Formal Verification of Railway Timetables - Using the UPPAAL Model Checker

Haxthausen, Anne E.; Hede, Kristian

Published in:
From Software Engineering to Formal Methods and Tools, and Back

Link to article, DOI:
10.1007/978-3-030-30985-5_25

Publication date:
2019

Document Version
Early version, also known as pre-print

Citation (APA):
Haxthausen, A. E., & Hede, K. (2019). Formal Verification of Railway Timetables - Using the UPPAAL Model Checker. In *From Software Engineering to Formal Methods and Tools, and Back* (pp. 433-448). Springer. https://doi.org/10.1007/978-3-030-30985-5_25

Formal Verification of Railway Timetables - Using the UPPAAL Model Checker

Anne E. Haxthausen and Kristian Hede

DTU Compute, Technical University of Denmark, Kongens Lyngby, Denmark.
aeha@dtu.dk and krhede@gmail.com

Abstract. This paper considers the challenge of validating railway timetables and investigates how formal methods can be used for that. The paper presents a re-configurable, formal model which can be configured with a timetable for a railway network, properties of that network, and various timetabling parameters (such as station and line capacities, headways, and minimum running times) constraining the allowed train behaviour. The formal model describes the system behaviour of trains driving according to the given railway timetable. Model checking can then be used to check that driving according to the timetable does not lead to illegal system states. The method has successfully been applied to a real world case study: a time table for 12 trains at Nærumbanen in Denmark.

Keywords: formal methods · model checking · UPPAAL · railways · timetables

1 Introduction

This paper considers the challenge of validating time tables in the railway domain.

Background. As timetables specify how the trains should run in the railway network, they have a great influence on the railway operations. Therefore, for any railway operator, it is of very high priority that the timetables are feasible as well as robust against delays. However, the process of creating train timetables [6] is very complicated and goes through many steps taking various wishes, requirements and scheduling constraints into account. Traditionally, time tables are created in a stepwise manner, where schedules are manually adjusted until all constraints are met.[1] When a timetable has been created in this way, it should be verified that it is feasible and satisfies all the many stated requirements and scheduling constraints. For instance, it should be checked that at any time the number of trains waiting at a station does not exceed the station capacity and that the minimal headways between trains are satisfied. Tools for automated checking of all such requirements are highly needed. Today there are

[1] Research in optimisation models and techniques for generating optimal timetables have been done [6], but these are very rarely used by the railway operators.

some timetabling tools [10] like RailSys and TPS on the market which can be used for managing timetables and estimating the effect of train delays. The tools typically have functionality for displaying time tables graphically (showing the planned train positions as a function of time). In order to verify a timetable, one can inspect such graphs and manually check whether the requirements are fulfilled. Although this is very useful, it would be desirable to have a tool that could automatically check all requirements.

Contribution. In this paper we propose a fully automated method for verifying timetables. This method is formal and based on model checking. The idea is as follows: To check a timetable for a given railway network, one should create a real-time model simulating how trains move around in the network, from station to station over time, according to the timetable. During the simulation, a number of required properties of timetables should be checked. If any of these fail, the system should go into an error state. Then one can model check that all trains reach their final destination without ending in an error state. We have explored this idea using UPPAAL[3], which is an integrated tool environment for modelling, validation and verification of real-time systems modelled as networks of timed automata. We choose to use UPPAAL as our system is a real-time system and the UPPAAL symbolic model checker has shown to very be effective. The UPPAAL model is re-configurable: It can be configured with a timetable and a railway network description including various scheduling parameters (such as station and line capacities, headways, and minimum running times) constraining the allowed train behaviour. The verification method has successfully been applied to a real world case study: a time table for 12 trains at Nærumbanen Local Railway in Denmark.

To our best knowledge it is a novelty to verify railway timetables by model checking.

Related work. Over the last decade formal methods have been widely used for developing railway systems, cf., the surveys given in [2, 5]. According to these surveys, formal methods have mostly been applied to railway interlocking systems[2] (as e.g. in [11, 8, 7, 12, 1]) and other control system components, but less for railway operations, where formal methods have primarily been used for capacity analyses (as e.g. in [4]).

Paper overview. First, in Sect. 2 and Sect. 3, short, informal introductions to the UPPAAL modelling language and to the domain of railway timetables are given, respectively. Then Sect. 4 presents a formal UPPAAL model of trains driving according to a given railway time table, whereupon, in Sect. 5, it is described how the verification of model instances is done. Finally, Sect. 6 gives a conclusion and states ideas for future work.

[2] An interlocking system is a signalling system component responsible for the safe routing of trains through a railway network.

2 The UPPAAL Modelling Language

This section gives an ultra short, informal introduction to the major UPPAAL modelling language constructs used in this paper. The reader is assumed familiar with the theory of timed automata. For more details, especially on semantics of the concurrency construct, the reader should consult [3].

In UPPAAL a system is modelled as a network of parallel timed automata (called processes) which are finite-state machines extended with time in the form of real-valued clocks which progress synchronously and with data variables of simple data types (bounded integers, arrays, etc.).

The specification of a system model consists of (1) templates for timed automata, (2) declarations of clocks, data variables, constants, channels, and functions which can be used in the templates, and (3) a system declaration which is a parallel composition of processes which are instances of the templates. The processes can communicate asynchronously via shared variables or synchronously via channels.

