Formal Modelling and Verification of a Distributed Railway Interlocking System Using UPPAAL

Laursen, Per Lange ; Trinh, Van Anh Thi ; Haxthausen, Anne Elisabeth

Published in:

Leveraging Applications of Formal Methods, Verification and Validation: Applications

Link to article, DOI:

[10.1007/978-3-030-61467-6_27](https://doi.org/10.1007/978-3-030-61467-6_27)

Publication date:

2020

Document Version

Peer reviewed version

[Link back to DTU Orbit](#)

Citation (APA):

Laursen, P. L., Trinh, V. A. T., & Haxthausen, A. E. (2020). Formal Modelling and Verification of a Distributed Railway Interlocking System Using UPPAAL. In *Leveraging Applications of Formal Methods, Verification and Validation: Applications* (pp. 415-433). Springer. Lecture Notes in Computer Science Vol. 12478 https://doi.org/10.1007/978-3-030-61467-6_27

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Formal Modelling and Verification of a Distributed Railway Interlocking System Using UPPAAL

Per Lange Laursen, Van Anh Thi Trinh, and Anne Elisabeth Haxthausen

DTU Compute, Technical University of Denmark, Kongens Lyngby, Denmark
`perlangelaursen@gmail.com`, `anh-van@live.com`, `aeha@dtu.dk`

Abstract. This paper investigates the modelling and model checking of a real-world distributed railway interlocking system algorithm using UPPAAL. Interlocking systems for specific railway networks are verified by instantiating a generic (re-configurable) model with configuration data that describes the network and involved trains. There are three variants of the generic model: (1) The first variant includes the minimum required operations such as reserving a segment for a train, locking a point in a fixed position, and moving a train. (2) A restricted variant that uses a more strict operational order. (3) A variant that extends the first variant with a cancel operation that removes reservations and locks. Verification experiments are carried out on instances of all variants in order to check their correctness and compare their performance. The scalability of the three variants has been investigated with networks of varying sizes. Finally, for a real-world railway network, instances of the three model variants have been successfully verified.

Keywords: Distributed systems, Railway interlocking systems, Formal verification, Model Checking, UPPAAL

1 Introduction

The aim of this paper is to report on the experiences using UPPAAL [4] for model checking a real-world distributed railway interlocking system.

Background. A railway interlocking system is a signalling system component responsible for controlling the switching of points in a railway network and the issuing of movement authorities to the trains running in that network, such that train collisions and derailments are avoided. Usually, interlocking systems are *centralised* control systems. For small local railways, this can be a quite expensive solution, so it has long been of interest to examine the possibilities of using *distributed* interlocking systems, as discussed in [9]. In distributed interlocking systems, the control is delegated to a collection of control components physically distributed along the tracks and inside the trains. These components each have their own state space for storing relevant data. With this approach, the components communicate with each other in order to cooperate in controlling the

trains and the switch points in the railway network. For a survey over different distributed control algorithms, see [11]. Although being a less expensive solution, the required communication also makes it more difficult to verify the safety of trains in the system. For that, the CENELEC standard EN 50128 [5] strongly recommends to use formal verification methods.

Contribution and Related Work. There are many examples of formal verification of railway control systems, see e.g. [1, 2, 6, 8, 13, 15, 16, 18, 21], but only very few for *distributed* interlocking systems, see e.g. [10, 12, 14].

The distributed interlocking concept considered in this paper is based on the RELIS 2000 interlocking system of INSY GmbH. This system was first modelled and verified in [14] using RSL [20] and the RAISE theorem prover. As verification by means of theorem proving is rather time consuming, while verification by means of model checking is fully automated, the use of the latter was later, in [12], investigated for the same case study. In [12], RSL* and the SAL symbolic model checker were used only for a proof of concept, and hence efficiency was here of less concern. It actually turned out that the tool did not scale up well for larger networks, in which case compositional reasoning was used instead.

The main goal of our study is to investigate whether UPPAAL has better scaling, such that larger networks can be verified. UPPAAL has been chosen, as it uses a symbolic on-the-fly verification technique, which has generally proved very efficient for many applications, see for instance [19, 22]. UPPAAL additionally comes with an appealing graphical user interface, which may affect the modelling experience positively compared to modelling and model checking through a terminal for instance. Furthermore, we will investigate what it means for the verification performance if the control algorithm is restricted to a variant having a more specific execution order. We also use the opportunity to model additional functionality not included in [12]: an operation for cancelling reservations. Finally, we will also examine how well suited UPPAAL is as a modelling tool compared to RSL* and SAL.

Our control algorithm variants are similar to the ones presented in [12, 14], but our models differ of course wrt the choice of data types as UPPAAL offers fewer data types than RSL and RSL*. They also differ from [12] by using channels instead of shared variables for modelling the communication between control components. A major difference from [12] is: For improved efficiency, a train in our models (like in [14]) only reserves route segments and locks points in the order that they are needed, leading to some simplifications. Furthermore, in contrast to [12, 14], points are modelled in more detail, taking into account that a point can be in a switching state.

