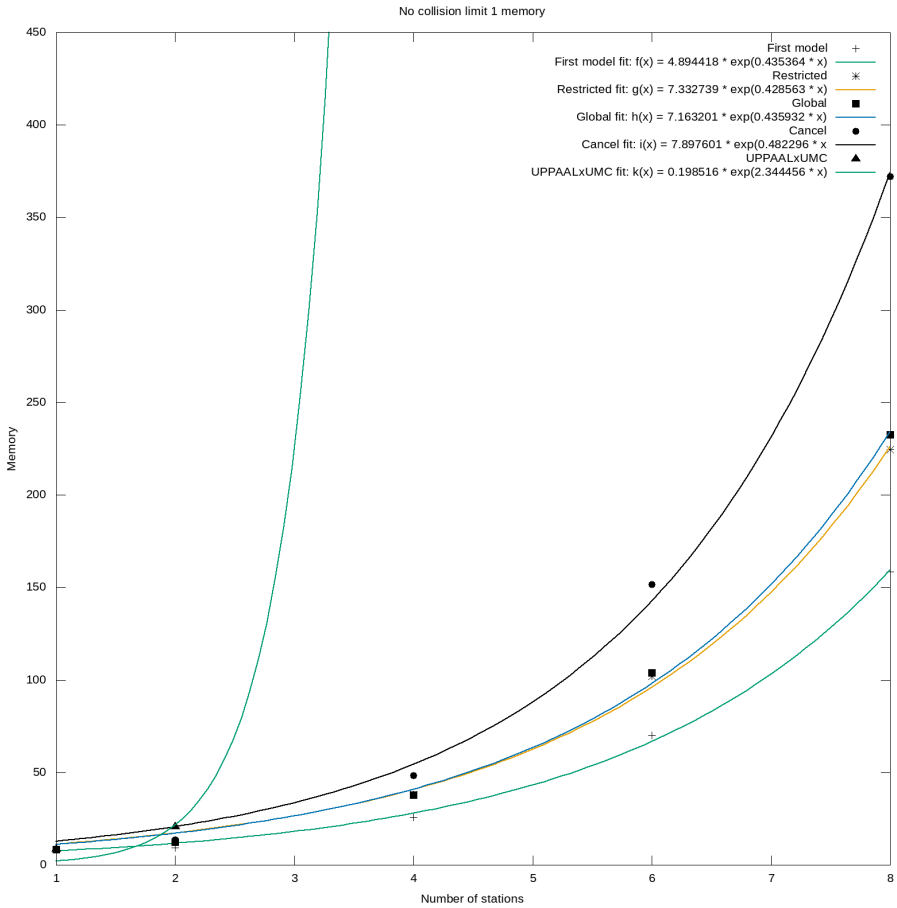



**Figure 13.10:** Memory Usage Peak in MB for 'No Collision' with reservation limit and lock limit set to 1 for varying number of stations, for each of the model variants.

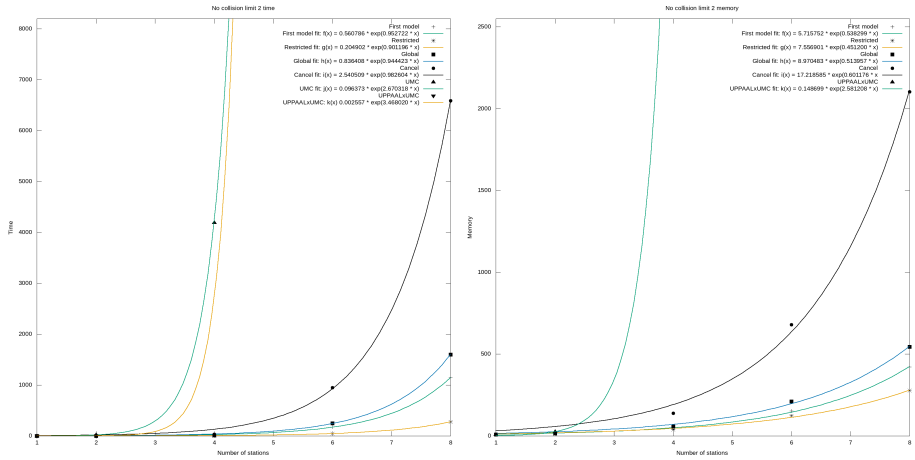



**Figure 13.11:** Elapsed time (left) and memory usage peak (right) for checking the 'No collision' property for varying number of stations, for each of the model variants with reservation limit and lock limit set to 2.

Chapter 14

# Discussion

After having modelled and verified various control system variants in UPPAAL and one of them in UMC, one easily notices the differences between the two tools. This chapter summarizes and discusses the most significant differences between the two tools as well as compares them with the tools used in [Gei18].

Section 14.1 describes how modelling with one tool differs from the other and section 14.2 discusses the difference in verification both in terms of property formulation and in terms of efficiency. Section 14.3 then discusses which features the two tools offer that can be used to model check and explore a model. Finally, sections 14.4 and 14.5 compare this project with [Gei18] with regard to both modelling and verifying properties.

## 14.1 Modelling with UPPAAL and UMC

**UPPAAL Locations and UMC States**
As explained in chapter 2, both UPPAAL and UMC are based on a state machine representation, but while UMC's state machines are on a textual form, UPPAAL uses a graphical notation. This may make UPPAAL more appealing and intuitive to use.

Another very significant difference related to the representation through state machines is that UPPAAL's state machines are timed state machines. With the use of clocks and urgent or committed locations, it is possible to enforce or prevent progress, which is not possible with UMC. On the other hand, UMC states can be composite, which means that states can be sub-states of other states. Neither of these features were, however, used in this project since trains should have the freedom to move whenever they wish to and there is also no need to have nested states.

Besides the differences between UPPAAL locations and UMC states, the two tools also have differences in the way edges and transitions can be used.

**UPPAAL Edges and UMC Transitions**
As discussed in section 10.1.2, UPPAAL edges and UMC transitions are generally of the same form, but still with noticeable differences. Both types can consist of a guard, a synchronization statement and some update statements/actions. The biggest difference here lies in the synchronization statement.

UPPAAL's synchronization statements include both the emission and reception on a channel while UMC triggers are only for the reception of asynchronous signals or synchronous operations. The emission - or operation call/signal sending - is instead a part of the UMC actions. Since the actions is a *list* of actions, this means that a UMC object can actually send multiple signals and call multiple operations from the same state as it receives a signal or operation call. With UPPAAL, it is only possible to do a single synchronization from a location, though the single synchronization can be on a broadcast channel. A broadcast channel will allow multiple instances to receive an emission at the same time, but it is still only the same emission.

The possibility of calling operations and sending signals during a transition where also a signal or operation call is received, means that the number of required states in the UMC model can actually be reduced. For example, in the UPPAAL models, a `CB` moves to its `SegmentChecked` location when it receives a reservation request and it then returns to its `Idle` location while emitting on the requesting `Train`'s (negative) acknowledgement channel. The `SegmentChecked` state in the UMC model could be omitted since it would be possible for a `CB` in UMC to return an answer to the `Train` in the same transition as the one in which it received the request.

Another advantage of UMC objects' way of communicating is how data can be shared among objects. With UPPAAL, one had to create a channel for each kind of data that could be sent between two specific instances and then use select statements to define what data an instance would accept. This can quickly result in a large number of channels if the value range becomes large, but with

UMC, it is possible to send data in the same manner as how arguments are used in functions. UMC is also more flexible in terms of the data type that can be sent. For example, `Train` objects can send references to themselves such that a `CB` can use the reference to send signals back.

**Global and Local Declarations**
A frequently used feature of UPPAAL was the possibility to declare variables, constants and functions locally and globally. This gave a clear distinction between the individual state spaces of `Train`'s and `CB`'s and the globally known network information. The possibility to define and use functions additionally improved the readability of the state machines and reduced the need for repetitive code.

With UMC, it was only possible to have local declarations, which did not include functions. This means that the update of a `Train`'s position had to be done differently.

**Instantiation of UPPAAL instances and UMC Objects**
In the UPPAAL implementations, data belonging to specific `CB`s and `Train`s was declared as global constants and fetched during the initialization step. The idea behind doing this was to make the instantiation of objects as general as possible. This would allow it to happen automatically, which would not only improve the instantiation process, but also the verification process.

Since UMC offers no way to declare global constants, the fetching of data to UMC objects was implemented as passing the data as arguments during instantiation. Compared to UPPAAL, UMC instantiation is fortunately more flexible, since the parameters for a specific class are not defined beforehand. This means that for each instantiation, one can choose to assign values to any number of variables declared locally within the class. Furthermore, arrays do not have to be instantiated with a fixed length beforehand, so it is easy to instantiate `Train`s with routes of different lengths. Unfortunately, the manual instantiation also means that it is not possible to iterate over all objects of the same class during verification.

## 14.2 Verifying with UPPAAL and UMC

To formulate and verify properties in UMC, abstractions must be formulated so specific UMC configurations or evolutions can be referred to. The formulation of abstractions can be cumbersome in itself, but experiments also showed that the number of abstractions affected the efficiency of the verification process

significantly. When verifying a property, one should therefore only include the abstractions relevant for that property, although this would mean that verifying multiple properties requires modification of the list of abstractions between each verification.

Even when the list of abstractions was reduced to only those that were needed for the verification of a specific property, UPPAAL turned out to verify properties much more efficiently - even with the revised model. This clearly shows that the implementation of UPPAAL has focused on optimization of verification with its on-the-fly and symbolic model checking technique [yPD98]. The focus on efficiency is also visible from the offered options such as 'State Space Reduction' and the 'Reuse' of state space. UMC is instead mainly used in academia, research and for experimental purposes. It verifies also with an on-the-fly approach but in a recursive depth-first manner, where the explicit configurations and evolutions are created as a part of the resulting graph. [TBFGM11]. Besides that and as mentioned before, the UPPAAL model was created first and the focus of the project has been on that tool and this too has influenced the design of the railway control system.

## 14.3 Model Checking with UPPAAL and UMC

Besides the different verification options offered by UPPAAL, the tool also includes a simulator. This makes it possible to see and fire enabled edges while following the change in variable values and the change in location of each instance of a system.

This is optimal for manual testing and debugging, since it is possible to use it for exploring generated diagnostic traces when a property fail or when a property asks for a path to a certain state.

UMC also offers a kind of simulator but the UPPAAL simulator is overall more user-friendly due to its more graphical and simple user interface. On the other hand, UMC can draw computation trees that include the reachable configurations and evolutions labelled after the model's abstractions.

## 14.4 Comparison with the RSL-SAL Models

The RAISE Specification Language supports many of the same data types as the ones supported by UPPAAL and UMC and more. But one of the more

distinct differences between RSL/RSL-SAL and the others is that RSL/RSL-SAL do not use state machine representations. Besides the general syntax and construction in the different languages and tools, the actual modelling of the engineering concept here also deviates from the modelling of it with RSL-SAL, which is described in [GH18].

A significant difference in the tools is UPPAAL's and UMC's use of templates and classes with local variables and the possibility to communicate using channels and signals. RSL-SAL, on the hand, only uses global data and for information exchange, shared variables are used. Futhermore, maps and sets were used in RSL-SAL to store information, while sets of data was modelled as arrays in UPPAAL and UMC.

Although arrays may be more fit for ordered sequences such as routes, maps and sets having no fixed size are a better fit for storing obtained reservations and locks as done in the RSL-SAL implementation. Using sets for these is far more intuitive and straightforward and it also allows trains to obtain reservations and locks in any order. This was avoided in the UPPAAL and UMC solutions by the use of indices, which results in less flexible models.

Another big difference between the RSL-SAL solution and the ones presented here is the representation of the network, i.e. how segments are or can be connected. In the RSL-SAL implementation, a network was modelled as a set of connections of two segments and end points of one segment. This is slightly different here since points, which allows for two different connections, are modelled as a single array and not two different connections. By doing this, the information that a control box needs can be saved at a single index in the global array, which makes it easier for it to retrieve it.

With the limited number of data types in UPPAAL and UMC, the solutions here mainly used integers for representing various concepts, which made the verification relatively efficient - at least for the UPPAAL models - but as previously mentioned, RSL-SAL's different collection types made the implementation more straightforward. Also the use of records made the implementation easier to comprehend although it slowed down the verification [Gei18, p.12].

## 14.5 Comparison with RSL-SAL Verification Results

The design of both the UPPAAL models and the UMC model are as previously mentioned based on the same engineering concept as the one used in [Gei18]. It

is hence interesting to compare the verification of one of the RSL-SAL models with the verification of a model presented in this project in order to determine which tool is most efficient for verifying the distributed railway control system.

The models that are chosen for comparison are this project's restricted model (see section 9.1) with reservation and lock limits set to 1 and the 'Restricted System Model' from [Gei18, sec. 6.6]. To prevent confusion between the two, the latter will be referred to as *RSL-SAL v3*. The UMC model will not be considered in the comparison since its results have already been compared to UPPAAL's in section 13.8.

The comparison between the chosen UPPAAL model and the RSL-SAL model (as shown in section 14.5.3) points towards UPPAAL being faster and more memory efficient, but the comparison is not completely conclusive since there are differences in both the used algorithms (see section 14.5.1) and the test machines (see section 14.5.2), which affects the results as well.[1]

### 14.5.1 Difference Between Algorithms

The key difference between the algorithms that are being compared is that RSL-SAL v3 allows for the request of the reservations at the two upcoming control boxes in any order. Trains in this project's restricted model will always reserve at the control box that is closest to it first.

Both models will, however, only make a full reservation of segment before doing anything else, thereby making the non-determinism a smaller issue for RSL-SAL v3 compared to the restricted model.

### 14.5.2 Differences Between Test Machines

A possibly more influential difference between the experiments with the two tools is that the results from the two models originate from two different test machines. The machine used for testing the restricted model here used an AMD Ryzen 7 2700X CPU clocked at 4GHz with 64GB of memory. On the other hand, RSL-SAL v3 was tested on an Intel(R) Xeon(R) CPU E5-2660 v3 clocked at 2.60GHz with 125GB of memory ([Gei18, p. 119]).

According to [Use], the AMD Ryzen 7 2700X is significantly faster than Intel(R) Xeon(R) CPU E5-2660 v2 in both single and multi-core performance. The

[1] The author has also later revised the RSL-SAL models [GH18]

Intel(R) Xeon(R) CPU E5-2660 v3 is a newer version of Intel(R) Xeon(R) CPU E5-2660 v2 and therefore potentially faster than the benchmarked results, but the results for this newer version were not available.

Another difference is that other users had access to and used the machine during the experimentation with RSL-SAL v3, which could skew the results even more in favour of UPPAAL [Gei18, p. 119]. The experimentation with the restricted model only had one user using it and no additional processes were running in the background.

### 14.5.3 Data Comparison

The configurations that will be compared under the 'No Collision' property are:

1. Example with Two Trains
2. Shared Segment
3. Large Network

Tables 14.1 and 14.2 show the average time and memory usage for the 'No Collision' property for the compared models. The data for RSL-SAL v3 originates from Appendix E.3 in [Gei18] where the time measurements have been converted into seconds first before calculating the average time usage. The average memory usage is calculated from the raw data.

| Configuration | Restricted Model | RSL-SAL v3 |
|---|---|---|
| 1 | 0.024 | 713.863 |
| 2 | 0.016 | 176.953 |
| 3 | 0.017 | |

**Table 14.1:** Verification time in seconds for the restricted model and RSL-SAL v3

The tables show that there are no results for the *Large Network* configuration using RSL-SAL v3, which is due to large time and memory consumption on the test machine as mentioned in [Gei18, p. 121-122].

When looking at the memory usage in table 14.2, it is clear that UPPAAL has a significantly lower memory usage than the SAL symbolic model checker. This is more directly comparable than the time measurements since the model checkers will generate the same state space regardless of the system. The memory

| Configuration | Restricted | RSL-SAL v3 |
|---|---|---|
| 1 | 8552 | 598193 |
| 2 | 9800 | 411864 |
| 3 | 12177 | |

**Table 14.2:** Memory usage in KB for the restricted model and RSL-SAL v3

exhaustion that occurred during the RSL-SAL/SAL symbolic model checker experimentation will therefore most likely also occur on the AMD Ryzen 2700X system because it only has 64GB of memory compared to the Intel(R) Xeon(R) CPU E5-2660 v3 system's 125GB of memory.

In order to get more comparable results, it would be beneficial to either model check RSL-SAL v3 on the AMD Ryzen 2700X machine or model check the restricted UPPAAL model on the Intel(R) Xeon(R) CPU E5-2660 v3 machine. By reducing the number of variables in the comparison, it would be easier to show with more certainty whether the UPPAAL model checker is more efficient than the SAL symbolic model checker. However, as the expressive power of the different tools are very different, it is a challenge to create identical models and thereby get a direct comparison.

Chapter 15

# Conclusion

In this project, model checking was used to analyse and verify different variants of a distributed railway control system with respect to the prevention of derailment and collision of trains. These variants have primarily been modelled using UPPAAL and one of them was also modelled using UMC.

The first control system variant was based on an existing case study and it was modelled and verified using both UPPAAL and UMC with only differences that were necessary due to the differences in the modelling languages' expressive power. Experiments with the two models and tools showed that for the created models, UPPAAL was far superior in terms of efficiency. Regarding the modelling process, the tools are somewhat similar. Both are based on state machines with guards, triggers and updates, but when formulating the properties that should be verified, UPPAAL has the advantage that it can iterate over all instances of a system, which will allow one to formulate general properties instead of formulating configuration dependent properties. UMC, on the other hand, has some advantages with regard to instantiation with configuration data and it has more options for communication between objects.

From the first model, a series of other variants was created in UPPAAL in order to either improve or extend the first model after having analyzed the results from initial experiments. The first improvement to the model was a restriction of the possible evaluation sequence that can occur with the first model. This

restriction improved the general performance and made it possible to verify larger networks without resulting in a state space explosion.

The third variant introduces an additional 'cancel' operation, which allows a train control computer to cancel obtained reservations and locks. The introduction of this operation preserved the safety properties, but the increase in number of operations also resulted in a higher memory usage and a longer verification time. Still, the modelling of this variant showed that only very few parts of the first model required changes, so it is very easy to add additional operations to the first model without affecting the safety of the system.

The fourth variant is not actually a new variant of the railway control system, but a new implementation variant. That is, it focuses more on the actual implementation of the first model rather than possible modifications to the case study. In this model, the control components exclusively use the global declarations for their static data instead of copying it to their own local state spaces first and using the data from there. Experiments generally showed that this change had no significant influence on the verification results, but the memory usage was slightly higher for the global model, which was possibly due to certain optimizations made by UPPAAL for local declarations.

The final variant was created after having modelled the first variant in UMC and realizing that certain aspects of the modelling could not be replicated in UMC very well. For example, UPPAAL and UMC had no identical features for communication and there was no way to implement global variables, constants or functions in UMC. The final UPPAAL variant was hence created in order to get a model that was closer to what could be implemented in UMC and thereby make a more fair comparison of the tools. The fact that certain things could be modelled in UPPAAL and not in UMC of course already shows an advantage of one tool over the other but since the system was designed and modelled for UPPAAL first, it also took advantage of UPPAAL's features without considering UMC's. Had it been designed and modelled for UMC first, it may have had a completely different design, which may have given different results.

The experiments with the revised variant of the first model showed that the additional interleavings between the different state machines resulted in a much more similar graph to that for the UMC results. That is, the state space became much larger and the verification time became much longer, but the verification of the UPPAAL model was nevertheless still more efficient than the verification of the UMC model.

Generally, the results from the different experiments show that the UPPAAL models can verify the railway control system for relatively large networks within a reasonable time frame. This also included a real-world network, which was

model checked with two and three trains. The UMC model had a scalability problem, which could possibly be resolved or diminished by implementing a UMC model that makes better use of UMC's feature set.

Compared to the RSL-SAL model, which was model checked with the SAL symbolic model checker, the models here could verify the distributed railway control system for larger networks. Especially the experiments with *Nærumbanen* shows that the models introduced in this thesis are useful for verifying real-world examples, which are often larger than a couple of segments - though the scope of this project is still only within the sizes of local railway systems.

To ease the process of formulating configuration data for the models, a graphical tool was also created using a model-driven engineering approach. A user of the tool can draw a railway network and trains as a graph of control boxes connected by segments and freely standing train nodes. The tool can then generate code for any of the presented models with the concrete configuration data from the provided graph. This is not only useful for testing different kinds of configurations or for experimenting, but if one were to actually use the models of the railway control system to verify the railway control system for real-world railway networks, using the tool is both much faster than manually formulating the configuration data, and it is also less prone to make mistakes.

The most valuable outcome of this project is the learning of the two model checking tools' capabilities with regard to modelling and efficiency. Once one model was created, it was not difficult to make small modifications to that in order to create other variants, and the similarities between UPPAAL and UMC also made it easy to translate a model in one tool to a model in the other tool. Finally, the project showed that UPPAAL turned out to be able to verify networks of large sizes due to its efficiency. This makes it a suitable candidate for model checking distributed railway control systems in real-world contexts.

In future work, it would be interesting to see whether a UMC model created independently from the UPPAAL model would be able to verify the distributed railway control system for these larger networks.

# Bibliography


[BDL04] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on UPPAAL. In Marco Bernardo and Flavio Corradini, editors, *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, number 3185 in LNCS, pages 200–236. Springer–Verlag, September 2004.

[BtBF+18] Davide Basile, Maurice H. ter Beek, Alessandro Fantechi, Stefania Gnesi, Franco Mazzanti, Andrea Piattino, Daniele Trentini, and Alessio Ferrari. On the industrial uptake of formal methods in the railway domain - A survey with stakeholders. In Carlo A. Furia and Kirsten Winter, editors, *Integrated Formal Methods*, pages 20–29. Springer International Publishing, 2018.

[ECfES11] CENELEC European Committee for Electrotechnical Standardization. *EN 50128:2011 – Railway applications – Communications, signalling and processing systems – Software for railway control and protection systems.* 2011.

[Ecla] Eclipse. Eclipse Modeling Framework (EMF). `https://www.eclipse.org/modeling/emf/`. [Online; accessed 21-April-2019].

[Eclb] Eclipse. Graphical Modeling Project (GMP). `https://www.eclipse.org/modeling/gmp/`. [Online; accessed 22-April-2019].

[FGH+16] A. Fantechi, S. Gnesi, Anne Elisabeth Haxthausen, J. van de Pol, M. Roveri, and H. Treharne. Sardin - a safe reconfigurable distributed interlocking. In *Proceedings of 11th World Congress on*

*Railway Research (WCRR 2016)*. Ferrovie dello Stato Italiane, 2016.

[FH18] Alessandro Fantechi and Anne Elisabeth Haxthausen. Safety interlocking as a distributed mutual exclusion problem. In *Formal Methods for Industrial Critical Systems*, pages 52–66. Springer, 2018.

[FHN17] Alessandro Fantechi, Anne E. Haxthausen, and Michel B. R. Nielsen. Model checking geographically distributed interlocking systems using UMC. In *2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 278–286, 2017.

[Gei18] Signe Geisler. Model checking distributed railway interlocking systems. M.sc. thesis, Technical University of Denmark, Kongens Lyngby, June 2018.

[GH18] Signe Geisler and Anne E. Haxthausen. Stepwise development and model checking of a distributed interlocking system - using raise. In Klaus Havelund, Jan Peleska, Bill Roscoe, and Erik de Vink, editors, *Formal Methods*, pages 277–293, Cham, 2018. Springer International Publishing.

[HKL+10] Helle Hvid Hansen, Jeroen Ketema, Bas Luttik, Mohammad Reza Mousavi, and Jaco van de Pol. Towards model checking executable UML specifications in mCRL2. *Innovations in Systems and Software Engineering*, 6(1):83–90, Mar 2010.

[HP00] Anne E. Haxthausen and Jan Peleska. Formal Development and Verification of a Distributed Railway Control System. In *IEEE Transactions on Software Engineering*, volume 26, pages 687–701. IEEE, 2000.

[JMN+14] Philip James, Faron Möller, Hoang Nga Nguyen, Markus Roggenbach, Steve Schneider, Helen Treharne, Matthew Trumble, and David Williams. Verification of Scheme Plans Using CSP$|$$|$B. In Steve Counsell and Manuel Núñez, editors, *Software Engineering and Formal Methods*, volume 8368 of *Lecture Notes in Computer Science*, pages 189–204. Springer, 2014.

[LCPT16] Christophe Limbrée, Quentin Cappart, Charles Pecheur, and Stefano Tonetta. Verification of railway interlocking-compositional approach with ocra. In *International Conference on Reliability, Safety, and Security of Railway Systems*, pages 134–149. Springer, 2016.

[Maz] Franco Mazzanti. Umc v4.8b. `http://fmt.isti.cnr.it/umc/V4.8/umc.html`. [Online; accessed 27-May-2019].

[MFH17] Hugo Daniel Macedo, Alessandro Fantechi, and Anne E. Haxthausen. Compositional model checking of interlocking systems for lines with multiple stations. In Clark Barrett, Misty Davies, and Temesghen Kahsai, editors, *NASA Formal Methods: 9th International Symposium, NFM 2017, Proceedings*, volume 10227 of *Lecture Notes in Computer Science*, pages 146–162. Springer International Publishing, 2017.

[Nie16] M Nielsen. Model checking geographically distributed railway control systems. *DTU Compute, Technical University of Denmark, Tech. Rep.*, 2016.

[PG07] Juan Ignacio Perna and Chris George. Model Checking RAISE Applicative Specifications. In *Proceedings of the Fifth IEEE International Conference on Software Engineering and Formal Methods, 2007*, pages 257–268. IEEE Computer Society Press, 2007.

[TBFGM11] Maurice H Ter Beek, Alessandro Fantechi, Stefania Gnesi, and Franco Mazzanti. A state/event-based model-checking approach for the analysis of abstract system properties. *Science of Computer Programming*, 76(2):119–135, 2011.