A timed automaton consists of locations and edges.

In the UPPAAL graphical representation of timed automata, locations are shown as circles and may have a name shown in red colour. The *initial* location is shown by double circles. A location may be *committed* (shown as a circle with a C inside). If a process is in a committed location, the time must not pass and the next system transition must involve an edge from a committed location. A location may be labelled with an *invariant* (shown in purple colour) which is Boolean expression over variables and clocks. The process can stay in that location and let the time pass as long as the invariant is satisfied, but when the invariant becomes false, it must leave the location.

In the graphical representation, edges are shown as arrows. An edge may be labelled with (1) a guard (shown in green colour) which is Boolean expression over variables and clocks determining when the edge is enabled and can be fired, (2) updates of variables and clocks (shown in blue colour) that should be executed when the edge is fired, and (3) synchronisations of the form $c?$ or $c!$ over a channel c (shown in light blue colour). (An edge labelled with $c!$ in one process may synchronise with an edge labelled with $c?$ in another process provided that both edges are enabled.)

3 Domain Description

This section presents the definitions of basic terms of timetabling, as used in this paper, and requirements to timetables.

3.1 Basic concepts and terms

Railway networks. A *railway network* consists of *stations* and *open lines* between stations. A *station* can be a passenger station where people embark and disembark trains, or a technical station where e.g. trains can overtake each other.

An *open line* is a collection of tracks between two stations. In this paper it is assumed that each open line is either a *single track* or is a *double track* (has two tracks).[3] For double track lines it is assumed that there is one track dedicated to each driving direction.

Railway timetables. A *railway timetable* for a given railway network is a collection of individual *train timetables/schedules* for train journeys in the network. Each train timetable has a unique (train) identifier and a list of stations at which the train should stop, along with the arrival and departure times at those stops.

A train is said to *drive according to its timetable*, if it arrives at and departs from the stations in its timetable at the stated arrival and departure times.

Parameters in timetabling. The requirements to timetables are expressed in terms of a number of *scheduling parameters* described below. These depend among others on the track layout, train properties, and signalling system properties. It is out of the scope to explain how these parameters are determined - for an explanation of that, see e.g. [9]. Often some time supplements are added to / included in the minimum times mentioned below to make the timetable robust against small train delays.

The *capacity* of a station is the maximal number of trains that are allowed to be at the station at the same time. (In this paper, for simplicity, we will assume that if a station has capacity n, then it consists of n parallel tracks, each track having capacity for one train. Furthermore, we will assume that each station track is connected to each open line track.) The *minimum dwell time* of a train at a station is the minimum time it must wait at the station, such that there is time to open and close doors and let people enter and exit the train.

The *capacity* of an open line is the maximal number of trains that are allowed to be on each of its tracks at the same time. An open line also has a *minimum running time*, i.e. the time it must at least take for a train to drive between the two stations connected by the open line. Furthermore, an open line has a *minimum headway*, i.e. the minimum time that must pass between two following trains both entering the same track or both leaving the same track. Stations can also have different kinds of minimum headways between two trains entering/leaving the stations (depending on the signalling system). In this paper we just illustrate how this can be done for a minimum headway between two trains entering the station.

3.2 Requirements

For a timetable to be *valid*, a number of requirements must hold when the trains drive according to the timetables. Below we state examples of typical requirements, but these might differ for different railways.

[3] In Denmark this is the case for most railway networks, with only very few exceptions.

no overtaking: Trains must not overtake another train on the same track of an open line (as this is physically not possible and would prevent the plan from being executable.)
no opposing trains: Single track open lines must not be utilised in both directions simultaneously (as that would lead to deadlocks preventing the plan from being executable.)
open line running times: Trains must satisfy the minimum running times of the open lines.
open line capacities: The capacity of open lines must never be exceeded.
open line headways: The headway times of open lines must be respected.
station dwell times: Trains must satisfy the dwell times at the stations.
station capacities: The capacity of stations must never be exceeded.
station headways: The headway times of stations must be respected.

It is seen how the two first requirements concern internal inconsistencies in a given timetable (in this case conflicting train schedules)[4], while the remaining ones concern the timetabling parameters identified above.

4 UPPAAL Model

This section presents the UPPAAL model created for verifying timetables. The model can be found here: `http://www2.compute.dtu.dk/~aeha/RobustRailS/data/timetabling/uppaal-models/`. The model has been designed to be re-configurable, so that it can be re-used for a whole class of railway networks and timetables, without having to change the templates of the model, but only constant data representing the railway network and the timetables.

4.1 Overview

The model consists of the following three parts:

1. Global declarations of (1) the configuration data (constants) defining a railway network layout and timetabling parameters and a timetable for that network, (2) clocks, (3) variables, (4) types, and (5) functions.
2. Two templates: `Train` and `Initialiser`. A process instance of the `Train` template represents a single train, running according to one of the timetables in the global declarations. The `Initialiser` template is used to initialise the system.
3. The system declaration, which creates a single `Initialiser` process and a `Train` process for each of the trains in the timetable.

The following sections describe these parts, except the types and functions.

[4] Note that the interlocking system has the responsibility to ensure that collisions would not happen in such cases.