A totally different algorithm for a distributed interlocking system based on a two-phase commit protocol was modelled and verified in [10] using UMC [3].

Overview. First, in Sects. 2 and 3, short, informal introductions to the UPPAAL modelling language and the case study are given. Then, in Sects. 4 and 5, models of three different variants of the interlocking system are described. The

first variant (later referred to as the ‘First model’) describes each of the different operations that a train control computer can perform, i.e. reserve a segment, switch and lock a point, and move the train to a new segment. Each of these different operations can be attempted in any order by the train control computer. The second variant restricts the execution sequence of the different operations to a more ideal sequence, while the third variant includes an additional operation for cancelling reservations and locks. Desired properties are then presented in Sect. 6 and used for the experiments described in Sect. 7¹. A conclusion on the modelling and verification experiences is given in Sect. 8.

2 The UPPAAL Modelling Language

This section gives a very 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 and temporal logics. For more details, especially on the semantics of the concurrency construct, the reader should consult [4].

In UPPAAL, a system is modelled as a network of parallel timed automata (called processes), which are finite-state machines extended with (1) time in the form of real-valued clocks that progress synchronously and (2) data variables of simple data types (bounded integers, arrays, etc.). Even though UPPAAL can use real-valued clocks as a part of an automaton, it is still possible to define and work with untimed automata. In that case, time will not affect the resulting state space of an automaton. The models that are described as a part of this paper are all untimed.

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 unicast channels and broadcast channels.

Fig. 1. An example of a timed automaton in UPPAAL.

¹ The experiment models and the verified properties can be found at <https://github.com/perlangelaursen/DistributedRailwayControl>

A timed automaton consists of locations and edges. In the graphical representation of timed automata (for an example, see Fig. 1), locations are shown as circles and may have a name shown in purple colour. The *initial* location is shown by a double circle. A location may be labelled with an *invariant* (shown in pink colour), which is a Boolean expression over variables and clocks. The process can stay in that location and let time pass as long as the invariant is satisfied, but when the invariant becomes false, it must leave the location.

Edges are shown as arrows. An edge may be labelled with (1) some parameters of the form $id : type$ (shown in light green colour), (2) a guard (shown in green colour), which is a Boolean expression over variables and clocks determining when the edge is enabled and can be fired, (3) updates of variables and clocks (shown in blue colour), which are executed when the edge is fired, and (4) synchronisations of the form $c?$ or $c!$ over a channel c (shown in light blue colour). For a unicast channel c , an edge labelled with $c!$ in one process (the sender) may synchronise with an edge labelled with $c?$ in another process (the receiver) provided that both edges are enabled. For a broadcast channel c , an enabled edge labelled with $c!$ in one process may synchronise with all enabled edges labelled with $c?$ in other processes. If there are no receivers, then the sender can still execute the $c!$ action. A channel may be declared to be *urgent*. Whenever an enabled edge is able to synchronise over an urgent channel, this edge must be fired without any delay.

For the specification of desired properties, UPPAAL uses a subset of Timed Computation Tree Logic (TCTL) [4], where the outer-most formula must be an application of one of the two quantifiers **A** (along all paths) or **E** (along at least one path) or the leads-to operator $\rightarrow$, and these must not be nested.

3 Case Study

In this section, the considered case study from [14] is described informally.

Fig. 2. Railway network with control components and train components.

Railway Network. A railway network (see Fig. 2) is composed of a series of *segments*, *points*, and *sensors*. A segment is a linear section of a railway track.

Each of its two ends can be connected to another segment. A point allows for two different connections to the same segment when the railway branches into two different directions. The segment that can be connected to two different segments is called the *stem* segment while the other two are the *plus* and *minus* segments. The stem segment can only be connected to one of the two other segments at a time. To change the connection, the point must be switched. Each point also has a sensor, which is used to detect a train passing the critical section of the point.

Control Components. The distributed railway interlocking system consists of a *train control computer* (TCC) in each train and a *control box* (CB) at each place where segments can be connected (see Fig. 2). Control boxes placed at points are also called *switch boxes* since they are responsible for switching and locking their associated points.

Control Task. The task of the control system is to ensure the safety of trains in the railway network under control. The trains are considered *safe* if they never *collide* or *derail*. Two trains (potentially) collide with each other if they occupy the same segment at the same time, and a train (potentially) derails if it is passing a switching point or if it enters a point from the wrong side (an unconnected branch).

Overall Control Strategy. The control strategy to ensure safety is as follows. Collisions are prevented by requiring that a train must obtain a *reservation* of a segment before it is allowed to enter it. This means that the train gets exclusive access to the segment, and therefore a segment may only be reserved by one train at a time. Derailment is prevented by requiring that a train obtains a *lock* of a point in the correct position for its route before it is allowed to pass it. Locking a point prevents the point from being switched, so a point can only be switched if no train has obtained a lock for it.

