CHALMERS
UNIVERSITY OF TECHNOLOGY

Verifying the Logical Correctness of a Train Station

Master's thesis in Systems, Control and Mechatronics

BERIT-JANICE HÄRLE and DANIEL LOVÉN ÖBERG

EX046/2016

Department of Signals and Systems
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2016

Master's thesis EX046/2016

Verifying the Logical Correctness of a Train Station

BERIT-JANICE HÄRLE and DANIEL LOVÉN ÖBERG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Signals and Systems
Division of Automation Control
Chalmers University of Technology
Gothenburg, Sweden 2016

Verifying the Logical Correctness of a Train Station
BERIT-JANICE HÄRLE and DANIEL LOVÉN ÖBERG

Supervisor: Martin Fabian, Department of Signals and Systems
Examiner: Martin Fabian, Department of Signals and Systems

Master's Thesis 2016:EXO46/2016
Department of Signals and Systems
Division of Automation Control
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Satellite picture of Gothenburg train station [1]

Typeset in LaTeX
Gothenburg, Sweden 2016

Logical Verification of a Train Station
Berit-Janice Härle
Daniel Lovén-Öberg
Department of Signals and Systems
Chalmers University of Technology

Abstract

Trains offer low emissions and energy efficient transportation of passengers and goods, and are therefore essential to the infrastructure of the country. For the safety of the passengers, amongst others, it is necessary to control the trains to avoid crashes. By using a signaling system (European Rail Traffic Management System), it is possible to keep track of the train locations and react to anomalies. The railway switches and the behaviors of the trains can be modelled and then formally verified for safety. The goal of this work is to use extended finite automata to verify the logical correctness of the Gothenburg Central train station. A model is created in Supremica and UPPAAL, both tools that can verify discrete event systems. The model is then verified to ensure absence of collisions while guaranteeing that the trains can always reach their designated platforms. Initially, a simplified model is created to test the two tools, and to define properties to achieve successful verification. For larger models, the state space becomes too large for UPPAAL that suffers from memory limitations due to its 32-bit implementation. Therefore, Supremica, with its 64-bits hardware, is used to successfully verify the safety of the train station. This report also includes extensive information for both tools.

Keywords: Verification, Automata, Logical Correctness, Railways, Train Station, Safety

Sammanfattning

Tåg är ett miljövänligt och och energieffektivt transportsätt för passagerare och varor. Det är därför en viktig del av infrastrukturen i ett land. För passagerarnas säkerhet är det nödvändigt att reglera tågen för att undvika kollisioner. Genom att använda ett signalsystem (European Rail Traffic Management System) är det möjligt att hålla reda på tågens position och reagera på avvikelser. Järnvägsväxlarnas och tågens beteende kan modelleras och sedan verifieras. Målet med rapporten är att använda sig av ändliga tillståndsmaskiner till att verifiera den logiska korrektheten av Göteborgs centralstation. En modell skapas i Supremica och UPPAAL, båda verktyg för att verifiera diskreta händelsesystem. Modellen är sedan verifierad för att undvika kollisioner och samtidigt garantera att ett tåg alltid når sin angivna plattform. Först skapas en mindre modell för att testa de båda verktygen och olika idéer över hur man kan nå en lyckad verifiering. För större modeller blir tillståndsrummet för stort för UPPAAL som får minnesproblem då det endast kan utnyttja på 32-bitar. Därför används Supremica till att framgångsrikt verifiera säkerheten för tågstationen. Rapporten innehåller också omfattande information om båda programmen.

Keywords: Verifiering, Tillståndsmaskin, Logisk Korrekthet, Järnvägar, Tågstation, Säkerhet

Acknowledgements

We would like to thank our supervisor and examiner Martin Fabian for his support, motivation and always having an open ear. His fast responses to e-mails and his availability even for emergencies on the weekend is impressive and something we are extremely thankful for. Without Martin, the thesis would not have been nearly as successful.

Also, thank you to all of our families and friends for their understanding and mental support.

Berit-Janice Härle and Daniel Lovén Öberg, Gothenburg, June 2016

Contents

List of Figures

List of Tables

1
INTRODUCTION

One important part of the infrastructure of a country is the railway system. Due to the low emission rates and the energy efficiency in comparison to airplanes and vehicles, trains become essential for the transportation of goods and people [7]. Therefore a large number of trains run between cities and countries and are controlled by an automatic European signaling system. Before automatic signaling was introduced, timetables and flag officers were used for railway signaling. However, this could not be continued due to the increase of accidents by human error and high costs [8].

In 2014 in Sweden, 53 significant accidents occurred from which four were due to the collisions of trains. Over the last nine years, the amount of train collisions varied between one and four per year [9]. Even though this number might seem small, a lot of passengers or freight are affected and therefore it is necessary to reduce the collision risk. This can be done by modelling the railways, controlling the behaviour for the trains and verifying that no crashes occur.

There is a wide range of tools that are based on different methods to model and verify systems. Two of these tools are called UPPAAL and Supremica and offer modelling using automata and verification based on automata properties.

The task of the project is to test UPPAAL and Supremica to see if it is possible to build and verify a safety critical model using the Gothenburg station as a template. To do this, a model describing a train station is created and different techniques are used to specify the control for the trains. Thereafter the model, with specifications, is verified in different ways to ensure that no collisions can occur. Also, tutorials are to be created for both programs.

1.1 Scope

Due to Trafikverket not supplying information, assumptions have to be made. These include the functionality of the switches, the layout of the switches and platforms and the system behaviour used. Therefore, a standard railroad switch (see Section 2.1.2), an old map (see Appendix A.1) and the Route Locking and Sectional Release System (RLSRS, see Section 2.1.3) are chosen respectively.

The choice for the modelling and verification programs is predefined and based on

interest and experience. Therefore, UPPAAL and Supremica are used to complete the task.

1.2 Related Research

Verification of train systems is a current topic all around the world. However, the approaches and proofs differ. The following examples are only a few of many ideas available. It can be seen, that modelling and verification has not been done previously with UPPAAL or Supremica.

Similar work has been done using k-induction. The system is first described using a Kripke structure and then transformed into a transition function, on which a bounded model checking and inductive reasoning can prove certain properties. This has already been implemented and tested for the Danish Railway system from the University of Denmark and the University of Bremen, Germany. The results included the successful verification of safety properties for controlling large networks of realistic size [10].

The safety critical research group from the School of Railway Engineering in Iran has used the NuSMV, a symbolic model checker, to model and verify the interlocking control tables. The contents of the table and the behaviour of the train was analysed for conflicts. The tool set developed successfully minimizes the human interference in design, development and verification of the control table [11].

The safety properties of the European Train Control System are verified using compositional verification rules based on the Weakly monotonic time extension of DC by the Laboratory of Rail Traffic Control and Safety, Beijing Jiaotong University, Beijing. The proof divides the problem up into smaller sections which simplifies the system at hand. Resulting in a successful compositional verification of the system for controllability, reactivity and safety [12].

1.3 Outline

Chapter 2 describes the functionality of the European Train Control System (ETCS), automata and their properties. This is followed by Chapter 3 consisting of tutorials for the programs Supremica and UPPAAL which are used for the modeling and verification of the system. Next the modelling of the system in both programs is described in Chapter 4, followed by the verification of the models in Chapter 5. Thereafter the modelling and verification of the Gothenburg train station is described Chapter 6. Lastly, the conclusions drawn are explained in Chapter 7 and possible future work in Chapter 8.

2
THEORETICAL BACKGROUND

This chapter explains the required theory necessary to understand the thesis. First the fundamentals of railways are described, followed by automata theory and ending with formal verification.

2.1 Fundamentals of Railways

Understanding the fundamentals of railways and switches is essential before modelling the system. Therefore, this section provides the basic knowledge needed to understand the task and the models.

2.1.1 European Train Control System

The European Union (EU) started the European Rail Traffic Management System (ERTMS) project, a Europe-wide standard for signalling, to increase the competitiveness of cross-border operation and signalling of trains. The ERTMS consists of three parts: the Global System for Mobiles - the Railway (GSM-R), the European Traffic Management Layer (ETML) and the European Train Control System (ETCS) [2].

The ETCS is a system which uses signals to control the trains and prevent collisions. It requires equipment on the track and a controller in the train itself, as seen in Figure 2.1. The Eurobalise on the track reports the position and the signal state. It gets its information from the interlocking which processes the track release and sends a signal. The train has a GSM-R antenna, a ETCS computer and a receiver. The receiver processes the information received from the Eurobalise and the ETCS computer reacts accordingly [2].

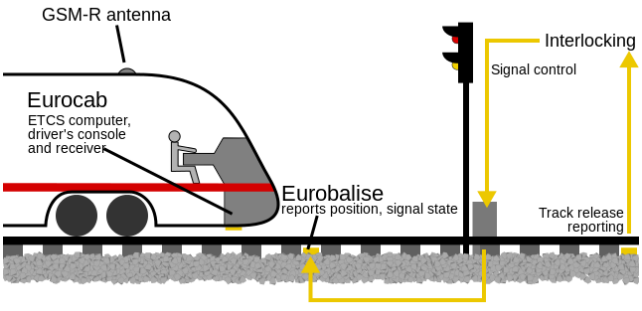

Figure 2.1. ETCS components [2].

Figure 2.2 shows a lane divided into multiple blocks with two trains on it. When a train enters a block, the light turns red and hinders the following train to enter this block. This means, that the train has to have stopped by the time it reaches the red block. To achieve this, a braking trajectory is calculated by the ETCS computer, and is updated regularly [2].

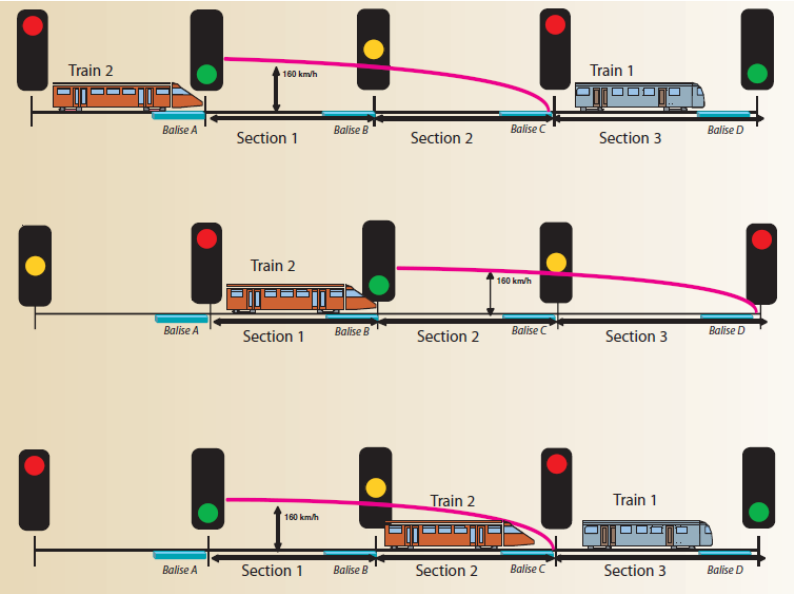

Figure 2.2. ETCS functionality [2].

2.1.2 Railway Switches

A railway switch is made out of three major parts: the point blades, the diverging route and the mainline route, see Figure 2.3. The point blades are the only movable parts and can be switched manually or automatically. The non-movable parts of the switch are called the stock rails. To make sure that the point blades can only touch one side of the stock rails at a time, a stretch bar is holding the blades together at a certain distance, see Figure 2.4.

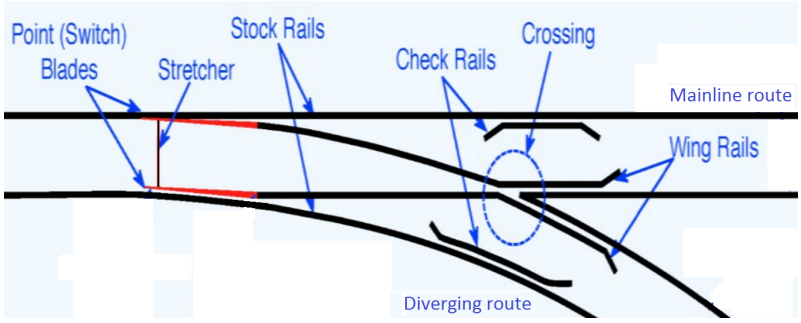

Figure 2.3. An overview of a railroad switch [3].

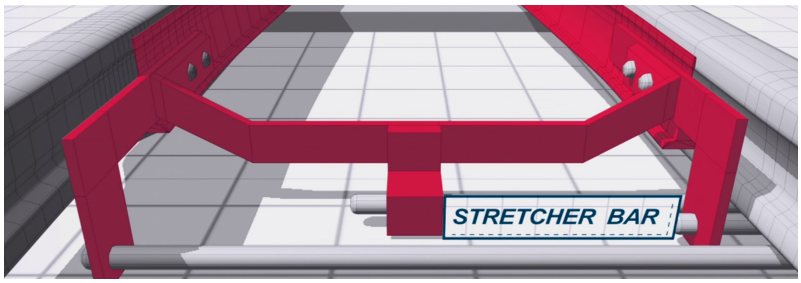

Figure 2.4. Closeup of the stretcher bar [3].

The direction of the train is decided by the position of the point blades. When a train approaches, only one side of the point blades are used (see Figure 2.5) along with one side of the stock rails (see Figure 2.6). When the train passes the crossing there is a small gap in the rail. To make sure that the train does not derail, there are Wing rails and Check rails on each side to lead the train in the right direction [3]. Unless a switch is locked, the direction of the point blades are irrelevant when a train comes from either the mainline route or the diverging route. The train will force the point blades in the desired direction [13].

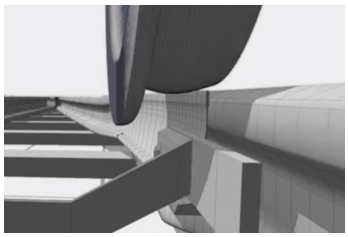

Figure 2.5. Pointblade [3].

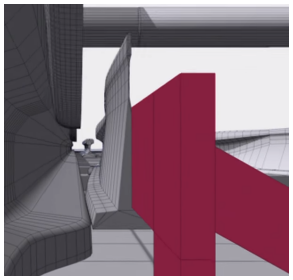

Figure 2.6. Stockrail [3].

2.1.3 Train Station

A common system for train stations is called the Route Locking and Sectional Release System (RLSRS). This means that a train sets an inbound route to the train station by claiming all the switches necessary to reach the desired platform and the platform itself. After the train has passed a switch it will release it for other trains to claim [14].

When a train is near a station the speed is usually very low, because of this the ETCS described in Section 2.1.1 will behave slightly different. The braking trajectory will be disregarded because of the short braking distance.

2.1.3.1 Gothenburg Train Station

The Gothenburg train station is the second largest train station in Sweden, serving 27 million passengers per year. Figure 2.7 shows the railway lines leading into the Gothenburg train station and to the 19 platforms.

Figure 2.7. Gothenburg train station [1].

2.2 Automata

Systems can be modelled and verified using automata (see Definition 1) which are, simply put, machines that change from one *state* (often graphically represented by circles) to another by the means of *transitions* (often depicted as arrows). A state symbolises a certain configuration or situation of a system in which specific rules, policies and physical laws hold. Each transition is associated with an event which represents an incident that changes the state of the automaton. The initial state, the state in which an automaton starts in, is indicated by a transition without an origin [15].

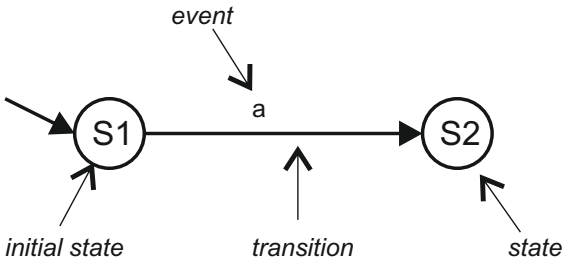

Figure 2.8. An example of an automaton.

Definition 1 *An automaton is a tuple* $\mathcal{A} = \langle Q,\ \Sigma,\ \delta,\ q_0 \rangle$ *with*

- Q, *a finite set of states*
- Σ, *a finite set of events (i.e. the alphabet)*
- $\delta \subseteq Q \times \Sigma \times Q$, *the transition function which describes, for each state and event, the next state after a certain event*
- q_0, *the initial state of the automaton*

2.2.1 Extended Finite Automata (EFA)

An extended finite automaton works exactly as an automaton, however, has some more functions in the transitions. While a common automaton has transitions that can only be associated to events, the EFA can use so called trigger conditions (guards). When a transition is fired, the EFA can also perform data operations (actions) [16]. This means that guards and actions can be associated with transitions e.g. a transition can only happen if the guard $x == 1$ is fulfilled and the action sets $y = 1$.

2.2.2 Marked/Forbidden States

A `marked state` is a state that is desired to be reached e.g. when a certain process is completed. It can either be graphically represented by a filled-in or a double circle. On the other hand, a `forbidden state` is a state which should not be reached and is graphically represented by a cross going through a circle [15].

2.2.3 (Co-)Reachability

A state is defined as `reachable` when a path of events from the initial state exists which leads to that state. If all states in an automaton are reachable, the automaton is `accessible`. If there exists a path from a state to the marked state, the state is `coreachable` and if all states in an automaton are coreachable, the automaton is `coaccessible`. An automaton is said to be `trim` if it is both accessible and coaccessible [15].

In Figure 2.9 the unreachable state (red) is still coreachable because a path to the marked state (grey) exists. On the other hand the green states are not coreachable but reachable.

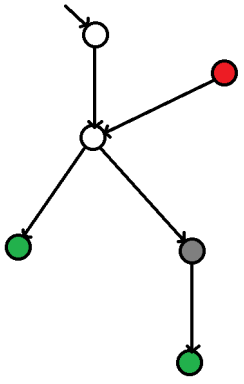

Figure 2.9. Non-reachable (red) and non-coreachable (green) states.

2.2.4 Plants and Specifications

Plants and specifications are two different kinds of automata. Physical systems are always modelled as `plants`. The `specification` is, as the name suggests, one or more automata, that specifies how the plants are supposed to behave. An example is a keyboard modelled as a plant. When letter `F` is pressed the letter `F` will appear on the screen. The specification can control in which order and what letter should appear on the screen [15].

2.2.5 Synthesize and Synchronize

Automata can be `synchronized` which means, that common events must happen simultaneously while other, uncommon events, can occur within the restrictions of the corresponding automaton. `Common events` occur when the automata have the same events in their alphabet, see Definition 1. This means that if a common event *press* can happen in only one automaton, it cannot occur since it is blocked by the other automata with the same event *press*. This feature is often used to control the behaviour of the plant. A `synchronized composition` is a representation of all the synchronized automata combined [15].

A `supervisor`, also called a controller, is generated by synthesising and it dis-/enables specific events in the plant. It is devised such that it, together with the plant, fulfills the specification, meaning, that at least one marked state can be reached and no deadlock exists without directly restricting uncontrollable events [15].

2.2.6 Deadlock and Livelock

A `deadlock state` is a state that has no outgoing transitions and therefore the automaton cannot execute any other transition. This is not a problem if the deadlocked state is also a marked state since this is the desired behaviour. On the other hand, a problem exists if the deadlocked state is not marked since a desired state cannot be reached. Similarly, a `livelock` occurs when an automaton is caught in a loop of unmarked states that it cannot leave [17]. In figures 2.10 and 2.11 the grey state is a marked state and represents a wanted deadlock while the red states indicate an unwanted deadlock and a livelock respectively.

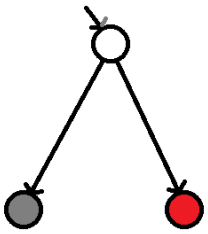

Figure 2.10. Automaton with an unwanted deadlock (red).

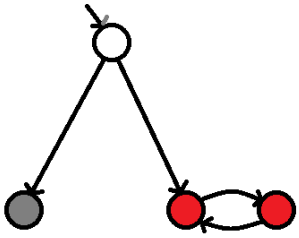

Figure 2.11. Automaton with a livelock (red).

`Blocking states` are states from which a deadlock and a livelock can no longer be avoided [17]. This means that, if a state's only transition is to an unwanted deadlock, livelock or another blocking state, that state is a blocking state as well. Two (or more) automata are `conflicting` if, and only if, their synchronous composition is blocking [15]. An automaton with no blocking state is referred to as `non-blocking`.

2.2.7 Uncontrollable Events

Physical systems can have uncontrollable events which need to be included in the plant. An `uncontrollable event` represents occurrences that the system cannot avoid or forbid. An example of an uncontrollable event is an opponent's move in a tic tac toe game. An `uncontrollable state` is where an uncontrollable event can occur in the plant even though the specification does not allow it. If a supervisor can reach an uncontrollable state, the system is uncontrollable [17].

2.2.8 Non-deterministic Automata

When an automaton cannot determine the end state after a specific sequence of events, it is referred to as `non-deteministic`. This normally occurs if there are multiple initial states or multiple outgoing transitions from the same state to different states, labeled by the same event [17].

2.3 Formal Verification

Formal verification is (dis-)proving the correctness of a system, based on a specified property or formal specification [18]. The most common properties, according to [19], to verify a state or a particular situation are the following:

- `Reachability` - can be reached

  For example: "$x < 1$ is always true", "the state $S1$ can always be reached" or "$x == 1$ always holds when in state $S1$".

- `Safety` - can `never` be reached (i.e. the negation of reachability)

  For example: "$x < 1$ can never happen", "memory overflow can never happen" or "both processes are never in the critical section".

- `Liveness` - will ultimately occur

  For example: "the elevator will arrive eventually, if it is called", "$x == 1$ will ultimately be satisfied" or "there will be sunshine after the rain".

- `Fairness` - will (or not) occur infinitely often

For example: "the door will open infinitely often" or "if something is requested infinitely often, it will be granted infinitely often".

- `Deadlock` - no transition is possible (i.e. the system cannot run indefinitely)

2.3.1 The State Space Explosion Problem

The most common problem when trying to verify systems is called the state space explosion problem. This means that the amount of states to verify will exponentially grow when adding models. For example: if an automaton has two states, `true` or `false` the total amount of states is $2^1 = 2$. When two of these automata exist, the states increase to $2^2 = 4$ and for n automata there are 2^n states. For $n = 21$ there are over a million states and for $n = 30$ there are over a billion. Now also consider if the amount of states per automata increases. Say the automaton has `true`, `false` and `maybe` as states. Then the base changes to 3^n and so on. Suddenly n is only 19 for the states to go over a billion. If a system becomes too large, the verification time will increase to over a lifetime. Therefore, it is important to keep the models as small as possible.