4.2 Railway network data

The railway network data (describing the network layout and the timetabling parameters) are represented by a collection of constants as shown in Listing 1.1, where the constants are configured for the network of *Nærumbanen* in Denmark which has 9 stations connected one by one by single track open lines. The first constants give the number of stations, the number of open lines, and the station ids. The `stationTable` is an array with one entry for each station giving the capacity (`capacity`) and minimum headway (`HWT`) for that station. The `openLineTable` is an array with one entry for each open line (represented as a pair of the two stations it is connecting) stating the kind of line (`false` means single track and `true` means double track) and giving the minimum running time (`MRT`), the capacity (`capacity`), and the minimum headway (`HWT`) for that open line.

```
const int STATIONS  = 9; //number of stations
const int OPENLINES = 8; //number of open lines

//station ids for station numbers 0 .. STATIONS-1:
const int remisen = 0;
...
const int narum = 8;

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

const OpenLineTableEntry openLineTable[OPENLINES] =
{{{remisen,jagersborg},false,2,1,1},
{{jagersborg,norgaardsvej},false,1,1,1},
{{norgaardsvej,lyngbylokal},false,1,1,1},
{{lyngbylokal,fuglevad},false,1,1,1},
{{fuglevad,brede},false,2,1,1},
{{brede,orholm},false,2,1,1},
{{orholm,ravnholm},false,1,1,1},
{{ravnholm,narum},false,2,1,1}};
```

Listing 1.1. Example of network data for *Nærumbanen*.

4.3 Timetable data

The railway timetable is represented by a collection of constants as shown in Listing 1.2, where the constants are configured for an extract of the timetable of

Nærumbanen in Denmark. `stops` is an array of timetables – one for each train, and each timetable is an array of stop entries. Each stop entry gives the station id (`StationID`), the arrival time (`AT`), the departure time (`DT`), and the minimum dwell time[5] (`DWT`) for a stop. For the shown train timetables the minimum dwell time has been chosen to be zero for stations where the trains need not to stop.

```
const int TRAINS = 2; //number of trains in the timetable
const int TRAINSTOPS[TRAINS] = {8,9}; //number of stops for
                                       each train time table
const int MAXLENGTH = 9; //max number of stops for any train

/* timetable for each train */
const StopEntry stops[TRAINS][MAXLENGTH] = {
//timetable for train 0
{{jagersborg,26,30,1},
{norgaardsvej,31,31,0},
{lyngbylokal,33,33,0},
{fuglevad,35,35,0},
{brede,37,37,0},
{orholm,39,40,1},
{ravnholm,41,41,0},
{narum,43,47,1},
{-1,-1,-1,-1}},
//timetable for train 1
{{remisen,0,34,0},
{jagersborg,36,40,1},
{norgaardsvej,41,41,0},
{lyngbylokal,43,43,0},
{fuglevad,45,45,0},
{brede,47,47,0},
{orholm,49,50,1},
{ravnholm,51,51,0},
{narum,53,57,1}}};
```

Listing 1.2. Example of a timetable for Nærumbanen.

4.4 Clocks

To express time constraints in the model, a number of clocks are introduced in addition to the system clock `time`:

```
clock TrainClock[TRAINS];
clock openLineLastEntered[OPENLINES][2];
clock openLineLastExit[OPENLINES][2];
clock stationLastEntered[STATIONS];
```

For each train `id`, `TrainClock[id]` is used to record how long time the train has been in its current location. For each open line number `o` (which is an index in the

[5] To allow for different minimum dwell times at the same station for different trains, this scheduling parameter has been placed here and not in `stationTable`.

openLineTable) and driving direction d $\in$ 0 .. 1, openLineLastEntered[o][d] and openLineLastExit[o][d] records the time elapsed since a train last time entered o and exited o in direction d, respectively. Direction 0 is used for the direction towards the first station found in the open line entry openLinetable[o], and direction 1 is used for the other direction. For each station id s, stationLastEntered[s] records the time elapsed since a train last time entered s.

4.5 Variables

To express requirement checks in the model, a number of variables are introduced:

```
int currentStop[TRAINS];
int[0, TRAINS] trainsAtStation[STATIONS];
int[0, TRAINS] trainsAtOpenLine[OPENLINES][2];
queueEntry queue[OPENLINES][2][TRAINS];
```

For each train id t, currentStop[t] stores the station id of the station at which the train is currently waiting or towards which it is currently moving (when it is on an open line). For each station s, trainsAtStation[s] stores the number of trains currently waiting at s. For each open line number o and direction d $\in$ 0 .. 1, trainsAtOpenLine[o][d] stores the number of trains currently driving on o in direction d. For each open line number o and direction d, queue[o][d] is a queue (an array) of entries for trains currently present on o in direction d. Each entry contains the id of a train and the time it is planned to leave the open line. The trains appear in the order they entered o, with the head in queue[o][d][0].

4.6 The Initialiser template

The Initialiser template is shown in Figure 1. The initial location is commit-

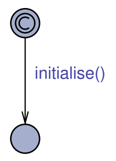


Fig. 1. The Initialiser Template.

ted making the outgoing edge of that location, the first to be fired in the system - without any delay. This edge invokes a function to initialise global variables and clocks.