Distributed State Space. Information about obtained reservations and locks are stored in the state space of a train's TCC, which also has information about the train's route and its current position in the route. TCCs obtain these reservations and locks by sending requests to control boxes (see Fig. 3).

In the state space of a control box, the control box stores information about reservations of its associated segments. If the control box is a switch box (i.e. has an associated point), it also stores information about which sections are currently connected by the point and whether the point is locked for a train.

Reserving a Segment. A control box is associated with and responsible for the segments that it is placed by. If a train wishes to request a reservation of a segment, it should send this request to the control boxes placed at each end of that segment. This means that a *full reservation* of a segment is obtained by

Fig. 3. Interactions between control components and point components.

requesting and successfully receiving the reservation of the segment at each of the segment's two control boxes.

A train only requests a reservation of a segment in its route at a control box in its route. It will only do so if it does not currently have that reservation. Different from [12], three additional conditions have been added:

- The train does not yet have the maximum allowed number of reservations,
- it has reserved all the segments between its current position and the segment in question and
- it always reserves a segment at the control box closest to its position first.

The first mentioned condition makes it possible to experiment with different values when figuring out how to improve the verification process. The two other conditions reduce the number of possible execution sequences by only allowing reservations to be made in the order that the train needs them while moving along its route.

When a control box receives a request for reserving a segment, it checks that

- it is in fact associated with that segment and
- the segment is not already reserved by another train.

If reservation is possible, the control box returns a positive acknowledgement to the requesting TCC, otherwise a negative acknowledgement is returned. If a request is positively acknowledged, both the control box and the TCC update their state spaces accordingly.

Locking a Point. To prevent a train from moving between unconnected segments or passing a switching point, it is necessary to first lock the point in the correct position.

A train sends a lock request to a switch box with the IDs of the segment that it is currently on and the segment that it wants to move to. It only requests locks in its route that it does not already have, and for connections of segments that are adjacent in the route and for which it already has the reservations at the switch box. Similarly to reservations, additional conditions have been added in order to improve the verification process:

- The train does not yet have the maximum allowed number of locks and
- it has already obtained the necessary locks between its current position and the switch box.

The receiving switch box will then check that

- it has not locked its point for another train,
- the received segments are the stem segment and either the plus or the minus segment and
- the received segments have been reserved by the requesting train.

If locking is possible, the switch box will switch the point if the segments are not already connected. It will then lock it and finally return a positive acknowledgement to the requesting train and both will update their state spaces. Otherwise, it will return a negative acknowledgement.

Releasing Reservations and Locks. Sensors detect trains in their critical sections by becoming *active*, and if no train is in the critical section, *passive*. When a sensor goes from active to passive, it triggers the release of reservations and locks at the associated control box. A TCC either uses the Global Positioning System (GPS) or track components signalling their location to know when it should release its reservations and lock data.

4 First Model

Our UPPAAL models have been designed to be re-configurable, so that they can be reused for a whole class of railway networks by only instantiating them with some configuration data: A model instance consists of (1) declarations of constants (provided as configuration data) defining a railway network, control parameters, and initial data of the control components, (2) channels, (3) functions, (4) templates, and (5) a system. The four last items are the same for all instances of the same model, and the same holds for the constant names, but the values of the constants are individual for each instance.

This section describes our first model. Details can be found in the report [17].

Configuration Data. Configuration data consists of (1) integer constants `NTRAIN`, `NCB`, `NPOINT`, `NSEG`, and `NROUTELENGTH` defining the number of trains, control boxes, points, segments, and route lengths, respectively, (2) constants `resLimit` and `lockLimit` defining the limits of how many reservations and locks

each train is allowed to have at a time, and (3) constant arrays defining the railway network and initial data for the state spaces of the TCCs and CBs. For example, the array `segRoutes`

```
const segV_id segRoutes[NTRAIN][NROUTELENGTH] = {...},...;
```

stores in `segRoutes[i]` an array of the segments of the route of the train with ID `i`. (All components of a specific kind are given an integer ID in the interval $[0; N - 1]$, where N is the number of components of that kind.)

Templates. The templates for the railway components are: `Train`, `CB` and `Point`. These templates each have a single parameter named `id`, which can be instantiated with the ID of a train, a control box, and a point, respectively, resulting in a process modelling the behaviour of the entity with that ID. In addition, an `Initializer` template and an urgent broadcast channel, `start`, is used for initiating the initialisation of the instances of the other templates.

Sensors could also have been modelled as instances of a `Sensor` template and channel synchronisation could then have been used to model a sensor's sensing of passing trains and the forwarding of its status to its associated control box. However, both the communication between a train and a sensor and the communication between a sensor and a control box (which is a wired connection) can be seen as taking no time. Hence, one can model this communication as a single synchronisation directly between the train and the control box, so there is no need for a separate `Sensor` template.