[Uni] Uppsala Universitet. Uppaal web help. `https://www.it.uu.se/research/group/darts/uppaal/help.php?file=Introduction.shtml`. [Online; accessed 27-May-2019].

[Use] UserBenchmark. UserBenchmark Compare. `https://cpu.userbenchmark.com/Compare/Intel-Xeon-E5-2660-v2-vs-AMD-Ryzen-7-2700X/m13068vs3958`. [Online; accessed 25-May-2019].

[VHP17] Linh Hong Vu, Anne E. Haxthausen, and Jan Peleska. Formal modelling and verification of interlocking systems featuring sequential release. *Science of Computer Programming*, 133, Part 2:91 – 115, 2017. http://dx.doi.org/10.1016/j.scico.2016.05.010.

[yPD98] Wang yi, Paul Pettersson, and Mats Daniels. Automatic verification of real–time communicating systems by constraint–solving. 06 1998.

Appendix A

# First Model

*`Initializer` and `Point` do not have any local declarations.

## A.1 Global Declarations

```
//Configuration Data - Part 1
const int NTRAIN = ...; //Number of trains
const int NCB = ...; //Number of control boxes
const int NPOINT = ...; //Number of points
const int NSEG = ...; //Number of segments
const int NROUTELENGTH = ...; //Maximum length of a route (in number of
    segments)

//TYPES
typedef int[0, NTRAIN-1] t_id; //Train IDs
typedef int[0, NCB-1]  cB_id; //Control box IDs
typedef int[0, NPOINT-1] p_id; //Point IDs
typedef int[0, NSEG-1] seg_id; //Segment IDs
typedef int[-1, NTRAIN-1] tV_id; //Train IDs with -1
typedef int[-1, NCB-1] cBV_id; //Control box IDs with -1
typedef int[-1, NPOINT-1] pV_id; //Point IDs with -1
typedef int[-1, NSEG-1] segV_id; //Segment IDs with -1
typedef int[0, NROUTELENGTH] cBRoute_i;
    //Indices of arrays of length NROUTELENGTH
typedef int[0, NROUTELENGTH-1] segRoute_i;
    //Indices of arrays of length NROUTELENGTH-1

typedef struct {
    cB_id cb;
```

```
    seg_id seg;
} reservation; //Reservation type for reservation of segment seg at control
     box cb

//Configuration Data - Part 2
const int[0,NROUTELENGTH] resLimit = ...; //Maximum number of full segment
     reservations allowed at a time per train
const int[0,NROUTELENGTH-1] lockLimit = ...; //Maximum number of locks
     allowed at a time per train

const segV_id segRoutes[NTRAIN][NROUTELENGTH]= {...}; //segRoutes[t] =
     segment route of train t
const cBV_id boxRoutes[NTRAIN][NROUTELENGTH+1] = {...}; //boxRoutes[t] =
     control box route of train t
const segV_id cBs[NCB][3]= {...}; //cBs[cb] = definition of control box cb
const pV_id points[NCB] = {...}; //points[cb] = ID of point associated with
     control box cb
const reservation initialRes[NTRAIN] = {...};//initialRes[t] = initial
     reservation of train t
bool pointInPlus[NPOINT] = {...}; //pointInPlus[p] = initial position of
     point p

//Channels
chan reqSeg[NCB][NTRAIN][NSEG]; //reqSeg[cb][t][s] = train t requests
     reservation of segment s at control box cb
chan reqLock[NCB][NTRAIN][NSEG][NSEG]; //reqLock[cb][t][s1][s2] = trian t
     requests switching/locking for connection between segment s1 and segment
      s2 at control box cb
chan OK[NTRAIN]; //OK[t] = control box returns acknowledgement to train t
chan notOK[NTRAIN]; //notOK[t] = control box returns negative acknowledge to
     train t
chan pass[NCB]; //pass[cb] = train enters critical section of control box cb
chan passed[NCB]; //passed[cb] = train leaves critical section of control box
      cb
chan switchPoint[NPOINT]; //switchPoint[p] = control box switches point p
chan OKp[NPOINT]; //OKp[p] = point p returns acknowledgement to control box
urgent broadcast chan start; //Initializer starts all processes

//Function that returns the ID of segment connected to segment s at control
     box cb
int nextSegment(cB_id cb, seg_id s){
    int s1 = cBs[cb][0];
    int s2 = cBs[cb][1];
    if(points[cb] > -1 && !pointInPlus[points[cb]]){
        s2 = cBs[cb][2];
    }

    if(s == s1){
        return s2;
    } else {
        return s1;
    }
}

////////////////////////////////////
//Well-formedness Functions
bool initialResIsConsistent(t_id id){
    return initialRes[id].cb == boxRoutes[id][1] && initialRes[id].seg ==
         segRoutes[id][0];
}

bool reservationIsWellFormed(reservation res){
    return cBs[res.cb][0] == res.seg || cBs[res.cb][1] == res.seg || cBs[res.
         cb][2] == res.seg;
}
```

```
bool sharesSegmentS(cB_id i, cB_id j, seg_id s){
    return  (i != j) &&
            (cBs[i][0] == s || cBs[i][1] == s || cBs[i][2] == s) &&
            (cBs[j][0] == s || cBs[j][1] == s || cBs[j][2] == s);
}

bool routesAreConsistent(t_id id){
    cBV_id bRoute[NROUTELENGTH+1] = boxRoutes[id];
    segV_id sRoute[NROUTELENGTH] = segRoutes[id];

    for(i:int[0,NROUTELENGTH-1]){
        if((bRoute[i+1] != -1) == (sRoute[i] == -1)){
            return false;
        }
        if(bRoute[i+1] != -1 && !sharesSegmentS(bRoute[i], bRoute[i+1],
             sRoute[i])){
            return false;
        }
    }
    return true;
}

bool sharesSegment(cB_id i, cB_id j){
    return (i != j) &&
            ((cBs[i][0] != -1 && (cBs[i][0] == cBs[j][0] || cBs[i][0] == cBs[
                j][1] || cBs[i][0] == cBs[j][2])) ||
            (cBs[i][1] != -1 && (cBs[i][1] == cBs[j][0] || cBs[i][1] == cBs[j
                ][1] || cBs[i][1] == cBs[j][2])) ||
            (cBs[i][2] != -1 && (cBs[i][2] == cBs[j][0] || cBs[i][2] == cBs[j
                ][1] || cBs[i][2] == cBs[j][2])));
}

bool boxRouteIsWellFormed(cBV_id route[NROUTELENGTH+1]){
    for(i:int[0,NROUTELENGTH-1]){
        if(route[i] == -1 && route[i+1] != -1){
            return false;
        }
        if(route[i+1] != -1 && !sharesSegment(route[i], route[i+1])){
            return false;
        }

    }
    return true;
}
bool canConnect(seg_id s1, seg_id s2){
    for(i:cB_id){
        if(cBs[i][0] == s1 && (cBs[i][1] == s2 || cBs[i][2] == s2)){
            return true;
        }
        if (cBs[i][0] == s2 && (cBs[i][1] == s1 || cBs[i][2] == s1)){
            return true;
        }
    }
    return false;
}

bool segRouteIsWellFormed(segV_id route[NROUTELENGTH]){
    int i = 0;
    if(route[0] == -1){
        return false;
    }

    for(i:segRoute_i){
```

```
        for(j:segRoute_i){
            if(j != i && route[i] == route[j] && route[i] != -1){
                return false;
            }
        }
    }

    while(i <= NROUTELENGTH - 2){
        if(route[i] == -1 && route[i+1] != -1){
            return false;
        }
        if(route[i+1] != -1 && !canConnect(route[i], route[i+1])){
            return false;
        }
        i++;
    }
    return true;
}

int pointIsWellFormed(cBV_id id){
    if(points[id] != -1){
        for(i : cB_id){
            if(i != id && points[i] == points[id]){
                return false;
            }
        }
    }
    return (points[id] == -1) == (cBs[id][2] == -1);
}

int otherBoxes(cB_id id, segV_id s){
    segV_id cB[3] = cBs[id];
    int found = 0;
    for(i:cB_id){
        if(id != i && (cBs[i][0] == s || cBs[i][1] == s || cBs[i][2] == s)){
            found++;
        }
    }
    return found;
}

int sharedSegments(cB_id i, cB_id j){
    int count = 0;
    if (cBs[i][0] != -1 && (cBs[i][0] == cBs[j][0] || cBs[i][0] == cBs[j][1]
         || cBs[i][0] == cBs[j][2])){
        count++;
    }
    if (cBs[i][1] != -1 && (cBs[i][1] == cBs[j][0] || cBs[i][1] == cBs[j][1]
         || cBs[i][1] == cBs[j][2])){
        count++;
    }
    if (cBs[i][2] != -1 && (cBs[i][2] == cBs[j][0] || cBs[i][2] == cBs[j][1]
         || cBs[i][2] == cBs[j][2])){
         count++;
    }
    return count;
}

bool cBIsWellFormed(cB_id id){
    segV_id cB[3] = cBs[id];

    //Invalid definitions
    if(cB[0] == -1 || (cB[1] == -1 && cB[2] != -1) || (cB[0] == -1 && cB[1]
         == -1)){
        return false;
```

```
    }
    if((cB[0] != -1 && (cB[0] == cB[1] || cB[0] == cB[2])) ||
        (cB[1] != -1 && (cB[1] == cB[0] || cB[1] == cB[2])) ||
        (cB[2] != -1 && (cB[2] == cB[0] || cB[2] == cB[1]))){
        return false;
    }

    //Case: []--x--
    if(cB[1] == -1){
        return otherBoxes(id, cB[0]) == 1;
    }

    //Case: --x--[]--y--
    if(cB[2] == -1){
        return otherBoxes(id, cB[0]) == 1 && otherBoxes(id, cB[1]) == 1;
    }

    //Case: Switch box
    for(i:cB_id){
        if (i != id && sharedSegments(i,id) > 1 && !(cBs[i][0] != cB[0] &&
             cBs[i][1] == cB[1] && cBs[i][2] == cB[2])){
            return false;
        }
    }
    return otherBoxes(id, cB[0]) == 1 && otherBoxes(id, cB[1]) == 1 &&
         otherBoxes(id, cB[2]) == 1;
}
```

## A.2 Train's Local Declarations

```
segV_id segments[NROUTELENGTH]; //Segment route
cBV_id boxes[NROUTELENGTH+1]; //Control box route

int[0,NROUTELENGTH] routeLength; //Route length (in number of segments)
segV_id curSeg; //Segment ID of actual current position

bool requiresLock[NCB]; //requiresLock[cb] = true if control box b is a
     switch box

cBRoute_i lockIndex = 1; //Index of control box that needs locking next
segRoute_i index = 0; //Index of current position in TCC's state space

int[0,1] resBit = 0; //Bit used to determine which end of a segment needs
     reservation next
cBRoute_i resCBIndex = 1; //Index of control box that next reservation is
     required at
cBRoute_i resSegIndex = 0; //Index of segment that needs reservation next

segV_id headSeg = -1; //Segment ID of actual head position of train - used
     only if train is in DoubleSegment
cB_id locks = 0; //Number of locks

void updateLockIndex(){
    while(lockIndex < NROUTELENGTH && !requiresLock[lockIndex]){
        lockIndex++;
    }
}

void initialize() {
    //Segments
    for(i : segRoute_i) {
        if(segRoutes[id][i]>-1) {
```

```
            routeLength++;
        }
    }
    curSeg = segRoutes[id][0];

    //Control boxes
    for(i : cBRoute_i) {
        if(boxRoutes[id][i] > -1){
            requiresLock[i] = points[boxRoutes[id][i]] > -1;
        }
    }

    //Locks and reservations
    resSegIndex = 1;
    updateLockIndex();
}

bool possibleToLock() {
    return lockIndex < routeLength && locks < lockLimit && ((resBit == 0 &&
        resSegIndex > lockIndex) || (resBit == 1 && resSegIndex >= lockIndex
        ));
}

bool hasArrived() {
    return index == routeLength-1;
}

bool possibleToReserve() {
    return resSegIndex < routeLength && resSegIndex - 1 - index < resLimit;
}

bool possibleToPass() {
    return resSegIndex > index + 1 && lockIndex > index + 1 && index + 1 <
        routeLength;
}

void updateResInfo(){
    resBit = resBit^1;
    resSegIndex = (resBit==0) ? resSegIndex + 1 : resSegIndex;
    resCBIndex = (resBit==1) ? resCBIndex + 1 : resCBIndex;
}

void updateLocationInfo(){
    curSeg = headSeg;
    headSeg = -1;
    if(requiresLock[index + 1]){
        locks--;
    }
    index++;
}

void updateHeadInfo(){
    headSeg = nextSegment(boxRoutes[id][index+1], curSeg);
}

void updateLockInfo(){
    locks++;
    lockIndex++;
    updateLockIndex();
}

bool isWellFormed(){
    return segRouteIsWellFormed(segRoutes[id]) &&
           boxRouteIsWellFormed(boxRoutes[id]) &&
           routesAreConsistent(id) &&
```

```
            reservationIsWellFormed(initialRes[id]) &&
            initialResIsConsistent(id);
}
```

## A.3 CB's Local Declarations

```
segV_id segments[3]; //Associated segments
segV_id connected =-1; //Segment ID of segment currently connected to
     segments[0]
pV_id point = -1; //Associated point

tV_id res[3] = {-1, -1, -1}; //res[i] = train ID of train with reservation of
      segments[i]
int[-1,2] result = -1; //Result of reservation and lock checks
tV_id tid = -1; //Train ID of communicating train
tV_id lockedBy = -1; //Train ID of train with lock

//Lock check results
const int ERROR = 0;
const int NOSWITCH = 1;
const int DOSWITCH = 2;

void initialize() {
    segments = cBs[id];
    point = points[id];
    connected = segments[1];

    if(point != -1 && !pointInPlus[point]){
        connected = segments[2];
    }

    for(i : t_id) {
        if (initialRes[i].cb == id) {
            seg_id s = initialRes[i].seg;
            if(s == segments[0]){
                res[0] = i;
            } else if (s == segments[1]){
                res[1] = i;
            } else {
                res[2] = i;
            }
        }
    }
}

int[-1,2] checkSegment(seg_id sid) {
    for(i:int[0,2]) {
        if(segments[i] == sid && res[i] == -1) {
            return i;
        }
    }
    return -1;
}

int[0,2] checkLock(seg_id s1, seg_id s2){
  if(lockedBy == -1 && (segments[0] == s1 && exists(i:int[1,2]) segments[i]
       == s2) || (segments[0] == s2 && exists(i:int[1,2]) segments[i] == s1))
       {
    if ((s1 == segments[0] && s2 == connected) || (s2 == segments[0] && s1 ==
          connected)){
        return NOSWITCH;
    } else
```

```
        return DOSWITCH;
    }
    return ERROR;
}


void clear(){
     lockedBy = -1;

     res[0] = -1;
     if(connected == segments[1]) {
        res[1] = -1;
     } else {
        res[2] = -1;
     }
}

void updateConnected(){
    if(connected == segments[1]){
        connected = segments[2];
    } else {
        connected = segments[1];
    }
}

void resetVariables(){
    tid = -1;
    result = -1;
}

void updateLockInfo(){
    lockedBy = tid;
    resetVariables();
}

void updateResInfo(){
    res[result]=tid;
    resetVariables();
}

bool isWellFormed(){
    return cBIsWellFormed(id) && pointIsWellFormed(id);
}
```

## A.4 System Declarations

```
system Initializer, Train, CB, Point;
```

Appendix B

# Configuration Data for UPPAAL Checks

## B.1 Check 1

```
const int NTRAIN = 2;
const int NCB = 3;
const int NPOINT = 1;
const int NSEG = 2;
const int NROUTELENGTH = 2;

...

const int[0, NROUTELENGTH] lockLimit = 1;
const int[0, NROUTELENGTH-1] resLimit = 1;
const segV_id segRoutes[NTRAIN][NROUTELENGTH] = {{0,1}, {1,-1}};
const cBV_id boxRoutes[NTRAIN][NROUTELENGTH+1] = {{0,1,2}, {1,2,-1}};
const segV_id cBs[NCB][3] = {{0, -1, -1}, {0, 1, -1}, {1, -1, -1}};
const reservation initialRes[NTRAIN] = {{1, 0}, {2, 1}};
const pV_id points[NCB] = {-1, -1, -1};
bool pointInPlus[NPOINT] = {true};
```

## B.2 Check 2

```
const int NTRAIN = 2;
const int NCB = 3;
const int NPOINT = 1;
const int NSEG = 2;
const int NROUTELENGTH = 2;
```

```
...

const int[0, NROUTELENGTH] lockLimit = 1;
const int[0, NROUTELENGTH-1] resLimit = 1;
const segV_id segRoutes[NTRAIN][NROUTELENGTH] = {{0,1}, {1,0}};
const cBV_id boxRoutes[NTRAIN][NROUTELENGTH+1] = {{0,1,2}, {2,1,0}};
const segV_id cBs[NCB][3] = {{0, -1, -1}, {0, 1, -1}, {1, -1, -1}};
const reservation initialRes[NTRAIN] = {{1, 0}, {1, 1}};
const pV_id points[NCB] = {-1, -1, -1};
bool pointInPlus[NPOINT] = {true};
```

## B.3 Check 3

```
const int NTRAIN = 2;
const int NCB = 4;
const int NPOINT = 1;
const int NSEG = 3;
const int NROUTELENGTH = 2;

...

const int[0, NROUTELENGTH] lockLimit = 1;
const int[0, NROUTELENGTH-1] resLimit = 1;
const segV_id segRoutes[NTRAIN][NROUTELENGTH] = {{0,1}, {1,2}};
const cBV_id boxRoutes[NTRAIN][NROUTELENGTH+1] = {{0,1,2}, {1,2,3}};
const segV_id cBs[NCB][3] = {{0, -1, -1}, {0, 1, -1}, {1, 2, -1}, {2, -1,
    -1}};
const reservation initialRes[NTRAIN] = {{1, 0}, {2, 1}};
const pV_id points[NCB] = {-1, -1, -1, -1};
bool pointInPlus[NPOINT] = {true};
```

## B.4 Check 4 + 5

```
const int NTRAIN = 1;
const int NCB = 4;
const int NPOINT = 1;
const int NSEG = 3;
const int NROUTELENGTH = 2;

...

const int[0, NROUTELENGTH] lockLimit = 1;
const int[0, NROUTELENGTH-1] resLimit = 1;
const segV_id segRoutes[NTRAIN][NROUTELENGTH] = {{0,1}};
const cBV_id boxRoutes[NTRAIN][NROUTELENGTH+1] = {{1,3,0}};
const segV_id cBs[NCB][3] = {{1, -1, -1}, {0, -1, -1}, {2, -1, -1}, {0, 1,
    2}};
const reservation initialRes[NTRAIN] = {{3, 0}};
const pV_id points[NCB] = {-1, -1, -1, 0};
bool pointInPlus[NPOINT] = {true}; //false for Check 5
```

## B.5 Check 6 + 7

```
const int NTRAIN = 1;
```

```
const int NCB = 4;
const int NPOINT = 1;
const int NSEG = 3;
const int NROUTELENGTH = 2;

...

const int[0, NROUTELENGTH] lockLimit = 1;
const int[0, NROUTELENGTH-1] resLimit = 1;
const segV_id segRoutes[NTRAIN][NROUTELENGTH] = {{0,2}};
const cBV_id boxRoutes[NTRAIN][NROUTELENGTH+1] = {{1,3,2}};
const segV_id cBs[NCB][3] = {{1, -1, -1}, {0, -1, -1}, {2, -1, -1}, {0, 1,
    2}};
const reservation initialRes[NTRAIN] = {{3, 0}};
const pV_id points[NCB] = {-1, -1, -1, 0};
bool pointInPlus[NPOINT] = {true}; //false for Check 7
```

## B.6 Check 8 + 9

```
const int NTRAIN = 1;
const int NCB = 4;
const int NPOINT = 1;
const int NSEG = 3;
const int NROUTELENGTH = 2;

...

const int[0, NROUTELENGTH] lockLimit = 1;
const int[0, NROUTELENGTH-1] resLimit = 1;
const segV_id segRoutes[NTRAIN][NROUTELENGTH] = {{1,0}};
const cBV_id boxRoutes[NTRAIN][NROUTELENGTH+1] = {{0,3,1}};
const segV_id cBs[NCB][3] = {{1, -1, -1}, {0, -1, -1}, {2, -1, -1}, {0, 1,
    2}};
const reservation initialRes[NTRAIN] = {{3, 1}};
const pV_id points[NCB] = {-1, -1, -1, 0};
bool pointInPlus[NPOINT] = {true}; //false for Check 9
```

## B.7 Check 10 + 11

```
const int NTRAIN = 1;
const int NCB = 4;
const int NPOINT = 1;
const int NSEG = 3;
const int NROUTELENGTH = 2;

...

const int[0, NROUTELENGTH] lockLimit = 1;
const int[0, NROUTELENGTH-1] resLimit = 1;
const segV_id segRoutes[NTRAIN][NROUTELENGTH] = {{2,0}};
const cBV_id boxRoutes[NTRAIN][NROUTELENGTH+1] = {{2,3,1}};
const segV_id cBs[NCB][3] = {{1, -1, -1}, {0, -1, -1}, {2, -1, -1}, {0, 1,
    2}};
const reservation initialRes[NTRAIN] = {{3, 2}};
const pV_id points[NCB] = {-1, -1, -1, 0};
bool pointInPlus[NPOINT] = {true}; //false for Check 11
```

## B.8 Check 12 + 13

```
const int NTRAIN = 2;
const int NCB = 4;
const int NPOINT = 1;
const int NSEG = 3;
const int NROUTELENGTH = 2;

...

const int[0, NROUTELENGTH] lockLimit = 1;
const int[0, NROUTELENGTH-1] resLimit = 1;
const segV_id segRoutes[NTRAIN][NROUTELENGTH] = {{0,2}, {1,0}};
const cBV_id boxRoutes[NTRAIN][NROUTELENGTH+1] = {{1,3,2}, {0,3,1}};
const segV_id cBs[NCB][3] = {{1, -1, -1}, {0, -1, -1}, {2, -1, -1}, {0, 1,
    2}};
const reservation initialRes[NTRAIN] = {{3, 0}, {3, 1}};
const pV_id points[NCB] = {-1, -1, -1, 0};
bool pointInPlus[NPOINT] = {true}; //false for Check 13
```

## B.9 Check 14 + 15

```
const int NTRAIN = 2;
const int NCB = 4;
const int NPOINT = 1;
const int NSEG = 3;
const int NROUTELENGTH = 2;

...

const int[0, NROUTELENGTH] lockLimit = 1;
const int[0, NROUTELENGTH-1] resLimit = 1;
const segV_id segRoutes[NTRAIN][NROUTELENGTH] = {{0,1}, {2,0}};
const cBV_id boxRoutes[NTRAIN][NROUTELENGTH+1] = {{1,3,0}, {2,3,1}};
const segV_id cBs[NCB][3] = {{1, -1, -1}, {0, -1, -1}, {2, -1, -1}, {0, 1,
    2}};
const reservation initialRes[NTRAIN] = {{3, 0}, {3, 2}};
const pV_id points[NCB] = {-1, -1, -1, 0};
bool pointInPlus[NPOINT] = {true}; //false for Check 15
```

## B.10 Check 16 + 17

```
const int NTRAIN = 2;
const int NCB = 4;
const int NPOINT = 1;
const int NSEG = 3;
const int NROUTELENGTH = 2;

...

const int[0, NROUTELENGTH] lockLimit = 1;
const int[0, NROUTELENGTH-1] resLimit = 1;
const segV_id segRoutes[NTRAIN][NROUTELENGTH] = {{2,0}, {1,0}};
const cBV_id boxRoutes[NTRAIN][NROUTELENGTH+1] = {{2,3,1}, {0,3,1}};
const segV_id cBs[NCB][3] = {{1, -1, -1}, {0, -1, -1}, {2, -1, -1}, {0, 1,
    2}};
const reservation initialRes[NTRAIN] = {{3, 2}, {3, 1}};
```

```
const pV_id points[NCB] = {-1, -1, -1, 0};
bool pointInPlus[NPOINT] = {true};  //false for Check 17
```