4.7 The `Train` template

The `Train` template has a single parameter, `id`, which can be instantiated with the identifier of a train from the timetable. The resulting train process simulates the behaviour of the train driving perfectly according to its timetable in `stops[id]` without any deviations. During the simulation, the timetable is checked to satisfy the requirements stated in Sect. 3.2.

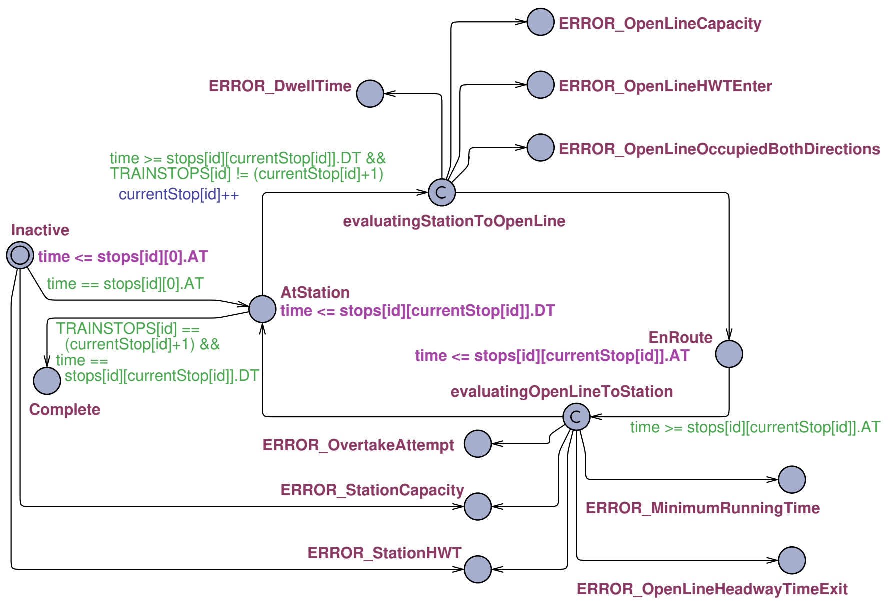

ERROR_OpenLineCapacity
ERROR_DwellTime
ERROR_OpenLineHWTEnter
ERROR_OpenLineOccupiedBothDirections
time >= stops[id][currentStop[id]].DT &&
TRAINSTOPS[id] != (currentStop[id]+1)
currentStop[id]++
evaluatingStationToOpenLine
Inactive
time <= stops[id][0].AT
time == stops[id][0].AT
AtStation
time <= stops[id][currentStop[id]].DT
TRAINSTOPS[id] ==
(currentStop[id]+1) &&
time ==
stops[id][currentStop[id]].DT
Complete
EnRoute
time <= stops[id][currentStop[id]].AT
evaluatingOpenLineToStation
time >= stops[id][currentStop[id]].AT
ERROR_OvertakeAttempt
ERROR_StationCapacity
ERROR_MinimumRunningTime
ERROR_StationHWT
ERROR_OpenLineHeadwayTimeExit

Fig. 2. The `Train` Template only showing state invariants and edge labels expressing conditions ensuring that the train drives according to its time table.

Figure 2 shows the template with all its locations and edges, but for readability only with those state invariants and edge labels that enforce the train to drive according to its time table. Later, the remaining edge labels will be explained.

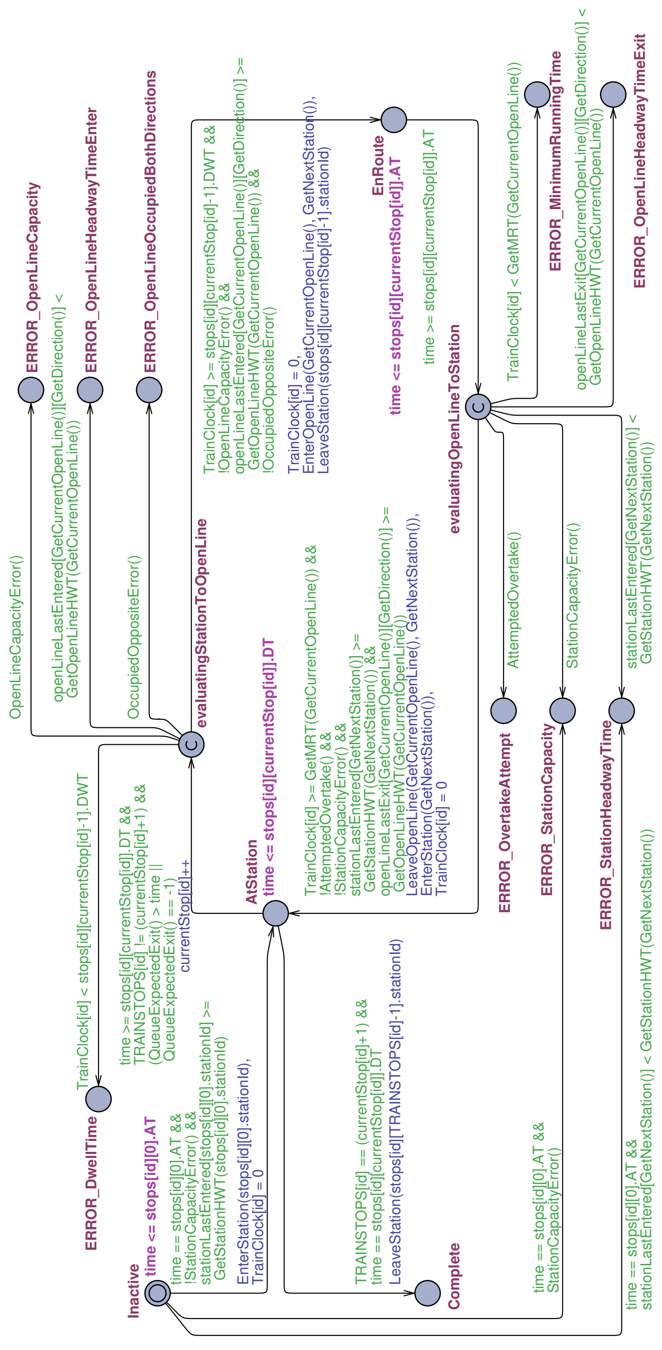
As it can be seen, a train has the following locations:

- `Inactive` which is the initial location.
- `AtStation` which reflects that at the current time the train is waiting at a station according to the timetable.
- `EnRoute` which reflects that at the current time the train is driving on the open line between two consecutive stations in its timetable.
- `evaluatingStationToOpenLine` which is a committed, intermediate location between `AtStation` and `EnRoute`. It is an auxiliary location (not existing in a real system) which is only used for checking that neither leaving the current station nor entering the next open line would break a requirement.

- `evaluatingOpenLineToStation` which is a committed, intermediate location between `EnRoute` and `AtStation`. It is an auxiliary location (not existing in a real system) which is only used for checking that neither leaving the current open line nor entering the next station would break a requirement.
- An error location for each possible error (breaking a requirement). For instance, if the open line capacity is exceeded at time `t`, the error state `ERROR_OpenLineCapacity` will be reached at that time.
- `Complete` which is a location that is reached if and when the train has successfully run through its whole timetable and reached its destination without any errors have been detected. If all trains reach this location, the complete railway timetable has successfully been validated.

Figure 2 shows those state invariants and edge labels that enforce the train to drive according to its time table: A train starts in the `Inactive` location. At the start time of its timetable (i.e. the arrival time `stops[id][0].AT` of its first stop) it will enter the `AtStation` location modelling that it is at the first stop of its timetable. Whenever the `AtStation` location is entered, if the train has reached its final stop (`TRAINSTOPS[id] == currentStop[id] + 1`) and final time of the timetable, it will immediately go to the `Complete` location. Otherwise, it must leave the `AtStation` location and enter the `evaluatingStationToOpenLine` location at its departure time `stops[id][currentStop[id]].DT` for its current station. During that transition the `currentStop[id]` counter is incremented by one.
In the `evaluatingStationToOpenLine` location it is checked whether some requirements are broken. If that is the case, the system goes into an error location, and otherwise it goes to the `EnRoute` location.
Similarly, whenever the train is in the `EnRoute` location (i.e. is driving on the open line towards the next station in its route), it must leave the location and enter the `evaluatingOpenLineToStation` location at the arrival time `stops[id][currentStop[id]].AT` for the train's next station.
In the `evaluatingOpenLineToStation` location it is checked whether some requirements are broken. If that is the case, the system goes into an error location, and otherwise it goes to the `AtStation` location.

Figure 3 shows the full model with all edge labels included. The additional guards and variable/clock updates are used to express the checks for the requirements stated in Sect. 3.2. On each edge from the `evaluatingStationToOpenLine` and `evaluatingOpenLineToStation` locations to an error state for a certain kind of error, there is a guard expressing that this error is found. The guard of the edge from the `evaluatingStationToOpenLine`/`evaluatingOpenLineToStation` location to the `EnRoute`/`AtStation` location is a conjunction of the negated error conditions in the guards of each of the edges leading to an error location from the `evaluatingStationToOpenLine`/`evaluatingOpenLineToStation` location. The guards are expressed in terms of the clocks and variables declared in Sections Sect. 4.4 and Sect. 4.5. These clocks and variables are updated when a train is entering a station or an open line.