2.3.2 Verification Algorithms

There are many verification algorithms based on the automata theory, however, only the ones used are introduced here.

2.3.2.1 Monolithic

The monolithic algorithm verifies the model as whole without reducing it. This means that every state is checked for the specified property [17].

2.3.2.2 Compositional

One of the algorithms is called "Compositional" and can be used to verify nonblocking. Therefore the controllability of the events is unimportant so that specifications can be regarded as plants. One benefit of this algorithm is how it deals with state space explosion by using the monolithic approach and abstracting at each step. Before synchronizing, all components are abstracted and then composed [20].

2.3.2.3 Partial Order

In order to understand this algorithm, it is important to define both independence and ample sets. The relation between events is known as independence and is used to identify redundant states in order to eliminate them. Ample sets are obtained when reducing automata and including the enabled transitions which are further used for synchronous composition. The algorithms vary depending on if controllability or conflict check are being conducted. Controllability is checked at every state before determining the ample set. Nonblocking requires the whole synchronized automaton to confirm that all reachable states reach the marked states [21].

2.3.3 Temporal Logic

Some tools, like UPPAAL, use `temporal logic` to specify properties for verification. By using the logical operators,

$$AND \wedge$$
$$OR \vee$$
$$NOT \neg$$
$$IMPLIES \rightarrow$$
$$EQUIVALENT\ TO \leftrightarrow$$

combined with the new `temporal logic operators`,

$$ALWAYS \Box$$
$$EVENTUALLY \Diamond$$
$$NEXT \bigcirc$$
$$UNTIL\ \mathcal{U}$$

it is possible to specify different properties.

Consider a system describing an elevator, temporal logic can be used to specify desired properties. When a person presses a button the elevator is supposed to arrive at that floor at some point in time. This can be specified as:

$$BUTTON2 \rightarrow \Diamond FLOOR2$$

If this expression is true, the elevator will eventually arrive at the second floor if someone has pressed the button. This means that, from a state where the button for the second floor has been pressed, a state where the elevator is at the second floor will eventually be reached. This statement does not cover everything though. It is important to know that it was not a coincidence that the elevator arrived at the second floor. One has to make sure that every time the button is pressed, the elevator will arrive. Therefore the specification is changed to:

$$\Box(BUTTON2 \rightarrow \Diamond FLOOR2)$$

Next, if someone presses a button, the light should be lit until the elevator reaches that floor.

$$\Box(BUTTON2 \rightarrow LIGHT2\ \mathcal{U}\ FLOOR2)$$

or both statements can be combined to:

$$\Box(BUTTON2 \rightarrow (\Diamond FLOOR2 \wedge (LIGHT2\ \mathcal{U}\ FLOOR2)))$$

Lets say that the elevator has a priority floor five and when that button is pressed it will immediately head to that floor. This can be expressed as:

$$\Box(BUTTON5 \rightarrow \bigcirc TO5)$$

There are two more well known expressions in temporal logic which are the two combinations between $\Box$ and $\Diamond$. $\Box\Diamond$ means that something always eventually will happen, this can be translated to `repeatedly`. $\Diamond\Box$ on the other hand means that at some point in time something will stay the same forever. This can be translated to `persistently`.

3
TOOLS

The following chapter describes the software tools used to model and verify the train station: Supremica and UPPAAL. Lastly, these tools are compared to other available tools.

3.1 Supremica

Supremica is a software tool for modeling and manipulating discrete event systems in the form of Finite Automata and Extended Finite Automata. The tool is jointly developed by Chalmers University of Technology in Gothenburg, Sweden, and the University of Waikato, Hamilton, New Zeeland. Supremica is a merge between what was originally a tool mainly focused at editing and simulating automata, called Waters (Waikato Analysis Tool for Events in Reactive Systems), and the original Supremica which had no real user interface. Though the two tools slowly merge into the Waters/Supremica tool kit, the old heritage still shines through in the three tabs, Editor, Simulator and Analyzer. Supremica implements state-of-the-art algorithms for verifying properties, and synthesizing supervisors for systems of huge state-spaces [20] [21] [22].

Figure 3.1 shows Supremica's GUI after starting the program. The menu-bar can be seen at the top with a shortcut menu underneath. The main tabs, Editor, Simulator and Analyser, are located to the left with each have sub-tabs described later on. At the bottom a console can be found which gives information to the program itself, summarizes the results from the analyser, states bad traces and much more. Currently, the Editor and the Components tab are active which allows editing of automata in the "New Module" section when created.

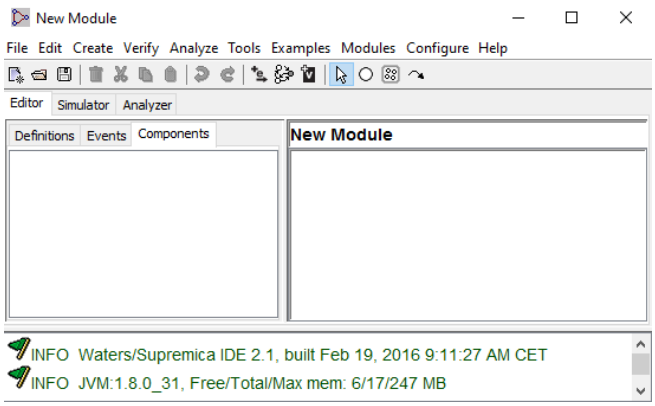

Figure 3.1. Supremicas GUI.

3.1.1 Editor

The tabs "Definitions", "Components" and "Events" belong to the "Editor"-tab. Constants and aliases are created in "Definitions" while plants, specifications and variables are defined in "Components" and events are created and edited in "Events".

3.1.1.1 Definitions

When using a value multiple times or to keep the model neat, it is important to use constants. Constants are created by defining a name and an expression, see Figure 3.2. This expression can either be a number or an array, for example, a number range (e.g. 0..3) or a set in hard brackets (e.g. $[apple]$ or $[apple, pear]$).

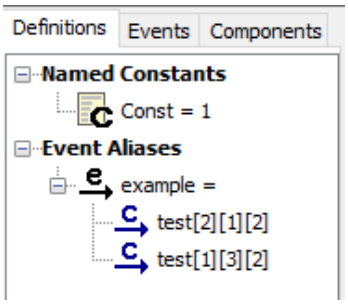

Figure 3.2. Example: constant and alias in Supremica.

A possibility to reduce large amounts of events on a transition is to use aliases. Aliases are a collection of events. One way to group multiple events is to have the box "Use simple expression" unchecked and the events can be pulled into the alias as seen in Figure 3.2. The alias then can be dragged onto the desired transition. Note that all events in the alias do not happen simultaneously but rather one of the events triggers the transition. Another possibility is to keep the box checked and use foreach-loops, see Section 3.1.1.3.

3.1.1.2 Components

The components section is where the plants, specifications and variables are created. Plants and specifications are created by using Supremica's Graphical User Interface (GUI) where states can be added and transitions can be created by dragging arrows between the states. Events are created in the events tab and dragged onto the transitions. By right clicking on the states it is possible to define the states as initial, marked or forbidden. Guards and actions can be added to a transition by right clicking it and selecting "Edge Properties...".

Variables are very useful because the value can be used as a guard and changed with an action. When adding a variable, the name, type and initial value are defined. The type can either be a number range (e.g. 0..3) or a set in hard brackets (e.g. $[busy, ready]$). The initial value is defined by a logical expression, like $var == 0$ or $var == busy|var == ready$ for multiple initial values.

It is possible to add a `Blocked Events List` to an automaton by right-clicking on the editor section of a component and choosing "Add Blocked Events List". Next, drag in all appropriate events. If the list is added to a plant, the events are disabled and cannot occur. If the list is added to the specification, the events are not allowed to happen however could, if they are uncontrollable.

3.1.1.3 Foreach Block

Supremica offers the possibility to create several instances of a specification, plant, constant, alias and variable by using a foreach-block. Such a block is initiated by defining the counter variable (e.g. i), its range (e.g. 0..10) and optionally a guard as a Boolean expression (e.g. $i == 4$). The guard only allows the transition to occur when i has the value four. When a plant/constant/specification/variable/alias is defined, the same counter variable is added to the name in hard brackets (e.g. $test[i]$) (see figures 3.3 and 3.4).

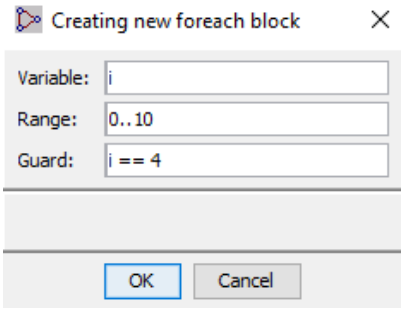

Figure 3.3. Example: Creating a foreach-loop in Supremica.

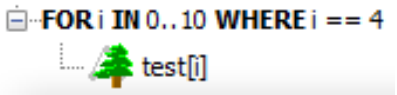

Figure 3.4. Example: finished foreach-loops in Supremica.

3.1.1.4 Events as Arrays

Events can be defined as an array by going to "more options" when adding a new event and defining one or several array ranges. Figure 3.5 shows an event called `test` which is an array of the size $3 \times 5 \times 6$. When adding this event to a specification or plant, the event appears in the local event column between the global events and current plant/specification as seen in Figure 3.6. If no specific position in the array is defined (e.g. $test$), it means that all events are on the transition. Therefore, in order to define a specific event, one must add hard brackets defining the exact position by renaming the local event (e.g. $test[1][3][2]$). When using foreach-blocks, numbers can be replaced by appropriate counter variables.

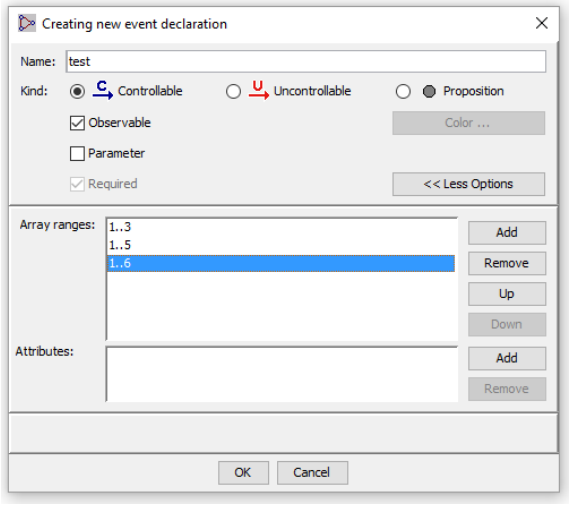

Figure 3.5. Example: Creating an event as an array in Supremica.

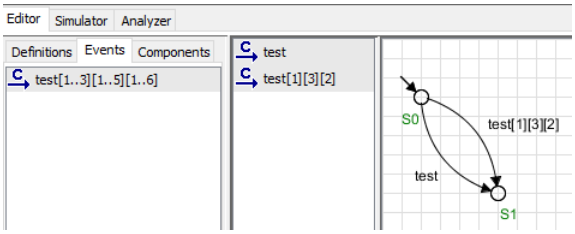

Figure 3.6. Example: Including array events in Supremica.

3.1.2 Verify Menu

There already exists the possibility to verify in the editor by using the checks available under the "verify" menu. The checks are for conflict and controllability. The `conflict check` makes sure that all states can reach a marked state and that no forbidden states can be reached. The `controllability check` makes sure that no uncontrollable state can be reached. Different algorithms/factories are available when checking and can be chosen under `Configure -> Options -> gui.analyser -> Model verifier factory used by Editor's Verify menu`. Not every check is available for every factory and therefore one might need to change factory depending on the check and on the size of the model. Some of the factories/algorithms in the tool are explained in Section 2.3.2.

3.1.3 Simulator

The simulator is very useful to do basic debugging and verification. The event-tab shows all events, if these are possible and why (see Figure 3.8) and by double clicking on the event, the according transitions are fired. The trace is saved and shown in the trace-tab (seen in Figure 3.7) where it is also possible to go back to a certain situation. In the Automata-tab all the plants and specifications are listed (see Figure 3.9) which are opened by double clicking and show the current state they are in.

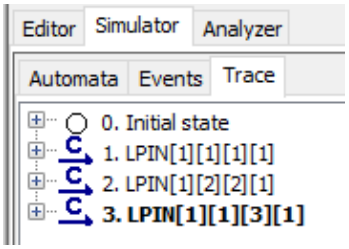

Figure 3.7. Example: Trace when Simulating.

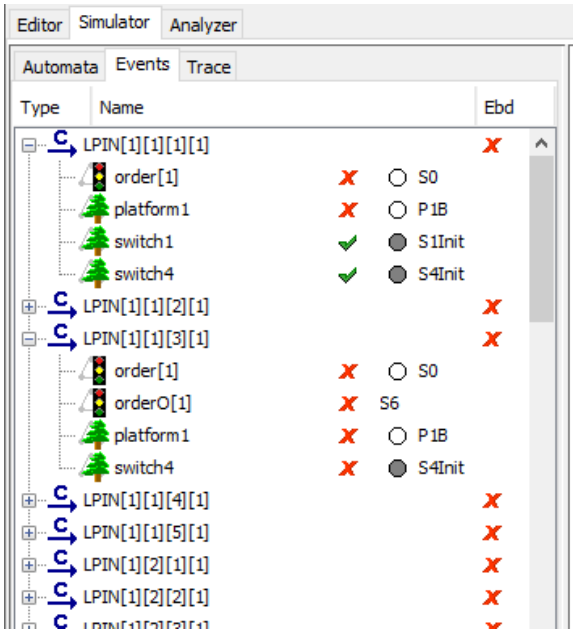

Figure 3.8. Example: Events in the Simulator.

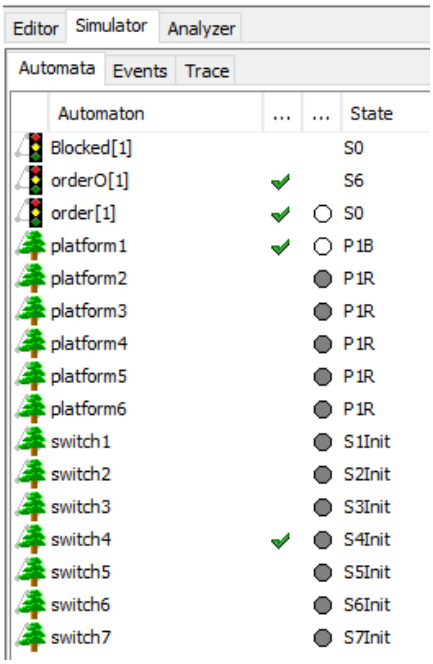

Figure 3.9. Example: Components shown in Simulator.

3.1.4 Analyser

In the analyzer, the plants and the specifications can be synchronized, synthesized and verified. They can also be graphically layed out, automatically, if the Graphviz [23] visualization package is installed. Note that Supremica currently only works with the Graphviz 2.28 version. When clicking the analyzer or the simulator, the plants and specifications are translated into objects by the compiler, if no errors exist.

3.1.4.1 Verify

To verify a system one must choose the plant(s), specification(s) and/or supervisor(s) which should be verified, right click and click "Verify". It is then possible to choose which property to verify, e.g. blocking, controllable and which algorithm to verify with. Also, if the property is not satisfied, a trace to a bad state can be generated by checking the "Show trace to bad states"-box. It is shown in the console and is the combination of events (string) that results in a state not satisfying the property. An example is a trace to a deadlocked state when trying to verify for non-blocking.

3.1.4.2 Find States

It is possible to find states in either a plant, specification or supervisor. This is useful for manual verification in order to check if a specific state does or does not exist. Under `Find States -> Free Form` it is possible to determine the state or a composition of states which should be checked using a specific syntax based on regular expressions.

The expression should start and end with .* where the dot symbolises "any sign" and the star "combination", thus together forming "any combination of signs". It is

possible to use an | (or-sign) and $[1-3]$ (range $1-3$). For example, $.*S[1-3] \mid S5.*$, where states with the state-labels $S1$, $S2$, $S3$ or state $S5$ are searched for.

3.1.5 Opening Files in Other Programs

It is possible to open and change the content of the .wmod files using any XML-aware editor, like MS Excel. This is extremely useful when wanting to use "Find & Replace" and it opens up many possibilities for the user but is also advanced.

3.1.6 Common Problems

When modelling with Supremica there are some problems which arise frequently. Therefore some solutions to these common problems are presented.

3.1.6.1 Increasing the Memory

When starting Supremica, the program shows the amount of memory as a second `INFO` string as seen in Figure 3.10. To increase the memory in Supremica, the memory of the Java machine has to be increased. This is done in `Windows` by creating a text file named "Supremica.bat" in `Supremica/dist` with the following content:

```
Java -Xms1024m -Xmx1024m -jar Supremica.jar
```

with $-Xms$ being the initial memory, $-Xmx$ the maximum amount of memory and m the amount of megabyte. Note that all numbers have to be factors of 1024. If it is not possible to save the *.bat file, include the full path of Supremica (right-click `Supremica.jar -> Properties -> Location:`) and paste it before "Supremica.jar" in the file.

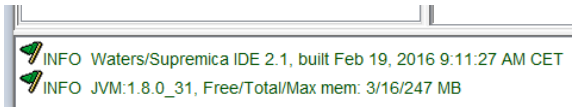

Figure 3.10. Example: Memory available.

If this does not work, it is possible to use the "Waters"-Supremica edition.

3.1.6.2 (Non-)Deterministic Variables

Supremica handles the variables non-deterministic by default. This means, that if their value is not specified for a transition, all values are possible. In order to change this, the following options need to be changed: In `Configure -> Options`, activate `Use normalizing EFSM compiler` and deactivate `Use per-event alphabet when compiling EFSM`. If this option is not available, a more recent Supremica version has to be used.

3.1.6.3 "Event removed due to Optimisation"

Supremica has an optimisation algorithm to reduce the amount of time when compiling. Due to this, events might be removed and therefore cannot be triggered in the simulation and appear in light gray. If this optimisation is not wanted, go to `Configure -> Options -> gui` and remove the tick at "Remove redundant events, transitions, and components when compiling".

If this does not solve the problem, consider if the events are unable to fire due to the guard always being false. Consider the following example: a variable is defined as `platform=[ready, busy, train]` and on a transition in a plant the platform is set to *train*. This is not a problem, however the plant is in a foreach-loop with the variable *train* being the counter variable ranging from $1 - 5$. Therefore, when the platform is set to *train*, it is actually set to a value instead. When *platform* on a transition is then checked for being either *ready*, *busy* or *train* it will never be true because *platform* is a value.

3.1.6.4 "State encoding requires x bits, 64 is the maximum!"

When using `Verify -> Conflict/Controllability Check` the amount of bits needed can extend the maximum of 64 bits. In order to solve this, a different model verifier factory/algorithm is used. This can be changed at `Configure -> Options -> gui.analyzer -> Model verifier factory used by Editor's Verify menu`. The factories $Native$, $PartialOrder$, BDD, $TRCompositional$ and $Compositional$ have no limitations as to the number of bits. Note that this only works on Windows because dynamic link libraries must be available. Also, not all algorithms support other checks and therefore if another kind of check (e.g. controllability check) is done the algorithm might need to be changed again.

3.2 UPPAAL

UPPAAL was developed by the Uppsala University in Sweden and the Aalborg University in Denmark and was first released in 1995. It is used to model, simulate and verify real-time systems and has a friendly GUI. Drawbacks include that UPPAAL uses binary synchronisation, which can be restrictive [19]. UPPAAL does not work with plants or specifications however rather with parametrized templates, where in each an automaton is created and local declarations are defined. It does not distinguish between controllable and uncontrollable events.

UPPAAL's GUI when starting the program is shown in Figure 3.11. At the top, a menu-bar, some shortcuts and the main tabs Editor, Simulator, ConcreteSimulator, Verifier and Yggdrasil can be found. Currently, in the Editor tab, the current project with the global declarations, a template and the system declarations is seen on the left. To the right the name and parameters of the template are defined while underneath, where an automaton is edited, an initial state of an automaton can be seen. At the bottom, after compilation, the position and description of errors are displayed.

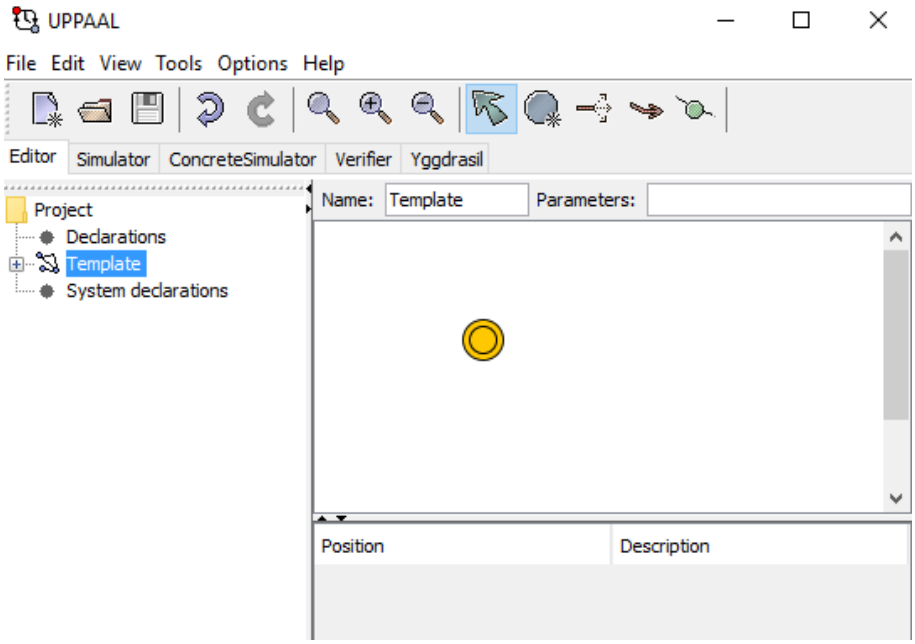

Figure 3.11. UPPAAL's GUI.

3.2.1 Editor

Under the editor-tab, it is possible to define global declarations ("Declarations"), insert Templates in which the automata are defined (`Edit -> Insert Template`) and to declare the systems ("System declarations"). The local declarations of the template can be found when expanding the system.

3.2.1.1 Declarations

The syntax for both local and global declarations are equivalent and can be used to define bounded integers (max range is $[-32768, 32767]$), channels, arrays, records, clocks, types and methods. As in programming languages, variables can be read, written and used for simple arithmetic operations [4]. Listing 3.1 shows and explains the syntax for common declarations. All clocks (global or local) progress in the same pace and are very important for the verification of real-time systems.