Appendix C

# Other Models

## C.1 Restricted Model

### C.1.1 Train's Local Declarations

```
segV_id segments[NROUTELENGTH];
cBV_id boxes[NROUTELENGTH+1];

int[0,NSEG] routeLength;
segV_id curSeg;

bool requiresLock[NROUTELENGTH+1];

cBRoute_i lockIndex = 1;
segRoute_i index = 0;

int[0,1] resBit = 0;
cBRoute_i resCBIndex = 1;
cBRoute_i resSegIndex = 0;

segV_id headSeg = -1;
cB_id locks = 0;

void updateLockIndex(){
    while(lockIndex < NROUTELENGTH && !requiresLock[lockIndex]){
        lockIndex++;
    }
}

void initialize() {
```

```
    //Segments
    for(i : segRoute_i) {
        segments[i] = segRoutes[id][i];
        if(segments[i]>-1) {
            routeLength++;
        }
    }
    curSeg = segments[0];

    //Control boxes
    for(i : cBRoute_i) {
        boxes[i] = boxRoutes[id][i];
        if(boxes[i] > -1){
            requiresLock[i] = points[boxes[i]] > -1;
        }
    }

    //Locks and reservations
    resSegIndex = 1;
    updateLockIndex();
}

bool possibleToLock() {
    return lockIndex < routeLength && locks < lockLimit && ((resBit == 0 &&
        resSegIndex > lockIndex) || (resBit == 1 && resSegIndex >= lockIndex
        ));
}

bool hasArrived() {
    return index == routeLength-1;
}

bool possibleToReserve() {
    return resSegIndex < routeLength && resSegIndex - 1 - index < resLimit;
}

bool stillPossibleToReserve() {
    int[0,1] tempBit = resBit^1;
    int[0,NSEG]  tempSegIndex = (tempBit == 0) ? resSegIndex + 1 :
        resSegIndex;
    return tempSegIndex < routeLength && tempSegIndex - 1 -index < resLimit;
}

bool possibleToPass() {
    return resSegIndex > index + 1 && lockIndex > index + 1 && index + 1 <
        routeLength;
}

void updateResInfo(){
    resBit = resBit^1;
    resSegIndex = (resBit==0) ? resSegIndex + 1 : resSegIndex;
    resCBIndex = (resBit==1) ? resCBIndex + 1 : resCBIndex;
}

void updateLocationInfo(){
    curSeg = headSeg;
    headSeg = -1;
    if(requiresLock[index + 1]){
        locks--;
    }
    index++;
}

void updateHeadInfo(){
    headSeg = nextSegment(boxes[index+1], curSeg);
```

```
}

void updateLockInfo(){
    locks++;
    lockIndex++;
    updateLockIndex();
}

bool hasReservations() {
    return resSegIndex > index + 1;
}

bool moreReservations() {
    return resSegIndex > index + 2;
}

bool stillAbleToPass() {
    return resSegIndex > index + 2 && lockIndex > index + 2 && index + 2 <
        routeLength;
}

bool isWellFormed(){
    return segRouteIsWellFormed(segRoutes[id]) &&
           boxRouteIsWellFormed(boxRoutes[id]) &&
           routesAreConsistent(id) &&
           reservationIsWellFormed(initialRes[id]) &&
           initialResIsConsistent(id);
}
```

# C.2 Extended Model

## C.2.1 Train's Local Declarations

```
segV_id segments[NROUTELENGTH];
cBV_id boxes[NROUTELENGTH+1];

int[0,NROUTELENGTH] routeLength;
segV_id curSeg;

bool requiresLock[NROUTELENGTH+1];

cBRoute_i lockIndex = 1;
segRoute_i index = 0;

int[0,1] resBit = 0;
cBRoute_i resCBIndex = 1;
cBRoute_i resSegIndex = 0;

segV_id headSeg = -1;
cB_id locks = 0;

void updateLockIndex(){
    while(lockIndex < NROUTELENGTH && !requiresLock[lockIndex]){
        lockIndex++;
    }
}

void initialize() {
    //Segments
    for(i : segRoute_i) {
```

```
        segments[i] = segRoutes[id][i];
        if(segments[i]>-1) {
            routeLength++;
        }
    }
    curSeg = segments[0];

    //Control boxes
    for(i : cBRoute_i) {
        boxes[i] = boxRoutes[id][i];
        if(boxes[i] > -1){
            requiresLock[i] = points[boxes[i]] > -1;
        }
    }

    //Locks and reservations
    resSegIndex = 1;
    updateLockIndex();
}

bool possibleToLock() {
    return lockIndex < routeLength && locks < lockLimit && ((resBit == 0 &&
        resSegIndex > lockIndex) || (resBit == 1 && resSegIndex >= lockIndex
        ));
}

bool hasArrived() {
    return index == routeLength-1;
}

bool possibleToReserve() {
    return resSegIndex < routeLength && resSegIndex - 1 - index < resLimit;
}

bool possibleToPass() {
    return resSegIndex > index + 1 && lockIndex > index + 1 && index + 1 <
        routeLength;
}

void updateResInfo(){
    resBit = resBit^1;
    resSegIndex = (resBit==0) ? resSegIndex + 1 : resSegIndex;
    resCBIndex = (resBit==1) ? resCBIndex + 1 : resCBIndex;
}

void updateLocationInfo(){
    curSeg = headSeg;
    headSeg = -1;
    if(requiresLock[index + 1]){
        locks--;
    }
    index++;
}

void updateHeadInfo(){
    headSeg = nextSegment(boxes[index+1], curSeg);
}

void updateLockInfo(){
    locks++;
    lockIndex++;
    updateLockIndex();
}

bool possibleToCancel(){
```

```
    return resCBIndex > index + 1;
}

void updateCancel(){
    resSegIndex = (resBit==0) ? resSegIndex - 1 : resSegIndex;
    resCBIndex = (resBit==1) ? resCBIndex - 1 : resCBIndex;
    resBit = resBit^1;

    if(requiresLock[resCBIndex] && lockIndex > resCBIndex){
        locks--;
        lockIndex = resCBIndex;
    }
}

bool isWellFormed(){
    return segRouteIsWellFormed(segRoutes[id]) &&
           boxRouteIsWellFormed(boxRoutes[id]) &&
           routesAreConsistent(id) &&
           reservationIsWellFormed(initialRes[id]) &&
           initialResIsConsistent(id);
}
```

### C.2.2 CB's Local Declarations

```
segV_id segments[3];
segV_id connected =-1;
pV_id point = -1;

tV_id res[3] = {-1, -1, -1};
int[-1,2] result = -1;
tV_id tid = -1;
tV_id lockedBy = -1;

const int ERROR = 0;
const int NOSWITCH = 1;
const int DOSWITCH = 2;

void initialize() {
    segments = cBs[id];
    point = points[id];
    connected = segments[1];

    if(point != -1 && !pointInPlus[point]){
        connected = segments[2];
    }

    for(i : t_id) {
        if (initialRes[i].cb == id) {
            seg_id s = initialRes[i].seg;
            if(s == segments[0]){
                res[0] = i;
            } else if (s == segments[1]){
                res[1] = i;
            } else {
                res[2] = i;
            }
        }
    }
}

int[-1,2] checkSegment(seg_id sid) {
    for(i:int[0,2]) {
        if(segments[i] == sid && res[i] == -1) {
```

```
            return i;
        }
    }
    return -1;
}

int[0,2] checkLock(seg_id s1, seg_id s2){
  if(lockedBy == -1 && (segments[0] == s1 && exists(i:int[1,2]) segments[i]
      == s2) || (segments[0] == s2 && exists(i:int[1,2]) segments[i] == s1))
      {
    if ((s1 == segments[0] && s2 == connected) || (s2 == segments[0] && s1 ==
        connected)){
        return NOSWITCH;
    } else
        return DOSWITCH;
    }
    return ERROR;
}


void clear(){
     lockedBy = -1;

     res[0] = -1;
     if(connected == segments[1]) {
        res[1] = -1;
     } else {
        res[2] = -1;
     }
}

void updateConnected(){
    if(connected == segments[1]){
        connected = segments[2];
    } else {
        connected = segments[1];
    }
}

void resetVariables(){
    tid = -1;
    result = -1;
}

void updateLockInfo(){
    lockedBy = tid;
    resetVariables();
}

void updateResInfo(){
    res[result]=tid;
    resetVariables();
}

void updateCancel(seg_id s){
    for(i:int[0,2]){
        if(segments[i] == s){
            if(res[i] == lockedBy){
                lockedBy = -1;
            }
            res[i] = -1;
        }
    }
}
```

```
bool isWellFormed(){
    return cBIsWellFormed(id) && pointIsWellFormed(id);
}
```

## C.3 Global Model

### C.3.1 Train's Local Declarations

```
int[0,NROUTELENGTH] routeLength;
segV_id curSeg;

bool requiresLock[NROUTELENGTH+1];

cBRoute_i lockIndex = 1;
segRoute_i index = 0;

int[0,1] resBit = 0;
cBRoute_i resCBIndex = 1;
cBRoute_i resSegIndex = 0;

segV_id headSeg = -1;
cB_id locks = 0;

void updateLockIndex(){
    while(lockIndex < NROUTELENGTH && !requiresLock[lockIndex]){
        lockIndex++;
    }
}

void initialize() {
    //Segments
    for(i : segRoute_i) {
        if(segRoutes[id][i]>-1) {
            routeLength++;
        }
    }
    curSeg = segRoutes[id][0];

    //Control boxes
    for(i : cBRoute_i) {
        if(boxRoutes[id][i] > -1){
            requiresLock[i] = points[boxRoutes[id][i]] > -1;
        }
    }

    //Locks and reservations
    resSegIndex = 1;
    updateLockIndex();
}

bool possibleToLock() {
    return lockIndex < routeLength && locks < lockLimit && ((resBit == 0 &&
        resSegIndex > lockIndex) || (resBit == 1 && resSegIndex >= lockIndex
        ));
}

bool hasArrived() {
    return index == routeLength-1;
}
```

```
bool possibleToReserve() {
    return resSegIndex < routeLength && resSegIndex - 1 - index < resLimit;
}

bool possibleToPass() {
    return resSegIndex > index + 1 && lockIndex > index + 1 && index + 1 <
        routeLength;
}

void updateResInfo(){
    resBit = resBit^1;
    resSegIndex = (resBit==0) ? resSegIndex + 1 : resSegIndex;
    resCBIndex = (resBit==1) ? resCBIndex + 1 : resCBIndex;
}

void updateLocationInfo(){
    curSeg = headSeg;
    headSeg = -1;
    if(requiresLock[index + 1]){
        locks--;
    }
    index++;
}

void updateHeadInfo(){
    headSeg = nextSegment(boxRoutes[id][index+1], curSeg);
}

void updateLockInfo(){
    locks++;
    lockIndex++;
    updateLockIndex();
}

bool isWellFormed(){
    return segRouteIsWellFormed(segRoutes[id]) &&
           boxRouteIsWellFormed(boxRoutes[id]) &&
           routesAreConsistent(id) &&
           reservationIsWellFormed(initialRes[id]) &&
           initialResIsConsistent(id);
}
```

### C.3.2 CB's Local Declarations

```
segV_id connected =-1;
pV_id point = -1;

tV_id res[3] = {-1, -1, -1};
int[-1,2] result = -1;
tV_id tid = -1;
tV_id lockedBy = -1;

const int ERROR = 0;
const int NOSWITCH = 1;
const int DOSWITCH = 2;

void initialize() {
    point = points[id];
    connected = cBs[id][1];

    if(point != -1 && !pointInPlus[point]){
        connected = cBs[id][2];
    }
```

```
    for(i : t_id) {
        if (initialRes[i].cb == id) {
            seg_id s = initialRes[i].seg;
            if(s == cBs[id][0]){
                res[0] = i;
            } else if (s == cBs[id][1]){
                res[1] = i;
            } else {
                res[2] = i;
            }
        }
    }
}

int[-1,2] checkSegment(seg_id sid) {
    for(i:int[0,2]) {
        if(cBs[id][i] == sid && res[i] == -1) {
            return i;
        }
    }
    return -1;
}

int[0,2] checkLock(seg_id s1, seg_id s2){
  if(lockedBy == -1 && (cBs[id][0] == s1 && exists(i:int[1,2]) cBs[id][i] ==
       s2) || (cBs[id][0] == s2 && exists(i:int[1,2]) cBs[id][i] == s1)){
    if ((s1 == cBs[id][0] && s2 == connected) || (s2 == cBs[id][0] && s1 ==
         connected)){
        return NOSWITCH;
    } else
        return DOSWITCH;
    }
    return ERROR;
}


void clear(){
     lockedBy = -1;

     res[0] = -1;
     if(connected == cBs[id][1]) {
        res[1] = -1;
     } else {
        res[2] = -1;
     }
}

void updateConnected(){
    if(connected == cBs[id][1]){
        connected = cBs[id][2];
    } else {
        connected = cBs[id][1];
    }
}

void resetVariables(){
    tid = -1;
    result = -1;
}

void updateLockInfo(){
    lockedBy = tid;
    resetVariables();
}
```

```
void updateResInfo(){
    res[result]=tid;
    resetVariables();
}

bool isWellFormed(){
    return cBIsWellFormed(id) && pointIsWellFormed(id);
}
```

Appendix D

# UMC Model

## D.1 The Train Class

```
Class Train is
Signals
    OK, notOK
Vars
    segments;
    boxes;

    curSeg:int;

    requiresLock;

    lockIndex:int = 1;
    index:int = 0;

    resBit:int = 0;
    resCBIndex:int = 1;
    resSegIndex:int = 1;

    headSeg:int = -1;
    locks:int = 0;

    resLimit = ...; //Maximum number of full segment reservations allowed at
        a time per train
    lockLimit = ...; //Maximum number of locks allowed at a time per train
Transitions
    SingleSegment -> Arrived {[index == segments.length-1]}
    Arrived -> Arrived

    SingleSegment -> DoubleSegment {
```

```
    [resSegIndex > index + 1 && lockIndex > index + 1 && index + 1 <
        segments.length] /

    //updateHeadInfo
    boxes[index+1].pass;
    headSeg = segments[index+1];
}

DoubleSegment -> SingleSegment {/
    //updateLocationInfo
    curSeg = headSeg;
    headSeg = -1;
    if(requiresLock[index +1]){
        locks--;
    };
    index++;

    boxes[index].passed;
}

SingleSegment -> Reserving {
    - [resSegIndex < segments.length && resSegIndex - 1 - index <
        resLimit] /

    boxes[resCBIndex].reqSeg(this, segments[resSegIndex])
}

Reserving -> SingleSegment {
    OK /

    //updateResInfo
    if (resBit == 0) {
        resBit = 1;
        resCBIndex++;
    } else {
        resBit = 0;
        resSegIndex++;
    }
}

Reserving -> SingleSegment {notOK}

SingleSegment -> Locking {
    [lockIndex < segments.length && locks < lockLimit && ((resBit == 0 &&
         resSegIndex > lockIndex) || (resBit == 1 && resSegIndex >=
        lockIndex))] /

    boxes[lockIndex].reqLock(this, segments[lockIndex-1],segments[
        lockIndex])
}

Locking -> SingleSegment {
    OK /

    //updateLockInfo
    locks++;
    lockIndex++;
    //updateLockIndex
    for i in lockIndex..segments.length {
        if(!requiresLock[i]){
            lockIndex++;
        } else {
            i = segments.length;
        }
    }
```

```
    }
    Locking -> SingleSegment {notOK}
end Train;
```

## D.2 The CB Class

```
Class CB is
Signals
    reqSeg(it:Train, s:int),
    reqLock(it:Train, s1:int, s2:int),
    pass,
    passed,
    OKp
Vars
    segments;
    connected;
    point:Point;

    res:Train[3];
    result:int := -1;
    t:Train := null;
    lockedBy:Train;

    ERROR:int := 0;
    NOSWITCH:int := 1;
    DOSWITCH:int := 2;
Transitions
    Idle -> SegmentChecked {
        reqSeg(it,s) /

        //checkSegment
        for i in 0..2 {
            if (segments[i] == s && res[i] == null) then {
                result = i;
            }
        };
        t = it;
    }

    SegmentChecked -> Idle {
        - [result > -1] /

        //updateResInfo
        res[result] = t;
        //resetVariables
        result = -1;

        t.OK;
        t = null;
    }

    SegmentChecked -> Idle {
        - [result == -1] /

        t.notOK;

        //resetVariables
        t = null;
        result = -1;
    }

    Idle -> LockChecked {
```

```
        reqLock(it,s1,s2) /

        //checkLock
        result = ERROR;
        if(lockedBy == null) then {
            if ((segments[0] == s1 && (segments[1] == s2 || segments[2] == s2
                 )) ||
                (segments[0] == s2 && (segments[1] == s1 || segments[2] == s1
                     ))) then {
                if((s1 == segments[0] && s2 == connected) || (s2 == segments
                     [0] && s1 == connected)) then {
                    result = NOSWITCH;
                } else {
                    result = DOSWITCH;
                }
            }
        };
        t = it;
    }

    LockChecked -> Switching {
        - [result == DOSWITCH] /

        point.switchPoint;
    }

    LockChecked -> Idle {
        - [result == ERROR] /

        t.notOK;
        //resetVariables
        t = null;
        result = -1
    }

    LockChecked -> Switched {- [result == NOSWITCH]}

    Switching -> Switched {
        OKp /
        //updateConnected
        if(connected == segments[1]) {
            connected = segments[2];
        } else {
            connected = segments[1];
        }
    }

    Switched -> Idle{
        /
        //updateLockInfo
        lockedBy = t;
        //resetVariables
        result = -1;

        t.OK;

        //resetVariables
        t = null;
    }

    Idle -> Passing {pass}
    Passing -> Idle {
        passed /

        //clear
```

```
        lockedBy = null;
        res[0] = null;
        if(connected == segments[1]){
            res[1] = null;
        } else {
            res[2] = null;
        }
    }
end CB;
```

## D.3 The Point Class

```
class Point is
Signals
    switchPoint
Vars
    cb:CB;
    inPlus:bool
Transitions
    Still -> Switching {switchPoint}
    Switching -> Still { - /
        inPlus = !inPlus;
        cb.OKp;}
end Point;
```

Appendix E

# Verification Result Tables

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.003 | 0.002 | 0.002 | 0.002 |
| Example with Two Trains | 0.020 | 0.018 | 0.013 | 0.017 |
| Shared Segment | 0.017 | 0.015 | 0.013 | 0.015 |
| Large Network | 0.011 | 0.014 | 0.017 | 0.014 |
| Station One | 0.013 | 0.009 | 0.012 | 0.011 |
| Station Two | 0.074 | 0.076 | 0.075 | 0.075 |
| Station Four | 1.690 | 1.715 | 1.755 | 1.720 |
| Station Six | 27.474 | 27.433 | 27.203 | 27.370 |
| Station Eight | 165.631 | 165.800 | 165.955 | 165.795 |
| Station Ten | 601.131 | 602.165 | 600.341 | 601.212 |
| Station Twelve | 1749.956 | 1751.553 | 1724.338 | 1741.949 |
| Two Trains Same Direction | 0.014 | 0.019 | 0.016 | 0.016 |
| Four Trains Same Direction | 1146.256 | 1130.966 | 1133.690 | 1136.971 |
| Nærumbanen (2T) | 24.112 | 24.158 | 24.186 | 24.152 |
| Nærumbanen (3T) | 648.154 | 655.805 | 660.029 | 654.663 |

**Table E.1:** First model: 'No derailment' 1, limits 1, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with one train | 5988 | 5996 | 6004 | 5996 |
| Example with two trains | 6512 | 6528 | 6538 | 6526 |
| Shared Segment | 7464 | 7472 | 7516 | 7484 |
| Large Network | 8928 | 8960 | 8972 | 8953 |
| Station One | 6472 | 6492 | 6500 | 6488 |
| Station Two | 9252 | 9288 | 9296 | 9279 |
| Station Four | 29580 | 29672 | 29688 | 29647 |
| Station Six | 69912 | 70192 | 70200 | 70101 |
| Station Eight | 205372 | 205396 | 205404 | 205391 |
| Station Ten | 313016 | 313220 | 313748 | 313328 |
| Station Twelfth | 560580 | 724492 | 723488 | 669520 |
| Two trains same direction | 7488 | 7496 | 7540 | 7508 |
| Four trains same direction | 726732 | 749296 | 749460 | 741829 |
| Nærumbanen (2T) | 78428 | 78824 | 78828 | 78693 |
| Nærumbanen (3T) | 384220 | 385180 | 385192 | 384864 |

**Table E.2:** First model: 'No derailment' 1, limits 1, Memory Usage Peak in KB

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.001 | 0.003 | 0.004 | 0.003 |
| Example with Two Trains | 0.015 | 0.021 | 0.021 | 0.019 |
| Shared Segment | 0.009 | 0.014 | 0.02 | 0.014 |
| Large Network | 0.017 | 0.017 | 0.015 | 0.016 |
| Station One | 0.012 | 0.011 | 0.012 | 0.012 |
| Station Two | 0.068 | 0.077 | 0.075 | 0.073 |
| Station Four | 1.679 | 1.723 | 1.721 | 1.708 |
| Station Six | 29.505 | 29.477 | 29.470 | 29.484 |
| Station Eight | 166.133 | 164.812 | 164.793 | 165.246 |
| Station Ten | 599.734 | 601.485 | 600.747 | 600.655 |
| Station Twelve | 1744.904 | 1745.973 | 1739.476 | 1743.451 |
| Two Trains Same Direction | 0.022 | 0.015 | 0.019 | 0.019 |
| Four Trains Same Direction | 1121.890 | 1139.341 | 1130.754 | 1130.662 |
| Nærumbanen (2T) | 24.004 | 24.091 | 24.111 | 24.069 |
| Nærumbanen (3T) | 664.594 | 660.006 | 657.312 | 660.637 |

**Table E.3:** First model: 'No derailment' 2, limits 1, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with one train | 5984 | 5996 | 5996 | 5992 |
| Example with two trains | 6500 | 6512 | 6516 | 6509 |
| Shared Segment | 7460 | 7464 | 7468 | 7464 |
| Large Network | 8920 | 8944 | 8948 | 8937 |
| Station One | 6464 | 6480 | 6480 | 6475 |
| Station Two | 9252 | 9276 | 9280 | 9269 |
| Station Four | 25772 | 25848 | 25860 | 25827 |
| Station Six | 69912 | 70168 | 70180 | 70087 |
| Station Eight | 158408 | 158508 | 158508 | 158475 |
| Station Ten | 312992 | 313008 | 313132 | 313044 |
| Station Twelfth | 559520 | 559532 | 559600 | 559551 |
| Two trains same direction | 7484 | 7488 | 7492 | 7488 |
| Four trains same direction | 726708 | 726852 | 726864 | 726808 |
| Nærumbanen (2T) | 78428 | 78800 | 78808 | 78679 |
| Nærumbanen (3T) | 382556 | 383184 | 383544 | 383095 |

**Table E.4:** First model: 'No derailment' 2, limits 1, Memory Usage Peak in KB

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.002 | 0.003 | 0.004 | 0.003 |
| Example with Two Trains | 0.022 | 0.024 | 0.015 | 0.020 |
| Shared Segment | 0.014 | 0.014 | 0.014 | 0.014 |
| Large Network | 0.011 | 0.015 | 0.021 | 0.016 |
| Station One | 0.007 | 0.01 | 0.008 | 0.008 |
| Station Two | 0.082 | 0.093 | 0.081 | 0.085 |
| Station Four | 1.702 | 1.911 | 1.738 | 1.784 |
| Station Six | 29.678 | 29.678 | 29.803 | 29.720 |
| Station Eight | 164.477 | 165.993 | 165.677 | 165.382 |
| Station Ten | 599.563 | 599.868 | 597.934 | 599.122 |
| Station Twelve | 1741.865 | 1741.334 | 1747.987 | 1743.729 |
| Two Trains Same Direction | 0.016 | 0.019 | 0.019 | 0.018 |
| Four Trains Same Direction | 1142.559 | 1141.663 | 1139.661 | 1141.294 |
| Nærumbanen (2T) | 24.081 | 24.345 | 24.339 | 24.255 |
| Nærumbanen (3T) | 671.008 | 653.381 | 663.635 | 662.675 |

**Table E.5:** First Model: 'No Collision', limits 1, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with one train | 6000 | 6016 | 6020 | 6012 |
| Example with two trains | 6528 | 6552 | 6572 | 6551 |
| Shared Segment | 7468 | 7520 | 7532 | 7507 |
| Large Network | 8932 | 8972 | 8988 | 8964 |
| Station One | 6484 | 6524 | 6536 | 6515 |
| Station Two | 9268 | 9308 | 9324 | 9300 |
| Station Four | 25792 | 25880 | 25896 | 25856 |
| Station Six | 69916 | 70200 | 70220 | 70112 |
| Station Eight | 158424 | 158440 | 158628 | 158497 |
| Station Ten | 312972 | 312984 | 313116 | 313024 |
| Station Twelfth | 723464 | 1051300 | 887412 | 887392 |
| Two trains same direction | 7496 | 7548 | 7560 | 7535 |
| Four trains same direction | 726716 | 726880 | 726904 | 726833 |
| Nærumbanen (2T) | 78436 | 78828 | 78888 | 78717 |
| Nærumbanen (3T) | 382548 | 383224 | 383544 | 383105 |

**Table E.6:** First model: 'No Collision', limits 1, Memory Usage Peak in KB

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.002 | 0.003 | 0.003 | 0.003 |
| Example with Two Trains | 0.023 | 0.018 | 0.013 | 0.018 |
| Shared Segment | 0.010 | 0.010 | 0.010 | 0.010 |
| Large Network | 0.013 | 0.020 | 0.020 | 0.018 |
| Station One | 0.012 | 0.008 | 0.010 | 0.010 |
| Station Two | 0.082 | 0.067 | 0.081 | 0.077 |
| Station Four | 1.679 | 1.740 | 1.715 | 1.711 |
| Station Six | 27.342 | 27.453 | 27.331 | 27.375 |
| Station Eight | 163.763 | 163.562 | 162.250 | 163.192 |
| Station Ten | 590.130 | 587.481 | 588.739 | 588.783 |
| Station Twelve | 1731.068 | 1730.117 | 1726.287 | 1729.157 |
| Two Trains Same Direction | 0.018 | 0.017 | 0.018 | 0.018 |
| Four Trains Same Direction | 1082.917 | 1090.144 | 1096.833 | 1089.965 |
| Nærumbanen (2T) | 23.900 | 23.947 | 23.908 | 23.918 |
| Nærumbanen (3T) | 662.400 | 662.405 | 669.196 | 664.667 |

**Table E.7:** First model: 'Will arrive', limits 1, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with one train | 5956 | 5960 | 5960 | 5959 |
| Example with two trains | 6468 | 6472 | 6472 | 6471 |
| Shared Segment | 7436 | 7436 | 7436 | 7436 |
| Large Network | 8888 | 8904 | 8908 | 8900 |
| Station One | 6436 | 6440 | 6440 | 6439 |
| Station Two | 9216 | 9232 | 9236 | 9228 |
| Station Four | 25736 | 25804 | 25804 | 25781 |
| Station Six | 69812 | 70064 | 70068 | 69981 |
| Station Eight | 158152 | 158152 | 158236 | 158180 |
| Station Ten | 312360 | 312528 | 312528 | 312472 |
| Station Twelfth | 557696 | 557984 | 557984 | 557888 |
| Two trains same direction | 7460 | 7460 | 7460 | 7460 |
| Four trains same direction | 726668 | 726800 | 726808 | 726759 |
| Nærumbanen (2T) | 78392 | 78704 | 78704 | 78600 |
| Nærumbanen (3T) | 382464 | 383120 | 383192 | 382925 |