Inactive
time <= stops[id][0].AT
time == stops[id][0].AT && !StationCapacityError() && stationLastEntered[stops[id][0].stationId] >= GetStationHWT(stops[id][0].stationId)
EnterStation(stops[id][0].stationId), TrainClock[id] = 0
ERROR_DwellTime
TrainClock[id] < stops[id][currentStop[id]-1].DWT
time >= stops[id][currentStop[id]].DT && TRAINSTOPS[id] != (currentStop[id]+1) && (QueueExpectedExit() > time || QueueExpectedExit() == -1)
currentStop[id]++
AtStation
time <= stops[id][currentStop[id]].DT
TRAINSTOPS[id] == (currentStop[id]+1) && time == stops[id][currentStop[id]].DT
LeaveStation(stops[id][TRAINSTOPS[id]-1].stationId)
Complete
time == stops[id][0].AT && StationCapacityError()
time == stops[id][0].AT && stationLastEntered[GetNextStation()] < GetStationHWT(GetNextStation())
evaluatingStationToOpenLine
OpenLineCapacityError()
ERROR_OpenLineCapacity
openLineLastEntered[GetCurrentOpenLine()][GetDirection()] < GetOpenLineHWT(GetCurrentOpenLine())
ERROR_OpenLineHeadwayTimeEnter
OccupiedOppositeError()
ERROR_OpenLineOccupiedBothDirections
TrainClock[id] >= stops[id][currentStop[id]-1].DWT && !OpenLineCapacityError() && openLineLastEntered[GetCurrentOpenLine()][GetDirection()] >= GetOpenLineHWT(GetCurrentOpenLine()) && !OccupiedOppositeError()
TrainClock[id] = 0, EnterOpenLine(GetCurrentOpenLine(), GetNextStation()), LeaveStation(stops[id][currentStop[id]-1].stationId)
EnRoute
time <= stops[id][currentStop[id]].AT
time >= stops[id][currentStop[id]].AT
evaluatingOpenLineToStation
TrainClock[id] < GetMRT(GetCurrentOpenLine())
ERROR_MinimumRunningTime
openLineLastExit[GetCurrentOpenLine()][GetDirection()] < GetOpenLineHWT(GetCurrentOpenLine())
ERROR_OpenLineHeadwayTimeExit
AttemptedOvertake()
StationCapacityError()
stationLastEntered[GetNextStation()] < GetStationHWT(GetNextStation())
ERROR_OvertakeAttempt
ERROR_StationCapacity
ERROR_StationHeadwayTime
TrainClock[id] >= GetMRT(GetCurrentOpenLine()) && !AttemptedOvertake() && !StationCapacityError() && stationLastEntered[GetNextStation()] >= GetStationHWT(GetNextStation()) && openLineLastExit[GetCurrentOpenLine()][GetDirection()] >= GetOpenLineHWT(GetCurrentOpenLine())
LeaveOpenLine(GetCurrentOpenLine(), GetNextStation()), EnterStation(GetNextStation()), TrainClock[id] = 0

Fig. 3. The complete `Train` Template.

Below we will give an overall idea of how the updates and checks for the various requirements are done.[6]

The **openline running times** requirement: In order to be able to measure how long time a train has been driving on an open line, the `TrainClock[id]` clock is reset to 0 each time the train enters the `EnRoute` location. Then, the following condition, when evaluated in the `evaluatingOpenLineToStation` location (i.e. when the train leaves the open line), expresses that the train has at least spent the minimum running time of the current open line in the `EnRoute` location:

```
TrainClock[id] >= GetMRT(GetCurrentOpenLine())
```

where `GetCurrentOpenLine` is an auxiliary function which returns the current open line `o` and `GetMRT(o)` returns the minimum running time of `o` as stated in the `openLineTable`: `openLineTable[o].MRT`.

The **dwell times** requirement is checked similarly using `TrainClock[id]`.

The **station headways** requirement is checked in a similar way using the `stationLastEntered[s]` clock which is reset (by an `EnterStation(s)` function call) each time a train enters station `s`.

The **open line headways** requirements for entering and leaving an open line `o` are checked in a similar way using the `openLineLastEntered[o][d]` and the `openLineLastExit[o][d]` clocks, respectively. These clocks are reset to 0 (by `EnterOpenLine(o, s)` and `LeaveOpenLine(o, s)` function calls, respectively, where `s` is the next station/stop) each time some train enters `o` and leaves `o` in direction `d`[7] (towards `s`), respectively.

The **station capacity** requirement: In order to be able to count the number of trains present on a station `s`, the `trainsAtStation[s]` variable is incremented by 1 (by an `EnterStation(s)` function call) each time some train enters `s` and is decremented by 1 (by a `LeaveStation(s)` function call) each time some train leaves `s`. Before the train is entering a new station `s`, the following condition is used to check that the capacity of that station will not be exceeded, if the train enters the station:

```
trainsAtStation[s] < GetStationCapacity(s)
```

where `GetStationCapacity(s)` returns the station capacity of `s` as stated in the `stationTable`: `stationTable[s].capacity`

The **open line capacity** requirement is checked in a similar way for an open line `o` using the `trainsAtOpenLine[o][d]` variables which are incremented/decremented (by an `EnterOpenLine(o, s)`/`LeaveOpenLine(o, s)` function call) each time a train enters/leaves `o` in direction `d` (towards `s`).

The **no opposing trains** requirement is checked before a train is entering a single track open line `o` in direction `d` by the condition

`trainsAtOpenLine[o][d']` $= 0$

where `d'` is the opposite direction of `d`.

[6] There is no space to show and explain the many auxiliary functions used in the edge labels to implement that.

[7] `d` can be found from `s` as explained in Sect. 4.4.

The **no overtaking** requirement is checked using the `queue[o][d]` variables (introduced in Sect. 4.5). Each time some train enters/leaves an open line `o` in direction `d` towards the next station `s`, a train entry for that train (keeping its name and planed arrival time at next station) is pushed/popped to/from the queue in `queue[o][d]` (by an `EnterOpenLine(o, s)/LeaveOpenLine(o, s)` function call). When a train is leaving its current open line `o` in direction `d`, the following condition is used to check that the first train in the queue `queue[o][d]` is the train it-self which means that it has not overtaken any train:

```
GetQueueFirstTrain(o, d) = id
```

where `GetQueueFirstTrain(o, d)` returns the train id in `queue[o][d][0]`.