Listing 3.1. Syntax for Declarations in UPPAAL.

```
const int a = 2; // constant integer with value 2
bool b[5], c[3]; // two boolean arrays with 5 and 3 elements
int [0, 4] d = 2; // integer variable with range 0 to 4 initialised to 2
int e[2][3] = {{1,2,3}, {4,5,6}}; // multidimensional array with set size and initialised
clock f, g; // clocks f and g
chan h; // channel h
struct {int i; bool j;} s1 = {2, true}; // a struct where members i and j are initiated

typedef int [1, nTrain] train_t ; // integer set train_t of size 1 to nTrain
```

To create multiple instances of a system, one needs to first use *typedef* to create a set (e.g. $typedef\ int[1, nTrain]\ train\_t$) and add the set (e.g. $const\ train\_t\ id$) to the parameters of the template. The variable id can then be added to the channels, variables, etc (if they were initiated appropriately e.g. $chan\ h[train\_t]$).

There is the possibility to define methods which can be used as guards or assignments. For a method to act as a guard, it has to return a value or `true/false` (i.e. declared as a `int` or `bool`), otherwise it is defined as a `void` to be used as an assignment.

Listing 3.2 shows an example method with a for-loop and an if-then-else-statement to show the required syntax.

Listing 3.2. Syntax for a for-loop and an if-then-else-statement in UPPAAL.

```
void example(){
    if (x == k){
        y = 7;
    }else{
        y = 5;};

    for(i = 1; i <= k; i ++){
            z[i] = 1;};
    }
```

For communication between sub-systems, channels are used. There are three different kinds of channels: normal, urgent and broadcast (see Listing 3.3). The normal

channel works as handshaking i.e. one channel sends and one receives. The sending channel is written with an exclamation mark `a!` while the receiver is indicated with a question mark `a?`. Notice that expressions are first updated on the sending transition and then on the receiving.

There are two kinds of sending channels: normal channel and broadcast channel. When the normal channel send a signal only one, randomly selected, receiving channel will listen and fire. If the signal on the channel cannot be received, it cannot be sent. A broadcast channel has one sender and multiple receivers, however the sender is indifferent to whether a receiver is listening. Important is that no clocks can be used as guards on receiving transitions and updates for the receiving channels are executed from left-to-right of the processes defined in the system definition.

Listing 3.3. Syntax to Initialise Channels in UPPAAL.

```
chan a;
urgent chan b;
broadcast chan c;
```

When a state is reached with an urgent channel leading away from this state, the transition is enabled without delay. Only other transitions which require no time to pass may be executed first. Clocks may not be used as guards on either sending or receiving transitions.

In global declarations it is possible to prioritise channels, for example with `chan priority a < b, c;` where channel a has a lower priority than b and c. Similarly, systems are prioritised in the system declarations with, for example, `system P < Q, R;` where system P has a lower priority than Q and R [24].

3.2.1.2 Templates

When editing locations/states there is a `Location`, `Comments` and `Test Code`-tab. In `Location`, the name can be defined and an invariant added. An "Invariant" is a condition which has to be fulfilled before the state can be left. As seen in Figure 3.12, an invariant can be $x < 3$ with x being a clock. This means, that the state has to change before the clock reaches the value 3. The idea behind the "Rate of Exponential" is that when the delay has no upper bound (i.e. no invariant present), a concrete delay is chosen which is based on the exponential probability distribution. For example, if a rate of 4 is chosen this means that the average delay will be $\frac{1}{4}$ time units. The rate is proportional to how active the process is [24].

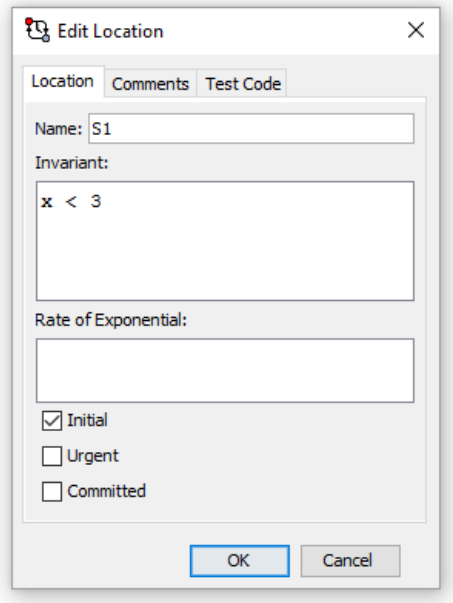

Figure 3.12. Example of a Location / State.

States can be defined as "Initial" and "Urgent" or "Committed". When a state is "Urgent", indicated with a "U" in the state, time pauses and the state must be left before the time continues. Therefore, only transitions which are time-independent are possible. Time also pauses in a "Committed" state, indicated with a "C" in the state, however only the transitions outgoing from that state are allowed. This is very useful for creating atomic sequences. Note that if multiple states are committed simultaneously, they will interleave [6]. Every template must have an initial state which is indicated with a double circle.

There are several possible labels for a transition: select, guard, sync and update. In `select` it is possible to create a local (i.e. for only the transition) variable or number. For example $test : int[1, 5]$ generates a random natural number between and including 1 and 5 and saves it in the local variable *test*. A `guard` is an expression which has to be fulfilled in order for the transition to be taken e.g. $test == 2$. The next label is `sync` which is short for synchronise and refers to the signal on the channel which should be sent or received . Lastly, the `update` assigns a value to a variable e.g. $test = 5$. Table 3.1 shows the colors the labels are represented by in UPPAAL. This increases the readability and understanding of the functionality of the transitions.

select	light brown
guard	green
sync	light blue
update	dark blue

Table 3.1. Colorcoding of the Transition Labels in UPPAAL.

Figure 3.13 shows an example edge label. A local, random number i (1 or 2) is generated and then the guard $i == 2$ & $j == 3$ is tested where j is a global variable declared in another automaton. If this guard is true, the channel *test*! is sent and

the value of the global variables y and j are assigned appropriately.

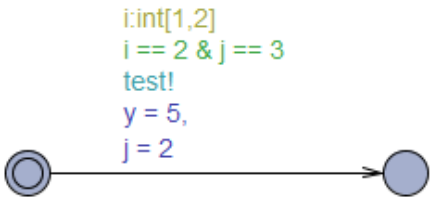

Figure 3.13. Example Edge Label in UPPAAL.

It is possible, by using a trick, to have an *urgent transition* using an urgent channel. This is done with a one-state automaton with a self-loop with the urgent channel and receiving this channel with a guard on the transition which should be urgent (see Figure 3.14).

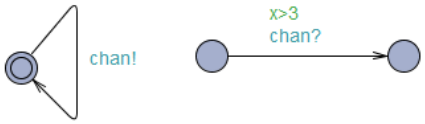

Figure 3.14. Example Urgent Transition in UPPAAL.

3.2.1.3 System Declarations

In `System Declarations`, the processes and templates which UPPAAL should consider need to be defined. The first example in Listing 3.4 shows how one declares the systems, the second shows how to create two processes identical to the template $A()$.

Listing 3.4. Syntax System Declarations in UPPAAL.

```
system test1, test2;

// Creating Identical  processes with one template
P1 = A();
P2 = A();
system P1, P2;
```

It is possible to declare a template as *dynamic* if it should be dynamically spawned. Parameters are limited to pass-by-value parameters or broadcasts channels. These templates are spawned by any other template with a transition with the update $spawnN$ where N is the dynamic template name. The dynamic templates end themselves with the update $exit()$.

3.2.2 Simulator

There are two available simulators: normal and concrete. The difference is that the concrete simulator includes time in the simulation. Figure 3.15 shows an example of the normal simulator. The section `Enabled Transitions` shows all possible transitions while `Simulation Trace` documents the transitions taken. The column in the middle shows the values of all local and global variables. All automata are shown in the section to the right, where the current states are marked red. The current transition choice is also indicated by a red transition in the corresponding automata. In the bottom right section, a trace shows the states the automata were in, based on the transitions taken.

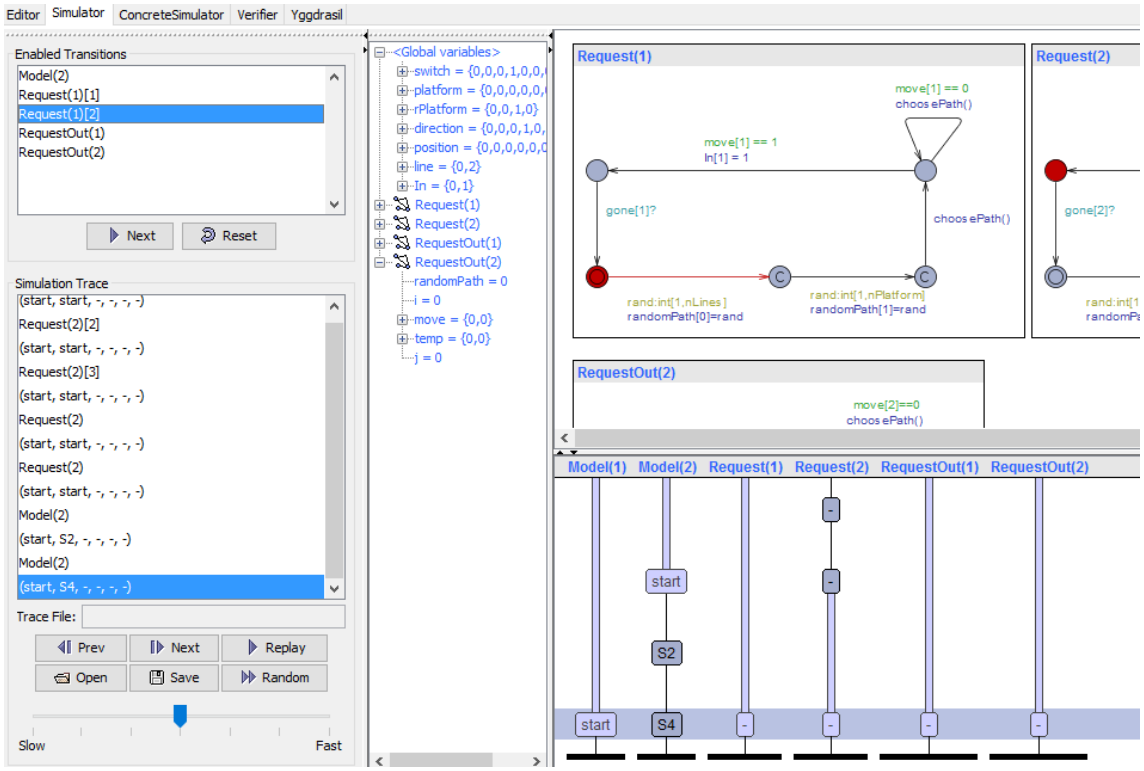

Figure 3.15. Example of the normal simulator in UPPAAL.

When choosing the transitions in the concrete simulator, the point of time in which the transition fires is defined by where the user clicks when activating the transition (see Figure 3.16). The exact time is shown under `delay` while the total simulation time can be seen in the simulation trace. The transitions may have different colors at different time spans. Red indicates that the transition should not, due to restrictions, be taken during that time. When green, the transition can be taken without violating any guards. If no color is present, the transition has to be taken before or after a specific time.

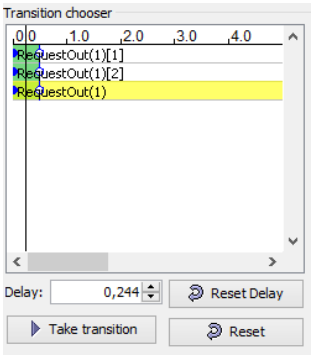

Figure 3.16. Example of choosing a transition in the concrete simulator in UPPAAL.

It is possible to use a Gantt chart to illustrate start and end times of processes, however it must first be defined in the system declarations. The line labels and the colors with the corresponding states need to be defined, as seen in Listing 3.5. For the systems `switch(1)` and all instances of `model`, some colors, depending on the current states are defined [24].

Listing 3.5. Example of Gantt Chart declaration in UPPAAL [24].

```
gantt{
    switch(1):
        Switch(1).Busy -> 0; // red
        Switch(1).Ready -> 1; // green
    model(i:id):
        Model(i).S1 -> 2; // blue
        Model(i).S2 -> 3; // magenta
}
```

3.2.3 Verifier

The verifier is used, as the name suggests, to verify the system. This is done by defining an appropriate logical expression in the `Query`-field, seen in Figure 3.17. Table 3.2 shows a summary of the queries available and which property they can be used to verify, while Figure 3.18 illustrates the queries. The yellow states are the states fulfilling ϕ and the dotted line is to illustrate that the path can be in livelock as long as the property is still fulfilled. A very useful property is `A[] not deadlock`, which tests for deadlocks. `deadlock` is expressed using a special state formula built into UPPAAL. If the query is proven to be true a button lights green, otherwise red (see Figure 3.17). When using abstraction and choosing specific options, the verifier may determine that the property is *maybe satisfied* which means, that due to the approximations the verifier cannot determine *true* or $false$ [4].

It is possible to see which trace caused a query to result negatively. This is done by defining if `some`, the `shortest` or the `fastest` trace should be shown in the simulator under `Options` and then `Diagnostic Trace`.

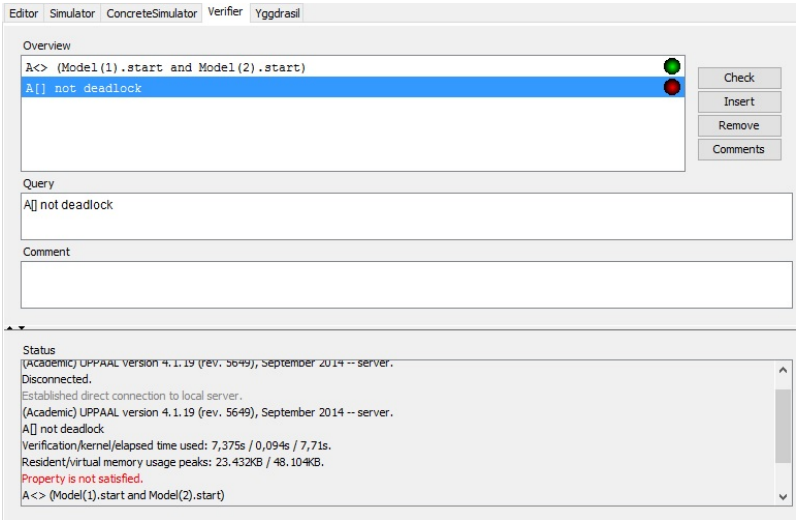

Figure 3.17. Example of the Verifier.

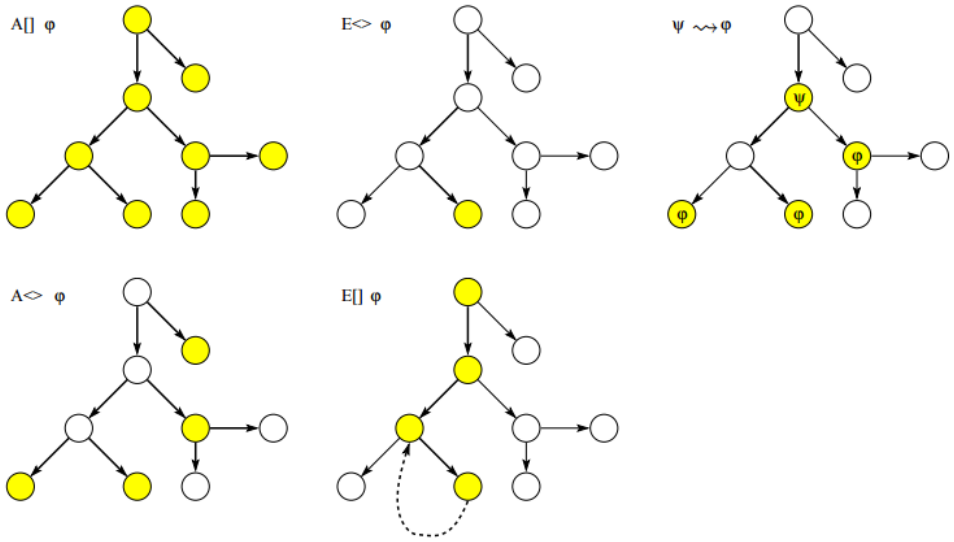

Figure 3.18. Query Language Illustrated [4].

Expression	Definition	Property
A[] ϕ	for all paths ϕ always holds	Safety
E<> ϕ	there exists a path where ϕ eventually holds	Reachability
$\Psi - - > \phi$	whenever Ψ holds ϕ will eventually hold	Liveness
A<> ϕ	for all paths ϕ will eventually hold	Liveness
E[] ϕ	there exists a path where ϕ always holds	Safety

Table 3.2. Query language with Ψ and ϕ as state formula expressions in UPPAAL [6].

3.2.4 YGGdrasil

`YGGdrasil` is used for offline testing. With an input of a deterministic model with no deadlocks, it generates traces from the model and converts them into test code. The traces are generated in three steps: `Query file`, `Depth Search` and `Single step`.

In `Query file` the input file in the `Verifier` is checked for reachability queries which are then executed. The resulting traces are used in `Depth Search` to perform a random depth first search. When no more coverage of the edges is possible, `Single step` checks the remaining edges [6].

3.2.5 Memory Issues

When verifying large models in UPPAAL, an error message may appear that states "Exhausted memory" and a link, see [25]. The memory is exhausted because of the state space explosion problem mentioned in Section 2.3.1. The link leads to a homepage with a conversation about this issue explaining that UPPAAL runs on 32-bit. This means that the maximum memory capacity of UPPAAL is 4GB of RAM.

There are multiple ways to reduce the size of the model:

- Reduce the amount of clocks
- Use committed states
- Reduce the amount of variables and decrease their specified range
- Avoid unbounded loops on integers

3.3 Additional Software Tools

There are multiple software tools which allow modelling and verification of systems. Table 3.3 summarises some of the features of the tools described below.

Symbolic Model Verifier (SMV) was developed at the Carnegie-Mellow University in Pittsburgh, USA, by K. L. McMillan under the guidance of E. M. Clarke. Due to the binary decision diagram (BDD) technology, it can verify very large systems. The input to the automata is based on shared variables and the automata are described textually. When verifying, a counter example is given if the property is not fulfilled [19]. There are two kinds of automata which can be modelled: a synchronous Mealy machine or an asynchronous network of abstract, nondeterministic processes. Both finite (scalars, booleans, and fixed arrays) and static, structured data types can be used [26].

An extension of SMV is the NuSMV2 which is an open source project and was developed by the University of Trento, University of Genova, Carnegie Mellon University and FBK-IRST. It combines BDD-based with SAT-based model checking and works with both synchronous and asynchronous finite state systems. Specifications can be done using Computational Tree Logic (CTL) and Linear Temporal Logic (LTL)

while heuristics partially control the state space explosion. A textual interface and a batch mode are available to the user. There are three possible simulation strategies: deterministic, random and interactive [27].

G. J. Holzmann at Bell Labs (Murray Hill, New Jersey, USA) developed the tool SPIN to simulate and verify distributed algorithms. Modelling is done with SPIN's specification language called Promela which can be compared to C programming with a few communication primitives. Communication between the processes is done using FIFO (First-In-First-Out), shared variables or rendez-vous. One of its key features are the available state space reduction methods: state compression, on-the-fly verification and hashing techniques [19].

VERIMAG developed the tool KRONOS which is used to analyse timed automata. Its model checking algorithm for timed temporal logic allows the verification of liveness properties. KRONOS is more for advanced users and has neither a simulator nor a GUI [19].

HYTECH was developed at the Cornell University and improved by the University of California, USA. It is used to analyse linear, hybrid automata which are implemented textually. HYTECH is the only tool handling parametric models which is useful for the design and verification of systems. There is no available simulator or verification using temporal logic and it can only handle small systems [19].

Tool	Simulator	System Size	GUI	Counter Example
SMV	no	large	no	yes
NuSMV2	yes	large	no	yes
SPIN	yes	large	yes	unknown
UPPAAL	yes	small	yes	yes
KRONOS	no	small	no	unknown
HYTECH	no	small	no	no
Supremica	yes	large	yes	yes

Table 3.3. Comparing Modelling and Verification Software Tools.

4
MODELLING

This chapter first describes how the model of the train station was simplified and then modeled. Two different approaches are modelled in UPPAAL and four in Supremica.

4.1 Simplification

The model of Gothenburg train station with 28 switches is large (see Figure 4.1) and therefore, in order to save time during the development phase, a smaller template model of the train station is initially used (see Figure 4.2). The train starts by entering from a line (circles without numbers) and then, depending on the specification, it arrives at the designated platform (squares). To do this, the train travels through switches (circles with numbers) that decide the direction for the train. Keep in mind that the train is using the RLSRS system described in Section 2.1.3. Therefore, the models created both for UPPAAL and Supremica will have a similar structure and some similar ideas.

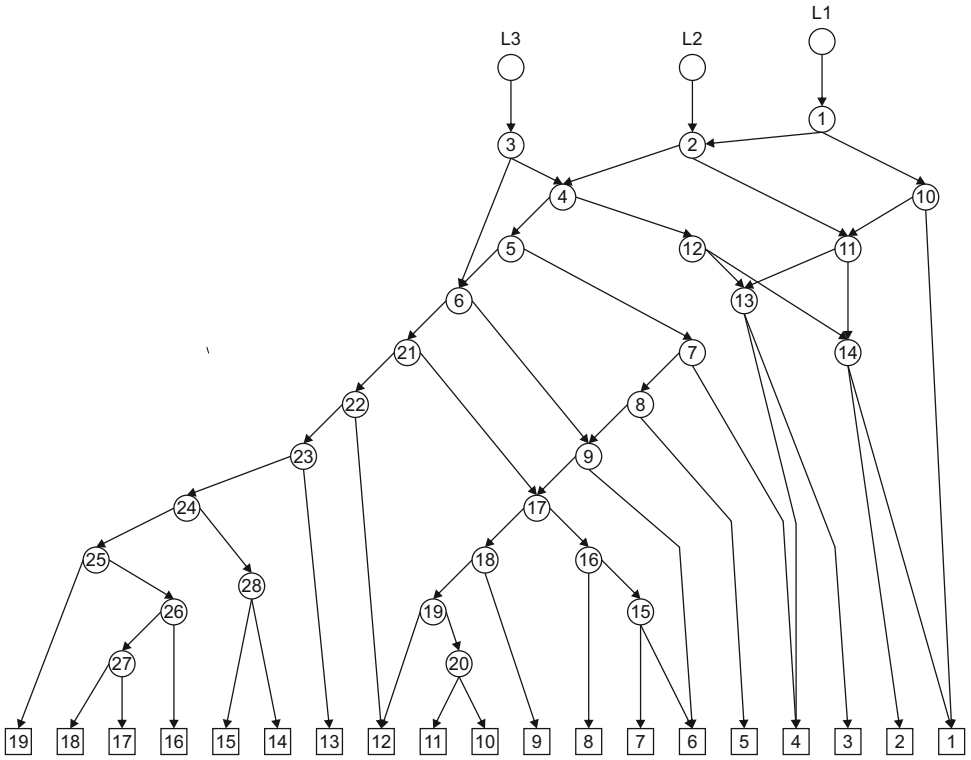

Figure 4.1. An overview of the train station.