System Declaration. The generic system declaration

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

creates automatically one `Initializer` process, a `Train(t)` process for each train/TCC `t`, a `CB(cb)` process for each control box `cb`, and a `Point(p)` process for each point `p`.

Channels. In addition to the broadcast channel used in the initialisation step, eight types of unicast channels, one for each of the interaction arrows shown in Fig. 3, are declared for binary communication between components.

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

For instance, when a train `t` wants to send a request to a control box `cb` for a reservation of segment `seg`, it needs to send its own ID and the segment ID along in the request. This request can be modelled as an output on the channel `reqSeg[cb][t][seg]`. The control box `cb` can synchronise with that by making an input on that channel. The channels `OK[t]` and `notOK[t]` can then

be used by the control box to send a positive and negative acknowledgement to **t**, respectively. **pass[cb]** and **passed[cb]** can be used by a train to signal to a control box **cb** that it has started and finished passing the sensor associated with that control box, respectively. **switchPoint[p]** can be used by a control box **cb** to request point **p** to switch to its other position and **OKp[cb]** can then be used by the point to notify that it has been locked in the requested position.

Synchronous communication of data over channels rather than via shared variables as in [12], ensures that data is never lost or altered before it is received.

The Initializer Template. The **Initializer** automaton has one edge from its initial location to its only other location. On this, it synchronises on an urgent broadcast channel, **start**, with all **Train** and **CB** instances to initiate their initialisation.

Fig. 4. The first model's **Train** template.

The Train Template. The **Train** template declares local variables (1) for storing the distributed state information described in Sect. 3 for a TCC and (2) for storing the actual position of the train itself. Below, the most important variables are explained.

- **routeLength** stores the number of segments in the train's route.
- **segments** and **boxes** are arrays in which the IDs of the segments and the control boxes along the train's route are stored, respectively.
- **index** stores the index of the segment in **segments** that the train *believes* that it is occupying.

- **resSegIndex** stores the index of the segment in **segments** that needs to be reserved next, and **resCBIndex** stores the index of the control box in **boxes** at which this segment should be reserved for the train.² Hence, the next reservation should be of segment **segments[resSegIndex]** at **boxes[resCBIndex]**.
- **lockIndex** stores the index of the control box in **boxes** that needs its associated point to be locked next.
- **curSeg** and **headSeg** stores the IDs of segments *actually* occupied by the train.³ If the train is only occupying one segment, the ID of that is stored in **curSeg** while **headSeg** = -1. If it is occupying two segments, the segment IDs occupied by the rear and the front of the train are stored in **curSeg** and **headSeg**, respectively. These variables are not used by the controller, but only for formulating the safety properties that should be verified in terms of the train’s actual position.

The **Train** automaton for a train with ID **id** can be seen in Fig. 4.

The outgoing edge from the **Initial** location synchronises with **Initializer** on the **start** channel and then copies configuration data to its local variables. The edge is only enabled if the data are wellformed.

After initialisation, it enters the location **SingleSegment**, which indicates that the train is located on a single segment.

From **SingleSegment**, there are two outgoing (request) edges used for requesting a segment reservation and for requesting a switching/locking of a point at a control box. These edges are guarded by functions that ensure that an operation is only initiated if the conditions stated in Sect. 3 are fulfilled. For instance, the guard for a reservation request is defined by the following function:

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

This function states that a segment reservation should only be requested if (1) there are more segments in the route that the train has not yet reserved, and (2) the maximum number of allowed reservations will not be exceeded by a reservation.

If a guard is true, the train can send its request to the control box by a synchronisation on the channel intended for that. E.g. for a reservation request, it will be **reqSeg[boxes[resCBIndex]][id][segments[resSegIndex]]**.

After that, the train waits for the control box to either send a positive or negative acknowledgement by synchronisation on **OK[id]** or **notOK[id]**, respectively. After a positive acknowledgement, it also updates its state space. In the case of a reservation, the update is made by the following function:

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

² As segment reservations and locks of points are made in sequential order in contrast to [12], no data structures for explicitly storing reservations and locks are needed – they can be derived from **resSegIndex**, **resCBIndex**, and **lockIndex**.

³ Trains are assumed to reside on at most two segments.

```

    resCBIndex = (resBit==1) ? resCBIndex+1 : resCBIndex;
}

```

Since a segment must be reserved at two control boxes, **resSegIndex** should not be incremented every time a reservation has been obtained, but only every second time. **resBit** stores an integer ranging from 0 to 1, which is flipped every time a reservation has been obtained. **resSegIndex** is then only incremented whenever **resBit** is 0. Similarly, **resCBIndex** should only be incremented every second time: whenever **resSegIndex** is not.

From **SingleSegment**, there is also an edge to **DoubleSegment** and one back from that to **SingleSegment**, for modelling that the train starts and finishes passing a control box. The first edge is guarded by a function checking that the passing is only initiated if the conditions stated for that in Sect. 3 are fulfilled. On both edges the location information of the train is updated.