**Table E.8:** First model: 'Will arrive', limits 1, Memory Usage Peak in KB

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.004 | 0.004 | 0.004 | 0.004 |
| Example with Two Trains | 0.025 | 0.027 | 0.028 | 0.027 |
| Shared Segment | 0.018 | 0.02 | 0.015 | 0.018 |
| Large Network | 0.017 | 0.028 | 0.023 | 0.023 |
| Station One | 0.017 | 0.016 | 0.015 | 0.016 |
| Station Two | 0.18 | 0.187 | 0.19 | 0.186 |
| Station Four | 8.239 | 8.075 | 8.076 | 8.130 |
| Station Six | 176.213 | 175.867 | 176.697 | 176.259 |
| Station Eight | 1127.661 | 1124.994 | 1128.751 | 1127.135 |
| Station Ten | 4311.035 | 4309.434 | 4313.687 | 4311.385 |
| Station Twelve | | | | |
| Two Trains Same Direction | 0.028 | 0.034 | 0.023 | 0.028 |
| Four Trains Same Direction | | | | |
| Nærumbanen (2T) | 114.581 | 114.372 | 114.491 | 114.481 |
| Nærumbanen (3T) | 4375.625 | 4437.235 | 4543.133 | 4451.998 |

**Table E.9:** First model: 'No derailment' 1, limits 2, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with one train | 5992 | 5996 | 5996 | 5995 |
| Example with two trains | 6536 | 6536 | 6536 | 6536 |
| Shared Segment | 7484 | 7488 | 7536 | 7503 |
| Large Network | 8948 | 8980 | 8984 | 8971 |
| Station One | 6480 | 6500 | 6508 | 6496 |
| Station Two | 9908 | 9940 | 9944 | 9931 |
| Station Four | 41412 | 41540 | 41556 | 41503 |
| Station Six | 152812 | 152908 | 152908 | 152876 |
| Station Eight | 422192 | 422384 | 422392 | 422323 |
| Station Ten | 954604 | 954948 | 954960 | 954837 |
| Station Twelfth | | | | |
| Two trains same direction | 7540 | 7544 | 7592 | 7559 |
| Four trains same direction | | | | |
| Nærumbanen (2T) | 122060 | 122396 | 122420 | 122292 |
| Nærumbanen (3T) | 1834760 | 1834804 | 1834812 | 1834792 |

**Table E.10:** First model: 'No derailment' 1, limits 2, Memory Usage Peak in KB

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.004 | 0.004 | 0.004 | 0.004 |
| Example with Two Trains | 0.015 | 0.025 | 0.018 | 0.019 |
| Shared Segment | 0.023 | 0.021 | 0.019 | 0.021 |
| Large Network | 0.026 | 0.023 | 0.023 | 0.024 |
| Station One | 0.01 | 0.012 | 0.017 | 0.013 |
| Station Two | 0.18 | 0.203 | 0.211 | 0.198 |
| Station Four | 8.055 | 8.083 | 8.074 | 8.071 |
| Station Six | 172.488 | 172.487 | 173.026 | 172.667 |
| Station Eight | 1138.254 | 1136.482 | 1139.334 | 1138.023 |
| Station Ten | 4346.405 | 4327.87 | 4331.854 | 4335.376 |
| Station Twelve | | | | |
| Two Trains Same Direction | 0.02 | 0.028 | 0.028 | 0.025 |
| Four Trains Same Direction | | | | |
| Nærumbanen (2T) | 113.146 | 112.966 | 112.577 | 112.896 |
| Nærumbanen (3T) | 4616.238 | 4564.949 | 4581.939 | 4587.709 |

**Table E.11:** First model: 'No derailment' 2, limits 2, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with one train | 5988 | 5988 | 5988 | 5988 |
| Example with two trains | 6528 | 6528 | 6532 | 6529 |
| Shared Segment | 7476 | 7480 | 7484 | 7480 |
| Large Network | 8952 | 8984 | 8996 | 8977 |
| Station One | 6468 | 6484 | 6484 | 6479 |
| Station Two | 9892 | 9916 | 9920 | 9909 |
| Station Four | 41404 | 41520 | 41528 | 41484 |
| Station Six | 152800 | 152876 | 152876 | 152851 |
| Station Eight | 422180 | 422364 | 422364 | 422303 |
| Station Ten | 1050988 | 1243744 | 1244084 | 1179605 |
| Station Twelfth | | | | |
| Two trains same direction | 7524 | 7528 | 7532 | 7528 |
| Four trains same direction | | | | |
| Nærumbanen (2T) | 122032 | 122124 | 122368 | 122175 |
| Nærumbanen (3T) | 1833100 | 1833140 | 1833140 | 1833127 |

**Table E.12:** First model: 'No derailment' 2, limits 2, Memory Usage Peak

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.004 | 0.004 | 0.004 | 0.004 |
| Example with Two Trains | 0.018 | 0.018 | 0.024 | 0.020 |
| Shared Segment | 0.015 | 0.02 | 0.023 | 0.019 |
| Large Network | 0.023 | 0.032 | 0.027 | 0.027 |
| Station One | 0.011 | 0.017 | 0.012 | 0.013 |
| Station Two | 0.183 | 0.215 | 0.187 | 0.195 |
| Station Four | 8.090 | 8.115 | 8.241 | 8.149 |
| Station Six | 176.156 | 175.635 | 175.459 | 175.750 |
| Station Eight | 1144.312 | 1146.043 | 1144.102 | 1144.819 |
| Station Ten | 4327.887 | 4350.655 | 4352.747 | 4343.763 |
| Station Twelve | | | | |
| Two Trains Same Direction | 0.024 | 0.029 | 0.022 | 0.025 |
| Four Trains Same Direction | | | | |
| Nærumbanen (2T) | 111.647 | 111.867 | 111.601 | 111.705 |
| Nærumbanen (3T) | 4707.002 | 4711.929 | 4683.026 | 4700.652 |

**Table E.13:** First model: 'No Collision', limits 2, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with one train | 5996 | 5992 | 5996 | 5995 |
| Example with two trains | 6548 | 6552 | 6576 | 6559 |
| Shared Segment | 7480 | 7532 | 7554 | 7522 |
| Large Network | 8960 | 9000 | 9016 | 8992 |
| Station One | 6492 | 6520 | 6532 | 6515 |
| Station Two | 9912 | 9952 | 9968 | 9944 |
| Station Four | 41416 | 41548 | 41560 | 41508 |
| Station Six | 152804 | 152904 | 152916 | 152875 |
| Station Eight | 422196 | 422388 | 422396 | 422327 |
| Station Ten | 955772 | 954604 | 954952 | 955109 |
| Station Twelfth | | | | |
| Two trains same direction | 7536 | 7588 | 7600 | 7575 |
| Four trains same direction | | | | |
| Nærumbanen (2T) | 122040 | 122144 | 122404 | 122196 |
| Nærumbanen (3T) | 1833096 | 1833144 | 1833148 | 1833129 |

**Table E.14:** First model: 'No Collision', limits 2, Memory Usage Peak in KB

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.003 | 0.004 | 0.003 | 0.003 |
| Example with Two Trains | 0.026 | 0.025 | 0.026 | 0.026 |
| Shared Segment | 0.021 | 0.022 | 0.017 | 0.020 |
| Large Network | 0.02 | 0.027 | 0.021 | 0.023 |
| Station One | 0.017 | 0.012 | 0.016 | 0.015 |
| Station Two | 0.191 | 0.196 | 0.203 | 0.197 |
| Station Four | 8.081 | 8.118 | 8.129 | 8.109 |
| Station Six | 178.496 | 176.697 | 177.664 | 177.619 |
| Station Eight | 1134.646 | 1132.922 | 1133.608 | 1133.725 |
| Station Ten | 4318.385 | 4306.554 | 4316.069 | 4313.669 |
| Station Twelve | | | | |
| Two Trains Same Direction | 0.021 | 0.025 | 0.031 | 0.026 |
| Four Trains Same Direction | | | | |
| Nærumbanen (2T) | 113.356 | 113.335 | 113.301 | 113.331 |
| Nærumbanen (3T) | 4768.012 | 4734.325 | 4725.702 | 4742.680 |

**Table E.15:** First model: 'Will arrive', limits 2, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with one train | 5960 | 5960 | 5956 | 5959 |
| Example with two trains | 6500 | 6504 | 6504 | 6503 |
| Shared Segment | 7452 | 7452 | 7452 | 7452 |
| Large Network | 8912 | 8928 | 8932 | 8924 |
| Station One | 6436 | 6440 | 6440 | 6439 |
| Station Two | 9860 | 9880 | 9880 | 9873 |
| Station Four | 41372 | 41456 | 41460 | 41429 |
| Station Six | 152688 | 152764 | 152764 | 152739 |
| Station Eight | 421808 | 421980 | 421980 | 421923 |
| Station Ten | 953328 | 953672 | 953672 | 953557 |
| Station Twelfth | | | | |
| Two trains same direction | 7504 | 7504 | 7504 | 7504 |
| Four trains same direction | | | | |
| Nærumbanen (2T) | 122016 | 122104 | 122116 | 122079 |
| Nærumbanen (3T) | 1833016 | 1833052 | 1833052 | 1833040 |

**Table E.16:** First model: 'Will arrive', limits 2, Memory Usage Peak in KB

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.004 | 0.003 | 0.004 | 0.004 |
| Example with Two Trains | 0.014 | 0.016 | 0.013 | 0.014 |
| Shared Segment | 0.015 | 0.018 | 0.018 | 0.017 |
| Large Network | 0.011 | 0.026 | 0.025 | 0.021 |
| Station One | 0.010 | 0.010 | 0.021 | 0.014 |
| Station Two | 0.075 | 0.097 | 0.079 | 0.084 |
| Station Four | 1.494 | 1.396 | 1.355 | 1.415 |
| Station Six | 22.872 | 23.017 | 22.706 | 22.865 |
| Station Eight | 135.759 | 133.936 | 134.133 | 134.609 |
| Station Ten | 475.160 | 477.251 | 474.018 | 475.476 |
| Station Twelve | 1357.894 | 1340.195 | 1289.723 | 1329.271 |
| Two Trains Same Direction | 0.022 | 0.024 | 0.026 | 0.024 |
| Four Trains Same Direction | 835.163 | 830.236 | 833.745 | 833.048 |
| Nærumbanen (2T) | 20.145 | 19.258 | 19.238 | 19.547 |
| Nærumbanen (3T) | 317.762 | 318.903 | 324.661 | 320.442 |

**Table E.17:** Restricted variant: 'No derailment' 1, limits 1, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with one train | 7640 | 7652 | 7656 | 7649 |
| Example with two trains | 8576 | 8584 | 8588 | 8583 |
| Shared Segment | 9816 | 9848 | 9852 | 9839 |
| Large Network | 12024 | 12072 | 12084 | 12060 |
| Station One | 8544 | 8552 | 8560 | 8552 |
| Station Two | 14256 | 14308 | 14320 | 14295 |
| Station Four | 37932 | 38164 | 38172 | 38089 |
| Station Six | 105136 | 105888 | 105888 | 105637 |
| Station Eight | 221068 | 222868 | 222868 | 222268 |
| Station Ten | 424840 | 428096 | 428268 | 427068 |
| Station Twelfth | 725592 | 732964 | 732964 | 730507 |
| Two trains same direction | 11664 | 11700 | 11704 | 11689 |
| Four trains same direction | 686480 | 686620 | 686636 | 686579 |
| Nærumbanen (2T) | 117620 | 118440 | 118644 | 118235 |
| Nærumbanen (3T) | 289164 | 290112 | 290112 | 289796 |

**Table E.18:** Restricted variant: 'No derailment' 1, limits 1, Memory Usage Peak in KB

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.003 | 0.004 | 0.004 | 0.004 |
| Example with Two Trains | 0.027 | 0.030 | 0.029 | 0.029 |
| Shared Segment | 0.014 | 0.017 | 0.018 | 0.016 |
| Large Network | 0.012 | 0.022 | 0.019 | 0.018 |
| Station One | 0.019 | 0.022 | 0.011 | 0.017 |
| Station Two | 0.067 | 0.077 | 0.078 | 0.074 |
| Station Four | 1.264 | 1.385 | 1.299 | 1.316 |
| Station Six | 21.984 | 22.251 | 22.444 | 22.226 |
| Station Eight | 135.149 | 134.701 | 136.291 | 135.380 |
| Station Ten | 471.003 | 472.720 | 472.425 | 472.049 |
| Station Twelve | 1356.472 | 1356.409 | 1308.059 | 1340.313 |
| Two Trains Same Direction | 0.024 | 0.022 | 0.026 | 0.024 |
| Four Trains Same Direction | 819.379 | 779.918 | 786.767 | 795.355 |
| Nærumbanen (2T) | 19.946 | 20.036 | 20.275 | 20.086 |
| Nærumbanen (3T) | 332.490 | 333.240 | 331.178 | 332.303 |

**Table E.19:** Restricted variant: 'No derailment' 2, limits 1, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with one train | 7608 | 7616 | 7628 | 7617 |
| Example with two trains | 8560 | 8568 | 8572 | 8567 |
| Shared Segment | 9792 | 9820 | 9832 | 9815 |
| Large Network | 12004 | 12052 | 12064 | 12040 |
| Station One | 8512 | 8520 | 8524 | 8519 |
| Station Two | 12408 | 12456 | 12464 | 12443 |
| Station Four | 36500 | 36728 | 36752 | 36660 |
| Station Six | 105660 | 106012 | 106048 | 105907 |
| Station Eight | 219100 | 220624 | 220980 | 220235 |
| Station Ten | 420864 | 424288 | 424460 | 423204 |
| Station Twelfth | 724864 | 727128 | 732708 | 728233 |
| Two trains same direction | 9936 | 9964 | 9972 | 9957 |
| Four trains same direction | 687808 | 687812 | 687784 | 687801 |
| Nærumbanen (2T) | 116124 | 116940 | 117144 | 116736 |
| Nærumbanen (3T) | 286900 | 287556 | 287852 | 287436 |

**Table E.20:** Restricted variant: 'No derailment' 2, limits 1, Memory Usage Peak in KB

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.003 | 0.004 | 0.003 | 0.003 |
| Example with Two Trains | 0.026 | 0.017 | 0.030 | 0.024 |
| Shared Segment | 0.014 | 0.016 | 0.017 | 0.016 |
| Large Network | 0.012 | 0.019 | 0.019 | 0.017 |
| Station One | 0.012 | 0.012 | 0.019 | 0.014 |
| Station Two | 0.065 | 0.111 | 0.114 | 0.097 |
| Station Four | 1.286 | 1.315 | 1.266 | 1.289 |
| Station Six | 23.164 | 23.206 | 23.325 | 23.232 |
| Station Eight | 127.860 | 124.623 | 124.672 | 125.718 |
| Station Ten | 474.403 | 459.789 | 444.659 | 459.617 |
| Station Twelve | 1337.102 | 1326.536 | 1320.822 | 1328.153 |
| Two Trains Same Direction | 0.024 | 0.020 | 0.025 | 0.023 |
| Four Trains Same Direction | 828.480 | 839.239 | 839.249 | 835.656 |
| Nærumbanen (2T) | 19.309 | 18.324 | 18.342 | 18.658 |
| Nærumbanen (3T) | 323.919 | 321.851 | 320.536 | 322.102 |

**Table E.21:** Restricted variant: 'No Collision', limits 1, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with one train | 7592 | 7608 | 7612 | 7604 |
| Example with two trains | 8544 | 8552 | 8560 | 8552 |
| Shared Segment | 9772 | 9808 | 9820 | 9800 |
| Large Network | 11980 | 12044 | 12506 | 12177 |
| Station One | 8516 | 8524 | 8532 | 8524 |
| Station Two | 12384 | 12436 | 12444 | 12421 |
| Station Four | 37948 | 38172 | 38184 | 38101 |
| Station Six | 101400 | 102152 | 102160 | 101904 |
| Station Eight | 223784 | 224888 | 224888 | 224520 |
| Station Ten | 421316 | 426156 | 426156 | 424543 |
| Station Twelfth | 725644 | 727908 | 732620 | 728724 |
| Two trains same direction | 9916 | 9948 | 9960 | 9941 |
| Four trains same direction | 688684 | 688700 | 688708 | 688697 |
| Nærumbanen (2T) | 118312 | 119176 | 119984 | 119157 |
| Nærumbanen (3T) | 287632 | 288220 | 288584 | 288145 |

**Table E.22:** Restricted variant: 'No Collision', limits 1, Memory Usage Peak in KB

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.002 | 0.003 | 0.003 | 0.003 |
| Example with Two Trains | 0.027 | 0.028 | 0.028 | 0.028 |
| Shared Segment | 0.012 | 0.014 | 0.014 | 0.013 |
| Large Network | 0.011 | 0.012 | 0.020 | 0.014 |
| Station One | 0.019 | 0.018 | 0.021 | 0.019 |
| Station Two | 0.077 | 0.064 | 0.076 | 0.072 |
| Station Four | 1.253 | 1.375 | 1.291 | 1.306 |
| Station Six | 21.982 | 22.389 | 22.220 | 22.197 |
| Station Eight | 134.299 | 134.362 | 135.154 | 134.605 |
| Station Ten | 470.486 | 452.370 | 444.189 | 455.682 |
| Station Twelve | 1279.219 | 1260.140 | 1274.903 | 1271.421 |
| Two Trains Same Direction | 0.021 | 0.018 | 0.014 | 0.018 |
| Four Trains Same Direction | 805.680 | 789.951 | 792.274 | 795.968 |
| Nærumbanen (2T) | 20.903 | 20.225 | 20.123 | 20.417 |
| Nærumbanen (3T) | 334.085 | 331.171 | 336.187 | 333.814 |

**Table E.23:** Restricted variant: 'Will arrive', limits 1, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with one train | 7520 | 7524 | 7524 | 7523 |
| Example with two trains | 8480 | 8480 | 8500 | 8487 |
| Shared Segment | 9700 | 9720 | 9724 | 9715 |
| Large Network | 11920 | 11920 | 11928 | 11923 |
| Station One | 8440 | 8440 | 8460 | 8447 |
| Station Two | 12312 | 12356 | 12364 | 12344 |
| Station Four | 37080 | 37260 | 37284 | 37208 |
| Station Six | 100220 | 100576 | 100608 | 100468 |
| Station Eight | 220080 | 221836 | 221916 | 221277 |
| Station Ten | 422496 | 427152 | 427152 | 425600 |
| Station Twelfth | 728620 | 731976 | 731903 | 730833 |
| Two trains same direction | 9844 | 9864 | 9872 | 9860 |
| Four trains same direction | 687712 | 687712 | 687716 | 687713 |
| Nærumbanen (2T) | 114784 | 115532 | 115716 | 115344 |
| Nærumbanen (3T) | 285640 | 286228 | 286300 | 286056 |

**Table E.24:** Restricted variant: 'Will arrive', limits 1, Memory Usage Peak in KB

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.004 | 0.004 | 0.004 | 0.004 |
| Example with Two Trains | 0.033 | 0.032 | 0.037 | 0.034 |
| Shared Segment | 0.015 | 0.015 | 0.015 | 0.015 |
| Large Network | 0.015 | 0.017 | 0.014 | 0.015 |
| Station One | 0.010 | 0.012 | 0.023 | 0.015 |
| Station Two | 0.119 | 0.118 | 0.139 | 0.125 |
| Station Four | 2.570 | 2.138 | 2.411 | 2.373 |
| Station Six | 48.013 | 47.405 | 47.224 | 47.547 |
| Station Eight | 298.282 | 296.721 | 295.815 | 296.939 |
| Station Ten | 1097.763 | 1042.993 | 1045.759 | 1062.172 |
| Station Twelve | 3231.455 | 3163.849 | 3091.770 | 3162.358 |
| Two Trains Same Direction | 0.022 | 0.024 | 0.021 | 0.022 |
| Four Trains Same Direction | 1537.286 | 1565.133 | 1525.910 | 1542.776 |
| Nærumbanen (2T) | 20.171 | 20.407 | 20.577 | 20.385 |
| Nærumbanen (3T) | 407.498 | 410.378 | 404.128 | 407.335 |

**Table E.25:** Restricted variant: 'No derailment' 1, limits 2, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with one train | 7612 | 7632 | 7640 | 7628 |
| Example with two trains | 8558 | 8604 | 8620 | 8594 |
| Shared Segment | 9792 | 9836 | 9796 | 9808 |
| Large Network | 12008 | 12072 | 12000 | 12027 |
| Station One | 8540 | 8556 | 8572 | 8556 |
| Station Two | 12608 | 12672 | 12684 | 12655 |
| Station Four | 39248 | 39488 | 39532 | 39423 |
| Station Six | 119492 | 120340 | 120344 | 120059 |
| Station Eight | 274880 | 276720 | 276768 | 276123 |
| Station Ten | 560252 | 565828 | 565840 | 563973 |
| Station Twelfth | 1017416 | 1264912 | 1015104 | 1099144 |
| Two trains same direction | 11504 | 11544 | 12076 | 11708 |
| Four trains same direction | 902860 | 902864 | 904716 | 903480 |
| Nærumbanen (2T) | 118712 | 119532 | 119740 | 119328 |
| Nærumbanen (3T) | 316156 | 316488 | 316500 | 316381 |

**Table E.26:** Restricted variant: 'No derailment' 1, limits 2, Memory Usage Peak in KB

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.002 | 0.002 | 0.003 | 0.002 |
| Example with Two Trains | 0.031 | 0.037 | 0.049 | 0.039 |
| Shared Segment | 0.015 | 0.014 | 0.016 | 0.015 |
| Large Network | 0.012 | 0.020 | 0.020 | 0.017 |
| Station One | 0.022 | 0.020 | 0.026 | 0.023 |
| Station Two | 0.116 | 0.145 | 0.138 | 0.133 |
| Station Four | 2.125 | 2.202 | 2.114 | 2.147 |
| Station Six | 47.666 | 47.449 | 47.622 | 47.579 |
| Station Eight | 292.541 | 291.094 | 293.485 | 292.373 |
| Station Ten | 1073.923 | 1076.563 | 1074.403 | 1074.963 |
| Station Twelve | 3323.860 | 3224.331 | 3152.559 | 3233.583 |
| Two Trains Same Direction | 0.012 | 0.023 | 0.026 | 0.020 |
| Four Trains Same Direction | 1566.301 | 1570.500 | 1523.703 | 1553.501 |
| Nærumbanen (2T) | 19.936 | 19.731 | 19.947 | 19.871 |
| Nærumbanen (3T) | 416.936 | 415.357 | 415.144 | 415.812 |

**Table E.27:** Restricted variant: 'No derailment' 2, limits 2, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with one train | 7580 | 7592 | 7592 | 7588 |
| Example with two trains | 8540 | 8564 | 8572 | 8559 |
| Shared Segment | 9744 | 9776 | 9784 | 9768 |
| Large Network | 11968 | 12020 | 12032 | 12007 |
| Station One | 8496 | 8524 | 8528 | 8516 |
| Station Two | 12552 | 12600 | 12620 | 12591 |
| Station Four | 40260 | 40448 | 40456 | 40388 |
| Station Six | 121108 | 121994 | 121956 | 121686 |
| Station Eight | 274252 | 276084 | 276132 | 275489 |
| Station Ten | 560432 | 563852 | 563852 | 562712 |
| Station Twelfth | 1011640 | 1019088 | 1019096 | 1016608 |
| Two trains same direction | 9764 | 9796 | 9804 | 9788 |
| Four trains same direction | 905036 | 905036 | 905472 | 905181 |
| Nærumbanen (2T) | 115860 | 116676 | 116884 | 116473 |
| Nærumbanen (3T) | 311716 | 312044 | 312044 | 311935 |

**Table E.28:** Restricted Variant: 'No derailment' 2, limits 2, Memory Usage Peak in KB

| Configuration | Second | First | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.004 | 0.004 | 0.004 | 0.004 |
| Example with Two Trains | 0.034 | 0.036 | 0.033 | 0.034 |
| Shared Segment | 0.015 | 0.014 | 0.015 | 0.015 |
| Large Network | 0.029 | 0.011 | 0.021 | 0.020 |
| Station One | 0.019 | 0.022 | 0.022 | 0.021 |
| Station Two | 0.142 | 0.102 | 0.136 | 0.127 |
| Station Four | 2.399 | 2.094 | 2.331 | 2.275 |
| Station Six | 47.752 | 47.224 | 47.632 | 47.536 |
| Station Eight | 271.339 | 290.211 | 269.255 | 276.935 |
| Station Ten | 1072.370 | 1069.352 | 1051.182 | 1064.301 |
| Station Twelve | 3154.039 | 3032.947 | 3265.269 | 3150.752 |
| Two Trains Same Direction | 0.011 | 0.015 | 0.014 | 0.013 |
| Four Trains Same Direction | 1513.903 | 1519.292 | 1466.833 | 1500.009 |
| Nærumbanen (2T) | 20.008 | 19.764 | 19.959 | 19.910 |
| Nærumbanen (3T) | 395.981 | 393.703 | 393.868 | 394.517 |