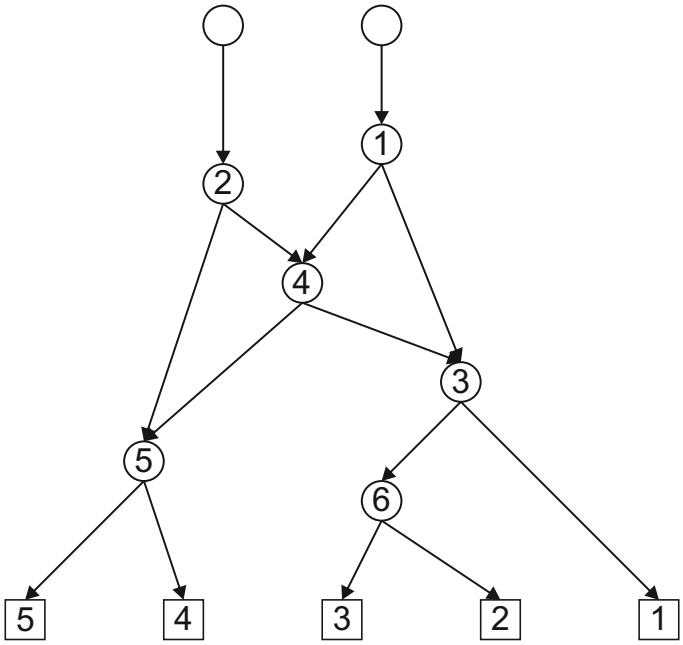

Figure 4.2. An overview of the simplified train station.

When creating switches, platforms and models of the train station, one model/-variable is made and duplicated depending on the amount of switches, platforms and trains respectively. In Supremica this is done with foreach-loops while in UPPAAL the appropriate parameters are given to the template as explained in Section 3.2.1.1.

4.2 UPPAAL

There are multiple ways to implement the automata to create a model describing the system. To show UPPAALs features, one approach will use a lot of UPPAALs different functionalities while the other approach will be more standard automata theoretic.

4.2.1 Approach 1: Standard approach

The first idea is to use the theory described in Section 2.2 rather than to exploit UPPAALs specific features.

4.2.1.1 Switches and Platforms

The switches are modelled as two 2-state automata, one for direction and one for their occupancy, see Figure 4.3. The switch can be either `ready` or `busy` and `left` or `right`. The platform is described by one 2-state automaton that has the states `ready` or `busy`. The channels are defined as receivers and the template parameter `[id]` is defined for the amount of models needed (trains, platforms, switches) as described in Section 3.2.1.1.

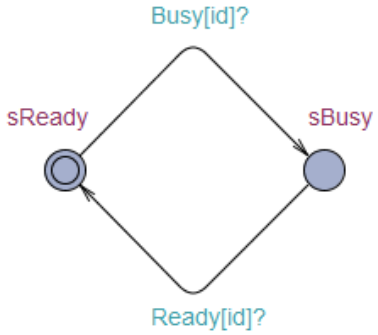

Figure 4.3. A 2-state automaton describing a switch occupancy - Approach 1.

4.2.1.2 The Trainmaker

Next an automaton that creates trains, defines a line and a platform is created, the `trainmaker`. By adding several outgoing transitions with different sending channels from a state, the model randomly selects a channel representing the incoming line. The `trainmaker` can only generate a new train and path if the maximum number of trains for the system has not been reached and all platforms are not occupied. The `trainmaker` also decides when a train is allowed to leave its platform, which can only happen if there is a train at any platform. Therefore, the variables `trainsAtPlatform` and `trainsInModel` are created. When the `trainmaker` asks for a train, it will determine which path the train should take. After that, it waits for a confirmation coming from the specification that a train has started to move until it allows another train to be created, see Figure 4.4.

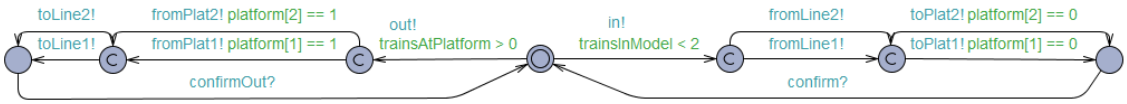

Figure 4.4. A `trainmaker` model with 2 lines and 2 platforms both for in and outgoing trains - Approach 1.

One specification is created for every possible line to platform combination. The `trainmaker` automaton sends a random line and platform combination using channels and only the appropriate specification listens and is allowed to execute. The use of broadcasted channels in the `trainmaker` and the selection of the route as shown in Figure 4.5 is done in various steps:

1. The `trainmaker` randomly determines that a train wants to enter the station, assuming the maximum amount of trains in the model has not been exceeded. It then randomly chooses from which line this train comes from, e.g. from line 1.
2. Due to the channel `fromLine1!` being broadcasted, all the listening receivers transition to their next state.
3. Next a platform is chosen, in this case platform 1, firing `toPlat1!` which only one specification listens to because no other specification fits this combination of line and platform.

4. The last step is to return the specifications that listened to the broadcasted event `fromLine1!` to their initial state. This is done by adding the broadcasted channel `confirm` confirming that an automaton has fulfilled the channels from `trainmaker` going from line 1 to platform 1. Therefore, all except that automaton, are back at their initial state and a train is added to the model.

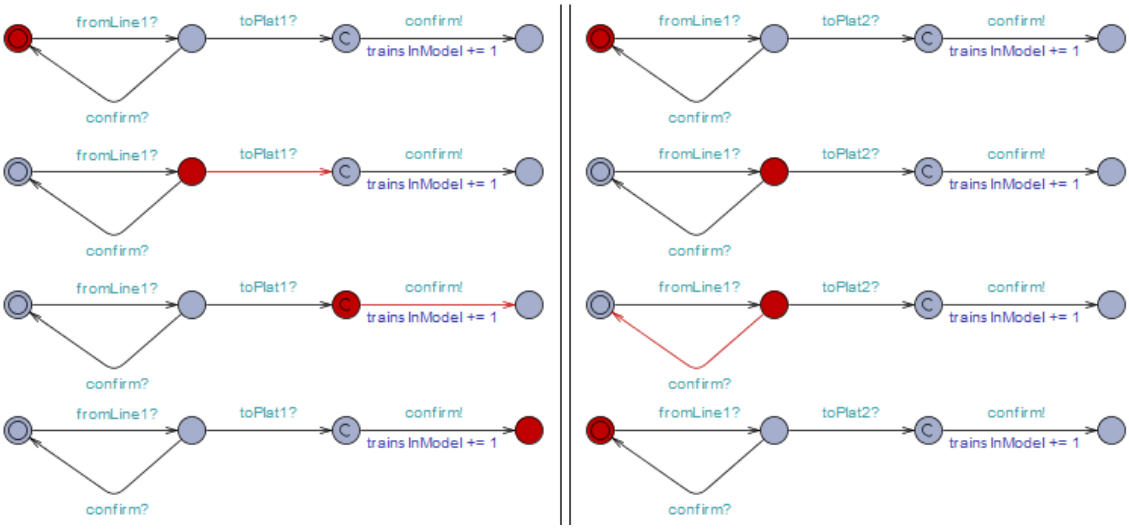

Figure 4.5. Steps taken in two specifications when the `trainmaker` creates a train coming from line one to platform one - Approach 1.

Observe that every step of this process is committed, meaning that the choice of path is done undisturbed from other channels. The same kind of process happens when the `trainmaker` asks a train to leave its platform.

4.2.1.3 Specification

After a route is selected, the model checks if the switches required are available. When using automata for the switches' occupancy, it is not possible to change all the switches simultaneously. However, by using committed states, it is possible to simulate this change from the model's point of view. The problem when using only automata is that the model becomes messy and large, as seen in Figure 4.6. The model checks the availability of the switches separately by using the channel that sets the switch to `busy`. Due to the property of the channels, the transition can only happen if the receiver accepts the channel. Therefore, the transition cannot happen if the switch is already `busy`. If a route has many switches, it could occur that all the switches except the last are `busy`, this means that the model has to set the other switches back to `ready` again. If all the switches are `ready`, the direction of the switch will have to be changed in several events as well.

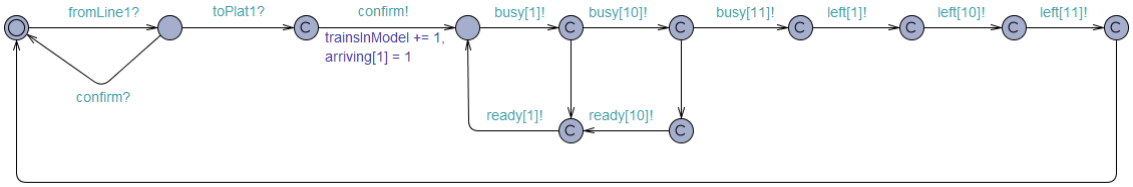

Figure 4.6. The specification of a train using 3 switches with switches and directions as automata - Approach 1.

In order to reduce the model, the automata for the direction and the occupancy of the switches are removed and replaced by variables (seen in Figure 4.7). When a route has been selected, the specification checks if the switches required are available, sets them to *busy* and their directions accordingly. To avoid similar problems later on the platform is also changed to a variable.

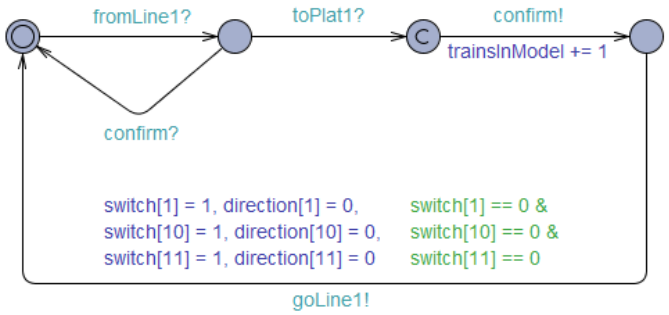

Figure 4.7. Three switch specification using variables - Approach 1.

4.2.1.4 Model of the Train Station

The model of the train station is built-up as explained in Section 4.1. Every state has two outgoing transitions with the direction of the switch as a guard (see Figure 4.8). When transitioning, the switch left by the train becomes `ready`. An automaton can only have one active state at a time so that a model of the train station is created for the amount of trains. Therefore it is necessary to ensure that a specification cannot initiate more than one model of the train station by using UPPAALs channel system. As described in Section 3.2.1.1, a normal signal is only received by one receiver. The channel `in!` is a normal channel so that only one of the models listens and initiates as shown in Figure 4.9. If `in!` was a broadcasted channel, it would trigger all the available models of the train stations and all trains would want to go simultaneously.

When the `trainmaker` sends a signal `fromlineX!` (where X is the number of the line) that decides from which line the train will come, the model of the train station receives the signal and positions the train to enter the station. Then the model of the train station listens to the chosen specification `goLineX?` to allow the train to start moving towards the desired platform. This will only happen when all required switches are `ready`.

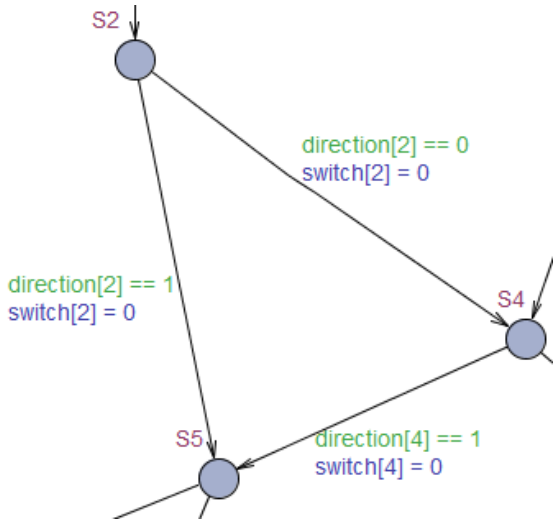

Figure 4.8. The model for incoming trains - Approach 1.

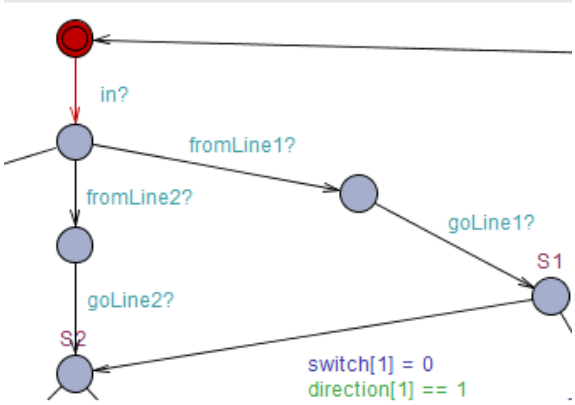

Figure 4.9. A description of how the model receives an incoming train - Approach 1.

There is also a model of the train station for the outgoing direction which is similar to the inbound model of the train station, and uses the same occupancy variables for the switches and platforms. Regarding the direction of the switches, a new variable `outDirection` is introduced that describes the direction a train should take to reach its desired line. Observe that the inbound switch direction does not affect the outgoing train as explained in Section 2.1.2.

When a train has reached a platform the variable `trainsAtPlatform` increases which allows the `trainmaker` to send the signal `out!` on the channel. When this happens, the specifications for the outgoing trains are initiated the same way as described in Figure 4.5. The only difference is that, when the train has left the station, all the models of the train stations have to be back at their initial state so that a new train can enter.

In order to do this both models of the of the train stations are affected by the specification: The inbound model is moved to a temporary state until the train leaves the system and then returns to its initial state. The outbound model gets triggered to move the train to its destination and then also goes back to the initial state as its final step. The `trainmaker` and the switches allow the specification to fire

`goPlatX!`. This moves the inbound model to its temporary state and immediately after, the channel `leavePlatX!` fires, allowing the outbound model to initiate, see figures 4.10, 4.11 and 4.12. When the train leaves the last switch towards a line, a channel `gone!` fires and resets both models back to their initial state, see Figure 4.13.

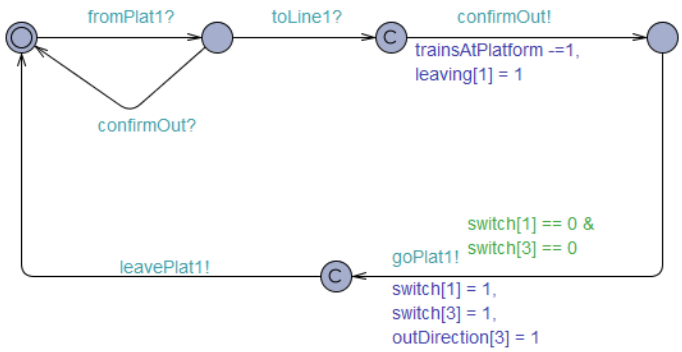

Figure 4.10. When switches are ready both `goPlat!` and `leavePlat!` will happen at the same time - Approach 1.

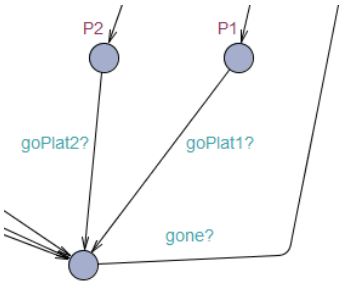

Figure 4.11. `goPlat` sets the inbound model of the train station in a temp-state waiting for `gone` - Approach 1.

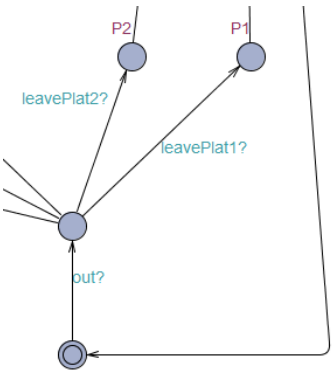

Figure 4.12. `leavePlat` starts the train in the oubound model of the train station - Approach 1.

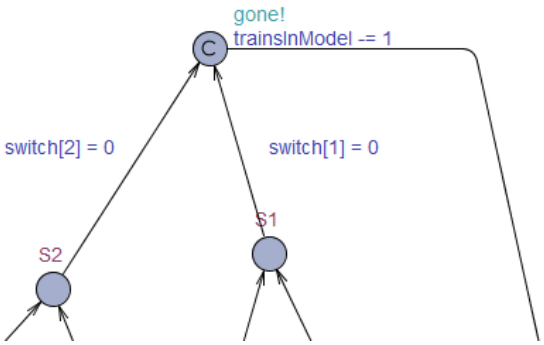

Figure 4.13. The outmodel of the train station is done and immediately fires `gone` - Approach 1.

Due to the platform being `busy` when a train arrives, the `trainmaker` tells many trains to head for the same platform. This is not a desired behaviour because another train should not select a platform which a train is already heading to. To avoid this, the variables `arriving` and `leaving` are introduced. When the `trainmaker` selects a platform for a train, another train cannot be assigned to the same platform until that train has started to leave the station. Similarly, if a train is already leaving a certain platform, the `trainmaker` cannot request a train to leave that platform (see Figure 4.14).

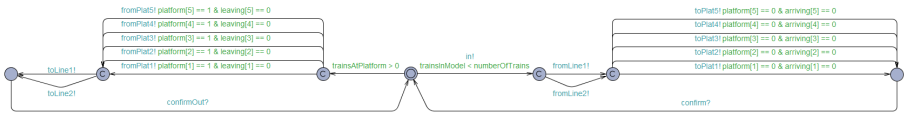

Figure 4.14. The complete `trainmaker` with two lines in and five platforms - Approach 1.

4.2.1.5 Reducing the Model of the Train Station

Due to the memory issues in UPPAAL described in Section 3.2.5, verification is not possible and some changes have to be made. Instead of having the intended five trains they are now reduced to two trains, however, the model is still too large.

To further reduce states, the in- and outbound model of the train station are combined. The same system as described above is used but the channel `gone!` is no longer needed because the model of the train station is already at its initial state when the train leaves the last switch. An additional variable `trainDir` is introduced to ensure that the train reaches the platform before leaving, see Figure 4.15.

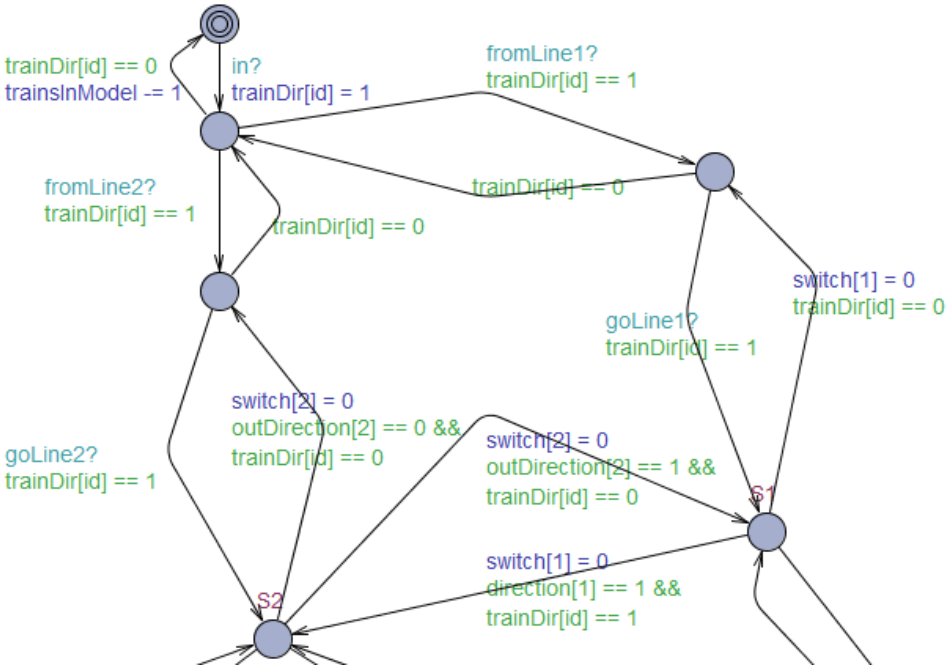

Figure 4.15. The in- and out-model of the train station combined with the `trainDir` variable implemented - Approach 1.

4.2.2 Approach 2: Using UPPAALs Programming Features

The next idea is based on Section 4.2.1 and therefore only the differences between the models are described. The main difference is that the programming features UPPAAL offers are used to check if a train is at a platform and to block the path the train will take. The switches, their directions and the platforms are immediately modelled as variables and only one model of the train station exists for both in- and outbound trains.

The variable `switch[id]` describes the behaviour of switch number *id* and the direction of the switch is given by the variable `direction[id]`. To describe the platforms, two variables are also needed. The variable `rPlatform[nPlatforms]` is 0 when the platform is free and 2 when busy, so there is no need to request the platform. The other variable implemented is the boolean `platform[nTrains][nPlatforms]` which saves if a train is at a specific platform.

Due to only having one model of the train station per train, the variable `In` is created and works identically as the variable `trainDir` from the previous model.

4.2.2.1 Request and RequestOut

Instead of having an automaton `trainmaker`, the automata `Request` and `RequestOut` are used to determine which line and platform should be used and to request the switches. There exist one automaton `Request` and `RequestOut` per train. Figure 4.16 shows that if the maximum number of trains in the model has not been reached, a random integer between 1 and *nLines* (number of lines) is generated and saved in the local variable `randomPath` while the amount of trains is increased. Immediately, another random integer between 1 and *nPlatforms* (number of platforms) is generated and saved in `randomPath`. Instantly, the method `choosePath()` (seen in Listing 4.1 for two lines and two platforms) checks if the appropriate switches and platforms are free. If they are, they are requested (i.e. `switch[id] = 1`) and the directions are set appropriately with the variable `direction[id]`.