Finally, there is an edge from **SingleSegment** to the **Arrived** location. This is fired when the train has arrived at its destination, i.e. is on the last segment of its route.

Fig. 5. The first model's CB template.

The CB Template. The CB template declares local variables for storing the distributed state at control boxes as described in Sect. 3. For instance, there is a **segments** array for storing the IDs of up to three segments that can be reserved at that CB, and an array, **res**, of the same length, which in **res[i]** stores the reservation status of the segment in **segments[i]**: The status is -1 if the segment is not reserved by a train, and otherwise it is the ID of the train that has reserved the segment.

The CB automaton for a control box with ID **id** can be seen in Fig. 5.

After initialisation, the CB enters the location **Idle**. From that, there is an outgoing edge on which it waits for a reservation request by synchronisation on a channel intended for that. After a synchronisation, it checks whether the

conditions stated in Sect. 3 for granting the requested reservation are fulfilled. Based on the result of that check, it sends either a positive or negative acknowledgement back to the requesting train, `tid`, by synchronisation on `OK[tid]` or `notOK[tid]`, respectively. If the request is granted, the CB updates its own state space and returns to the `Idle` location. The mentioned check is expressed using the following function:

```
int[-1,2] checkSegment(seg_id sid) {
  for(i:int[0,2]) {
    if(segments[i] == sid && res[i] == -1) {
      return i; // sid exists and is not reserved
    }
  }
  return -1; //sid either not found or already reserved
}
```

This function call (`checkSegment(j)`) checks whether it can find the requested segment `j` among the CB's associated segments and that it is not reserved. If the reservation is possible, the index of the segment is returned, otherwise -1 is returned.

From the `Idle` location, there are other similar loops for handling a locking request from a train and for releasing reservations and locks for a train at the control box when the train passes it. The handling of locking requests is more complicated as that may also involve the switching of a point. Due to space limitations, the interested reader is referred to [17] for details.

Note that a CB cannot synchronise for a request from a train or with a passing train, when it is not in `Idle` (i.e. it is already processing a request or being passed by a train). This design ensures that a control box can only communicate with one train at a time.

5 Other System Variants

Two additional variants of the model described in Sect. 4 have been developed to explore how the model can be *restricted* and also *extended*. While the restricted model enforces a more restricted sequence of operations, the extended model introduces a new *cancel* operation, which can be used to cancel reservations and locks.

Restricted Operation Sequence. Although `Trains` are free to request segments and locks and pass CBs whenever they wish as long as the guards are true, it is not always beneficial to do so. For example, if a `Train` has not yet obtained its maximum allowed number of segment reservations or locks, it may choose to either reserve more segments or request the locking of connections of segments that it has already reserved. Hence, to further reduce the number of possible execution sequences, this variant should limit the number of choices a `Train` has.

An example of such a restricted sequence could be formulated as follows: Obtain as many reservations as possible, obtain as many locks as possible and finally move forward in the route for as long as possible.

Fig. 6. The Train template with restricted options.

To ensure the deterministic execution path, the **SingleSegment** location has been divided into several locations: **SingleSegment**, **Reserving**, **PassOrLock** and **Locking** as seen in Fig. 6. Each of these locations will at most have one enabled edge to another location by considering both the current state and the potential future state of a **Train**. An example of this is when a **Train** is reserving a segment. It will make the reservation request and transition to the **Reserving** location. If the reservation was *successful*, it will go to either the **SingleSegment** location or the **PassOrLock** location depending on whether it can make more reservations or not. If the reservation was *unsuccessful*, then it will also go to either the **PassOrLock** location or the **SingleSegment** location this time depending on whether it already has at least one full reservation or not. The other operations are designed in the same manner by analysing what is possible in the next location before actually transitioning to it.

Extension with Cancel Operation. A situation that has not yet been discussed in the first model is the situation in which a train is unable to proceed on its route due to livelocks. This situation can for instance arise if a **Train** gets the reservation of a segment at one of the segment's associated CBs, while another **Train** gets the reservation of the same segment at the other associated CB. This prevents either of the two **Trains** to get the full reservation of the segment. Therefore, it may be beneficial to have a cancel operation that will allow a **Train** to cancel its obtained reservations and locks if needed.

The general strategy of the cancel operation is that a **Train** cancels reservations and locks in the opposite order of how it obtained them. For a description of how the cancel operation has been implemented, see [17].

6 Verification Properties

This section shows examples of the formalisation in TCTL of safety properties and other relevant properties to be verified. The complete list of properties and their formalisation can be found in [17].

Safety Properties. The safety properties to be verified are ‘no collision’ and ‘no derailment’. Here we show how the former of these is formalised.