**Table E.29:** Restricted variant: 'No Collision', limits 2, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with one train | 7560 | 7576 | 7576 | 7571 |
| Example with two trains | 8548 | 8560 | 8568 | 8559 |
| Shared Segment | 9744 | 9776 | 9784 | 9768 |
| Large Network | 11944 | 11996 | 12016 | 11985 |
| Station One | 8500 | 8508 | 8516 | 8508 |
| Station Two | 12540 | 12588 | 12608 | 12579 |
| Station Four | 39228 | 39448 | 39476 | 39384 |
| Station Six | 122276 | 123040 | 123044 | 122787 |
| Station Eight | 276256 | 277364 | 277368 | 276996 |
| Station Ten | 560320 | 560176 | 561456 | 560651 |
| Station Twelfth | 1266208 | 1796796 | 1019336 | 1360780 |
| Two trains same direction | 9744 | 9780 | 9792 | 9772 |
| Four trains same direction | 903736 | 903792 | 904192 | 903907 |
| Nærumbanen (2T) | 112928 | 113756 | 113976 | 113553 |
| Nærumbanen (3T) | 313656 | 313988 | 314000 | 313881 |

**Table E.30:** Restricted variant: 'No Collision', limits 2, Memory Usage Peak in KB

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.003 | 0.004 | 0.002 | 0.003 |
| Example with Two Trains | 0.031 | 0.035 | 0.034 | 0.033 |
| Shared Segment | 0.010 | 0.016 | 0.016 | 0.014 |
| Large Network | 0.014 | 0.019 | 0.021 | 0.018 |
| Station One | 0.018 | 0.019 | 0.021 | 0.019 |
| Station Two | 0.109 | 0.114 | 0.136 | 0.120 |
| Station Four | 2.189 | 2.502 | 2.125 | 2.272 |
| Station Six | 46.999 | 46.819 | 47.045 | 46.954 |
| Station Eight | 296.215 | 296.269 | 295.207 | 295.897 |
| Station Ten | 1060.366 | 1043.868 | 1053.478 | 1052.571 |
| Station Twelve | 3291.267 | 3136.088 | 3102.526 | 3176.627 |
| Two Trains Same Direction | 0.013 | 0.012 | 0.019 | 0.015 |
| Four Trains Same Direction | 1498.865 | 1498.277 | 1493.474 | 1496.872 |
| Nærumbanen (2T) | 20.173 | 19.960 | 20.082 | 20.072 |
| Nærumbanen (3T) | 432.761 | 400.223 | 409.341 | 414.108 |

**Table E.31:** Restricted variant: 'Will arrive', limits 2, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with one train | 7500 | 7500 | 7500 | 7500 |
| Example with two trains | 8472 | 8472 | 8476 | 8473 |
| Shared Segment | 9668 | 9688 | 9692 | 9683 |
| Large Network | 11900 | 11924 | 11924 | 11916 |
| Station One | 8428 | 8428 | 8428 | 8428 |
| Station Two | 12468 | 12508 | 12520 | 12499 |
| Station Four | 39064 | 39228 | 39260 | 39184 |
| Station Six | 119216 | 119604 | 119612 | 119477 |
| Station Eight | 272788 | 273536 | 273548 | 273291 |
| Station Ten | 715344 | 716436 | 559096 | 663625 |
| Station Twelfth | 1013140 | 1020800 | 1020828 | 1018256 |
| Two trains same direction | 9684 | 9704 | 9708 | 9699 |
| Four trains same direction | 907728 | 907852 | 908220 | 907933 |
| Nærumbanen (2T) | 114440 | 115208 | 115376 | 115008 |
| Nærumbanen (3T) | 312232 | 312752 | 312764 | 312583 |

**Table E.32:** Restricted variant: 'Will arrive', limits 2, Memory Usage Peak in KB

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.005 | 0.004 | 0.003 | 0.004 |
| Example with Two Trains | 0.036 | 0.046 | 0.040 | 0.041 |
| Shared Segment | 0.014 | 0.014 | 0.016 | 0.015 |
| Large Network | 0.019 | 0.018 | 0.017 | 0.018 |
| Station One | 0.015 | 0.018 | 0.018 | 0.017 |
| Station Two | 0.125 | 0.133 | 0.149 | 0.136 |
| Station Four | 2.321 | 2.735 | 2.390 | 2.482 |
| Station Six | 46.617 | 45.514 | 45.616 | 45.916 |
| Station Eight | 276.247 | 276.290 | 277.621 | 276.719 |
| Station Ten | 1019.511 | 1017.561 | 1002.800 | 1013.291 |
| Nærumbanen (2T) | 30.402 | 31.323 | 30.141 | 30.622 |
| Nærumbanen (3T) | 609.778 | 609.493 | 608.663 | 609.311 |

**Table E.33:** Global variant: 'No derailment' 1, limits 1, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with one train | 7656 | 7676 | 7676 | 7669 |
| Example with two trains | 8404 | 8412 | 8420 | 8412 |
| Shared Segment | 9924 | 9956 | 9960 | 9947 |
| Large Network | 11924 | 11972 | 11980 | 11959 |
| Station One | 8360 | 8364 | 8372 | 8365 |
| Station Two | 12254 | 12572 | 12584 | 12470 |
| Station Four | 37716 | 37956 | 37964 | 37879 |
| Station Six | 103552 | 104284 | 104372 | 104069 |
| Station Eight | 231528 | 233348 | 233428 | 232768 |
| Station Ten | 451236 | 454584 | 451232 | 452351 |
| Nærumbanen (2T) | 113668 | 114688 | 114792 | 114383 |
| Nærumbanen (3T) | 309836 | 310420 | 310736 | 310331 |

**Table E.34:** Global variant: 'No derailment' 1, limits 1, Memory Usage Peak in KB

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.004 | 0.004 | 0.005 | 0.004 |
| Example with Two Trains | 0.026 | 0.027 | 0.051 | 0.035 |
| Shared Segment | 0.021 | 0.020 | 0.022 | 0.021 |
| Large Network | 0.016 | 0.017 | 0.027 | 0.020 |
| Station One | 0.022 | 0.021 | 0.025 | 0.023 |
| Station Two | 0.125 | 0.134 | 0.164 | 0.141 |
| Station Four | 2.394 | 2.338 | 2.348 | 2.360 |
| Station Six | 45.077 | 44.229 | 45.653 | 44.986 |
| Station Eight | 289.866 | 288.213 | 288.014 | 288.698 |
| Station Ten | 1027.780 | 1021.841 | 1022.455 | 1024.025 |
| Nærumbanen (2T) | 29.375 | 29.997 | 29.294 | 29.555 |
| Nærumbanen (3T) | 634.538 | 625.064 | 623.036 | 627.546 |

**Table E.35:** Global variant: 'No derailment' 2, limits 1, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with one train | 7648 | 7668 | 7668 | 7661 |
| Example with two trains | 8384 | 8408 | 8416 | 8403 |
| Shared Segment | 9904 | 9932 | 9936 | 9924 |
| Large Network | 11924 | 11968 | 11980 | 11957 |
| Station One | 8328 | 8356 | 8360 | 8348 |
| Station Two | 12516 | 12564 | 12572 | 12551 |
| Station Four | 37712 | 37952 | 37360 | 37675 |
| Station Six | 103552 | 104280 | 104360 | 104064 |
| Station Eight | 231512 | 233328 | 233400 | 232747 |
| Station Ten | 451224 | 454576 | 454584 | 453461 |
| Nærumbanen (2T) | 111964 | 112992 | 113096 | 112684 |
| Nærumbanen (3T) | 309812 | 310700 | 310728 | 310413 |

**Table E.36:** Global variant: 'No derailment' 2, limits 1, Memory Usage Peak in KB

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.004 | 0.004 | 0.004 | 0.004 |
| Example with Two Trains | 0.036 | 0.034 | 0.042 | 0.037 |
| Shared Segment | 0.020 | 0.019 | 0.020 | 0.020 |
| Large Network | 0.019 | 0.027 | 0.025 | 0.024 |
| Station One | 0.025 | 0.015 | 0.025 | 0.022 |
| Station Two | 0.125 | 0.129 | 0.128 | 0.127 |
| Station Four | 2.685 | 2.651 | 2.694 | 2.677 |
| Station Six | 44.791 | 45.379 | 44.759 | 44.976 |
| Station Eight | 286.995 | 287.993 | 285.826 | 286.938 |
| Station Ten | 1022.700 | 1029.362 | 1027.356 | 1026.473 |
| Nærumbanen (2T) | 27.643 | 28.318 | 28.409 | 28.123 |
| Nærumbanen (3T) | 589.217 | 596.547 | 592.764 | 592.843 |

**Table E.37:** Global variant: 'No Collision', limits 1, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with one train | 7632 | 7652 | 7656 | 7647 |
| Example with two trains | 8392 | 8400 | 8408 | 8400 |
| Shared Segment | 9900 | 9936 | 10192 | 10009 |
| Large Network | 11920 | 11972 | 11980 | 11957 |
| Station One | 8344 | 8352 | 8360 | 8352 |
| Station Two | 12504 | 12560 | 12572 | 12545 |
| Station Four | 37696 | 37892 | 37948 | 37845 |
| Station Six | 103528 | 104260 | 104352 | 104047 |
| Station Eight | 231500 | 233320 | 233340 | 232720 |
| Station Ten | 451224 | 454576 | 454680 | 453493 |
| Nærumbanen (2T) | 111960 | 112988 | 112996 | 112648 |
| Nærumbanen (3T) | 309820 | 310408 | 310736 | 310321 |

**Table E.38:** Global variant: 'No Collision', limits 1, Memory Usage Peak in KB

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.003 | 0.004 | 0.003 | 0.003 |
| Example with Two Trains | 0.037 | 0.048 | 0.045 | 0.043 |
| Shared Segment | 0.021 | 0.021 | 0.021 | 0.021 |
| Large Network | 0.015 | 0.016 | 0.020 | 0.017 |
| Station One | 0.025 | 0.025 | 0.025 | 0.025 |
| Station Two | 0.124 | 0.137 | 0.133 | 0.131 |
| Station Four | 2.578 | 2.538 | 2.654 | 2.590 |
| Station Six | 46.681 | 46.718 | 47.014 | 46.804 |
| Station Eight | 285.616 | 285.069 | 285.284 | 285.323 |
| Station Ten | 1012.653 | 1003.455 | 1008.457 | 1008.188 |
| Nærumbanen (2T) | 28.386 | 28.808 | 28.951 | 28.715 |
| Nærumbanen (3T) | 642.802 | 642.096 | 635.737 | 640.212 |

**Table E.39:** Global variant: 'Will arrive', limits 1, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with one train | 7572 | 7580 | 7580 | 7577 |
| Example with two trains | 8316 | 8316 | 8316 | 8316 |
| Shared Segment | 9828 | 9848 | 9852 | 9843 |
| Large Network | 11856 | 11856 | 11856 | 11856 |
| Station One | 8268 | 8268 | 8268 | 8268 |
| Station Two | 12428 | 12468 | 12480 | 12459 |
| Station Four | 37624 | 37808 | 37816 | 37749 |
| Station Six | 103380 | 103724 | 104060 | 103721 |
| Station Eight | 231172 | 232936 | 233012 | 232373 |
| Station Ten | 452100 | 455184 | 455188 | 454157 |
| Nærumbanen (2T) | 111888 | 112812 | 112820 | 112507 |
| Nærumbanen (3T) | 309736 | 310312 | 310464 | 310171 |

**Table E.40:** Global variant: 'Will arrive', limits 1, Memory Usage Peak in KB

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.009 | 0.008 | 0.009 | 0.009 |
| Example with Two Trains | 0.056 | 0.069 | 0.068 | 0.064 |
| Shared Segment | 0.033 | 0.040 | 0.041 | 0.038 |
| Large Network | 0.022 | 0.031 | 0.047 | 0.033 |
| Station One | 0.032 | 0.037 | 0.038 | 0.036 |
| Station Two | 0.316 | 0.404 | 0.374 | 0.365 |
| Station Four | 12.138 | 11.746 | 11.241 | 11.708 |
| Station Six | 254.857 | 253.484 | 255.691 | 254.677 |
| Station Eight | 1658.055 | 1580.792 | 1545.470 | 1594.772 |
| Station Ten | 6293.598 | 6145.904 | 6052.362 | 6163.955 |
| Nærumbanen (2T) | 132.497 | 131.916 | 133.614 | 132.676 |
| Nærumbanen (3T) | 4627.806 | 4562.283 | 4565.455 | 4585.181 |

**Table E.41:** Global variant: 'No derailment' 1, limits 2, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with one train | 9296 | 9320 | 9324 | 9313 |
| Example with two trains | 10116 | 10132 | 10144 | 10131 |
| Shared Segment | 11644 | 11936 | 12212 | 11931 |
| Large Network | 13876 | 13924 | 13940 | 13913 |
| Station One | 10052 | 10068 | 10080 | 10067 |
| Station Two | 16324 | 16388 | 16396 | 16369 |
| Station Four | 59620 | 59884 | 59900 | 59801 |
| Station Six | 211792 | 212548 | 212528 | 212289 |
| Station Eight | 540852 | 542156 | 542156 | 541721 |
| Station Ten | 1177352 | 1177396 | 1179200 | 1177983 |
| Nærumbanen (2T) | 152988 | 154016 | 154132 | 153712 |
| Nærumbanen (3T) | 1303376 | 1303416 | 1303432 | 1303408 |

**Table E.42:** Global variant: 'No derailment' 1, limits 2, Memory Usage Peak in KB

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.003 | 0.006 | 0.007 | 0.005 |
| Example with Two Trains | 0.061 | 0.052 | 0.067 | 0.060 |
| Shared Segment | 0.031 | 0.031 | 0.052 | 0.038 |
| Large Network | 0.029 | 0.032 | 0.034 | 0.032 |
| Station One | 0.030 | 0.016 | 0.021 | 0.022 |
| Station Two | 0.351 | 0.402 | 0.350 | 0.368 |
| Station Four | 11.451 | 11.260 | 10.112 | 10.941 |
| Station Six | 257.545 | 239.275 | 240.982 | 245.934 |
| Station Eight | 1757.166 | 1660.574 | 1666.525 | 1694.755 |
| Station Ten | 6251.048 | 6112.542 | 6116.494 | 6160.028 |
| Nærumbanen (2T) | 132.551 | 134.115 | 133.177 | 133.281 |
| Nærumbanen (3T) | 4328.896 | 4311.326 | 4319.310 | 4319.844 |

**Table E.43:** Global variant: 'No derailment' 2, limits 2, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with one train | 7656 | 7668 | 7672 | 7665 |
| Example with two trains | 10448 | 10472 | 10480 | 10467 |
| Shared Segment | 9948 | 9976 | 9980 | 9968 |
| Large Network | 11948 | 11996 | 12008 | 11984 |
| Station One | 8396 | 8424 | 8428 | 8416 |
| Station Two | 15408 | 15456 | 15464 | 15443 |
| Station Four | 58064 | 58308 | 58316 | 58229 |
| Station Six | 208876 | 209304 | 209308 | 209163 |
| Station Eight | 542196 | 543516 | 543516 | 543076 |
| Station Ten | 1178508 | 1178568 | 1178036 | 1178371 |
| Nærumbanen (2T) | 151420 | 152436 | 152560 | 152139 |
| Nærumbanen (3T) | 1303352 | 1303392 | 1303392 | 1303379 |

**Table E.44:** Global variant: 'No derailment' 2, limits 2, Memory Usage Peak in KB

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.006 | 0.006 | 0.007 | 0.006 |
| Example with Two Trains | 0.034 | 0.035 | 0.063 | 0.044 |
| Shared Segment | 0.018 | 0.037 | 0.033 | 0.029 |
| Large Network | 0.024 | 0.029 | 0.040 | 0.031 |
| Station One | 0.018 | 0.017 | 0.019 | 0.018 |
| Station Two | 0.306 | 0.398 | 0.349 | 0.351 |
| Station Four | 12.486 | 10.081 | 12.083 | 11.550 |
| Station Six | 252.827 | 246.848 | 249.513 | 249.729 |
| Station Eight | 1625.016 | 1620.067 | 1548.218 | 1597.767 |
| Station Ten | 6219.558 | 5960.382 | 6024.153 | 6068.031 |
| Nærumbanen (2T) | 133.816 | 133.661 | 133.648 | 133.708 |
| Nærumbanen (3T) | 4380.034 | 4368.982 | 4373.205 | 4374.074 |

**Table E.45:** Global variant: 'No Collision', limits 2, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 7648 | 7660 | 7660 | 7656 |
| Example with two trains | 8420 | 8428 | 8436 | 8428 |
| Shared Segment | 9924 | 9956 | 10216 | 10032 |
| Large Network | 11948 | 12000 | 12008 | 11985 |
| Station One | 8404 | 8412 | 8420 | 8412 |
| Station Two | 14920 | 14976 | 14984 | 14960 |
| Station Four | 58036 | 58280 | 58288 | 58201 |
| Station Six | 209552 | 210308 | 210312 | 210057 |
| Station Eight | 543948 | 544996 | 545256 | 544733 |
| Station Ten | 1176496 | 1177572 | 1178464 | 1177511 |
| Nærumbanen (2T) | 151368 | 152404 | 152424 | 152065 |
| Nærumbanen (3T) | 1303336 | 1303388 | 1303408 | 1303377 |

**Table E.46:** Global variant: 'No Collision', limits 2, Memory Usage Peak in KB

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.007 | 0.005 | 0.005 | 0.006 |
| Example with Two Trains | 0.063 | 0.054 | 0.063 | 0.060 |
| Shared Segment | 0.018 | 0.017 | 0.020 | 0.018 |
| Large Network | 0.023 | 0.042 | 0.038 | 0.034 |
| Station One | 0.018 | 0.029 | 0.030 | 0.026 |
| Station Two | 0.300 | 0.318 | 0.401 | 0.340 |
| Station Four | 12.274 | 10.647 | 12.419 | 11.780 |
| Station Six | 259.847 | 261.048 | 261.756 | 260.884 |
| Station Eight | 1670.962 | 1697.476 | 1604.006 | 1657.481 |
| Station Ten | 6131.157 | 5857.906 | 5887.561 | 5958.875 |
| Nærumbanen (2T) | 131.425 | 130.869 | 132.489 | 131.594 |
| Nærumbanen (3T) | 4587.688 | 4593.748 | 4603.034 | 4594.823 |

**Table E.47:** Global variant: 'Will arrive', limits 2, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with one train | 7584 | 7592 | 7592 | 7589 |
| Example with two trains | 8348 | 8348 | 8348 | 8348 |
| Shared Segment | 9860 | 9880 | 9884 | 9875 |
| Large Network | 11860 | 11900 | 11904 | 11888 |
| Station One | 8332 | 8332 | 8332 | 8332 |
| Station Two | 15332 | 15376 | 15384 | 15364 |
| Station Four | 58804 | 58980 | 58996 | 58927 |
| Station Six | 205536 | 205880 | 205900 | 205772 |
| Station Eight | 542224 | 543556 | 543556 | 543112 |
| Station Ten | 1172864 | 1172864 | 1172880 | 1172869 |
| Nærumbanen (2T) | 151304 | 152240 | 152248 | 151931 |
| Nærumbanen (3T) | 1303276 | 1303312 | 1303312 | 1303300 |

**Table E.48:** Global variant: 'Will arrive', limits 2, Memory Usage Peak in KB

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.003 | 0.005 | 0.005 | 0.004 |
| Example with Two Trains | 0.007 | 0.005 | 0.007 | 0.006 |
| Shared segment | 0.025 | 0.032 | 0.030 | 0.029 |
| Large network | 0.027 | 0.029 | 0.041 | 0.032 |
| Station One | 0.024 | 0.025 | 0.031 | 0.027 |
| Station Two | 0.228 | 0.239 | 0.286 | 0.251 |
| Station Four | 5.665 | 5.602 | 5.315 | 5.527 |
| Station Six | 107.828 | 110.020 | 106.935 | 108.261 |
| Station Eight | 633.381 | 628.627 | 630.462 | 630.823 |
| Nærumbanen (2T) | 44.972 | 44.176 | 45.270 | 44.806 |
| Nærumbanen (3T) | 1137.892 | 1136.126 | 1138.605 | 1137.541 |

**Table E.49:** Extended variant: 'No derailment' 1, limits 1, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with one train | 7828 | 7848 | 7848 | 7841 |
| Example with two trains | 7820 | 7844 | 7844 | 7836 |
| Shared segment | 10220 | 10224 | 10492 | 10312 |
| Large network | 12300 | 12352 | 12356 | 12336 |
| Station One | 8716 | 8744 | 8752 | 8737 |
| Station Two | 13564 | 13624 | 13632 | 13607 |
| Station Four | 48324 | 48548 | 48584 | 48485 |
| Station Six | 151016 | 151900 | 151908 | 151608 |
| Station Eight | 370992 | 372952 | 373952 | 372632 |
| Nærumbanen (2T) | 127520 | 128484 | 128716 | 128240 |
| Nærumbanen (3T) | 552016 | 552700 | 552700 | 552472 |

**Table E.50:** Extended variant: 'No derailment' 1, limits 1, Memory Usage Peak in KB

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.003 | 0.004 | 0.004 | 0.004 |
| Example with Two Trains | 0.005 | 0.005 | 0.005 | 0.005 |
| Shared segment | 0.024 | 0.023 | 0.031 | 0.026 |
| Large network | 0.029 | 0.034 | 0.037 | 0.033 |
| Station One | 0.030 | 0.037 | 0.037 | 0.035 |
| Station Two | 0.278 | 0.277 | 0.256 | 0.270 |
| Station Four | 4.868 | 5.811 | 4.912 | 5.197 |
| Station Six | 108.516 | 107.877 | 107.639 | 108.011 |
| Station Eight | 631.406 | 636.037 | 635.308 | 634.250 |
| Nærumbanen (2T) | 44.166 | 42.726 | 43.928 | 43.607 |
| Nærumbanen (3T) | 1107.463 | 1105.787 | 1134.785 | 1116.012 |

**Table E.51:** Extended variant: 'No derailment' 2, limits 1, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with one train | 7796 | 7808 | 7816 | 7807 |
| Example with two trains | 7808 | 7824 | 7828 | 7820 |
| Shared segment | 9976 | 10256 | 10260 | 10164 |
| Large network | 12260 | 12312 | 12324 | 12299 |
| Station One | 8692 | 8696 | 8716 | 8701 |
| Station Two | 13544 | 13596 | 13604 | 13581 |
| Station Four | 48284 | 48508 | 48544 | 48445 |
| Station Six | 150992 | 151872 | 151872 | 151579 |
| Station Eight | 370960 | 372916 | 372928 | 372268 |
| Nærumbanen (2T) | 125848 | 126812 | 127032 | 126564 |
| Nærumbanen (3T) | 550272 | 550296 | 550272 | 550280 |

**Table E.52:** Extended variant: 'No derailment' 2, limits 1, Memory Usage Peak in KB

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.004 | 0.004 | 0.004 | 0.004 |
| Example with Two Trains | 0.004 | 0.007 | 0.008 | 0.006 |
| Shared segment | 0.026 | 0.019 | 0.031 | 0.025 |
| Large network | 0.023 | 0.045 | 0.031 | 0.033 |
| Station One | 0.025 | 0.034 | 0.036 | 0.032 |
| Station Two | 0.266 | 0.244 | 0.250 | 0.253 |
| Station Four | 5.340 | 5.289 | 5.214 | 5.281 |
| Station Six | 107.415 | 108.704 | 106.545 | 107.555 |
| Station Eight | 633.211 | 636.107 | 642.983 | 637.434 |
| Nærumbanen (2T) | 42.576 | 43.328 | 43.498 | 43.134 |
| Nærumbanen (3T) | 1121.852 | 1129.423 | 1129.642 | 1126.972 |

**Table E.53:** Extended variant: 'No Collision', limits 1, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with one train | 7784 | 7800 | 7804 | 7796 |
| Example with two trains | 7788 | 7800 | 7804 | 7797 |
| Shared segment | 10176 | 10180 | 10448 | 10268 |
| Large network | 12256 | 12304 | 12324 | 12295 |
| Station One | 8680 | 8708 | 8716 | 8701 |
| Station Two | 13528 | 13588 | 13596 | 13571 |
| Station Four | 48280 | 48504 | 48540 | 48441 |
| Station Six | 150972 | 151848 | 151860 | 151560 |
| Station Eight | 370944 | 372896 | 372908 | 372249 |
| Nærumbanen (2T) | 125832 | 126796 | 127016 | 126548 |
| Nærumbanen (3T) | 550268 | 550300 | 550272 | 550280 |

**Table E.54:** Extended variant: 'No Collision', limits 1, Memory Usage Peak in KB

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.002 | 0.004 | 0.004 | 0.003 |
| Example with Two Trains | 0.005 | 0.006 | 0.005 | 0.005 |
| Shared segment | 0.029 | 0.019 | 0.028 | 0.025 |
| Large network | 0.030 | 0.046 | 0.032 | 0.036 |
| Station One | 0.023 | 0.046 | 0.030 | 0.033 |
| Station Two | 0.222 | 0.231 | 0.234 | 0.229 |
| Station Four | 4.714 | 4.932 | 4.823 | 4.823 |
| Station Six | 107.899 | 107.418 | 104.468 | 106.595 |
| Station Eight | 653.839 | 654.680 | 653.906 | 654.142 |
| Nærumbanen (2T) | 42.939 | 42.902 | 42.686 | 42.842 |
| Nærumbanen (3T) | 1123.635 | 1124.439 | 1093.512 | 1113.862 |

**Table E.55:** Extended variant: 'Will arrive', limits 1, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with one train | 7724 | 7732 | 7732 | 7729 |
| Example with two trains | 7740 | 7744 | 7748 | 7744 |
| Shared segment | 9904 | 9928 | 9928 | 9920 |
| Large network | 12184 | 12224 | 12224 | 12211 |
| Station One | 8612 | 8612 | 8624 | 8616 |
| Station Two | 13444 | 13488 | 13492 | 13475 |
| Station Four | 48132 | 48264 | 48276 | 48224 |
| Station Six | 150296 | 151156 | 151164 | 150872 |
| Station Eight | 369096 | 370960 | 370964 | 370340 |
| Nærumbanen (2T) | 125760 | 126624 | 126864 | 126416 |
| Nærumbanen (3T) | 550176 | 550176 | 550180 | 550177 |