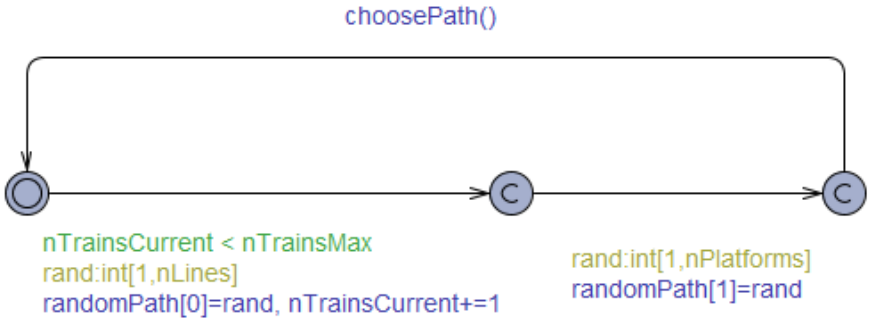

Figure 4.16. Automaton `Request` Version 1 - Approach 2.

In order to avoid a deadlock when the required switches or platforms are not free, the variable `move` and a self-loop are implemented, seen in Figure 4.17. If the path is possible, the variable is set to 1 (i.e. `move == 1`). If not, the variable stays 0 and

the self-loop continues until the path is available.

The next problem is that a new request for that train can happen before the train has been in the station or gone out. Therefore, the channel `gone` is implemented which sets the automaton to the initial state when the train is available again. Also, the last step is to add the counter `const nTrain_t id` in the parameters of the template and *id* in the appropriate channels and variables so that each train has a corresponding `Request` automaton.

Lastly, the variable `In` is added to the transition with variable `move` to initiate movement in the `Model`, described later on (see Figure 4.17).

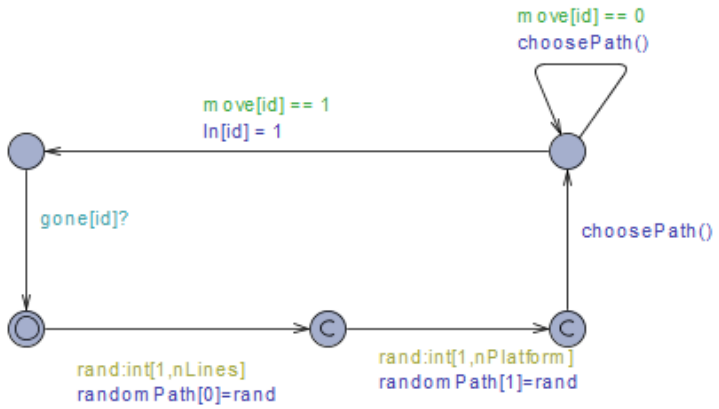

Figure 4.17. Automaton `Request` Version 2 - Approach 2.

Listing 4.1. Example of Local Declarations `Request` - Approach 2.

```
int  randomPath[2]; // Array randomPath with integers of size [1,2]
bool move[nTrain_t];
int  i;
void choosePath(){
    move[id]=0; // reset
    if  (randomPath[0]==1){ //Line 1
        line[id]  = 1;

        // L1P1
        if (randomPath[1] == 1 & switch[1] == 0 & switch[3] == 0
        & platform[id][1]  == 0 & rPlatform[1]  == 0){

            switch[1] = 1; switch[3] = 1;
            direction [1]  = 0; direction [3]  = 0;
            move[id] = 1; rPlatform[1]  = 1;};

        // L1P2
        if (randomPath[1] == 2 & switch[1] == 0 & switch[3] == 0
        & switch[6] == 0  & platform[id][2]  == 0 & rPlatform[2]  == 0){

            switch[1] = 1; switch[3] = 1; switch[6] = 1;
```

```
            direction [1] = 0; direction [3] = 1; direction [6] = 0;
            move[id] = 1; rPlatform[2] = 1;};

}else if (randomPath[0]==2){ //Line 2
    line [id] = 2;

        // L2P1
        if (randomPath[1] == 1 & switch[2] == 0 & switch[4] == 0
        & switch[3] == 0 & platform[id][1] == 0 & rPlatform[1] == 0){

            switch[2] = 1; switch[4] = 1; switch[3] = 1;
            direction [2] = 0; direction [4] = 0; direction [3] = 0;
            move[id] = 1; rPlatform[1] = 1;};

        // L2P2
        if (randomPath[1] == 2 & switch[2] == 0 & switch[4] == 0 &
        switch[3] == 0 & switch[6] == 0 & platform[id][2] == 0 & rPlatform[2] == 0){

            switch[2] = 1; switch[4] = 1; switch[3] = 1; switch[6] = 1;
            direction [2] = 0; direction [4] = 0; direction [3] = 1; direction [6] = 0;
            move[id] = 1; rPlatform[2] = 1;};

    }else{
    line [id] = 0;
    };
}
```

The `RequestOut` automaton is based on the `Request` automaton. The method `atPlatform()`, shown in Listing 4.2, checks if the specific train is at a station and sets the variable `temp` to 1 if this is true. Then a random number between 1 and $nLines$ is generated and saved in the variable `randomPath` to determine which line the train should exit through. Again, the method `choosePath()` checks if the switches are free, requests them and sets the variable `move` to 1. If not, the self-loop continues to check and waits until the path is available.

Figure 4.18 shows the `RequestOut` automaton, with the guard `platform[id][1] == 1 | platform[id][2] == 1 | platform[id][3] == 1 | platform[id][4] == 1 | platform[id][5] == 1` which checks if that train is at a platform before the method `atPlatform()` is executed. This should reduce the amount of transitions. Again, the variable `In` is included when $move == 1$. The method `initialise()` sets all values of `position[id][nSwitch]` to 1 which will be explained later on.

Listing 4.2 shows an example of the local declarations for `RequestOut` for two Lines and Platforms. Some variables may have not been explained yet.

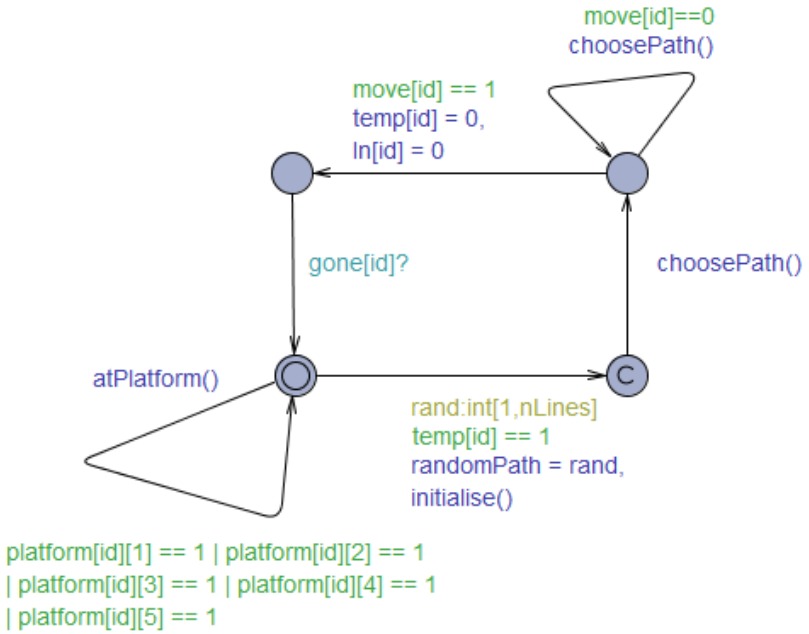

Figure 4.18. Automaton `RequestOut` - Approach 2.

Listing 4.2. Example of Local Declarations `RequestOut` - Approach 2.

```
int  randomPath;
int  i;
bool move[nTrain_t];
bool temp[nTrain_t];
int  j;

void initialise  (){
      for(j  = 1; j  <= nSwitch; j++){
            position[id][j] = 1;
        };
}

void atPlatform(){
    temp[id] = 0;
    for(i  = 1; i  <= nPlatform; i++){
        if (platform[id][i] == 1){
            temp[id] = 1;};
    };
}

void choosePath(){
   move[id] = 0;

    if  (randomPath == 1){ // Line 1
        line[id] = 1;

        // L1P1
        if (platform[id][1]  == 1 & switch[1] == 0 & switch[3] == 0){
```

```
            switch[1] = 1; switch[3] = 1;
            direction [1] = 0; direction [3] = 0;
            move[id] = 1;
            position[id][1] = 0; position[id][3] = 0;};

        // L1P2
        if (switch[1] == 0 & switch[3] == 0 & switch[6] == 0
        & platform[id][2] == 1){

            switch[1] = 1; switch[3] = 1; switch[6] = 1;
            direction [1] = 0; direction [3] = 1; direction [6] = 0;
            move[id] = 1;
            position[id][1] = 0; position[id][3] = 0; position[id][6] = 0;};

}else if (randomPath == 2){ //Line 2
    line[id] = 2;

        // L2P1
        if (switch[2] == 0 & switch[4] == 0 & switch[3] == 0
        & platform[id][1] == 1){

            switch[2] = 1; switch[4] = 1; switch[3] = 1;
            direction [2] = 0; direction [4] = 0; direction [3] = 0;
            move[id] = 1;
            position[id][2] = 0; position[id][4] = 0; position[id][3] = 0;};

        // L2P2
        if (switch[2] == 0 & switch[4] == 0 & switch[3] == 0
        & switch[6] == 0 & platform[id][2] == 1){

            switch[2] = 1; switch[4] = 1; switch[3] = 1; switch[6] = 1;
            direction [2] = 0; direction [4] = 0; direction [3] = 1; direction [6] = 0;
            move[id] = 1;
            position[id][2] = 0; position[id][4] = 0; position[id][3] = 0;
            position[id][6] = 0;};

    }else{
    line[id] = 0;
    };
}
```

4.2.2.2 Model of the Train Station

The `Model` automaton of the train station is similar to the one already described in Section 4.2.1 and many lessons learnt are applied. The automaton can only be in one state at a time, however, the goal is to implement multiple trains. Therefore, each train needs its own `Model` automaton of the train station which is done by

implementing the template parameter `const nTrain_t id` and adding $[id]$ to the appropriate channels or variables that are train specific.

Listing 4.3 shows an example label for the transition $S1$ to $S4$ which can be seen in Figure 4.19. The model of the train station is similar to the one in Figure 4.8 where the direction of the switch is used as a guard (`direction[i]`, with i being the switch number) to block specific transitions and the variable `switch[id]` is reset to 0 after leaving a switch. When having only one model of the train station for both in- and outbound trains, the variable `In` is added which defines if the train is in- (`In[id] == 1`) or outbound (`In[id] == 0`).

Listing 4.3. Label for Transition $S1$ to $S4$ - Approach 2.

```
direction [1]  == 1 // check if  direction  is  correct
& In[ id ]  == 1 // check if  train  is  inbound

switch[1] = 0 // reset switch 1 – free
```

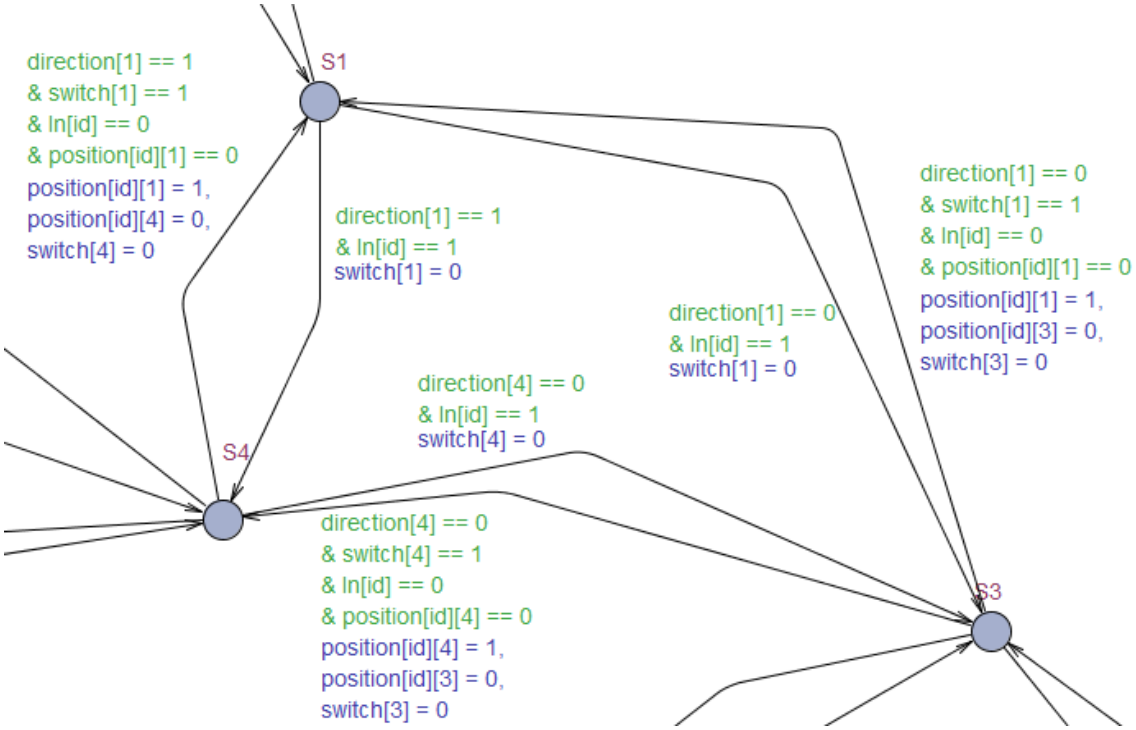

Figure 4.19. Transition between Switches in `Model` automaton - Approach 2.

The labels from Listing 4.3 cannot be used for outbound trains because a few problems occur and therefore they are changed to Listing 4.4. Firstly, both outgoing transitions may be possible if the direction is correct. Therefore, the guards are extended to check if a switch is also `busy`, since the train should go to the busy switch. However, another train can already be at the busy switch but has not yet released it. Therefore the variable `position[i]`, with i being the number of the switch, is introduced which is set to 1 when a train is at that switch and 0 when the switch is free again. All values of `position` are reset to 1 using the method `initialise()`

in the automaton `RequestOut` when wanting to leave a platform. This is done to lead the train back to the initial state.

Listing 4.4. Label for Transition $S4$ to $S1$ - Approach 2.

```
direction [1]  == 1 // check if switch 1 is facing correct direction
& switch[1] == 1 // check if switch 1 is free
& In[ id]  == 0 // check if train is outbound
& position[ id][1]  == 0 // check if a train is at switch 1

position[ id][1]  =  1, // set position – train is at switch 1
position[ id][4]  = 0, // reset position – train not at switch 4
switch[4] = 0 // reset switch 4 – free
```

The label of the transitions going into a platform (see Figure 4.20) is similar to Listing 4.3, however the variable `platform[id][nPlatform]` is set to 1, showing that the train is at the platform. When the train leaves the platform, the variable `rPlatform[platform]` is reset to 0.

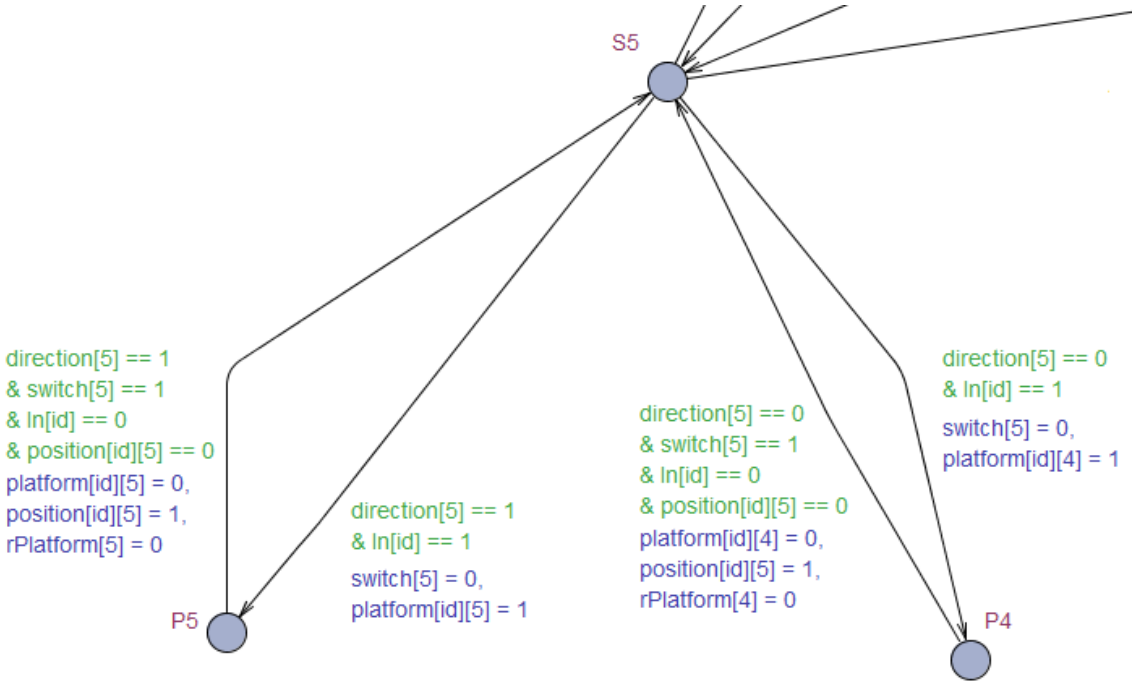

Figure 4.20. Transition in and from the Platforms in `Model` automaton - Approach 2.

All trains are initially at the state *start* before an appropriate path is chosen (see Figure 4.21). Depending on the line chosen (i.e. `line[id] == 1` or `line[id] == 2`) the train can go to either switch 1 or 2. When going to the initial state the channel `gone[id]!` resets both automata `Request` and `RequestOut` to their initial state.

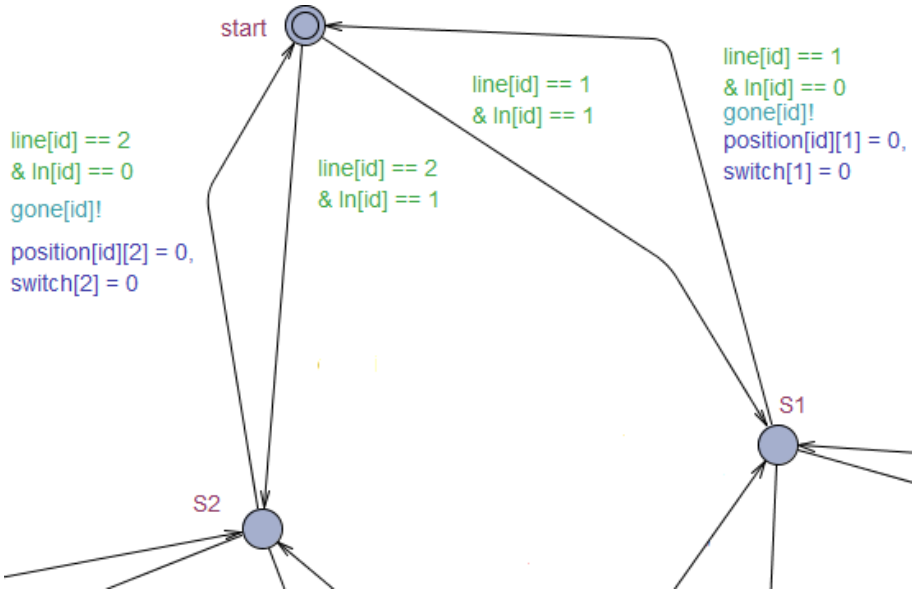

Figure 4.21. Lines in `Model` - Approach 2.

Listing 4.5 shows the global declarations.

Listing 4.5. Global Declarations - Approach 2.

```
// GLOBAL

// constants
const int nSwitch = 6;
const int nPlatform = 5;
const int nLines = 2;
const int nTrainsMax = 2;


// id to make multiple of something
typedef int [1, nSwitch] switch_t;
typedef int [1, nTrainsMax] nTrain_t;
typedef int [1, nPlatform] nPlatform_t;


// Channels
broadcast chan gone[nTrain_t];

// Multiple
int [0,2] switch[switch_t]; // 0 = free; 1 = requested; 2 = busy
int platform[nTrain_t][nPlatform_t]; // platform the train is at
bool rPlatform[nPlatform_t]; // 0 = free; 2 = busy
bool direction [switch_t]; // 0 = left , 1 = right
bool position [nTrain_t][switch_t]; // 0 = free; 1 = train there
int [0,3] line [nTrain_t];
bool ln[nTrain_t]; // 1 = true, 0 = false
```

4.3 Supremica

There are four possibilities of how to model the train station: with(out) variables and with(out) a model describing the relationship between the switches. While working with variables the problem of nondeterminisim surfaced which was solved much later on as described in Section 3.1.6.2. In UPPAAL, it became obvious that the model of the train station increases the amount of total state tremendously. Therefore, approaches without a model of the train station are investigated.

4.3.1 Approach 1: Without Variables and with a Model of the Train Station

The idea with the first approach in Supremica is to, by only using common events rather than variables, create a functioning model of a train station. A switch should only be *busy* when a train reserves it and become *ready* as soon as the train passes it (see Section 2.1.3). Therefore, if enough events are used, it is possible to have every train route set all the necessary switches to busy with one unique event. The events are created by using the array technique explained in Section 3.1.1.4 and are divided into incoming `LPIN` and outgoing events `PLOUT` in the format

`LPIN[lines][platforms][steps][trains]`
and
`PLOUT[platforms][lines][steps][trains]`.

The amount of steps is the number of maximum steps taken by a train in the model of the train station. This creates a large number of unused events because all trains do not have to move through the same (maximum) amount of switches. Events that are not used can be annoying when simulating in Supremica because those events are constantly available in the event list. Therefore, it is hard to see what events are actually changing the model of the train station so that an event alias for all unused events is manually created and put into a blocked list, see Section 3.1.1.2. Using one array is an easier way of creating many events instead of making several arrays depending on how many switches a certain train is passing.

In order to reduce the amount of states, instead of having a switch as two 2-state automata for *ready*/*busy* and $left/right$, the switch is described by three different states: *ready*, $left$ and *right*. Here, a switch set to any direction is regarded as *busy*, see Figure 4.22. The platforms are also modelled as 3-state automata with the states *ready*, *reserved* and *platform*, see Figure 4.24.