A concurrency issue: If one train t_1 enters an open line `o` at the same time t (according to the timetable) as another train t_2 leaves the same open line according to the timetable (both driving in the same direction `d`), the interleaving semantics of UPPAAL would lead to two different traces sequencing the two events: one in which t_1 enters first and t_2 leaves afterwards, and one in which the order is opposite. If we assume that there are no other trains on the line, the value of `trainsAtOpenLine[o][d]` will at time t in the first trace first be 2 and then 1, while in the other trace it will be 1 in both states. If the capacity of the line is 1, then the first trace will give rise to a false line capacity error. Therefore, we should ensure that only the second trace is considered. In order to ensure this, the following time constraint was added to the guard of the edge from the `AtStation` location to the `evaluatingStationToOpenLine` location:

```
QueueExpectedExit() > time
```

where `QueueExpectedExit()` returns the expected exit time of the train in the head of the queue in `queue[o][d]`: `queue[o][d][0].expectedExit`. This would mean, that a train can only enter the open line at time `time`, if the next train that will leave the same open line in the same direction will do that later on. Hence, for the example above, trace 1 would not anymore be possible.

Note that we do not have the same concurrency issue for stations, since if one train enters the station at the same time as another leaves it, there should be space for both of them at the same time, so both traces should be considered.

4.8 System declaration

The system declaration

```
system Train, Initialiser;
```

creates a single `Initialiser` process and a `Train` process `Train(id)` for each train identifier `id`, `id = 0, .. ,TRAIN-1`.

5 Verification

This section explains how a railway timetable can be verified by model checking a system model instantiated with data for that timetable against some properties.

5.1 Properties

To verify that a railway timetable is valid, it should be checked that eventually all the trains have reached their destination as expressed by the property

```
A<> forall(i: t_id) Train(i).Complete
```

If this property is true, it in particular follows that no error states have been reached and therefore all the requirements stated in Sect. 3.2 are fulfilled for trains driving according to the timetable.

However, if the property fails, one can't from the property itself conclude what went wrong. In such a case, if one wishes to know which kind of errors were encountered and which trains did not reach their destination, one can instead check

`A[] forall(i: t_id) not Train(i).`$ERROR\_S$

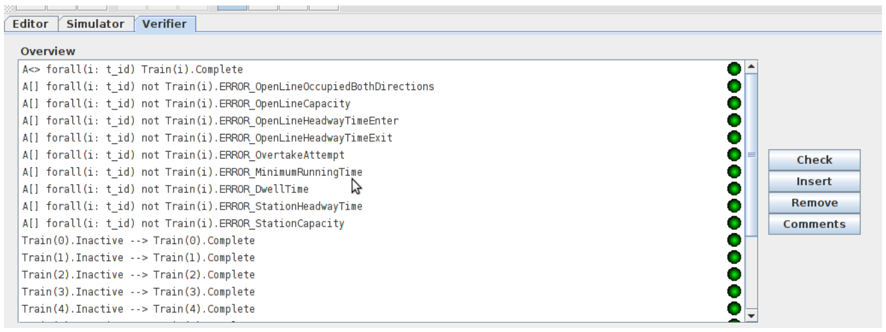
for each error location $ERROR\_S$, and

```
Train(i).Inactive --> Train(i).Complete
```

and for each train i.

5.2 Verification results

For an extended version of the model created in Sect. 3.2 for Nærumbanen (now with 12 trains), we used the symbolic model checker of UPPAAL to check the properties above. All properties were successfully verified as shown by green lamps in Figure 4.



```
A<> forall(i: t_id) Train(i).Complete
A[] forall(i: t_id) not Train(i).ERROR_OpenLineOccupiedBothDirections
A[] forall(i: t_id) not Train(i).ERROR_OpenLineCapacity
A[] forall(i: t_id) not Train(i).ERROR_OpenLineHeadwayTimeEnter
A[] forall(i: t_id) not Train(i).ERROR_OpenLineHeadwayTimeExit
A[] forall(i: t_id) not Train(i).ERROR_OvertakeAttempt
A[] forall(i: t_id) not Train(i).ERROR_MinimumRunningTime
A[] forall(i: t_id) not Train(i).ERROR_DwellTime
A[] forall(i: t_id) not Train(i).ERROR_StationHeadwayTime
A[] forall(i: t_id) not Train(i).ERROR_StationCapacity
Train(0).Inactive --> Train(0).Complete
Train(1).Inactive --> Train(1).Complete
Train(2).Inactive --> Train(2).Complete
Train(3).Inactive --> Train(3).Complete
Train(4).Inactive --> Train(4).Complete
```

Fig. 4. Verification results for the timetable for Nærumbanen in Denmark.

Verification[8] using the `A<> forall(i: t_id) Train(i).Complete` property took less than a second. The alternative (more informative) verification of all the detailed properties (for individual errors and individual trains) took 5 - 6 seconds.

[8] The experiments were done with an Intel(R) Core(TM) i5-5200U processor at 2.2 GHz with 12 GB RAM.