**Table E.56:** Extended variant: 'Will arrive', limits 1, Memory Usage Peak in KB

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.004 | 0.008 | 0.009 | 0.007 |
| Example with Two Trains | 0.081 | 0.076 | 0.084 | 0.080 |
| Shared segment | 0.036 | 0.043 | 0.047 | 0.042 |
| Large network | 0.039 | 0.082 | 0.065 | 0.062 |
| Station One | 0.028 | 0.038 | 0.039 | 0.035 |
| Station Two | 0.777 | 0.927 | 0.906 | 0.870 |
| Station Four | 32.451 | 36.382 | 33.524 | 34.119 |
| Station Six | 910.817 | 888.077 | 888.182 | 895.692 |
| Station Eight | 6530.169 | 6563.059 | 6564.713 | 6552.647 |
| Nærumbanen (2T) | 242.522 | 242.283 | 242.911 | 242.572 |
| Nærumbanen (3T) | 10033.654 | 10092.504 | 10099.487 | 10075.215 |

**Table E.57:** Extended variant: 'No derailment' 1, limits 2, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with one train | 7836 | 7860 | 7860 | 7852 |
| Example with two trains | 9024 | 9048 | 9056 | 9043 |
| Shared segment | 10308 | 10312 | 10580 | 10400 |
| Large network | 12384 | 12440 | 12444 | 12423 |
| Station One | 12036 | 12060 | 12068 | 12055 |
| Station Two | 18520 | 18568 | 18576 | 18555 |
| Station Four | 139412 | 139664 | 139672 | 139583 |
| Station Six | 680148 | 680496 | 680504 | 680383 |
| Station Eight | 2103844 | 2104664 | 2103808 | 2104105 |
| Nærumbanen (2T) | 240248 | 240920 | 240948 | 240705 |
| Nærumbanen (3T) | 3640400 | 3640416 | 3640424 | 3640413 |

**Table E.58:** Extended variant: 'No derailment' 1, limits 2, Memory Usage Peak in KB

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.007 | 0.007 | 0.007 | 0.007 |
| Example with Two Trains | 0.065 | 0.075 | 0.077 | 0.072 |
| Shared segment | 0.044 | 0.066 | 0.059 | 0.056 |
| Large network | 0.047 | 0.062 | 0.060 | 0.056 |
| Station One | 0.038 | 0.035 | 0.035 | 0.036 |
| Station Two | 0.913 | 0.926 | 0.793 | 0.877 |
| Station Four | 32.448 | 32.247 | 32.239 | 32.311 |
| Station Six | 920.258 | 917.515 | 922.833 | 920.202 |
| Station Eight | 6620.755 | 6610.525 | 6595.127 | 6608.802 |
| Nærumbanen (2T) | 249.023 | 253.583 | 252.875 | 251.827 |
| Nærumbanen (3T) | 10004.897 | 10072.137 | 9968.254 | 10015.096 |

**Table E.59:** Extended variant: 'No derailment' 2, limits 2, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with one train | 7812 | 7828 | 7832 | 7824 |
| Example with two trains | 9000 | 9016 | 9024 | 9013 |
| Shared segment | 10076 | 10356 | 10360 | 10264 |
| Large network | 12352 | 12400 | 12412 | 12388 |
| Station One | 8716 | 8736 | 8744 | 8732 |
| Station Two | 16624 | 16680 | 16680 | 16661 |
| Station Four | 139368 | 139620 | 139628 | 139539 |
| Station Six | 680044 | 680392 | 680392 | 680276 |
| Station Eight | 2103808 | 2104628 | 2104632 | 2104356 |
| Nærumbanen (2T) | 238396 | 239228 | 239252 | 238959 |
| Nærumbanen (3T) | 3638684 | 3638700 | 3638712 | 3638699 |

**Table E.60:** Extended variant: 'No derailment' 2, limits 2, Memory Usage Peak in KB

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.006 | 0.006 | 0.006 | 0.006 |
| Example with Two Trains | 0.068 | 0.064 | 0.075 | 0.069 |
| Shared segment | 0.033 | 0.055 | 0.052 | 0.047 |
| Large network | 0.038 | 0.063 | 0.040 | 0.047 |
| Station One | 0.048 | 0.044 | 0.047 | 0.046 |
| Station Two | 0.915 | 0.932 | 0.835 | 0.894 |
| Station Four | 33.160 | 32.814 | 35.634 | 33.869 |
| Station Six | 946.866 | 952.055 | 954.846 | 951.256 |
| Station Eight | 6550.005 | 6608.182 | 6603.568 | 6587.252 |
| Nærumbanen (2T) | 254.086 | 251.442 | 250.039 | 251.856 |
| Nærumbanen (3T) | 9754.725 | 9692.086 | 9644.453 | 9697.088 |

**Table E.61:** Extended variant: 'No Collision', limits 2, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with one train | 7780 | 7792 | 7796 | 7789 |
| Example with two trains | 8980 | 9004 | 9016 | 9000 |
| Shared segment | 10252 | 10256 | 10524 | 10344 |
| Large network | 12336 | 12384 | 12400 | 12373 |
| Station One | 8708 | 8732 | 8740 | 8727 |
| Station Two | 16604 | 16656 | 16664 | 16641 |
| Station Four | 139356 | 139608 | 139616 | 139527 |
| Station Six | 680100 | 680448 | 680452 | 680333 |
| Station Eight | 2103804 | 2104608 | 2104624 | 2104345 |
| Nærumbanen (2T) | 238384 | 239212 | 239240 | 238945 |
| Nærumbanen (3T) | 3638672 | 3638708 | 3638728 | 3638703 |

**Table E.62:** Extended variant: 'No Collision', limits 2, Memory Usage Peak in KB

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.005 | 0.009 | 0.007 | 0.007 |
| Example with Two Trains | 0.081 | 0.076 | 0.074 | 0.077 |
| Shared Segment | 0.041 | 0.042 | 0.048 | 0.044 |
| Large Network | 0.040 | 0.054 | 0.072 | 0.055 |
| Station One | 0.025 | 0.025 | 0.030 | 0.027 |
| Station Two | 0.887 | 0.781 | 0.777 | 0.815 |
| Station Four | 36.392 | 31.455 | 36.080 | 34.642 |
| Station Six | 938.160 | 932.271 | 931.334 | 933.922 |
| Station Eight | 6647.814 | 6618.742 | 6608.544 | 6625.033 |
| Nærumbanen (2T) | 245.720 | 241.710 | 241.961 | 243.130 |
| Nærumbanen (3T) | 10032.713 | 9964.283 | 9963.516 | 9986.837 |

**Table E.63:** Extended variant: 'Will arrive', limits 2, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with one train | 7716 | 7720 | 7724 | 7720 |
| Example with two trains | 8908 | 8916 | 8920 | 8915 |
| Shared segment | 9996 | 10016 | 10020 | 10011 |
| Large network | 12276 | 12308 | 12312 | 12299 |
| Station One | 8624 | 8636 | 8640 | 8633 |
| Station Two | 16472 | 16512 | 16516 | 16500 |
| Station Four | 138064 | 138196 | 138212 | 138157 |
| Station Six | 675176 | 675516 | 675516 | 675403 |
| Station Eight | 2087188 | 2087984 | 2087984 | 2087719 |
| Nærumbanen (2T) | 238316 | 239136 | 239156 | 238869 |
| Nærumbanen (3T) | 3638500 | 3638592 | 3638596 | 3638563 |

**Table E.64:** Extended variant: 'Will arrive', limits 2, Memory Usage Peak in KB

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.074 | 0.072 | 0.072 | 0.073 |
| Example with Two Trains | 0.597 | 0.589 | 0.598 | 0.595 |
| Shared Segment | 0.278 | 0.278 | 0.279 | 0.278 |
| Large Network | 0.281 | 0.282 | 0.280 | 0.281 |
| Station One | 0.269 | 0.270 | 0.269 | 0.269 |
| Station Two | 14.436 | 14.517 | 14.455 | 14.469 |
| Station Four | | | | |
| Station Six | | | | |
| Station Eight | | | | |
| Nærumbanen (2T) | | | | |
| Nærumbanen (3T) | | | | |

**Table E.65:** UMC Version of First Model: 'No derailment' 1, limits 1, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.067 | 0.068 | 0.068 | 0.068 |
| Example with Two Trains | 0.409 | 0.408 | 0.412 | 0.410 |
| Shared Segment | 0.199 | 0.199 | 0.200 | 0.199 |
| Large Network | 0.252 | 0.252 | 0.252 | 0.252 |
| Station One | 0.233 | 0.234 | 0.226 | 0.231 |
| Station Two | 7.966 | 8.003 | 7.973 | 7.981 |
| Station Four | | | | |
| Station Six | | | | |
| Station Eight | | | | |
| Nærumbanen (2T) | | | | |
| Nærumbanen (3T) | | | | |

**Table E.66:** UMC Version of First Model: 'No derailment' 2, limits 1, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.063 | 0.063 | 0.064 | 0.063 |
| Example with Two Trains | 0.752 | 0.748 | 0.758 | 0.753 |
| Shared Segment | 0.342 | 0.341 | 0.341 | 0.341 |
| Large Network | 0.346 | 0.343 | 0.345 | 0.345 |
| Station One | 0.368 | 0.379 | 0.378 | 0.375 |
| Station Two | 12.400 | 12.355 | 12.352 | 12.369 |
| Station Four | | | | |
| Station Six | | | | |
| Station Eight | | | | |
| Nærumbanen (2T) | | | | |
| Nærumbanen (3T) | | | | |

**Table E.67:** UMC Version of First Model: 'No Collision', limits 1, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.063 | 0.063 | 0.063 | 0.063 |
| Example with Two Trains | 0.282 | 0.282 | 0.281 | 0.282 |
| Shared Segment | 0.134 | 0.139 | 0.134 | 0.136 |
| Large Network | 0.136 | 0.136 | 0.136 | 0.136 |
| Station One | 0.156 | 0.152 | 0.155 | 0.154 |
| Station Two | 2.422 | 2.427 | 2.428 | 2.426 |
| Station Four | | | | |
| Station Six | | | | |
| Station Eight | | | | |
| Nærumbanen (2T) | | | | |
| Nærumbanen (3T) | | | | |

**Table E.68:** UMC Version of First Model: 'Will arrive', limits 1, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.079 | 0.081 | 0.079 | 0.080 |
| Example with Two Trains | 0.688 | 0.690 | 0.691 | 0.690 |
| Shared Segment | 0.335 | 0.334 | 0.350 | 0.340 |
| Large Network | 0.338 | 0.337 | 0.337 | 0.337 |
| Station One | 0.305 | 0.303 | 0.314 | 0.307 |
| Station Two | 23.583 | 23.752 | 23.649 | 23.661 |
| Station Four | | | | |
| Station Six | | | | |
| Station Eight | | | | |
| Nærumbanen (2T) | | | | |
| Nærumbanen (3T) | | | | |

**Table E.69:** UMC Version of First Model: 'No derailment' 1, limits 2, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.074 | 0.073 | 0.074 | 0.074 |
| Example with Two Trains | 0.460 | 0.459 | 0.458 | 0.459 |
| Shared Segment | 0.230 | 0.231 | 0.231 | 0.231 |
| Large Network | 0.299 | 0.313 | 0.300 | 0.304 |
| Station One | 0.260 | 0.252 | 0.254 | 0.255 |
| Station Two | 12.805 | 12.790 | 12.778 | 12.791 |
| Station Four | | | | |
| Station Six | | | | |
| Station Eight | | | | |
| Nærumbanen (2T) | | | | |
| Nærumbanen (3T) | | | | |

**Table E.70:** UMC Version of First Model: 'No derailment' 2, limits 2, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.066 | 0.066 | 0.066 | 0.066 |
| Example with Two Trains | 0.886 | 0.886 | 0.894 | 0.889 |
| Shared Segment | 0.417 | 0.415 | 0.416 | 0.416 |
| Large Network | 0.420 | 0.419 | 0.420 | 0.420 |
| Station One | 0.429 | 0.430 | 0.429 | 0.429 |
| Station Two | 20.210 | 20.193 | 20.154 | 20.186 |
| Station Four | | | | |
| Station Six | | | | |
| Station Eight | | | | |
| Nærumbanen (2T) | | | | |
| Nærumbanen (3T) | | | | |

**Table E.71:** UMC Version of First Model: 'No Collision', limits 2, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.062 | 0.062 | 0.062 | 0.062 |
| Example with Two Trains | 0.275 | 0.264 | 0.262 | 0.267 |
| Shared Segment | 0.161 | 0.155 | 0.156 | 0.157 |
| Large Network | 0.419 | 0.418 | 0.422 | 0.420 |
| Station One | 0.170 | 0.169 | 0.170 | 0.170 |
| Station Two | 3.548 | 3.556 | 3.535 | 3.546 |
| Station Four | | | | |
| Station Six | | | | |
| Station Eight | | | | |
| Nærumbanen (2T) | | | | |
| Nærumbanen (3T) | | | | |

**Table E.72:** UMC Version of First Model: 'Will arrive', limits 2, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.008 | 0.011 | 0.014 | 0.011 |
| Example with Two Trains | 0.010 | 0.014 | 0.017 | 0.014 |
| Shared segment | 0.066 | 0.079 | 0.068 | 0.071 |
| Large network | 0.080 | 0.087 | 0.097 | 0.088 |
| Station One | 0.073 | 0.084 | 0.070 | 0.076 |
| Station Two | 1.723 | 1.706 | 1.800 | 1.743 |
| Station Four | 2086.282 | 2058.861 | 2082.121 | 2075.755 |
| Station Six | | | | |
| Station Eight | | | | |
| Nærumbanen (2T) | | | | |
| Nærumbanen (3T) | | | | |

**Table E.73:** Revised model: 'No derailment' 1, limits 1, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with one train | 7676 | 7692 | 7696 | 7688 |
| Example with two trains | 7688 | 7704 | 7708 | 7700 |
| Shared segment | 10224 | 10260 | 10264 | 10249 |
| Large network | 12224 | 12268 | 12284 | 12259 |
| Station One | 8548 | 8572 | 8600 | 8573 |
| Station Two | 22600 | 22656 | 22664 | 22640 |
| Station Four | 2347152 | 2347156 | 2347156 | 2347155 |
| Station Six | | | | |
| Station Eight | | | | |
| Nærumbanen (2T) | | | | |
| Nærumbanen (3T) | | | | |

**Table E.74:** Revised model: 'No derailment' 1, limits 1, Memory Usage Peak in KB

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.010 | 0.009 | 0.007 | 0.009 |
| Example with Two Trains | 0.014 | 0.018 | 0.017 | 0.016 |
| Shared segment | 0.055 | 0.072 | 0.064 | 0.064 |
| Large network | 0.066 | 0.084 | 0.087 | 0.079 |
| Station One | 0.075 | 0.088 | 0.077 | 0.080 |
| Station Two | 1.751 | 1.776 | 1.757 | 1.761 |
| Station Four | 2073.825 | 2117.443 | 2137.279 | 2109.516 |
| Station Six | | | | |
| Station Eight | | | | |
| Nærumbanen (2T) | | | | |
| Nærumbanen (3T) | | | | |

**Table E.75:** Revised model: 'No derailment' 2, limits 1, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with one train | 7632 | 7644 | 7644 | 7640 |
| Example with two trains | 7640 | 7656 | 7656 | 7651 |
| Shared segment | 10180 | 10208 | 10212 | 10200 |
| Large network | 12188 | 12228 | 12236 | 12217 |
| Station One | 8508 | 8512 | 8520 | 8513 |
| Station Two | 20620 | 20664 | 20762 | 20682 |
| Station Four | 2347124 | 2347116 | 2347108 | 2347116 |
| Station Six | | | | |
| Station Eight | | | | |
| Nærumbanen (2T) | | | | |
| Nærumbanen (3T) | | | | |

**Table E.76:** Revised model: 'No derailment' 2, limits 1, Memory Usage Peak in KB

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.010 | 0.010 | 0.012 | 0.011 |
| Example with Two Trains | 0.016 | 0.017 | 0.017 | 0.017 |
| Shared segment | 0.062 | 0.066 | 0.072 | 0.067 |
| Large network | 0.066 | 0.092 | 0.083 | 0.080 |
| Station One | 0.062 | 0.056 | 0.079 | 0.066 |
| Station Two | 1.713 | 1.782 | 1.785 | 1.760 |
| Station Four | 2101.282 | 2120.291 | 2110.154 | 2110.576 |
| Station Six | | | | |
| Station Eight | | | | |
| Nærumbanen (2T) | | | | |
| Nærumbanen (3T) | | | | |

**Table E.77:** Revised model: 'No Collision', limits 1, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with one train | 7636 | 7652 | 7652 | 7647 |
| Example with two trains | 7648 | 7664 | 7664 | 7659 |
| Shared segment | 10184 | 10216 | 10224 | 10208 |
| Large network | 12180 | 12228 | 12244 | 12217 |
| Station One | 8516 | 8532 | 8552 | 8533 |
| Station Two | 20620 | 20672 | 20680 | 20657 |
| Station Four | 2347104 | 2347108 | 2347108 | 2347107 |
| Station Six | | | | |
| Station Eight | | | | |
| Nærumbanen (2T) | | | | |
| Nærumbanen (3T) | | | | |

**Table E.78:** Revised model: 'No Collision', limits 1, Memory Usage Peak in KB

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.012 | 0.009 | 0.011 | 0.011 |
| Example with Two Trains | 0.008 | 0.007 | 0.008 | 0.008 |
| Shared segment | 0.065 | 0.079 | 0.076 | 0.073 |
| Large network | 0.066 | 0.066 | 0.075 | 0.069 |
| Station One | 0.075 | 0.062 | 0.078 | 0.072 |
| Station Two | 1.818 | 1.721 | 1.824 | 1.788 |
| Station Four | 2132.376 | 2037.542 | 2048.334 | 2072.751 |
| Station Six | | | | |
| Station Eight | | | | |
| Nærumbanen (2T) | | | | |
| Nærumbanen (3T) | | | | |

**Table E.79:** Revised model: 'Will arrive', limits 1, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 7572 | 7576 | 7576 | 7575 |
| Example with Wwo Trains | 7580 | 7588 | 7588 | 7585 |
| Shared Segment | 10112 | 10132 | 10132 | 10125 |
| Large Network | 12108 | 12144 | 12152 | 12135 |
| Station One | 8456 | 8456 | 8456 | 8456 |
| Station Two | 20444 | 20484 | 20488 | 20472 |
| Station Four | 2230892 | 2230908 | 2230912 | 2230904 |
| Station Six | | | | |
| Station Eight | | | | |
| Nærumbanen (2T) | | | | |
| Nærumbanen (3T) | | | | |

**Table E.80:** Revised model: 'Will arrive', limits 1, Memory Usage Peak in KB

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.014 | 0.015 | 0.017 | 0.015 |
| Example with Two Trains | 0.171 | 0.170 | 0.199 | 0.180 |
| Shared segment | 0.083 | 0.038 | 0.024 | 0.048 |
| Large network | 0.105 | 0.111 | 0.108 | 0.108 |
| Station One | 0.074 | 0.084 | 0.070 | 0.076 |
| Station Two | 2.505 | 2.606 | 2.612 | 2.574 |
| Station Four | 4164.950 | 4235.207 | 4300.883 | 4233.680 |
| Station Six | | | | |
| Station Eight | | | | |
| Nærumbanen (2T) | | | | |
| Nærumbanen (3T) | | | | |

**Table E.81:** Revised model: 'No derailment' 1, limits 2, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 7684 | 7700 | 7704 | 7696 |
| Example with Two Trains | 8960 | 8980 | 9008 | 8983 |
| Shared Segment | 10292 | 10332 | 10336 | 10320 |
| Large Network | 12288 | 12340 | 12352 | 12327 |
| Station One | 8584 | 8604 | 8632 | 8607 |
| Station Two | 26980 | 27028 | 27040 | 27016 |
| Station Four | 4533628 | 4533628 | 4533628 | 4533628 |
| Station Six | | | | |
| Station Eight | | | | |
| Nærumbanen (2T) | | | | |
| Nærumbanen (3T) | | | | |

**Table E.82:** Revised model: 'No derailment' 1, limits 2, Memory Usage Peak in KB

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.013 | 0.016 | 0.016 | 0.015 |
| Example with Two Trains | 0.187 | 0.198 | 0.196 | 0.194 |
| Shared segment | 0.082 | 0.020 | 0.037 | 0.046 |
| Large network | 0.090 | 0.102 | 0.090 | 0.094 |
| Station One | 0.083 | 0.080 | 0.080 | 0.081 |
| Station Two | 2.657 | 2.624 | 2.599 | 2.627 |
| Station Four | 4107.677 | 4339.988 | 4246.319 | 4231.328 |
| Station Six | | | | |
| Station Eight | | | | |
| Nærumbanen (2T) | | | | |
| Nærumbanen (3T) | | | | |

**Table E.83:** Revised model: 'No derailment' 2, limits 2, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 7644 | 7660 | 7660 | 7655 |
| Example with Two Trains | 8916 | 8932 | 8936 | 8928 |
| Shared Segment | 10252 | 10284 | 10288 | 10275 |
| Large Network | 12252 | 12288 | 12296 | 12279 |
| Station One | 8528 | 8544 | 8548 | 8540 |
| Station Two | 29932 | 26980 | 26980 | 27964 |
| Station Four | 4533584 | 4533576 | 4533584 | 4533581 |
| Station Six | | | | |
| Station Eight | | | | |
| Nærumbanen (2T) | | | | |
| Nærumbanen (3T) | | | | |

**Table E.84:** Revised model: 'No derailment' 2, limits 2, Memory Usage Peak in KB

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.014 | 0.020 | 0.015 | 0.016 |
| Example with Two Trains | 0.170 | 0.194 | 0.174 | 0.179 |
| Shared segment | 0.085 | 0.033 | 0.032 | 0.050 |
| Large network | 0.090 | 0.112 | 0.101 | 0.101 |
| Station One | 0.082 | 0.069 | 0.095 | 0.082 |
| Station Two | 2.607 | 2.726 | 2.557 | 2.630 |
| Station Four | 4102.210 | 4170.611 | 4313.341 | 4195.387 |
| Station Six | | | | |
| Station Eight | | | | |
| Nærumbanen (2T) | | | | |
| Nærumbanen (3T) | | | | |

**Table E.85:** Revised model: 'No Collision', limits 2, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 7636 | 7652 | 7652 | 7647 |
| Example with Two Trains | 8916 | 8936 | 8964 | 8939 |
| Shared Segment | 10248 | 10288 | 10292 | 10276 |
| Large Network | 12248 | 12296 | 12312 | 12285 |
| Station One | 8536 | 8560 | 8588 | 8561 |
| Station Two | 26932 | 26984 | 26992 | 26969 |
| Station Four | 4533576 | 4533580 | 4533572 | 4533576 |
| Station Six | | | | |
| Station Eight | | | | |
| Nærumbanen (2T) | | | | |
| Nærumbanen (3T) | | | | |

**Table E.86:** Revised model: 'No Collision', limits 2, Memory Usage Peak in KB

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with One Train | 0.014 | 0.008 | 0.014 | 0.012 |
| Example with Two Trains | 0.165 | 0.166 | 0.169 | 0.167 |
| Shared segment | 0.082 | 0.029 | 0.028 | 0.046 |
| Large network | 0.093 | 0.103 | 0.093 | 0.096 |
| Station One | 0.074 | 0.095 | 0.090 | 0.086 |
| Station Two | 2.696 | 2.539 | 2.519 | 2.585 |
| Station Four | 4103.429 | 4163.708 | 4172.222 | 4146.453 |
| Station Six | | | | |
| Station Eight | | | | |
| Nærumbanen (2T) | | | | |
| Nærumbanen (3T) | | | | |

**Table E.87:** Revised model: 'Will arrive', limits 2, Elapsed time in seconds

| Configuration | First | Second | Third | Average |
|---|---|---|---|---|
| Example with one train | 7584 | 7592 | 7592 | 7589 |
| Example with two trains | 8848 | 8856 | 8860 | 8855 |
| Shared segment | 10184 | 10204 | 10204 | 10197 |
| Large network | 12180 | 12216 | 12224 | 12207 |
| Station One | 8472 | 8472 | 8472 | 8472 |
| Station Two | 26764 | 26796 | 26800 | 26787 |
| Station Four | 4471212 | 4471196 | 4471212 | 4471207 |
| Station Six | | | | |
| Station Eight | | | | |
| Nærumbanen (2T) | | | | |
| Nærumbanen (3T) | | | | |