To explain this further, a small example is displayed in figures 4.22 and 4.23. A train can only go to platform one or five and randomly the event `LPIN[1][5][1][2]` is fired. This means that the train wants to go from line one to platform five and has already taken the first step towards the desired platform. The value 2 decides that the train represented by model two of the train station is activated. Since only one version of switch one is available, no other train can use that switch until it has

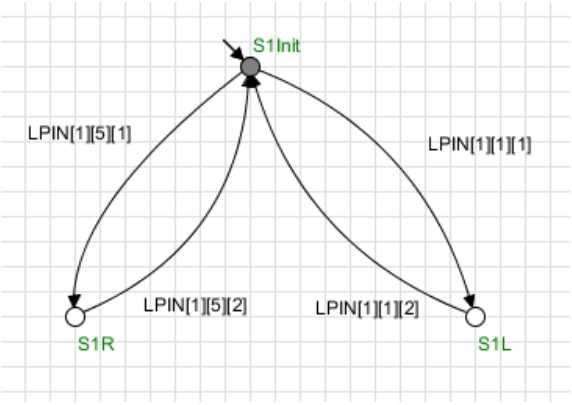

Figure 4.22. Example of a switch - Approach 1.

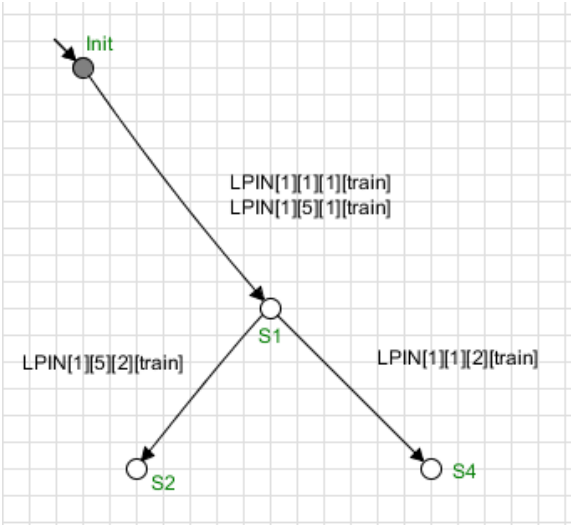

Figure 4.23. A small example of a model - Approach 1.

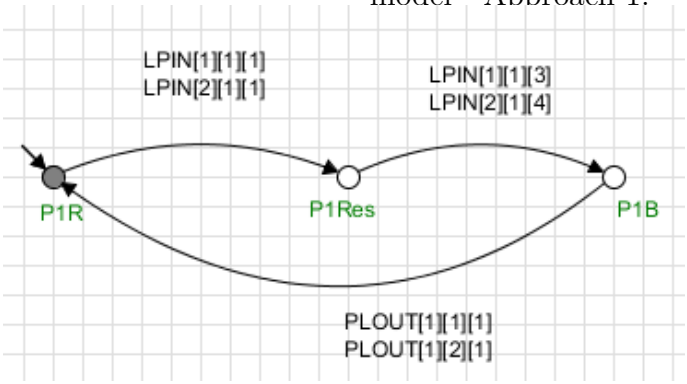

Figure 4.24. An example of a platform.

returned to its initial state.

All events that set a switch to *busy* are the first step of an event, representing a train coming from a line and heading to a platform or a train leaving a platform to some line. To set switch one to *ready* again, the train has to pass switch one and reach switch two. The event `LPIN[1][5][2][train]` sets the switch back to its *ready* position. Observe that [*train*] has the value 2 because no other train can be at switch one to trigger the event. If the train continues past switch two it is triggered by the event `LPIN[1][5][1][2]` and reset to its initial state by the event `LPIN[1][5][3][2]`.

When the train is heading out, with the events `PLOUT`, it needs to follow the desired path. Therefore, switches responsible for the outwards direction are created. However, to ensure a low state count, the events that steer the trains outwards are added to the already created switches, see Figure 4.25.

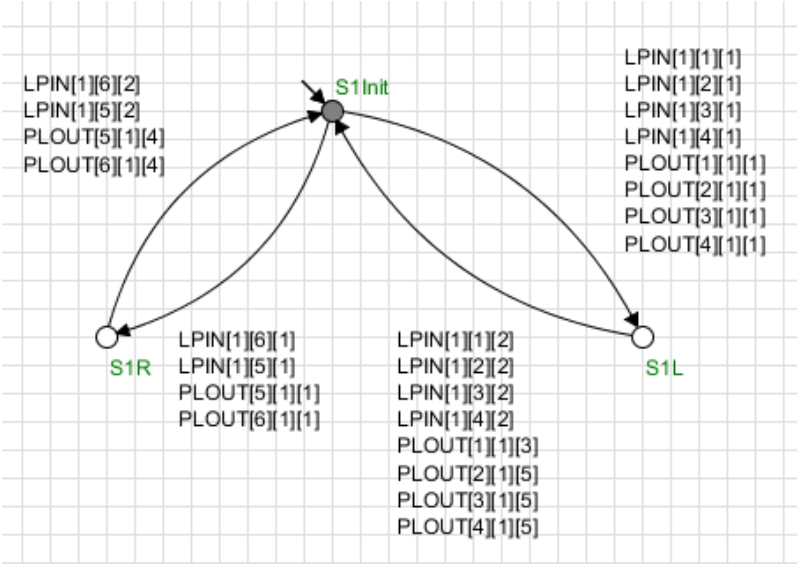

Figure 4.25. An example of a switch - Approach 1.

Finally, to make sure that a train reaches a platform before heading out again, a specification is created and the events `done` and `arrived` added to the model of the train station. The `arrived` event is added as a selfloop at the platforms while `done` is added after the last outgoing event, see figures 4.26 and 4.27. After that, two aliases are made for `incoming` and `outgoing` events respectively. Outgoing events can only happen when a train has reached the designated platform, see Figure 4.28.

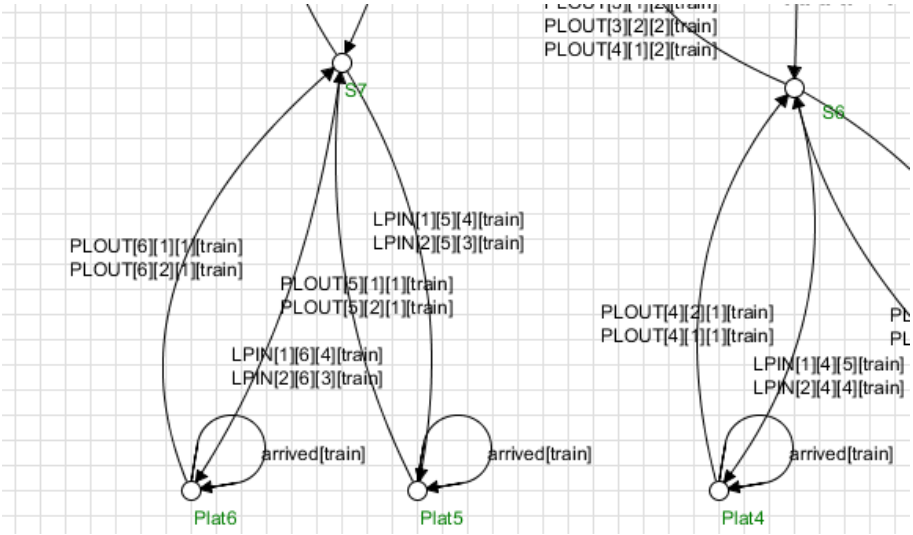

Figure 4.26. The event `arrived` shown in the model of the train station - Approach 1.

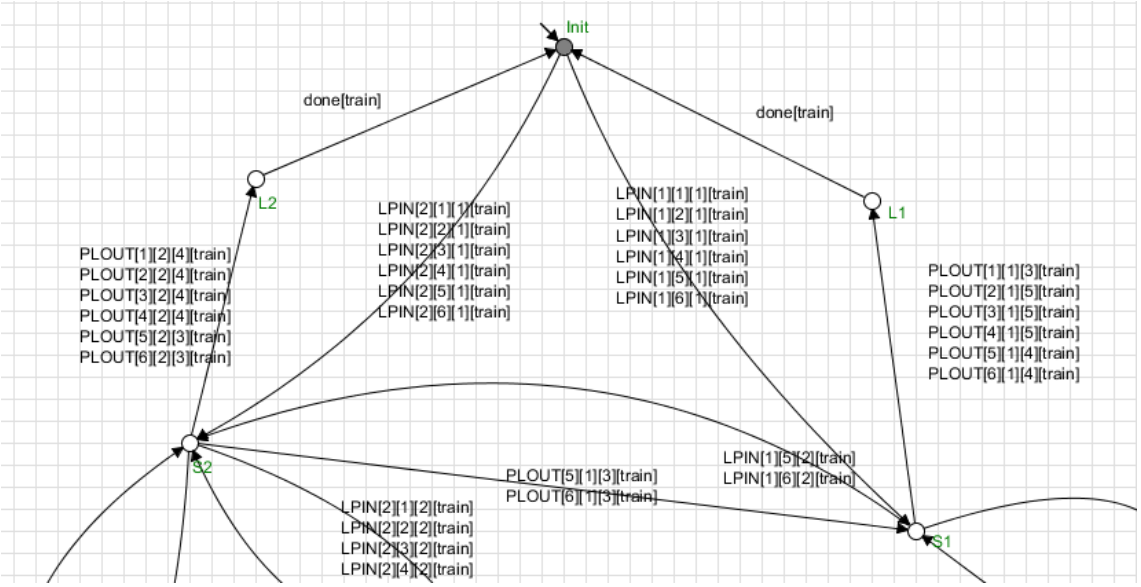

Figure 4.27. The event `done` shown in the model of the train station - Approach 1.

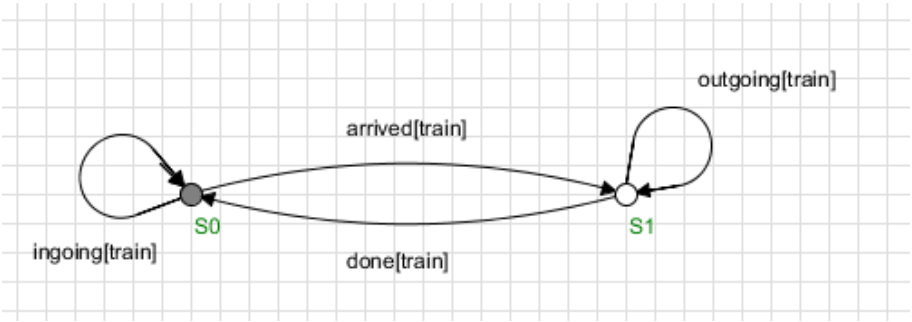

Figure 4.28. Specification to handle the train direction in the model - Approach 1.

4.3.2 Approach 2: Without Variables and no Model of the Train Station

Approach 2 is based on Section 4.3.1, however, it does not include the model of the train station and has an additional automaton `order`.

Figure 4.29 shows the new automaton `order`, now called `orderIn`, which ensures that the steps are done in the correct order from 1 to 5. This is done by using the alias `Step[step][train]` which holds all possible events for each step, duplicated for each train. The event `goOut[train]` is train specific and is used to activate automaton `orderOut` (see Figure 4.30).

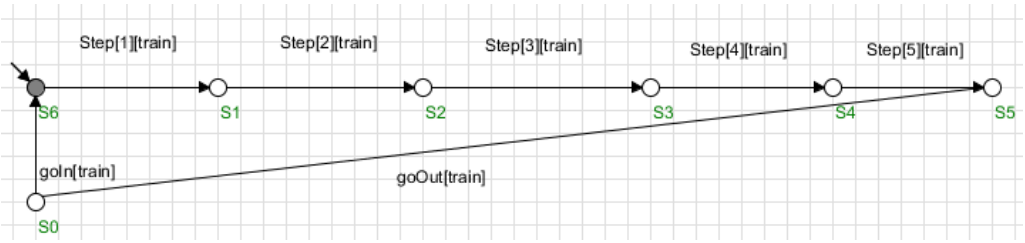

Figure 4.29. Automaton `orderIn` Version 1 - Approach 2.

Automaton `orderOut` ensures, similarly to `orderIn`, that the steps are done in the correct order using the event `StepO[step][train]`. The event `goIn[train]` activates the automaton `orderIn` again when the train has left the station.

Figure 4.30. Automaton `orderOut` Version 1 - Approach 2.

This works, if all the paths have the same amount of steps. However, some paths do not have five steps even though the events for five steps are generated. Therefore, events exist which can occur in any state and have no affect on the automaton, called "extra events". When creating the aliases `step[stepNumber][trainNumber]`, foreach-loops were used so that all events, independent of the maximum amount of steps per path, are included. This leads to a problem because these extra events are triggered while having no affect on the switches, however, disabling the correct events.

For example, assume that the automaton `orderIn` is in state $S3$, as seen in Figure 4.31, and switch 5 is to the left as shown in Figure 4.32. Only one event should be able to trigger to set the switch back to its initial state: `LPIN[2][1][4][1]`. However, the extra event `LPIN[1][1][4][1]` is included in the alias `step[4][train]` and can also be triggered. The problem is, that the event, which is responsible to reset the switch, is disabled so that a deadlock occurs. Therefore, the amount of steps each path has needs to be considered.

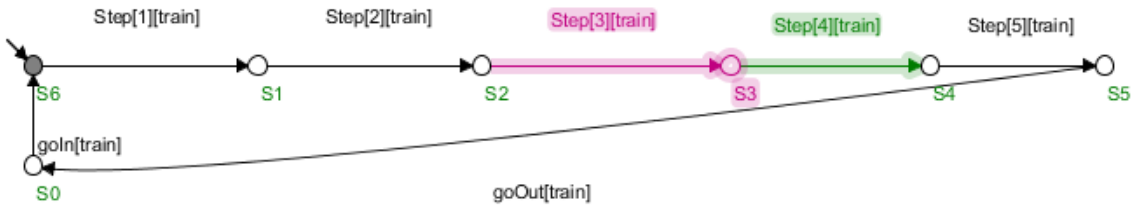

Figure 4.31. Simulation of Automaton `orderIn` Version 1 - Approach 2.

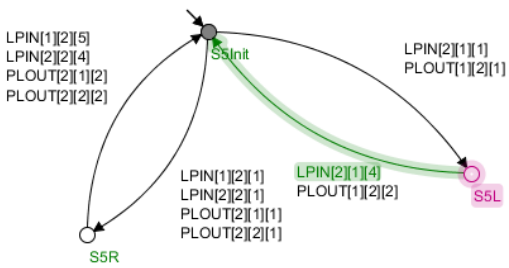

Figure 4.32. Simulation of Automaton `switch5` - Approach 2.

The first idea, which might come to mind is to add the extra events to a blocked list. However, it is necessary to use them to reach step 5 if the path is too short. Otherwise the train stops at step 3 and a deadlock occurs.

One solution is shown in Figure 4.33, with the adapted automaton `orderIn`. It can be seen that transitions are added for each step after step two. There exists no last step before step two, therefore no transition in state $S1$ is needed. The aliases `goOutX[train]`, with X being a number between three and five, hold the last steps of paths corresponding to the X. Therefore, if step four is the last step of a path, it is stored in the alias `goOut4[train]` and activates the automaton `orderOut`.

Figure 4.33. Automaton `orderIn` Version 2 - Approach 2.

The automaton `orderOut` (see Figure 4.34) works similarly to `orderIn`.

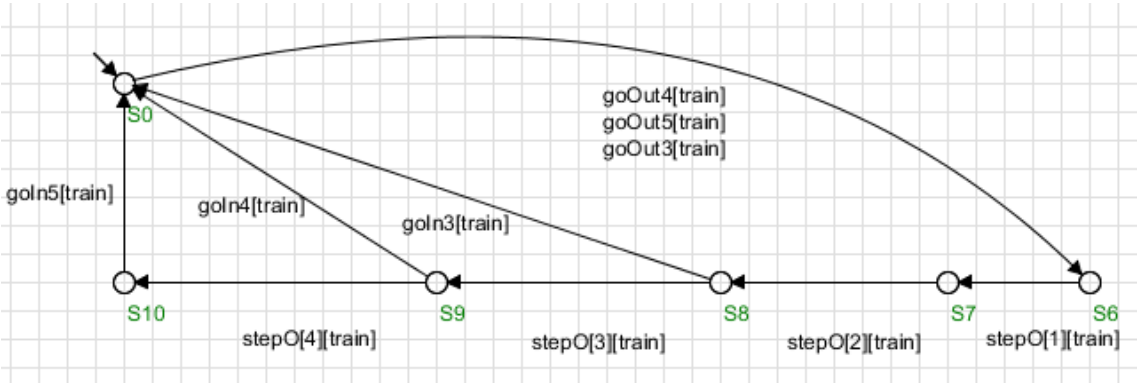

Figure 4.34. Automaton `orderOut` Version 2 - Approach 2.

Additionally, an automaton is created to hold a "blocked-list" which includes all the steps that should not happen because the last step has already been taken (see Figure 4.35).

Figure 4.35. Automaton `blocked` - Approach 2.

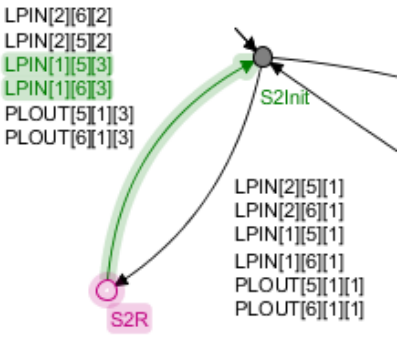

Figure 4.36. Part of automaton `switch2` - Approach 2.

This however does not work, because it becomes non-deterministic as seen in Figure 4.37. In this example, switch 2 is set to *right* and there are multiple events which can reset it back to its initial state (see Figure 4.36). The problem is that additional to the correct event found in the alias `goOut3[train]`, the alias `step[3][train]` also holds possible events. Therefore, the automaton becomes non-deterministic.

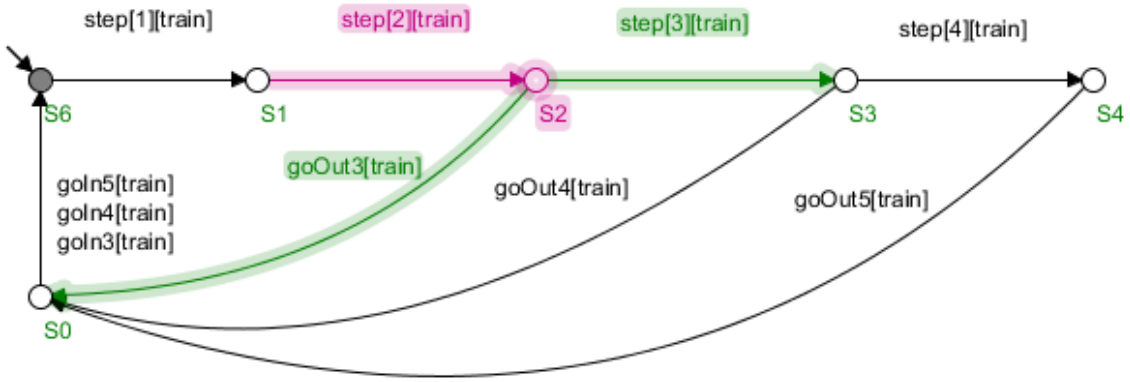

Figure 4.37. Simulation of automaton `orderIn` - Approach 2.

Therefore, it is necessary to explicitly specify the path the train should take. The implementation could either be one automaton for every possible path or one automaton which holds all possible paths, i.e. the model mentioned in the idea in Section 4.3.1. Therefore, when using no variables, a model of the train station is required.

4.3.3 Approach 3: With Variables and no Model of the Train Station

One idea is to connect the switches and create a path using events to activate the next switch in the path, as seen in Figure 4.38. Depending on whether the switch is facing to the *right* or the *left* (i.e. $S1\_state == 1$ or $S1\_state == 0$ respectively), the variable for switch four (`ToS4`) or switch three (`ToS3`) is set to one, which is used as a guard in the next switch, as seen in Figure 4.39.

Figure 4.40 shows what the specification for a train looks like. If switch $S1$ is facing to the right (i.e. $S1\_state == 1$) then it can be passed. If not, the event $S1R$ sets

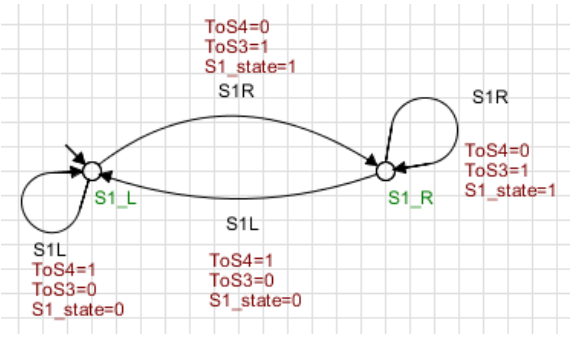

Figure 4.38. Connecting the Switches with Variables `Switch1`- Approach 3.

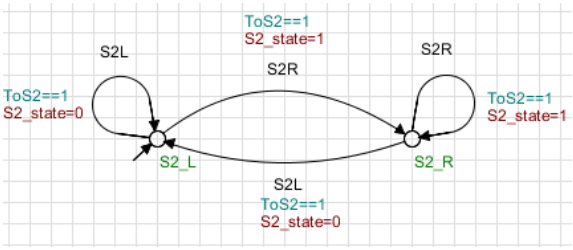

Figure 4.39. Connecting the Switches with Variables `Switch2` - Approach 3.

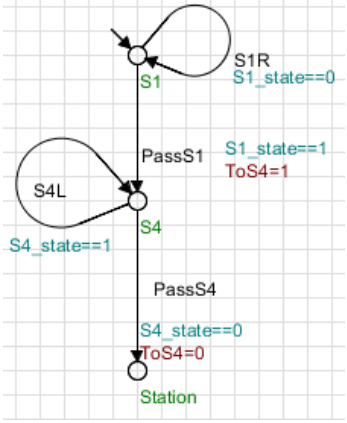

Figure 4.40. Train Specification - Approach 3

the switch to right. The same process is done for switch $S3$, however, with it facing left.

When using this idea, no model of the train station for each train is needed, as described in Section 4.3.1. However, this increases the amount of states because the specification *train* would be needed for each possible path from every incoming line to each platform. Therefore, for the simple model of the train station, ten automata for the in- and ten for the out-going paths are needed. The different automata have states which represent the same switches since the switches are a shared resource for the path. This means, that the amount of states and the amount of transitions is quite large. Therefore, this idea is disregarded.

4.3.4 Approach 4: With Variables and with a Model of the Train Station

To avoid unnecessary states the switches and platforms are created similarly to Section 4.3.1 but instead in variable form, see Figure 4.41. The train is unaffected by the direction of the switch when moving out of the station as mentioned in Section 2.1.2. However, to ensure that the train follows the desired outbound path, a variable `switchOut` with the values $left$ and $right$ is created. This was not necessary when working without variables because the events are divided up into steps and are therefore unique. But in this case one event for out- and inbound train exists respectively.