‘No collision’ means that at any time, no two different **Train** instances occupy any of the same segments. This property is formalised as:

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

The query states that in all paths, for all states after the initialisation step (i.e. when the **Initializer** automaton has reached the location **Initialized**), two **Train** instances with different IDs, *i* and *j*, have positions that do not include the same segments. With two position variables (**curSeg** and **headSeg**) for each **Train**, it should be checked that the values of these for one train are different from the values of these for the other train, but the value of **headSeg** for a train should only be checked when the train occupies two segments, i.e. if its automaton is in location **DoubleSegment**, which for a train with ID = *i* can be checked by the formula **Train(i).DoubleSegment**.

Other Properties. We have formulated other relevant properties to be verified. For example, *consistency properties*, which express the consistency between **Train** state spaces and **CB** state spaces, e.g. that whenever a **Train** has stored a segment reservation at a control box, the corresponding **CB** has stored the same segment as reserved for that train.

We have also formalised a *liveness property* that expresses that there exists a path in which all trains eventually arrive at their destinations:

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

7 Verification Experiments

A series of experiments has been conducted in order to determine whether the properties presented in Sect. 6 can be verified for instances of the three generic models for some specific networks having a typical RELIS 2000 layout⁴ and what the verification resource usage is⁵. Furthermore, a comparison of verification metrics of UPPAAL and SAL has been made.

⁴ The RELIS 2000 system was intended for local railways with up to 20 stations with 1–2 track segments each, connected by single lines, and operated by 2–3 trains.

⁵ The experiment models and the verified properties can be found at <https://github.com/perlangelaursen/DistributedRailwayControl>

The first set of networks focuses on the scalability of the system where the number of stations varies. A general description can be seen in Fig. 7 where two trains are positioned in each end of a network with n stations. Train t_0 's route is shown as a dashed line, while the route of t_1 is shown as a dotted line.

Fig. 7. Network with n stations.

Fig. 8. Nærumbanen with two trains.

The second set of networks is based on the local railway network *Nærumbanen* in Denmark. There are two separate situations for this network. The first situation represents the regular traffic where two trains start in each end of the network while their destinations are in the opposite ends as seen in Fig. 8. The second situation represents the rush hour traffic with an additional train t_2 . t_0 starts at segment s_0 with destination at s_{18} , t_1 starts at s_{18} with destination at s_{10} and t_2 starts at s_{10} with destination at s_0 .

For each model instance, each of the properties have been verified⁶ three times and the averages of the elapsed time and resident memory usage peaks (extracted from UPPAAL's model checker) have been calculated.

Table 1 shows the elapsed time and resident memory usage peaks for verification of the 'no collision' property for both experiment series for each of the model variants (with the limit control parameters set to 2).

The results show that the time and memory usage increases when the size of the network increases. The reason for the exponential growth is due to the increasing number of places where trains can pass each other, which results in more states that need to be checked. The results also show (as expected) that the restricted model generally has the lowest time and memory usage, and that the model including the cancel operation has the highest time and memory usage. This is because the restricted model has fewer interleavings than the other model

⁶ The experiments were performed with UPPAAL version 4.1.19 on a machine with Arch Linux OS, an AMD Ryzen 2700X processor clocked at 4.0 GHz and 64GB RAM memory clocked at 2666MHz.

Table 1. Verification metrics for ‘No collision’: elapsed time in seconds on the left-hand side of a column and memory usage peak in KB on the right-hand side. All models use limits = 2. Verifications that lasted longer than 24 hours (86400 seconds) were stopped.

Configuration	First model	Restricted model	Model with cancel
Station One	0,013 / 6515	0,021 / 8508	0,046 / 8727
Station Two	0,195 / 9944	0,127 / 12579	0,894 / 16641
Station Four	8 / 41508	2 / 39384	34 / 139527
Station Six	176 / 152875	48 / 122787	951 / 680333
Station Eight	1145 / 422327	277 / 276996	6587 / 2104345
Station Ten	4344 / 955109	1064 / 560651	23912 / 5207884
Station Twelfth	17905 / 3436440	3151 / 1360780	71971 / 11269876
Station Fourteen	47898 / 5680072	8540 / 2043916	stopped
Station Sixteen	stopped	26376 / 3225156	
Station Twenty		72986 / 6961996	
Nærumbanen (2T)	112 / 122196	20 / 113553	251 / 238945
Nærumbanen (3T)	4700 / 1834129	395 / 313881	9697 / 3638703

variants, while the model with the cancel operation has additional states. Within 24 hours, the restricted model was verifiable for up to 20 stations, while the two others had to be stopped for 16 and 14 stations, respectively.