**Table E.88:** Revised model: 'Will arrive', limits 2, Memory Usage Peak in KB

Appendix F

# Verification Result Graphs

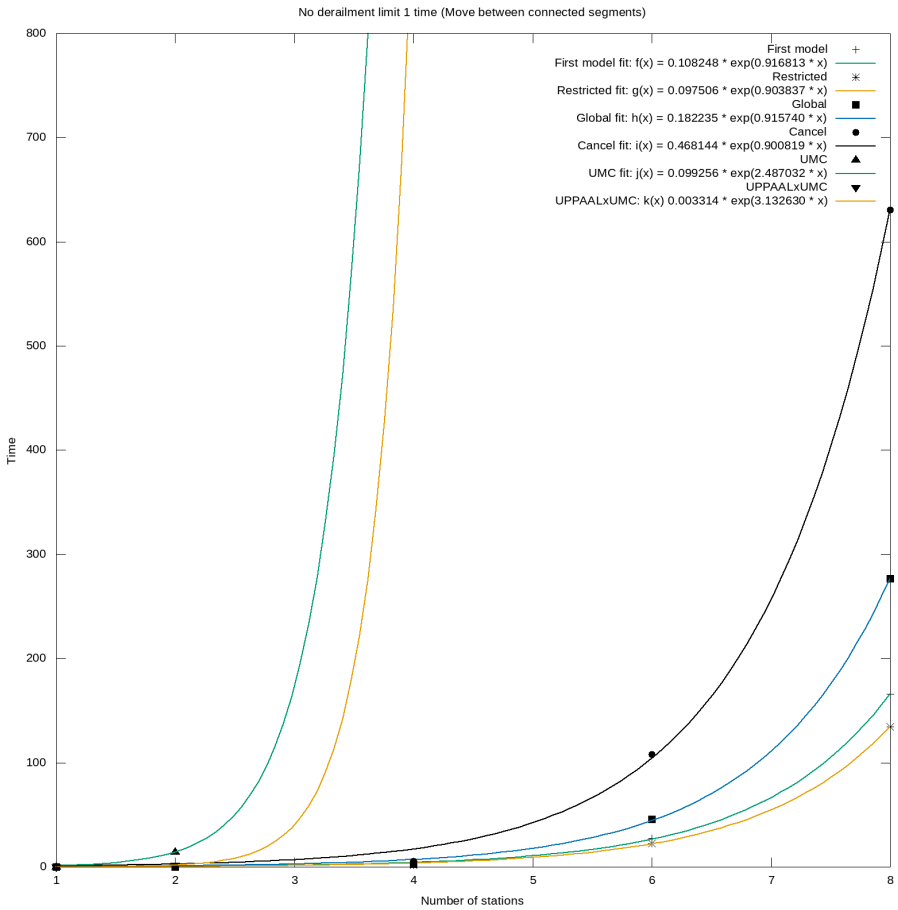



**Figure F.1:** Graphs for number of stations and time limit 1 in seconds

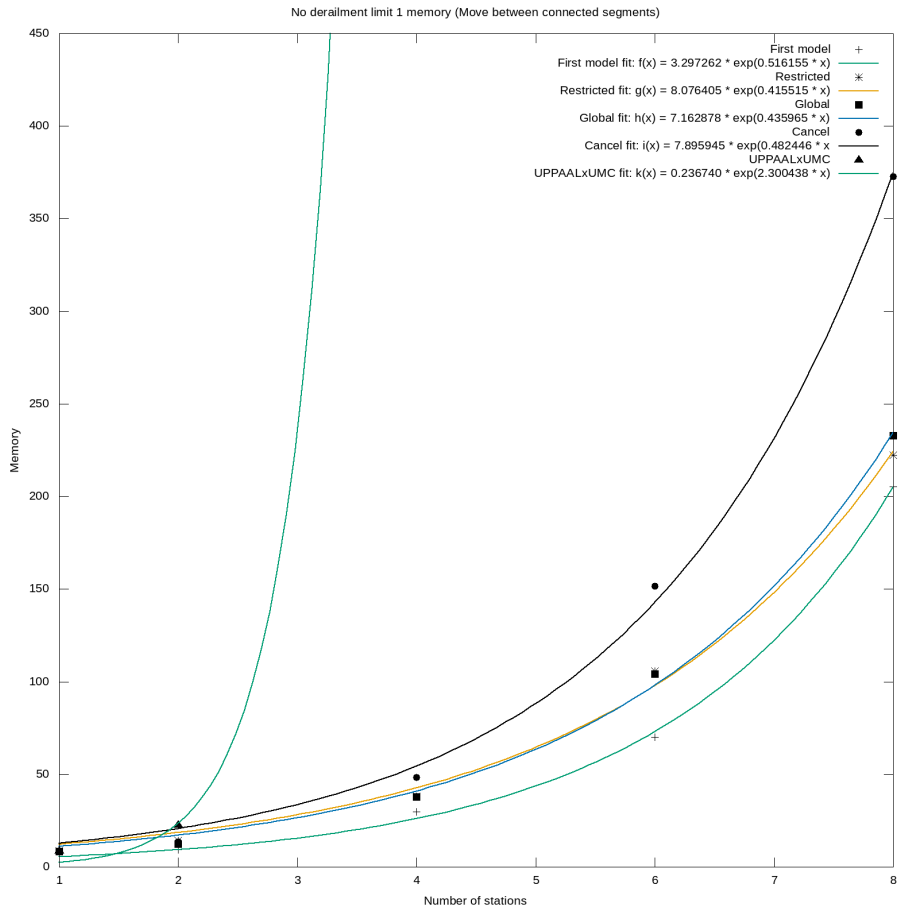



**Figure F.2:** Graphs for number of stations and memory limit 1 in MB

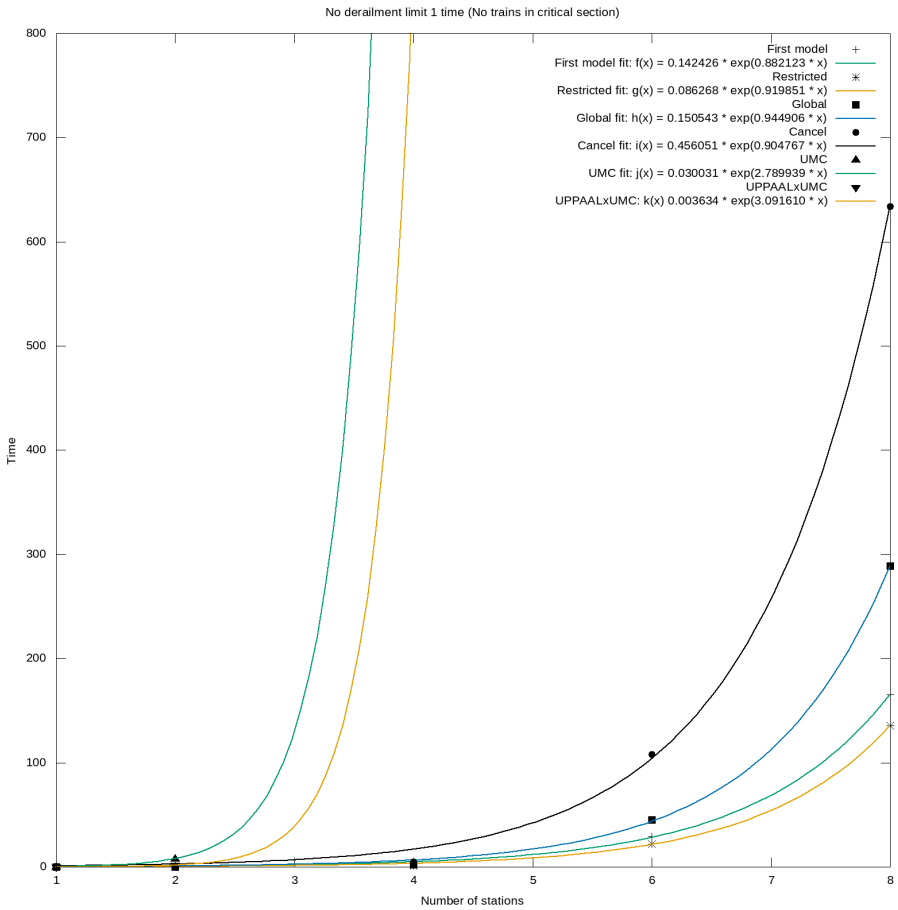



**Figure F.3:** Graphs for number of stations and time limit 1 in seconds

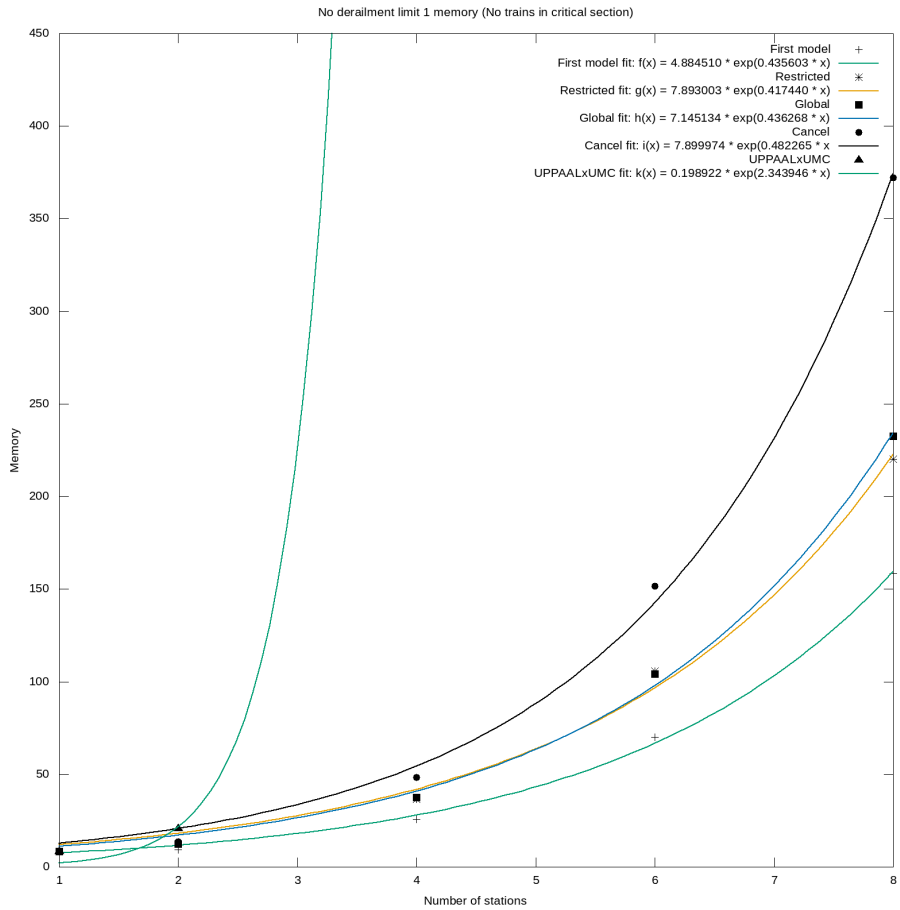



**Figure F.4:** Graphs for number of stations and memory limit 1 in MB

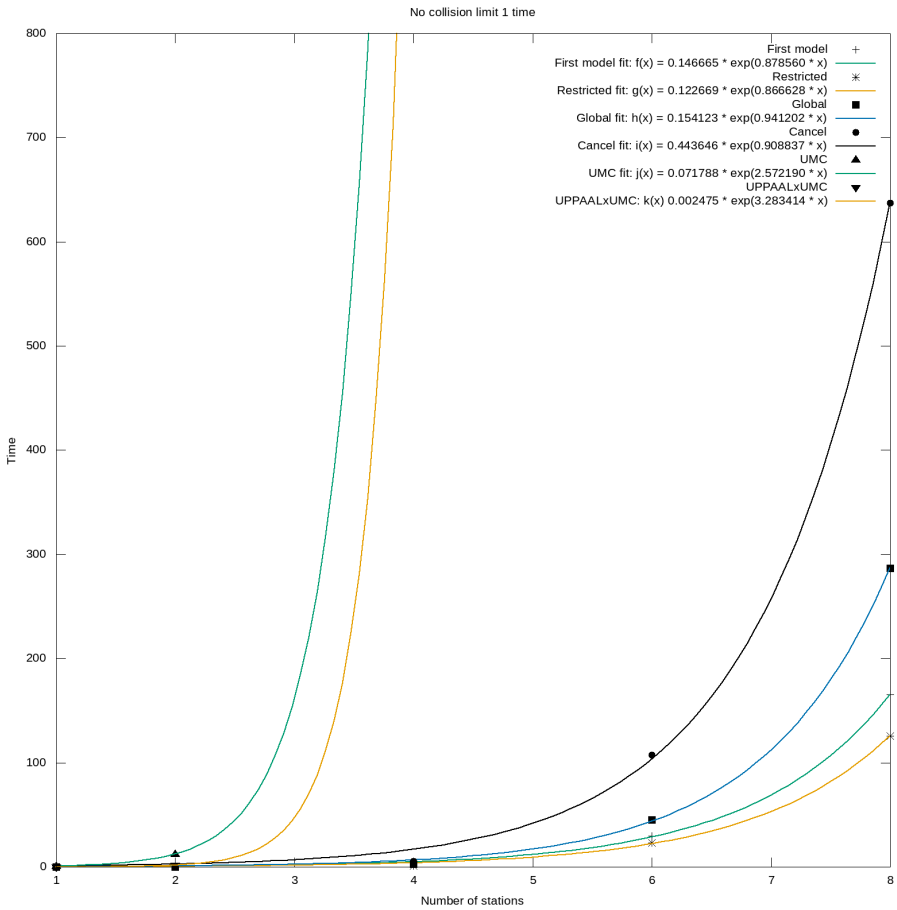



**Figure F.5:** Graphs for number of stations and time limit 1 in seconds

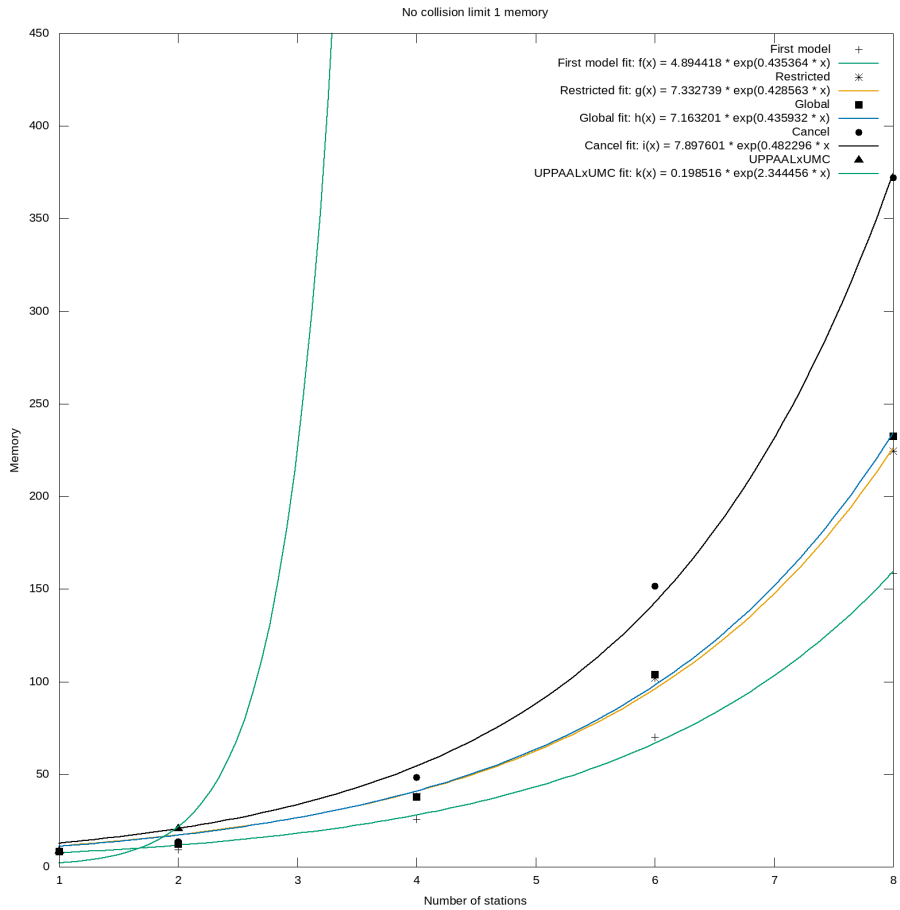



**Figure F.6:** Graphs for number of stations and memory limit 1 in MB

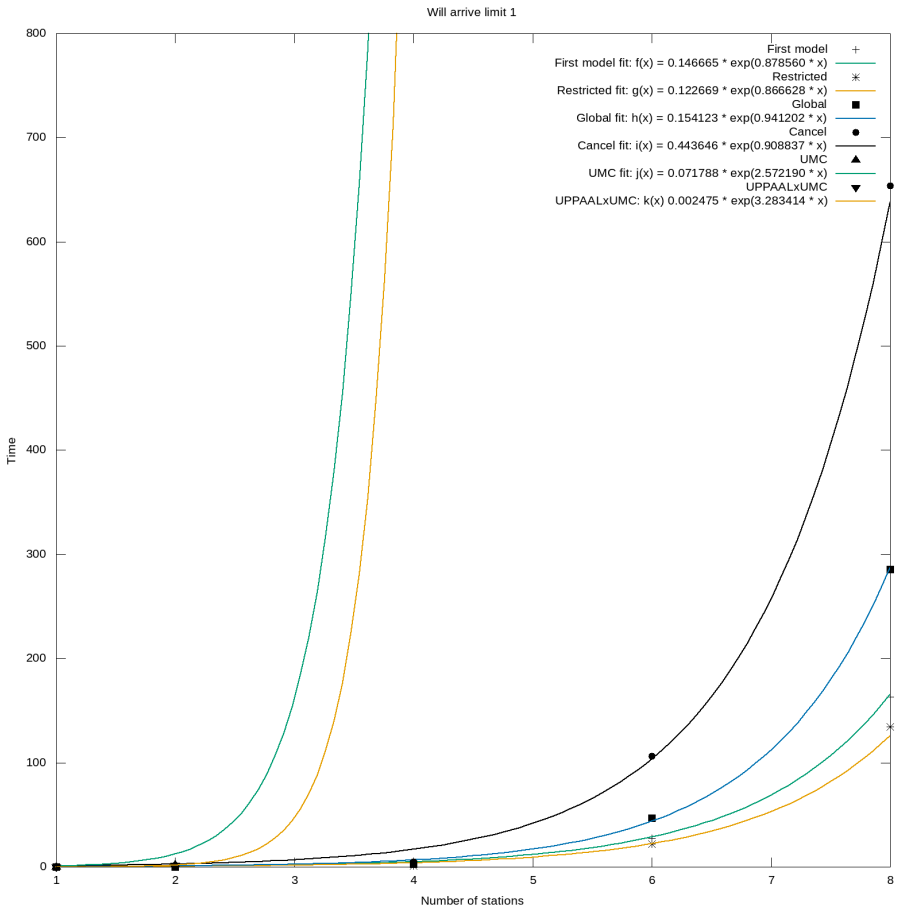



**Figure F.7:** Graphs for number of stations and time limit 1 in seconds

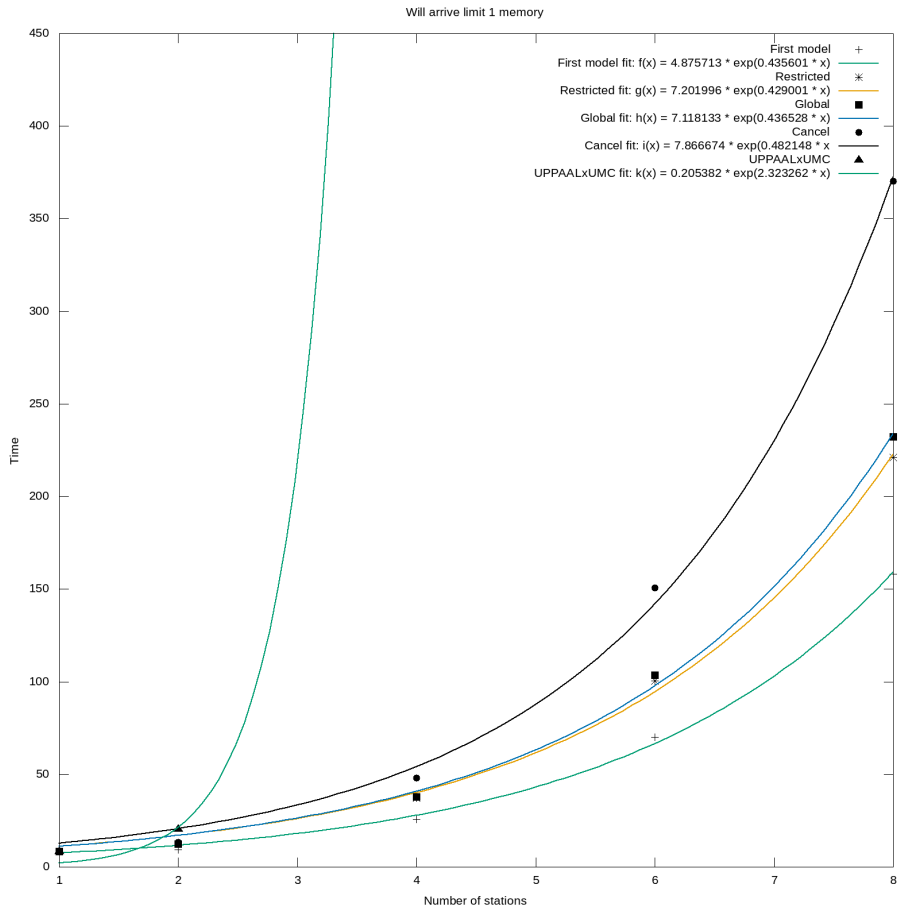



**Figure F.8:** Graphs for number of stations and memory limit 1 in MB

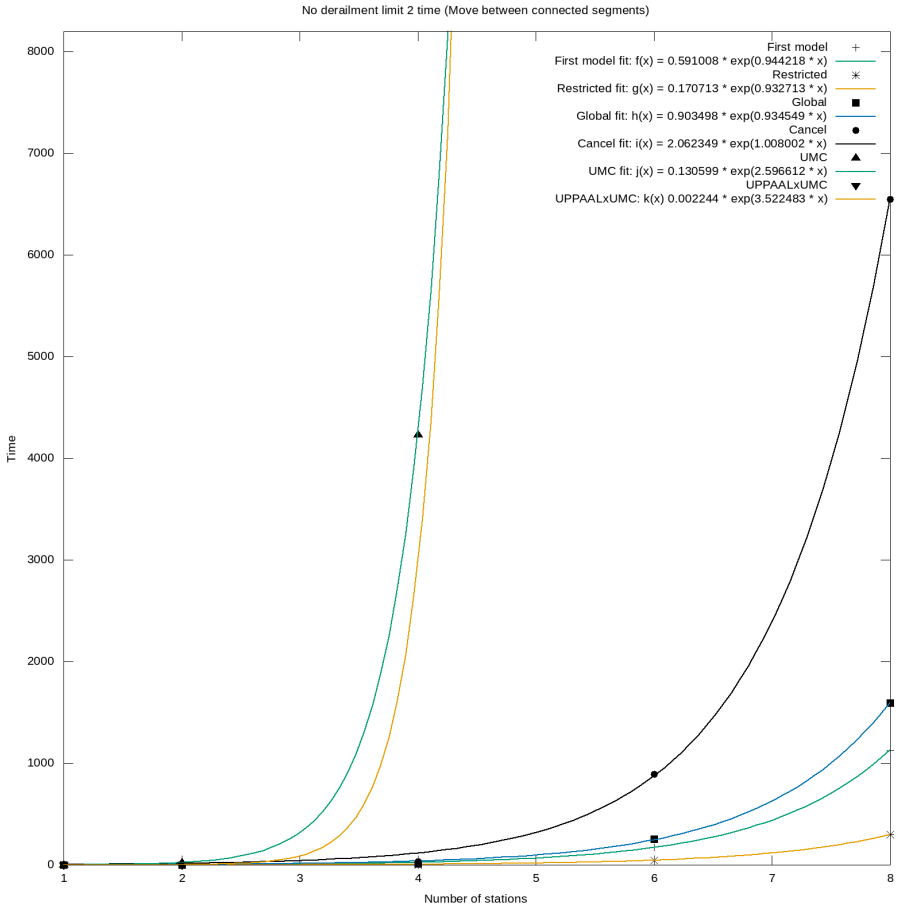



**Figure F.9:** Graphs for number of stations and time limit 2 in seconds

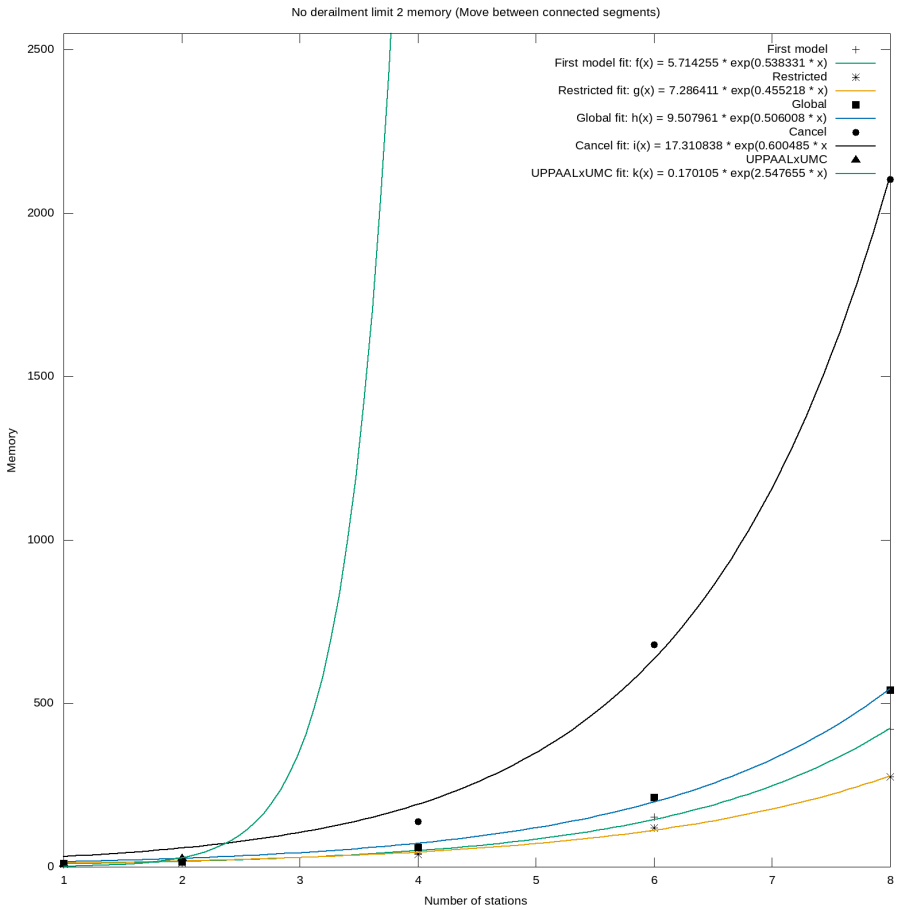



**Figure F.10:** Graphs for number of stations and memory limit 2 in MB

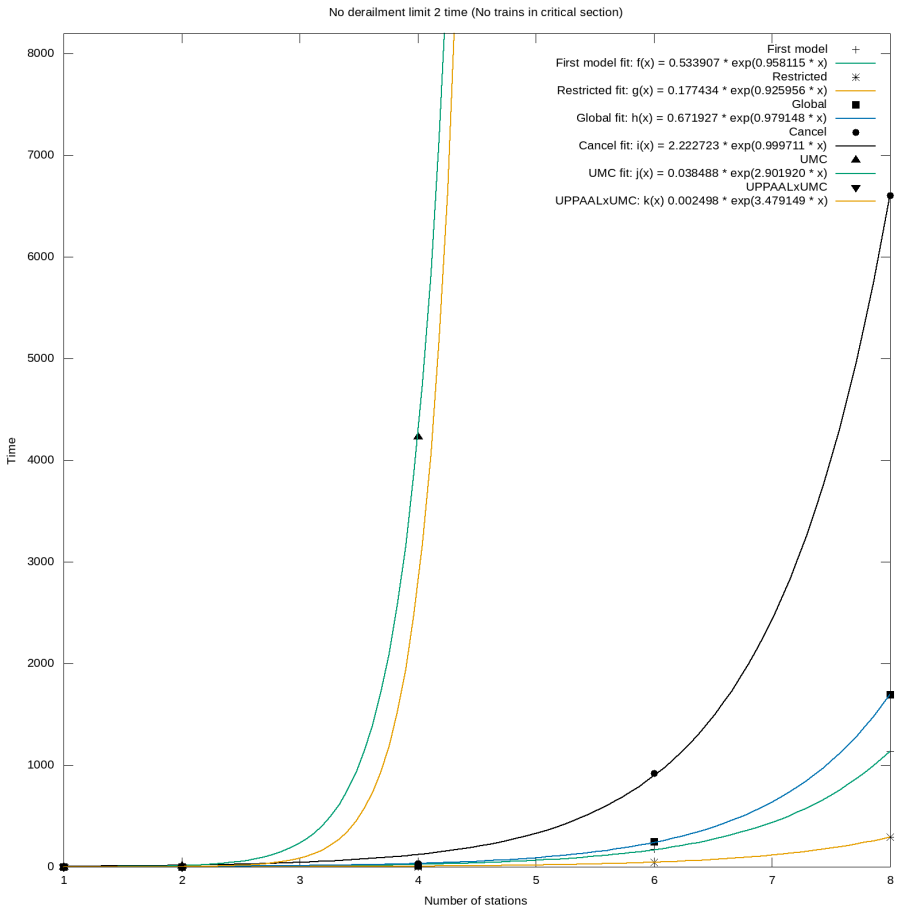



**Figure F.11:** Graphs for number of stations and time limit 2 in seconds

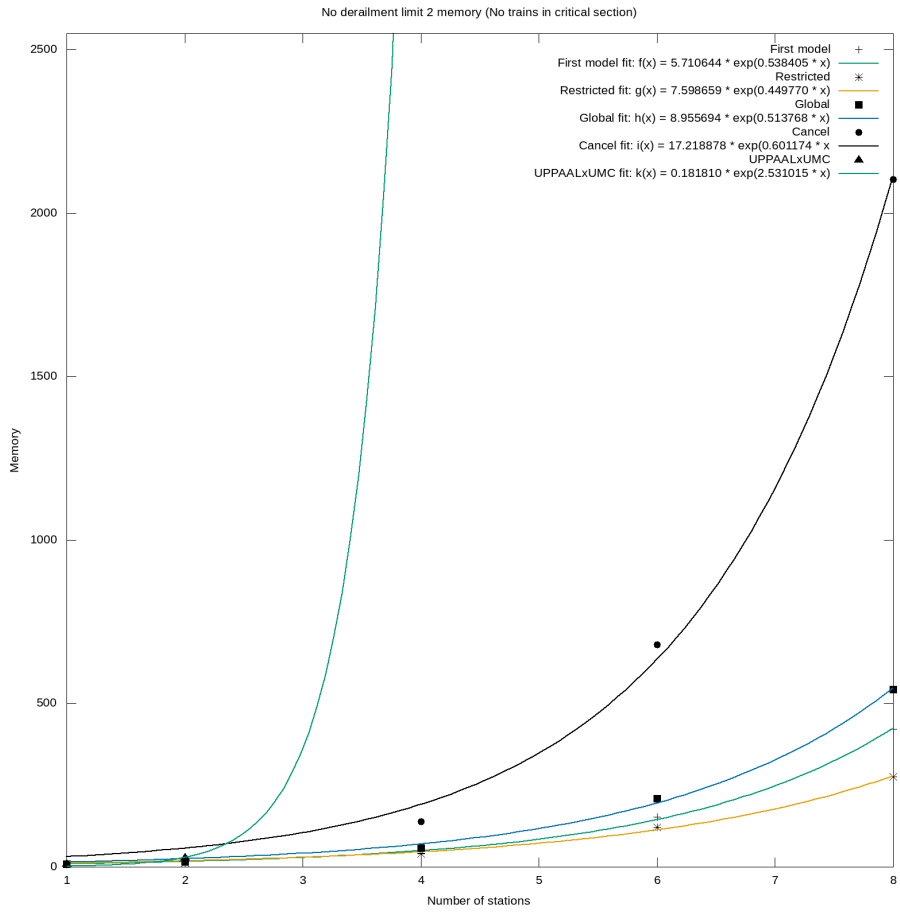



**Figure F.12:** Graphs for number of stations and memory limit 2 in MB

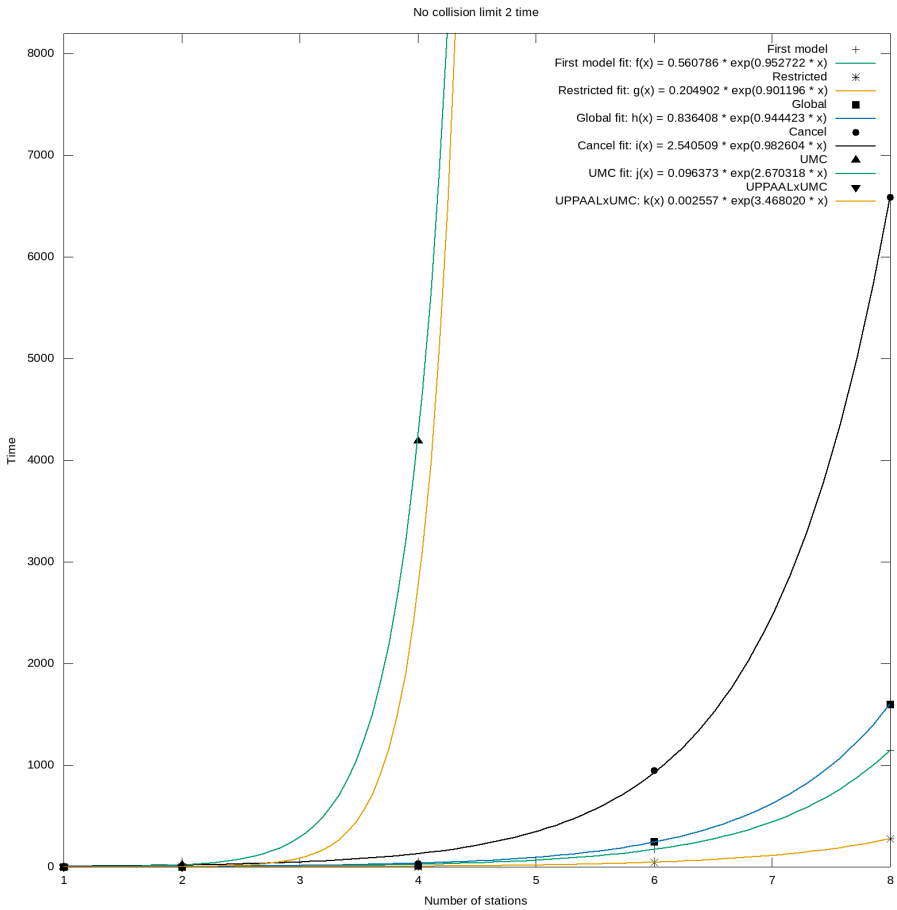



**Figure F.13:** Graphs for number of stations and time limit 2 in seconds

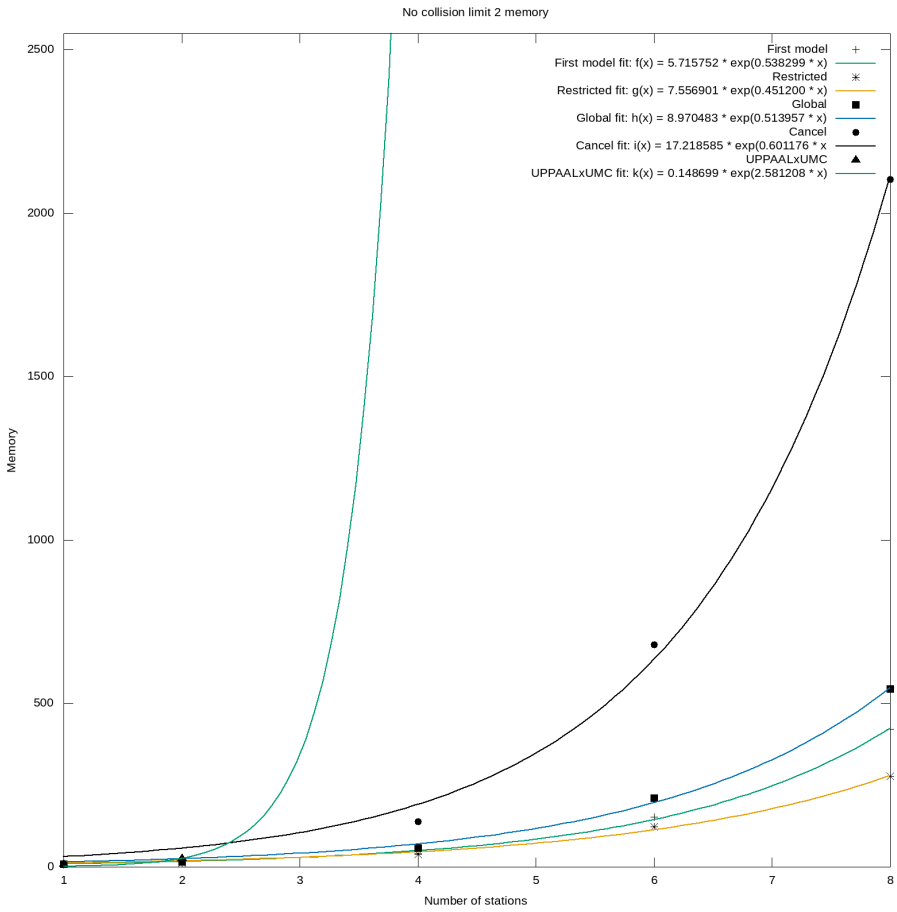



**Figure F.14:** Graphs for number of stations and memory limit 2 in MB

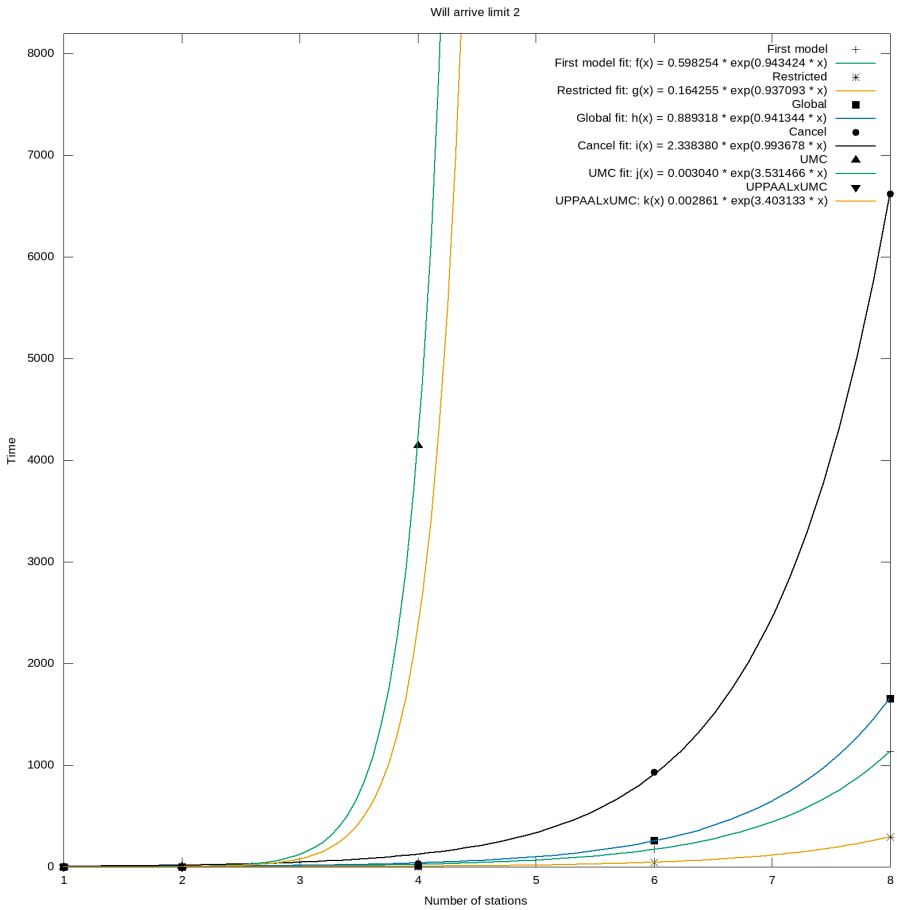



**Figure F.15:** Graphs for number of stations and time limit 2 in seconds

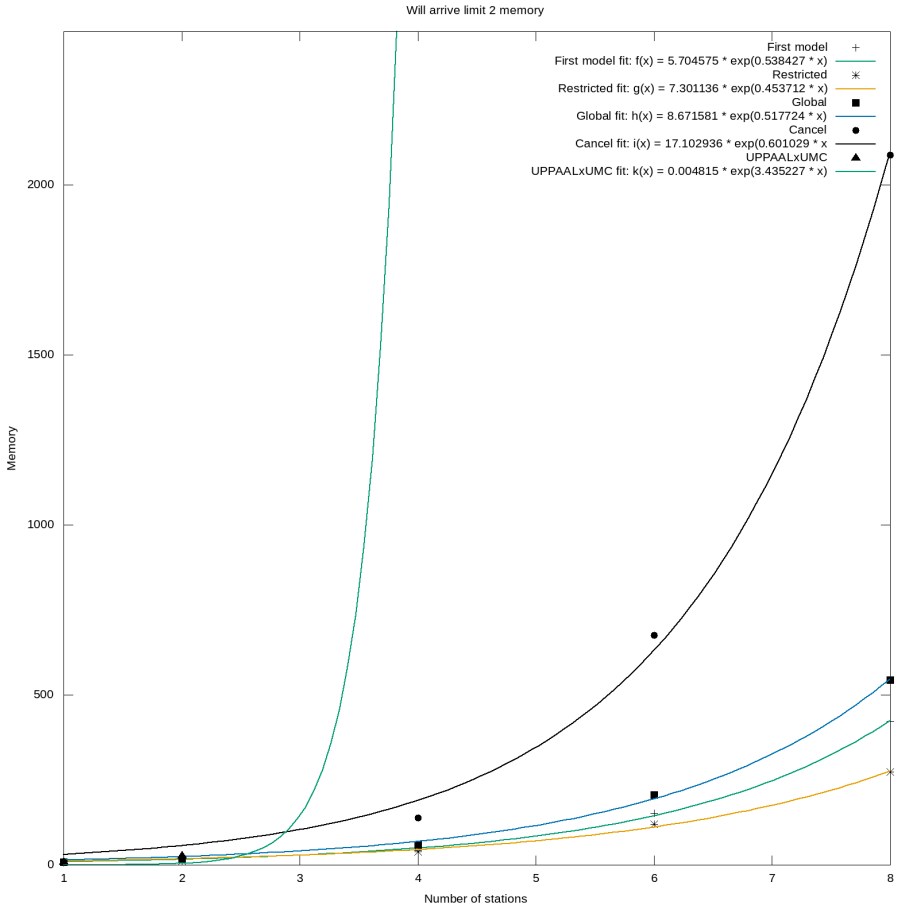



**Figure F.16:** Graphs for number of stations and memory limit 2 in MB

Appendix G

# Eclipse Plugin User Guide

## G.1 Required Tools

To use the plugin, the following tools must be installed:

- *Eclipse Modeling Tools*:
  `https://www.eclipse.org/downloads/packages/release/2019-03/r/eclipse-modeling-tools`
- *Graphical Modeling Framework (GMF) Tooling*:
  `https://www.eclipse.org/gmf-tooling/download.php`

## G.2 Installation

1. **Import the plugin projects**
   (a) Extract the projects from *s144449-s144456_FVDRCS.zip* > *Railway Model Generator Tool* into a folder
   (b) In *Eclipse Modeling*, select `File > Import...`

(c) Select `Existing Projects into Workspace...` and `Next >`

(d) Click `Browse...` and select the folder with the extracted plugin projects from step 1

(e) Click on `Finish`

2. **Set up a runtime configuration**

(a) Go to `Run > Run Configurations...`

(b) Create a new `Eclipse Application` launch configuration (see figure G.1)

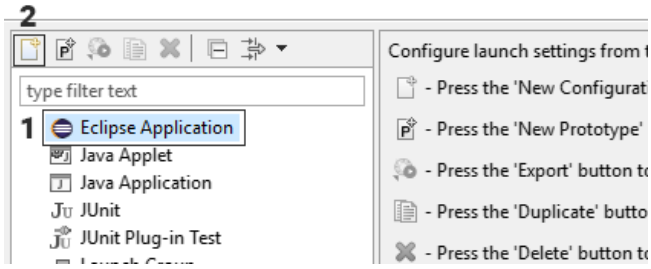



**Figure G.1:** Creating a new launch configuration for an `Eclipse Application`

(c) Go to the `Plug-ins` tab and ensure that `Launch with` includes the imported projects: Use the `all workspace and enabled target plug-ins` option or select the specific features with the `plug-ins selected below only` option.