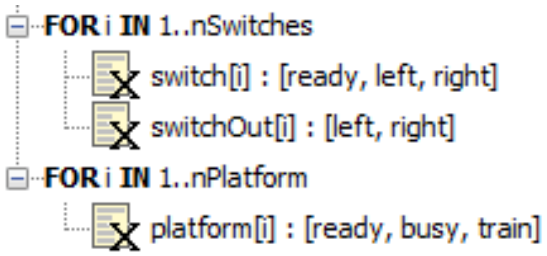

Figure 4.41. The switches and platforms as variables - Approach 4.

The event `trainIN[train]` is used to describe the train going inbound and the event `trainOut[train]` an outbound train. The build-up of the transitions labels is quite consistent throughout the model of the train station, therefore Switch $S1$, $S3$ and $S4$ are shown as examples in Figure 4.42. The transition from $S1$ to $S4$ can only be triggered if switch $S1$ shows in the specific direction $left$ (i.e. also $busy$). When this transition is triggered, the train is no longer at switch $S1$ so that its status is changed to $ready$, allowing other trains to use the switch. Switch $S3$ has outbound transitions $S3$ to $S4$ and $S3$ to $S1$. If more than one outbound transition exists, an extra guard is needed which defines which transition should be triggered ($switchOut[3] == left$ or $switchOut[3] == right$). Again, the transition sets the recent switch to $ready$ again.

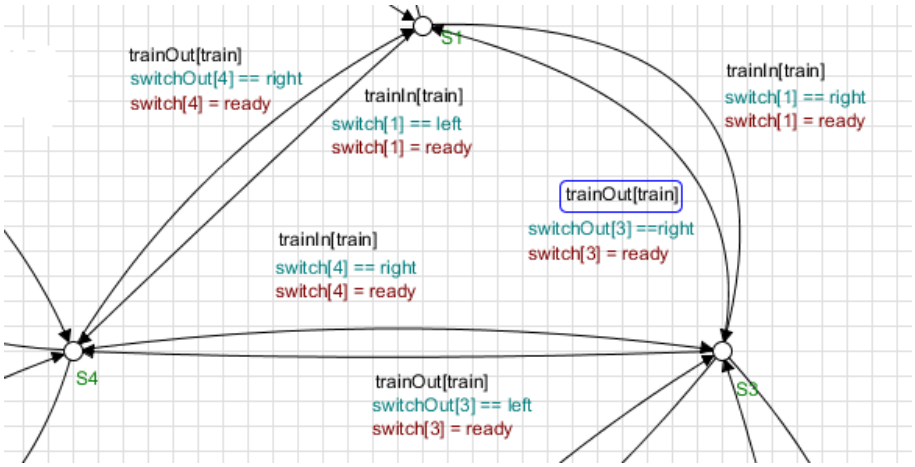

Figure 4.42. Going in and out of a Switch - Approach 4.

Figure 4.43 shows the labels of the transitions leading to and from a platform. This is done similarly to switches, however an action is included which defines, that a train is at the platform (i.e. $platform == train$). This is necessary because a train can only go outbound after it has reached a platform. The selfloop $arrived[train]$ is used in the specification, to ensure the correct order of in- and outbound.

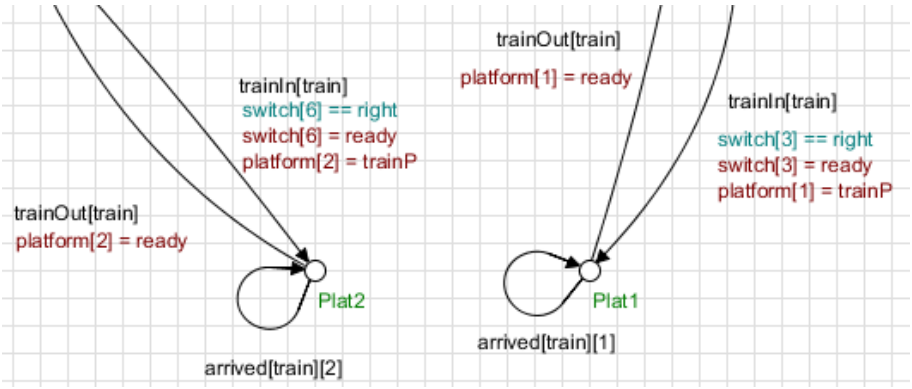

Figure 4.43. Going in and out of a Platform - Approach 4.

Figure 4.44 shows the events $line1[train]$ and $line2[train]$ from the initial state $Init$. These events are used in the specification and determine which line the train originates from.

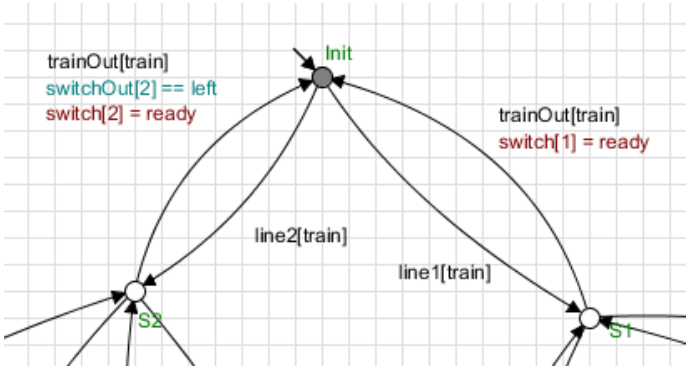

Figure 4.44. Going in and out of $Init$ - Approach 4.

The specification (see Figure 4.45) consists of two states $S0$ (initial state) and $S1$. The events $line1[train]$ and $line2[train]$ lead from $S0$ to $S1$ and initialize a train going inbound. Each transition has different guards which correspond to a specific line to platform combination. When at $S1$, the event $trainIn[train]$ in a self-loop occurs until the train has reached a platform. The event $arrived[train]$, again with guards corresponding to a specific line to platform combination, sets the specification back to $S0$ where the event $trainOut[train]$ in a self-loop occurs until the train is back in the initial sate in the model of the train station.

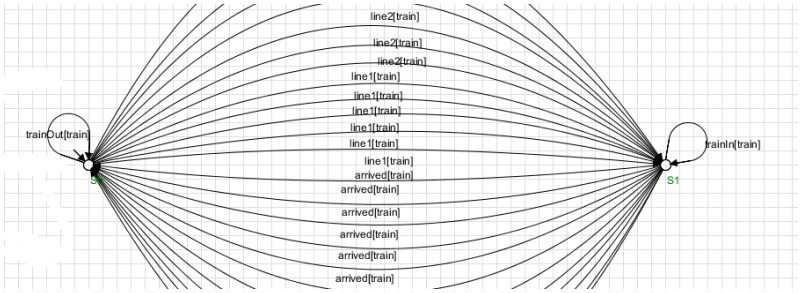

Figure 4.45. Specification - Approach 4.

Figures 4.46 and 4.47 give an overview of the transition labels for both in- and outbound trains. The guards check if a specific combination is possible and the actions set the corresponding switches and platforms to *busy* or in a specific direction.

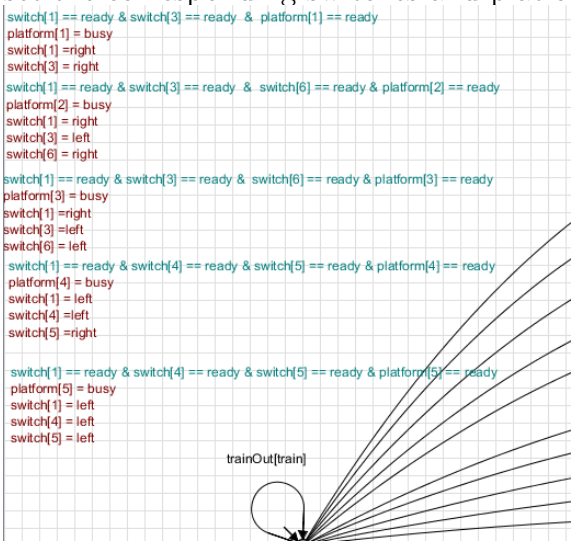

Figure 4.46. Inbound transition labels for the Specification - Approach 4.

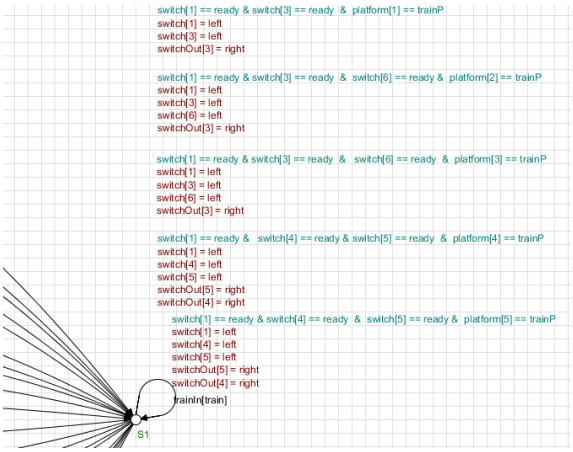

Figure 4.47. Outgoing transition labels for the Specification - Approach 4.

5
VERIFICATION

When verifying, one first has to decide which properties are supposed to be verified for the system. Therefore the following properties are defined:

- The trains can continuously run i.e. no deadlock state exists.
- The trains do not crash i.e. switches and platforms are only occupied by one train.
- The trains behave as required i.e. the specified platform or line is reached. This is not a necessary requirement to check to ensure safety but rather if the modeling is correct.

Different approaches for the simplified model of the train station with two lines and five platforms are verified and then an appropriate approach is chosen and used to model the Gothenburg train station. The results of the verifications are then presented in Chapter 6.

5.1 UPPAAL

To get an overall idea if the model behaves as expected, the simulator is used to observe the behaviour. Notice, that the verification is done only for two trains due to the memory issues described in Section 3.2.5. Verification for both approaches are equivalent unless specified differently.

5.1.1 Deadlock verification

The specification in UPPAAL is done using temporal logic, as explained in Section 2.3.3, and the verifier itself, as described in Section 3.2.3 specifically Table 3.2. To check for deadlock, the built in `deadlock` command is used along with $A[]$:

```
A[] not deadlock
```

This implies that no matter where the trains are, there will never be a deadlocked state.

5.1.2 Avoid same state verification

The next part is to ensure that the trains will not crash by specifying the forbidden states and ensuring that these states are never reached. Forbidden states are defined

as when two trains are at the same state in their model of the train station. This is done by using the following temporal logic expression:

```
A[] not((model(1).S1 and model(2).S1) or (model(1).S2 and model(2).S2)...
```

Here *model*(1) is the model of the train station describing the first train and $S1$ is a state in that model of the train station, in this case representing switch one. This is done for all switches and all platforms to ensure that the trains never collide.

5.1.3 Platform/Line chosen is reached

Next is to ensure that only the specified platform is reached by that specific train. This is checked by using the expression $\Psi \rightarrow \phi$. Which, according to Table 3.2, means that whenever Ψ holds ϕ will eventually hold.

When a trains destination has been determined, no other trains can affect the train to eventually reach the destination. Other trains can only delay a train by occupying the platform or switches that the train has to use. Therefore, it is enough to do this verification for only one train, i.e. one model of the train station.

For the *first* model described in Section 4.2, the verification starts by naming the state where a train specifies its path. In Figure 5.1 the state `L1P2.go` is a state where a train has decided that it wants to go to platform two. From that state, the model of the train station describing a train should eventually reach platform two, described as `model(1).P2`:

```
L1P2.go -> model(1).P2
```

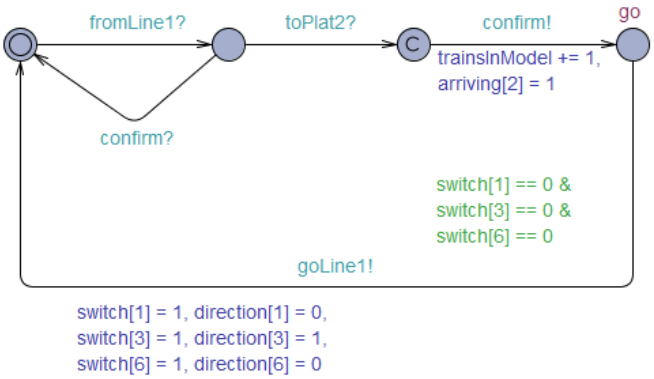

Figure 5.1. The state `go` top right.

When using this command in the verifier, the property is not satisfied and a counterexample is given. In this case the counterexample is that the train remains at the initial state and does not move. While this is a true counterexample, in reality the trains are supposed to move and will move. In order to solve this, UPPAAL's clock feature is used to force the train to move. Section 3.2.1.2 `Templates` describes how an invariant on a state can be connected to a clock to give a limited time a state

can be active.

A clock `x` is declared and an invariant `x<1` added to every state of the model of the train station except the initial state. This means that every second the train has to move, see Figure 5.2. Notice that it is important to reset the clock on every transition.

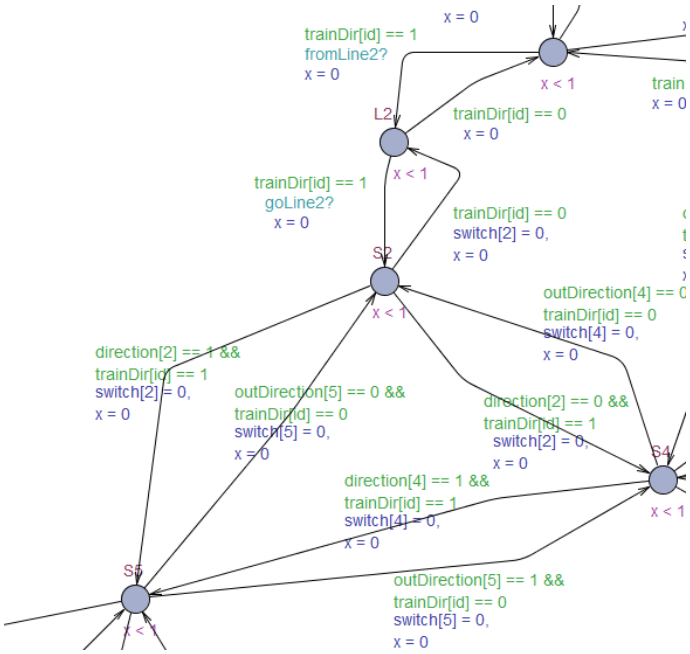

Figure 5.2. The invariants `x<0` and updates `x=0` added.

Checking the property again with the same logical expression results in the property being satisfied. However, to complete the verification it needs to be ensured, that the train does not reach any other platform than the specified one. So, to make sure that the train headed to platform two cannot reach any other platform, another expression is defined:

```
L1P2.go -> (model(1).P1 or model(1).P3 or model(1).P4 or model(1).P5)
```

This property is, as desired, not satisfied. All the platforms and outbound trains to a specific line are tested similarly.

For the *second model* described in Section 4.2, the logical expression is changed a little. The variable `rPlatform` is 1 when a platform is requested and 0 if not. Therefore, when this variable is 1, the specified platform will eventually be reached. So for `rPlatform[2]` requesting the platform two, the expression becomes:

```
rPlatform[2] == 1 -> model(1).P2
```

This property is satisfied, and in the same manner, the other platforms using the following expression is not satisfied:

```
rPlatform[2] == 1 -> (model(1).P1 or model(1).P3 or model(1).P4
or model(1).P5)
```

The same principle as described above is also used for a train heading outbound from a platform to a specific line.

5.2 Supremica

Supremica offers different possibilities to verify the model, as already mentioned in Section 3.1.4.1. Depending on the size of the model, different verification methods are appropriate. For smaller models, it is possible to automatically verify controllability and non-blocking in the analyzer. The feature "find states" can be used on the synchronised/synthesised automaton to check for unwanted states. However, when the automata gets too large, it is not possible to use these features. Instead, "conflict check" and "controllability check" in the menu in "Verify" when in the "Editor"-tab can be used.

This verification is done only for approach one, without variables and with a model of the train station from Section 4.3.1 because approaches two and three in sections 4.3.2 and 4.3.3 were not successfully modelled. Approach four, with variables and no model of the train station in Section 4.3.4, has a larger amount of states after synchronization (for one to three trains) and is more complicated to implement than when using the first approach.

5.2.1 Deadlock verification

A possibility to verify for deadlock (without using the analyzer) is using the conflict check as described in Section 3.1.2. All initial states are set as marked states and therefore should always be reachable. If all states can reach the initial states, it means that the model is non-deadlock.

5.2.2 Avoid same state verification

To get a better understanding of Supremica and how properties can be verified, two different alternatives are presented.

5.2.2.1 Alternative 1

A way to check if trains share a switch/platform simultaneously is to look at the events. This means that two trains should not be able to go to the switch/platform if the first train has not left it. Basically, the event leading *into* the switch (i.e. a train is at that switch) should not happen twice unless the event leading *from* that switch/platform (i.e. a train has left) has occurred. Similarly two trains should not be able to leave a switch/platform twice before a train has entered. To specify this, strings of events that have this undesired behaviour will reach a forbidden state, see

figures 5.3 and 5.4. A conflict check is then executed.

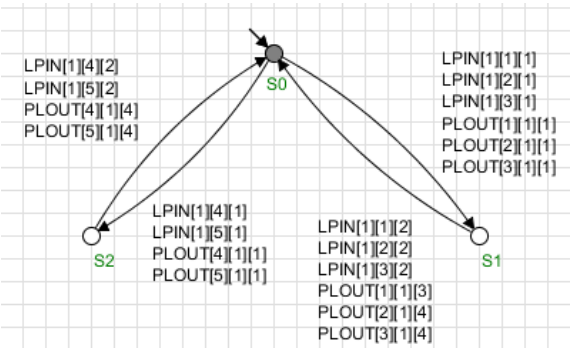

Figure 5.3. Plant of a switch.

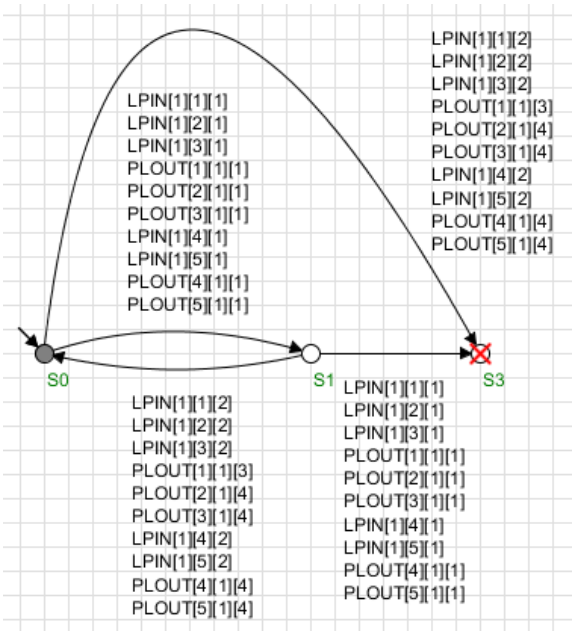

Figure 5.4. Specification of the switch.

5.2.2.2 Alternative 2

When working with automata there is a way of checking if two states are active simultaneously by using uncontrollable events. As mentioned in Section 2.2.7 a supervisor is controllable if it is able to follow all uncontrollable events that the plant executes. Using this, it is possible to add a selfloop in two states with a common uncontrollable event `crash`. This event is then blocked by a specification.

If the two states are active simultaneously the system will be uncontrollable because event `crash` will be able to fire, see figures 5.5 and 5.6. In this example, the two automata are not allowed to simultaneously be in their respective $S1$ states. Lets say that event `one` is fired. If event `three` is fired next, then both automata will be in $S1$ and the uncontrollable event `crash` can trigger. With `crash` being blocked in the specification this will render the system uncontrollable.

In the model of the train station every switch and platform receives its own selfloop with all `crash` combinations of that model of the train station to any other model of the train station. It is important that the events in the respective selfloop are shared only with one other train. If three trains share the same event in a selfloop that means that all three trains have to be in the same state for the event to trigger. These events where created using aliases.

Figure 5.7 shows the aliases with the uncontrollable events for the models of the train station with 5 trains. The alias $check[train][switch]$ is train and switch dependent and it is important to differ between the values $train$ takes, because the events differ according to the value of train. This is done by using foreach-loops and

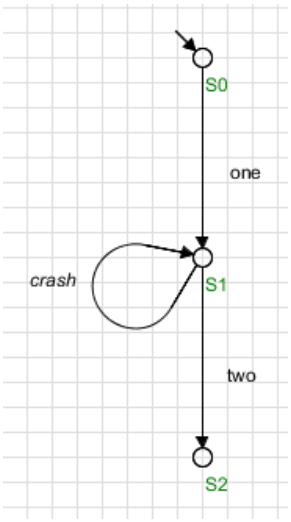

Figure 5.5. Example automaton one.

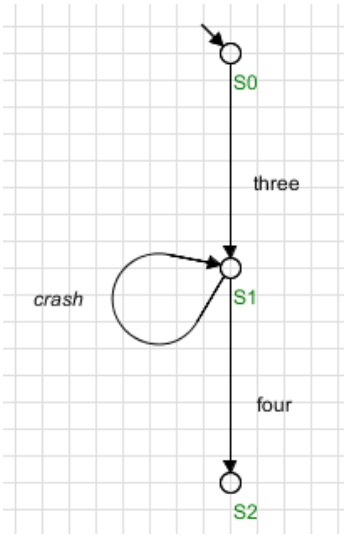

Figure 5.6. Example automaton two.

their guards which create a switch statement known from programming. Also, it is important that all events included affect two different trains and, therefore, the foreach-loop with $trainN$ is valid for a different value range. For example, train 1 is connected to trains $2-5$, train 2 to trains $1, 3-5$ and train 3 to trains $1-2, 4-5$. Due to this, multiple foreach-loops with the same guard are sometimes necessary to include all values. The constants $nTest$ and $nTest2$ are used to trick the program because it is not possible have a range [4..4] or [5..5].