It could be interesting to compare our UPPAAL results with the SAL results in [12] for some common network. The railway network in Fig. 13 on P. 30 in [12] can be used for that, as it is the same as our *Station One*. The third model variant in [12] uses the Just-In-Time principle to reserve segments and lock points, which is close to our ‘First model’ with the limit parameters set to 1, but it differs by being less restrictive with respect to the order of reservations. For a more fair comparison of the verification performance, we have adapted the model from [12] to a new model (Model 4) that follows our more restricted order. Table 2 shows a comparison of the verification performance using the UPPAAL and the SAL model checkers, respectively, for these two models instantiated with the *Station One* network.⁷ As it can be seen, UPPAAL had a much better performance only spending 0.017 seconds and 9.5 MB RAM to verify the UPPAAL model, whereas SAL spent around 6 seconds and 105–110 MB RAM for the RSL* model. The same was tried for the *Station Ten* network. In this case, UPPAAL spent 1858 seconds and 675 MB, while RSL*-SAL did not scale up to that.

8 Conclusion

This paper described the modelling and verification of three variants of a real-world distributed railway control system algorithm using UPPAAL.

Having modelled the first variant, it was easy to refine it to one enforcing a more restricted operation sequence, which improved the verification metrics and

⁷ The verification using UPPAAL was this time kindly performed by Signe Geisler on her machine used in [12] to make the comparison possible. In [12], an Intel(R) Core(TM) i7-8650U CPU clocked 1.90 GHz with 31GB of memory was used.

Table 2. Verification metrics for verifying ‘No collision’ for *Station One* using SAL and UPPAAL.

Model Checker Model		Time (sec)	Memory (MB)
SAL	RSL* Model 4	6.03 - 6.18	105 - 110
UPPAAL	UPPAAL First model with limit 1	0.017	9.5

scalability significantly compared to the former. Extending the first model with an additional operation was even more straightforward than refining it since no changes to existing locations or edges were made, but only new locations and edges were added. Verification of the extended model costs more in terms of space and time than for the two other models, but the new operation introduced in this model is required in real-world systems to prevent livelocks. In conclusion, the first model was modelled in such a way that it was easy to both improve the verification process by refining it, but also extend it if new functionalities were to be introduced. One could also attempt to combine the two new variants by extending the restricted version with a cancel operation. The algorithm should then be re-specified, for example by allowing cancellations as a special operation at any time possible. This would introduce more interleavings, but still be more restricted than the extended model.

Compared to the use of RSL* and SAL in [12], some concepts have been more difficult to express in UPPAAL, which led to some less elegant solutions, e.g. the use of -1 as a filler because no list data structure of variable length was available. On the other hand, UPPAAL’s model checker is significantly faster and less memory consuming than the SAL symbolic model checker, and consequently the use of UPPAAL scaled up to larger networks much better. Furthermore, UPPAAL’s graphical tool generally made the modelling process more visual and straightforward, while the simulator and diagnostic trace function also made it simple to debug and test the different model variants.

In future work, we wish to explore the use of other tools, e.g. UMC, nuXmv, SPIN, and mCRL2, to model and verify the same railway control system and compare with that. It could also be interesting to experiment with variants of the UPPAAL models using shared variables for communication instead of channels, and to extend the model with the possibility of message loss (due to wireless communication) and check that the system still remains safe. Furthermore, it could be interesting to experiment with stochastic variants of the UPPAAL models and use UPPAAL STRATEGO [7] to synthesise schedulers.

Acknowledgements. We would like to express our gratitude to Jan Peleska from whom the case study originates and with whom the second author had the great pleasure to verify the same case study by theorem proving [14]. We are grateful to Signe Geisler for repeating some of our experiments on her laptop making it possible to compare the verification performance of some of our verification experiments using UPPAAL with some of her experiments using SAL.

Finally, we would like to thank the anonymous reviewers for useful suggestions for the presentation.

References

1. 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). https://doi.org/10.1007/978-3-319-98938-9_2
2. Basile, Davide and Beek, Maurice H. ter and Ferrari, Alessio and Legay, Axel: Modelling and analysing ERTMS L3 moving block railway signalling with Simulink and Uppaal SMC. In: Larsen, K.G., Willemse, T. (eds.) *Formal Methods for Industrial Critical Systems*. Lecture Notes in Computer Science, vol. 11687, pp. 1–21. Springer Verlag (2019). https://doi.org/10.1007/978-3-030-27008-7_1
3. ter Beek, M.H., Fantechi, A., Gnesi, S., Mazzanti, F.: A state/event-based model-checking approach for the analysis of abstract system properties. *Science of Computer Programming* **76**(2), 119–135 (2011)
4. 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: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*. pp. 200–236. No. 3185 in *Lecture Notes in Computer Science*, Springer-Verlag (September 2004)
5. CENELEC – European Committee for Electrotechnical Standardization: EN 50128:2011 – Railway applications – Communications, signalling and processing systems – Software for railway control and protection systems (2011)
6. Comptier, M., Deharbe, D., Perez, J.M., Mussat, L., Pierre, T., Sabatier, D.: Safety analysis of a CBTC system: A rigorous approach with Event-B. In: Fantechi, A., Lecomte, T., Romanovsky, A. (eds.) *Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification*. Lecture Notes in Computer Science, vol. 10598, pp. 148–159. Springer Verlag (2017). https://doi.org/10.1007/978-3-319-68499-4_10
7. David, A., Jensen, P.G., Larsen, K.G., Mikučionis, M., Taankvist, J.H.: UPPAAL STRATEGO. In: Baier, C., Tinelli, C. (eds.) *Tools and Algorithms for the Construction and Analysis of Systems*. Lecture Notes in Computer Science, vol. 9035, pp. 206–211. Springer Berlin Heidelberg (2015)
8. 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)
9. Fantechi, A., Gnesi, S., Haxthausen, A., van de Pol, J., Roveri, M., Treharne, H.: SaRDIn - A safe reconfigurable distributed interlocking. In: *Proc. 11th World Congress on Railway Research (WCRR 2016)*. Ferrovie dello Stato Italiane, Milano (2016)
10. Fantechi, A., Haxthausen, A.E., Nielsen, M.B.R.: Model checking geographically distributed interlocking systems using UMC. In: *25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. pp. 278–286 (2017). <https://doi.org/10.1109/PDP.2017.66>