(d) Click on `Apply` to save the runtime configuration or `Run` to save it and start it immediately

## G.3 Use of Tool

**Create a `Network`**

1. Run the plugin with the created runtime configuration
2. Go to `File > New > Other...` and search `network`
3. Select `Network Diagram`[1]

[1] This option automatically creates a corresponding `network` file

4. Give the `network_diagram` file a name and click on `Next > Finish`
5. Add a new node or edge to the `Network` by using the tools from the tool palette on the right-hand side of the editor (see figure G.2). Select a tool and click on the canvas to place it. Segments can be added by holding down the cursor on one `ControlBox` and dragging and dropping it onto another.

**Set Properties**
Properties can be set for a node/edge in the `Properties` window (`Window > Show View > Properties`).

- `Network`:
  - *Name*: Used in the names of generated model files
  - *Reservation Limit*
  - *Lock Limit*
- `Regular Box`:
  - *ID*
- `Switch Box`:
  - *ID*
  - *Stem*: Segment used as stem segment
  - *Plus*: Segment used as plus segment
  - *Minus*: Segment used as minus segment
  - *Connected*: Segment initially connected to stem segment
- `Segment`:
  - *ID*
- `Train`:
  - *ID*
  - *Route*: List of segments in route

**Generate Model**
To generate model code from the created `Network`, right-click on an empty spot on the canvas and select `Generate Railway Control System Model` and the desired model to generate (see figure G.3). The new file will be created in the same directory as the `network_diagram` file.

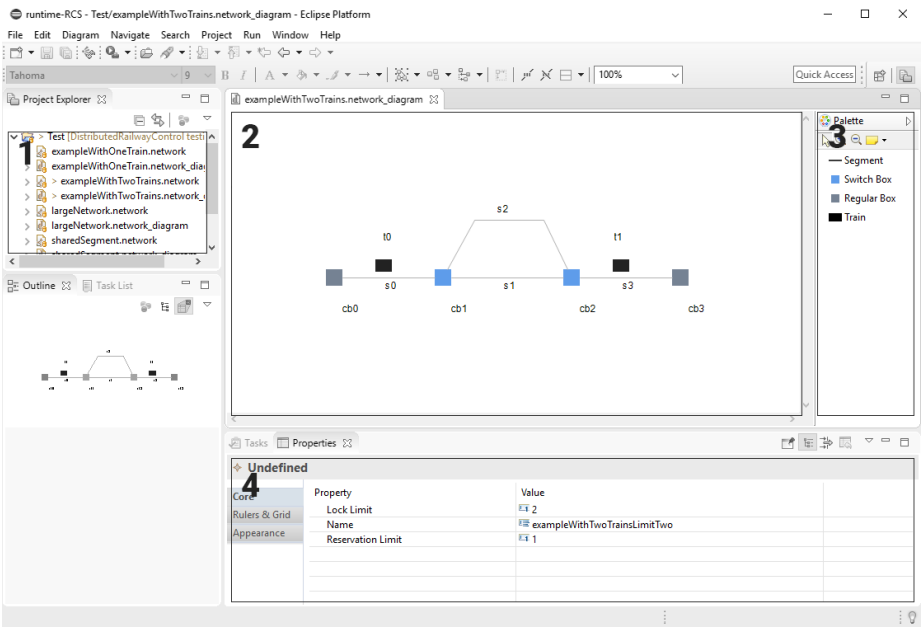



**Figure G.2:** Editor for creating specific railway networks and trains. 1. Project Explorer, 2. Canvas, 3. Tool Palette, 4. Properties

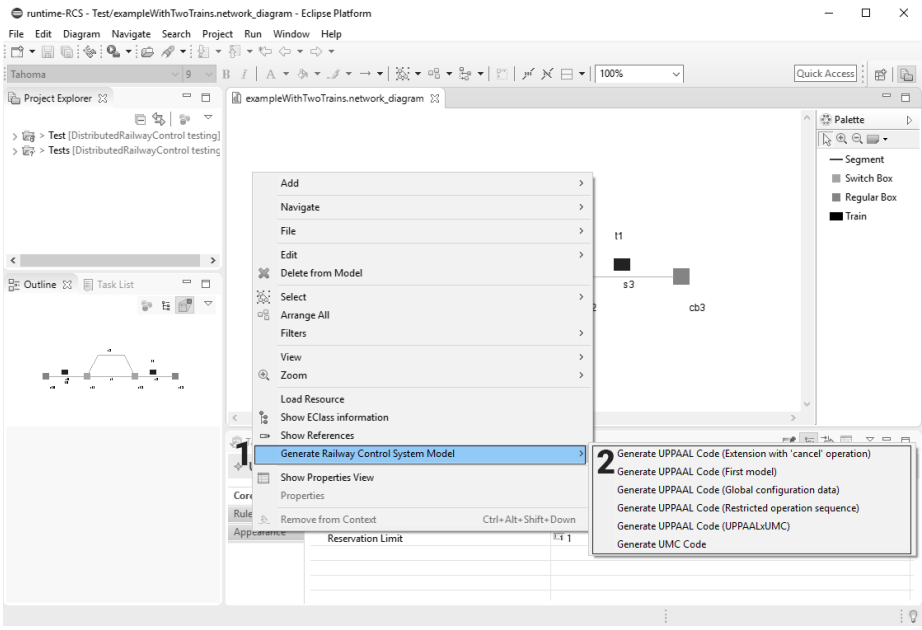



**Figure G.3:** Generate model code

Appendix H

# UPPAAL Example Configuration

## H.1 Example with One Train

```
const int NCB = 4;
const int NSEG = 4;
const int NTRAIN = 1;
const int NPOINT = 2;
const int NROUTELENGTH = 3;

...

const int[0, NROUTELENGTH] lockLimit = 2;
const int[0, NROUTELENGTH-1] resLimit = 2;

const cBV_id boxRoutes[NTRAIN][NROUTELENGTH+1] = {{0,1,2,3}};
const segV_id segRoutes[NTRAIN][NROUTELENGTH]= {{0,1,3}};
const segV_id cBs[NCB][3]= {{0,-1,-1}, {0,1,2}, {3,1,2}, {3,-1,-1}};
const pV_id points[NCB] = { -1, 0, 1, -1 };
const reservation initialRes[NTRAIN] = {{1, 0}};
bool pointInPlus[NPOINT] = {true, true};
```

Appendix I

# UMC Example Configuration

## I.1 Example with One Train

```
Objects
t0:Train(segments=>[0,1,3], curSeg => 0, boxes => [cb0,cb1,cb2,cb3],
    requiresLock => [false,true,true,false], lockIndex => 1);

cb0:CB(segments => [0,-1,-1], res => [null,null,null], connected => null);
cb1:CB(segments => [0,1,2], res => [t0,null,null], connected => 1, point =>
    p0);
cb2:CB(segments => [3,1,2], res => [null,null,null], connected => 1, point =>
     p1);
cb3:CB(segments => [3,-1,-1], res => [null,null,null], connected => null);

p0:Point(inPlus => true, cb => cb1);
p1:Point(inPlus => true, cb => cb2);
```