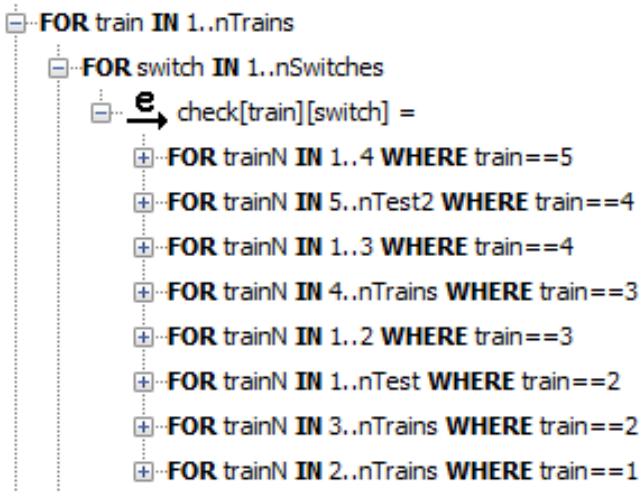

Figure 5.7. Aliases for the uncontrollable events.

The foreach-loops all include the event `cont[switch][train][trainN]`, however it is also important to include the event `cont[switch][trainN][train]` which is exactly the opposite. For example, `cont[1][2][1]` is equivalent to `cont[1][1][2]` and therefore both train 1 and 2 need to include both events.

5.2.3 Platform/Line chosen is reached

Another property that can be verified using the conflict check, described in Section 3.1.2, is if the train actually arrives at the platform or line specified. Two train

specific specifications are added: one to reach the platforms and one to reach the lines.

In figures 5.8 and 5.9 the five platforms are represented by the states $P1$ to $P5$. The events leading to these states are the first step of the path leading to these platforms, independent of the line (i.e. `LPIN[X][Y][1][train]` with X being the line number and Y the platform number). The events leading back to the initial state correspond to the last steps of each path leading to that specific platform.

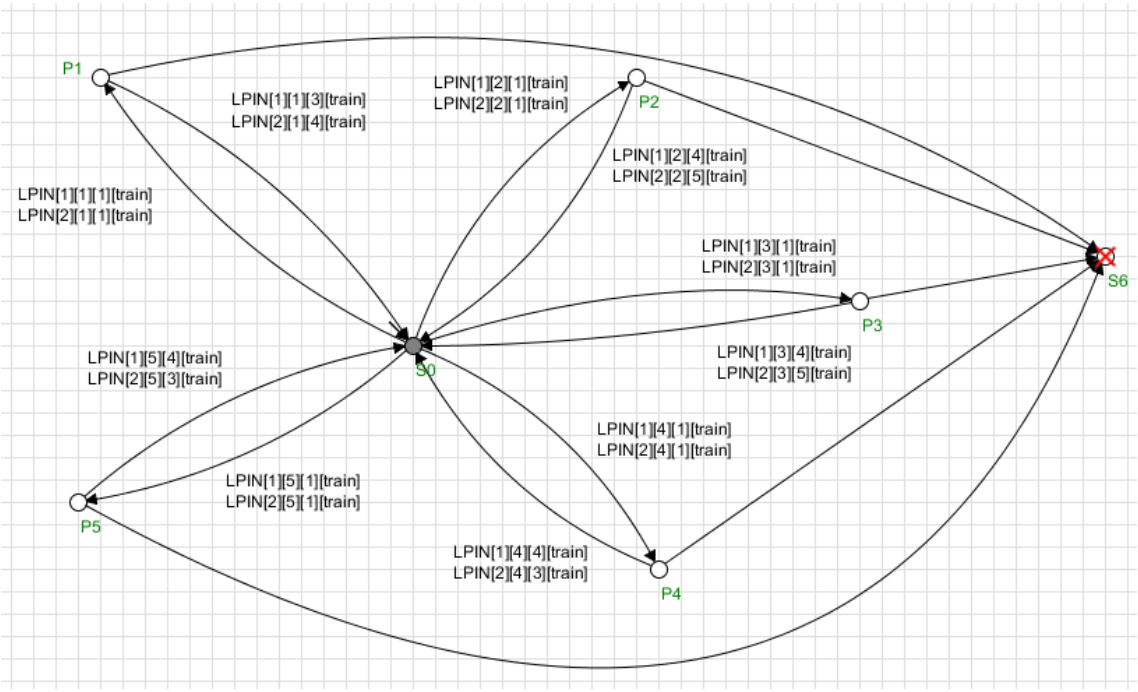

Figure 5.8. Specification to reach Platforms.

Figure 5.9 shows the events on the transitions to the forbidden state. In general, the events leading to any platform and any events leading back from all other platforms are included. The specification for the outbound trains work similarly.

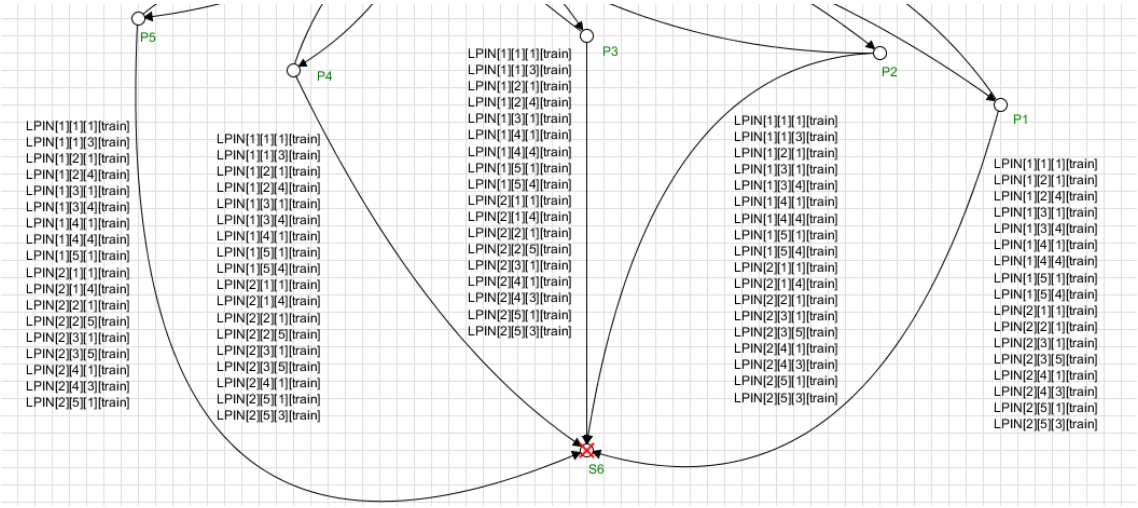

Figure 5.9. Specification to reach Platforms - Transitions to forbidden state.

5.3 Results

Due to the limited memory size of the programs and state space explosion, not all models can be verified for the required amount of trains, platforms, lines and switches. Therefore, it is interesting to measure the amount of time needed to verify specific models. Note that verification options in UPPAAL are limited to the search order while Supremica offers different algorithms which verify using different methods. The verification times for both programs with different models are measured by the same computer and recorded. The verification times vary slightly (±10%) for every test so that the average time of five tests are recorded.

5.3.1 UPPAAL

In UPPAAL, three or more trains cause memory issues for both models for both deadlock and same state checks so that the memory is exhausted after about seven minutes. Table 5.1 shows the different times depending on the model, amount of trains and verification property for the simplified model of the train station with two lines and five platforms.

Approach	Amount of Trains	Verification	Time (s)
Standard	1	deadlock	0.162
Programming	1	deadlock	0.203
Standard	2	deadlock	13.485
Programming	2	deadlock	58.501
Standard	2	Same platform	5.513
Programming	2	Same platform	20.796
Standard	2	Same switch	7.750
Programming	2	Same switch	21.834
Standard	1	Reach platform	0.003
Programming	1	Reach platform	0.029

Table 5.1. Results for UPPAAL for the simplified model of the train station.

It can also be seen from Table 5.1 that the programming approach has a higher verification time than the standard approach. This is due to how UPPAAL handles the programming features used in comparison to additional automata. It is a wonderful example to underline that different ways of modelling can affect the verification.

5.3.2 Supremica

Supremica, compared to UPPAAL, has the possibility to increase its memory to handle larger models (see Section 3.1.6). To compare with UPPAAL, the small model of the train station is verified first with two trains and then with three trains because, as explained in Section 6.2, that is the maximum amount of active trains in the model.

When verifying for the smaller model of the train station, Supremicas analyzer (see Section 3.1.4) is used. However, for the larger model of the train station the system becomes too large for the analyzer to work with, and the conflict and controllability check have to be used instead.

The conflict and controllability checks do not measure the verification time automatically and therefore the results are estimated using a stopwatch. Controllability refers to both a check for switches and platforms compared to UPPAAL where only one switch/platform can be verified at a time. Table 5.2 shows the verification times for the approach with no variables and a model of the train station in Supremica. As already mentioned in Section 2.3.2, the compositional algorithm is only possible for nonblocking verification and is compared with the monolithic approach. The plants and specifications are first synchronized and then verified.

Trains	Verification	Time (h)	Algorithm
2	nonblocking	0.045	Compositional
2	nonblocking	0.130	Monolithic
3	nonblocking	1.5	Compositional
3	nonblocking	12	Monolithic
2	Controllability	0.040	Monolithic
3	Controllability	1.2	Monolithic
1	Reach state	0.002	Monolithic

Table 5.2. Results for Supremica for the simplified model of the train station.

From Table 5.2 it can be seen that the monolithic algorithm takes longer time than the compositional algorithm. This is expected because, as described in Section 2.3.2, the compositional algorithm uses the monolithic approach in an optimised way.

The verification time for trains reaching the desired platform is very small with 0.002 seconds. This is of course partially because the verification only needs one train. Additionally, not every state has to be checked but rather the path of the train followed.

It can be seen from tables 5.1 and 5.2 that the verification time dramatically increases when adding a second train. As discussed in Section 2.3.1, the states increase exponentially when adding models of the train stations. Every time the amount of trains is increased, a complete model of the train station is added.

6

GOTHENBURG TRAIN STATION

6.1 Modelling

After working with both tools and verifying, it becomes clear that UPPAAL cannot handle the size of the train station and therefore Supremica is used to model and verify. This is due to UPPAAL working on 32 bits (see Section 3.2.5) limiting its memory to 4GB while Supremica runs on 64 bits (see Section 3.1.6.1).

The verification was successfully done for the first approach in Supremica, from Section 4.3.1, without variables and with a model of the train station. Therefore, this approach is used to model the Gothenburg train station. The modelling of the train station is equivalent to the simplified model and will therefore, not be explained again.

6.2 Verification

The verification of the simplified and the train station model is a little different. This is due to problems that arise for the train station model which do not exist in the simplified model. The complete model cannot be verified for all 19 trains due to memory issues. Therefore, it is necessary to consider alternative approaches for verification. The question is, how can the model be verified without exceeding the memory limit?

6.2.1 Deadlock

First, the general structure of the model is checked. The new idea is that, assuming that the trains will not affect and block each other (verified later), no deadlock verification with more than one train is needed. The deadlock check ensures, that the model is correct and that a train cannot e.g. go inbound but not outbound afterwards. Again, the initial states are set to marked and a conflict check is performed.

6.2.2 Only one train can claim a switch/platform

Since the shared resources, the switches and platforms, are all claimed simultaneously, trains should not be able to interfere or block each other. However, it is

necessary to verify that only one train can use a switch/platform at a time. This is done by creating a forbidden state which is reached when a switch is claimed twice before it has been released, see Section 5.2.1.

Compared to the simple model of the train station, it is now realised that two trains are enough to verify for an arbitrary amount of trains in the model. This is because all trains share switches and platforms. There is no new switch or platform that appears from adding more trains to the model. Therefore, if one train has claimed a switch and there is no possibility for another train to claim that same switch, then more trains will not make a difference.

One thing to note is that the deadlock check can be included by also setting the initial states to marked.

6.2.3 The trains will not diverge from its route

The last part is to make sure that the trains will only pass the switches they have claimed and that a train ends up at its desired line/platform. If a train diverges from a switch, this would mean that the direction of the switch is wrong. This cannot happen since the system has already been checked for non-blocking through conflict check.

6.3 Results

As seen in Table 6.1, the monolithic and the compositional algorithm were not used for the conflict check. The monolithic algorithm cannot handle more than 64-bits as discussed in Section 3.1.6.4 which is required for the large model of the train station. The available algorithms are tested and the fastest algorithm chosen: Partial Order.

Trains	Verification	Time (s)	Algorithm
1	Conflict	under 1	Partial order
2	Conflict	66	Partial order

Table 6.1. Results for Supremica for the Gothenburg train station.

7
CONCLUSION

The task was to model and verify the logical correctness of the Gothenburg train-station using the programs Supremica and UPPAAL. Additionally, a tutorial for the two programs should be included for future reference.

First, the Gothenburg station with three lines and 19 platforms was reduced to a simple model with two lines and five platforms. This allowed faster implementation and testing for different approaches and ideas. Initially UPPAAL was used to model two different approaches (with and without UPPAALs programming features), however, this could not be continued due to memory issues. Therefore the knowledge gained was used to model four different approaches in Supremica: with(out) variables and with(out) a model of the train station. Only the two approaches using a model of the train station were successfully implemented and functioned as expected.

The properties which were verified for the simplified model of the train station were:

- No deadlock
- Trains are not at the same Switch/Platform
- Platform selected is reached

In UPPAAL, these properties could only be verified for the simplified models of the train station and for two trains. The verification was successful in Supremica for the approach with a model and no variables. The verification for a model of the train station and variables was not done.

The verified approach in Supremica was used to model the Gothenburg train station. However, due to memory issues, the verification was slightly changed. This resulted in the following properties and the corresponding amount of trains needed to verify:

- No deadlock (one train)
- Train cannot claim an already busy switch or platform (two trains)
- The trains follow the specified path (two trains)

This resulted in a successful verification of the Gothenburg train station for an arbitrary amount of trains.

The whole modelling procedure was a process, so that the models in Supremica are more advanced than in UPPAAL. Nevertheless, even if the models in UPPAAL would have been improved, memory issues would most likely still be present. This

process can also be seen when verifying, as this is done differently for the train station than with the simplified model of the train station.

Elaborate tutorials for both Supremica and UPPAAL were successfully written and describe the main features of the program including editor, simulation and verification. Additionally, common problems and tricks were included to aid the user.

The results underline the difficulty present throughout the modelling and verification phase: state space explosion. This is a prevailing problem when verifying automata and solutions or new approaches to deal with this issue are continuously searched for. It is also interesting to notice the effect using different algorithms on the verification time has while verifying with Supremica.

8

FUTURE WORK

The fourth approach in Supremica, with variables and a model of the train station, has not been verified for other properties than not deadlock. Therefore, the other properties can be verified and the approach used to model the Gothenburg train station.

As already described in the theory section about railways, Section 2.1, outside of a train station, the signalling varies slightly and breaking curves for the trains are constantly recalculated. Due to long distances between the switches, the railways are divided into sections and a train reserves a section, instead of a switch. Also, there always exists at least one section between trains because the speed a train travels at is much larger and there needs to be enough time and space to break. Therefore, it would be interesting to model the railways between two cities or before entering the Gothenburg train station.

Another idea is to implement alternative routes for the trains. This means that if a platform is free but the switches are not, an alternative route to the platform is checked. If that route is available, it is pursued.

Furthermore, investigations to the efficiency of the algorithms depending on the different scenarios could be conducted. These verification algorithms could be improved in order to handle much larger systems and/or use less memory.

Bibliography

[1] Google. Google maps. Website, 2016. https://www.google.de/maps (2016-01-20).

[2] Directorate-General for Mobility European Commission and Directorate-General for Energy Transport. *ERTMS – delivering flexible and reliable rail traffic*. European Union, 2006.

[3] Network Rail. An introduction to switches & crossings - network rail engineering education (12 of 15). Youtube, 2012. https://www.youtube.com/watch?v=ZuR5QTlfOzk (2016-03-15).

[4] K.Larsen G. Behrmann, A. David. A tutorial on uppaal 4.0. 2006.

[5] Daniel Bergqvist. Bangårdsritningar. Website, 2016. http://www.bangardar.se/filer/ritningar/J1-Jpeg/Jvm1184_J1_-0326.jpg (2016-01-25).

[6] T. Amnell M. Stigge, A. David. Uppaal 4.0: a small tutorial. 2009.

[7] U. Yıldırım S. Türk A. Sonat M. Söylemez, M. Durmuş. The application of automation theory to railway signalization systems: The case of turkish national railway signalization project. In *18th IFAC World Congress*, pages 10752–10757, 2011.

[8] S. Kurtulan E. Dincel, O. Eris. Automata-based railway signaling and interlocking system design. *IEEE Antennas and Propagation Magazine*, 55(4):308–319, 2013.

[9] European Railway Accident Information. Sweden 2014, version 1, validated accidents, 2014. https://erail.era.europa.eu/csi-data.aspx?country=25&year=2014&public=1 (2016-05-30).

[10] A. Haxthausen L. Vu and J. Peleska. Formal modelling and verification of interlocking systems featuring sequential release. *Science of Computer Programming*, 2016.

[11] M. Yazdi A. Mirabadi. Automatic generation and verification of railway interlocking control tables using fsm and nusmv. *Transport Problems*, 4(1):103–110, 2009.

[12] C. Gao B. Chai T. Xu, T. Tang. Logic verification of collision avoidance system in train control systems. *IEEE*, pages 918–923, 2009.

[13] hnf1930. Railroad switches and how they work. Youtube, 2014. https://www.youtube.com/watch?v=Xfqt33x0QcQ (2016-03-15).

[14] S. van Hoesel P. Zwaneveld, L. Kroon. Routing trains through a railway station based on a node packing model. *European Journal of Operational Research*, 128:14 – 33, 2001.

[15] Bengt Lennartson. Lecture notes in introduction to discrete event systems, 2014.

[16] Extended finite state machine. Wikipedia, 2016. https://en.wikipedia.org/wiki/Extended_finite-state_machine (2016-06-16).
[17] Martin Fabian. Lecture notes in discrete event control and optimisation, 2015.
[18] Formal verification. Wikipedia, 2016. https://en.wikipedia.org/wiki/Formal_verification (2016-04-10).
[19] A. Finkel et al. B. Bérard, M. Bidoit. *Systems and Software Verification*. Springer, 2001.
[20] Sahar Mohajerani. *On Compositional Approaches for Discrete Event Systems Verification and Synthesis*. PhD thesis, Chalmers University of Technology, 2015.
[21] A. M. Partial order reduction with compositional verification. Master thesis, University of Waikato, 2014.
[22] H. Flordal R. Malik K. Åkesson, M. Fabian. Supremica - an integrated environment for verification, synthesis and simulation of discrete event systems. In *8th International Workshop on Discrete Event Systems*, pages 384 – 385, 206.
[23] Graphviz. Graphviz - graph visualization software. Website, 2016. www.graphviz.org/ (2016-06-27).
[24] A. Legay M. Mikučionis D. Poulsen A. David, K. Larsen. Uppaal smc tutorial. *International Journal on Software Tools for Technology Transfer*, 17(4):397–415, 2015.
[25] Gerd Behrmann. Bug 63 - st9 bad alloc exception. Website, 2005. http://bugsy.grid.aau.dk/bugzilla/show_bug.cgi?id=63 (2016-03-01).
[26] K. L. McMillan. The smv system. 2000.
[27] C. Jochim R. Cavada, A. Cimatti. Nusmv 2.6 user manual.

A

Old Map

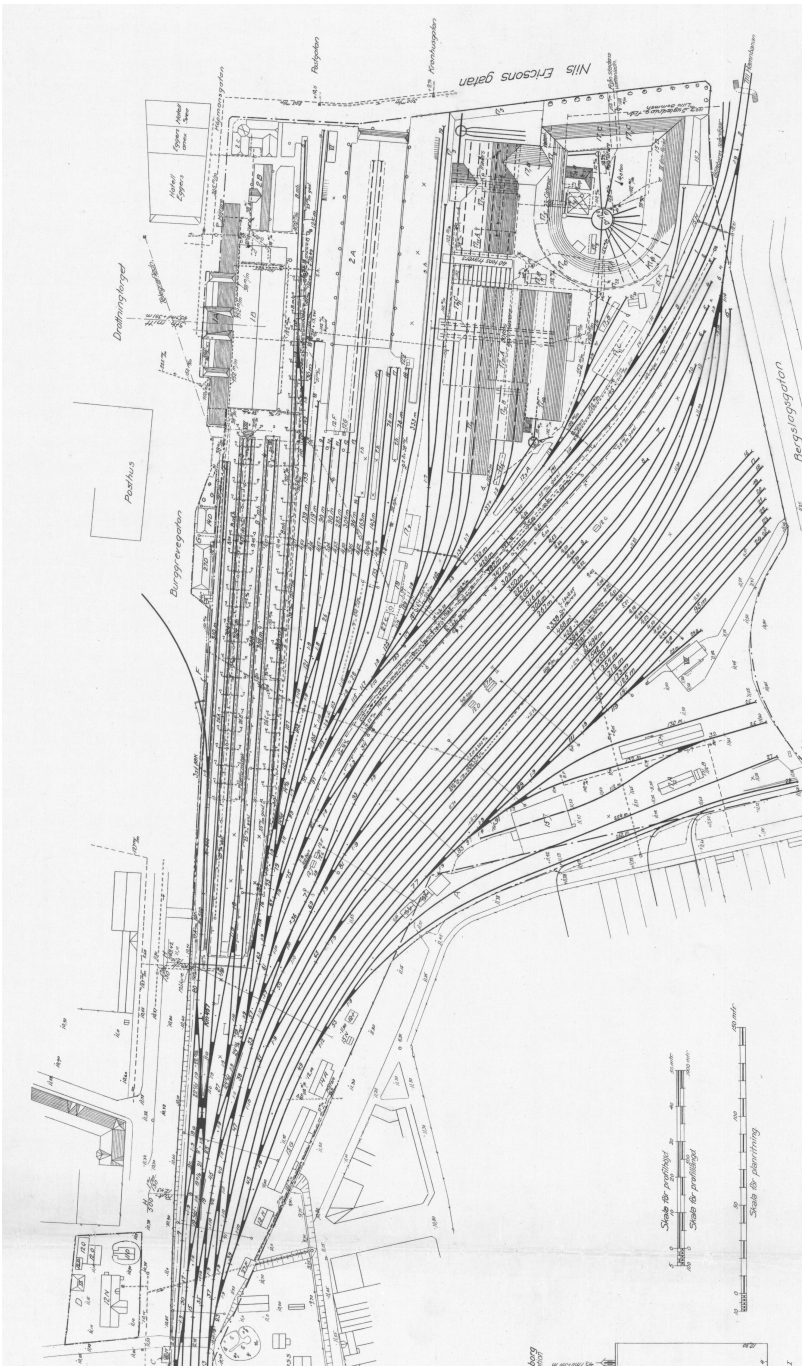

Figure A.1. Old Map of the Gothenburg Train Station [5].