11. Fantechi, A., Haxthausen, A.: Safety interlocking as a distributed mutual exclusion problem. In: Howar, F., Barnat, J. (eds.) *Formal Methods for Industrial Critical Systems*. Lecture Notes in Computer Science, vol. 11119, pp. 52–66. Springer (2018). https://doi.org/10.1007/978-3-030-00244-2_4
12. Geisler, S., Haxthausen, A.E.: Stepwise development and model checking of a distributed interlocking system using RAISE. *Formal Aspects of Computing* (published online first, 21 February 2020). <https://doi.org/10.1007/s00165-020-00507-2>
13. Hansen, H.H., Ketema, J., Luttik, B., Mousavi, M.R., van de Pol, J.: Towards model checking executable UML specifications in mCRL2. *Innovations in Systems and Software Engineering* **6**(1), 83–90 (Mar 2010). <https://doi.org/10.1007/s11334-009-0116-1>
14. Haxthausen, A.E., Peleska, J.: Formal Development and Verification of a Distributed Railway Control System. In: *IEEE Transactions on Software Engineering*, vol. 26, pp. 687–701. IEEE (2000)
15. Hoang, T.S., Butler, M., Reichl, K.: The hybrid ERTMS/ETCS level 3 case study. In: Butler, M., Raschke, A., Hoang, T.S., Reichl, K. (eds.) *Abstract State Machines, Alloy, B, TLA, VDM, and Z*. Lecture Notes in Computer Science, vol. 10817, pp. 251–261. Springer Verlag (2018). https://doi.org/10.1007/978-3-319-91271-4_17
16. James, P., Möller, F., Nguyen, H.N., Roggenbach, M., Schneider, S., Treharne, H., Trumble, M., Williams, D.: Verification of Scheme Plans Using CSP||B. In: Counsell, S., Núñez, M. (eds.) *Software Engineering and Formal Methods*, Lecture Notes in Computer Science, vol. 8368, pp. 189–204. Springer (2014)
17. Laursen, P.L., Trinh, V.A.T.: Formal modelling and verification of distributed railway control systems. Tech. rep., DTU Compute, Technical University of Denmark (2019), <https://github.com/perlangelaursen/DistributedRailwayControl/blob/master/s144449s144456-MSc-Thesis.pdf>
18. Limbrée, C., Cappart, Q., Pecheur, C., Tonetta, S.: Verification of railway interlocking-compositional approach with OCRA. In: *International Conference on Reliability, Safety, and Security of Railway Systems*. pp. 134–149. Springer (2016)
19. Mazzanti, F., Ferrari, A.: Ten diverse formal models for a CBTC automatic train supervision system. In: Gallagher, J.P., van Glabbeek, R., Serwe, W. (eds.) *Proceedings Third Workshop on Models for Formal Analysis of Real Systems and Sixth International Workshop on Verification and Program Transformation*. EPTCS, vol. 268, pp. 104–149 (2018). <https://doi.org/10.4204/EPTCS.268.4>, <http://arxiv.org/abs/1803.08668>
20. RAISE Language Group: Chris George, Peter Haff, Klaus Havelund, Anne E. Haxthausen, Robert Milne, Claus Bendix Nielsen, Søren Prehn, Kim Ritter Wagner, T.: *The RAISE Specification Language*. The BCS Practitioners Series, Prentice Hall Int. (1992)
21. 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), <http://www.sciencedirect.com/science/article/pii/S0167642316300570>, <http://dx.doi.org/10.1016/j.scico.2016.05.010>
22. Yi, W., Pettersson, P., Daniels, M.: Automatic verification of real-time communicating systems by constraint-solving. In: Hogrefe, D., Leue, S. (eds.) *Formal Description Techniques VII*. pp. 243–258. Springer (1995)