We also made examples with illegal timetables to test that the model checker would be able to catch the different kinds of errors. For instance, we changed the capacity of `orholm` station from 2 to 1, such that the station capacity would be exceeded when two trains should pass each other at the station.

Once an error has been detected, it is possible to identify how, where and when it occurred by choosing the 'Some' option of the Diagnostic Trace. Then the failing query should be chosen and verified by itself, resulting in a diagnostic trace, which will provide the entire diagnostic trace up until the state of the error.

6 Conclusion and Future Plans

This paper has shown how railway timetables can be formally verified to be feasible (executable) and satisfy a number of scheduling constraints. The UPPAAL model checker was used for that purpose. To our knowledge, it is the first time that model checking has been used for that. The approach is quite promising as it was applied successfully to a real-world case study for which the timetable for 12 trains on a railway with 9 stations was successfully verified in less than a second. In future work, it would be interesting to apply the method to larger examples.

It should be straight forward to extend the model to allow more than two tracks on open lines and take more details of station topologies into account (e.g. the number of platforms and which open line tracks they are connected to) and include scheduled track occupation information for trains in their timetables. Furthermore, it could be interesting to investigate how model checking could be used for investigating the effects of train delays.

We have also investigated how model checking can be used to generate timetables and we plan to describe those results in a future paper.

Acknowledgements

This paper is dedicated to Professor Stefania Gnesi on the occasion of her 65th birthday. The first author have had the great pleasure of meeting Stefania Gnesi at many occasions and would like to thank her for inspiration, discussions, and collaboration, especially in European Technical Working Group on Formal Methods in Railway Control.

The research was partially funded by the RobustRailS project (2012-17) granted by Innovation Fund Denmark under grant agreement 0603-00483B. The authors are grateful to the organisers of the Festschrift Symposium for giving the opportunity to prepare this paper. We would like to thank the anonymous reviewers for their comments, and we would like to thank Alex Landex for discussions and sharing of his deep knowledge about railway timetabling while he was employed as a researcher in railway operations at DTU Transport.

References

1. Banci, M., Fantechi, A., Gnesi, S.: Some Experiences on Formal Specification of Railway Interlocking Systems Using Statecharts. In: TRain Workshop at SEFM (2005)
2. Basile, D., ter Beek, M.H., Fantechi, A., Gnesi, S., Mazzanti, F., Piattino, A., Trentini, D., Ferrari, A.: On the industrial uptake of formal methods in the railway domain - A survey with stakeholders. In: Furia, C.A., Winter, K. (eds.) Integrated Formal Methods. pp. 20–29. Springer International Publishing (2018)
3. Behrmann, G., David, A., Larsen, K.G.: A Tutorial on UPPAAL. In: Bernardo, M., Corradini, F. (eds.) Formal Methods for the Design of Real-Time Systems: International School on Formal Methods for the Design of Computer, Communication, and Software Systems, Revised Lectures, pp. 200–236. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
4. Di Giandomenico, F., Fantechi, A., Gnesi, S., Itria, M.L.: Stochastic model-based analysis of railway operation to support traffic planning. In: Gorbenko, A., Romanovsky, A., Kharchenko, V. (eds.) Software Engineering for Resilient Systems. pp. 184–198. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
5. Fantechi, A.: Twenty-Five Years of Formal Methods and Railways: What Next? In: Counsell, S., Núñez, M. (eds.) Software Engineering and Formal Methods. Lecture Notes in Computer Science, vol. 8368, pp. 167–183. Springer (2014)
6. Hansen, I.A., Pachl, J.: Railway Timetable & Traffic: Analysis, Modelling, Simulation. Eurailpress (2008)
7. James, P., Moller, F., Nguyen, H.N., Roggenbach, M., Schneider, S., Treharne, H.: Techniques for modelling and verifying railway interlockings. International Journal on Software Tools for Technology Transfer **16**(6), 685–711 (2014)
8. Khan, U., Ahmad, J., Saeed, T., Hayat, S.: Real Time Modeling of Interlocking Control System of Rawalpindi Cantt Train Yard. In: 2015 13th International Conference on Frontiers of Information Technology (FIT). pp. 347–352 (2015). https://doi.org/10.1109/FIT.2015.28
9. Landex, A., Kaas, A., Hansen, S.: Railway Operation. Tech. rep., Technical University of Denmark, Centre for Traffic and Transport (2006)
10. Schittenhelm, B., Landex, A.: Jernbanesimuleringsværktøjer i Danmark (Eng.: Railway Simulation Tools in Denmark). In: Aalborg Trafikdage 2008, available from `www.trafikdage.dk/papers_2008/praesentationer/bernd_schittenhelm_158.pdf` (2008), in Danish
11. Vu, L.H., Haxthausen, A.E., Peleska, J.: Formal Modelling and Verification of Interlocking Systems Featuring Sequential Release. Science of Computer Programming **133, Part 2**, 91–115 (2017). https://doi.org/http://dx.doi.org/10.1016/j.scico.2016.05.010, `http://www.sciencedirect.com/science/article/pii/S0167642316300570`, http://dx.doi.org/10.1016/j.scico.2016.05.010
12. Winter, K.: Symbolic Model Checking for Interlocking Systems. In: Flammini, F. (ed.) Railway safety, reliability, and security: technologies and systems engineering. IGI Global (2012)