Stepwise Development and Model Checking of a Distributed Interlocking System - Using RAISE

Geisler Pedersen, Signe; Haxthausen, Anne Elisabeth

Published in:
Formal Methods

Link to article, DOI:
10.1007/978-3-319-95582-7_16

Publication date:
2018

Document Version
Peer reviewed version

Stepwise Development and Model Checking of a Distributed Interlocking System - Using RAISE

Signe Geisler(✉) and Anne E. Haxthausen

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

Abstract. This paper considers the challenge of designing and verifying control protocols for geographically distributed railway interlocking systems. It describes for a real-world case study how this can be tackled by stepwise development and model checking of state transition models in an extension of the RAISE Specification Language (RSL). This method also allows different variants of the control protocols to be explored.

Keywords: Stepwise development · Model checking
RAISE · Railway interlocking systems · Distributed systems

1 Introduction

This paper considers the challenge of formally modelling and verifying the real-world geographically distributed railway interlocking system presented in [8]. The engineering concept of this was originally developed by INSY GmbH Berlin for their railway control system RELIS 2000 designed for local railway networks.

1.1 Background

A railway *interlocking system* is a safety-critical system controlling the track side equipment and movement of trains in a railway network such that train collisions and derailments are avoided. Current computer-based interlocking systems usually have a *centralised* design, but in a few cases, as for instance described in [8], the control has been *geographically distributed* to processors deployed at the sensors and actuators (e.g. points) along the track layout and to onboard train control computers. One of the motivating factors for this is the lower cost, making it available as a solution for small, local railway networks, cf. the discussion in [3,8].

To verify the safety of *distributed* railway interlocking systems is even more challenging than for centralised systems. For centralised interlocking systems, there is a global notion of the state of the system, which can be observed by the control computer to make interlocking decisions. In contrast to this, in the

K. Havelund et al. (Eds.): FM 2018, LNCS 10951, pp. 277–293, 2018.
https://doi.org/10.1007/978-3-319-95582-7_16

geographically distributed approach, where each train is equipped with a train control computer, and additional control components are distributed throughout the railway network, the interlocking data must be distributed (but also duplicated to some extent) in the different control components. Furthermore, the control components must collaborate in order to take safe decisions, so communication between the control computers must be introduced. This adds additional threads which would not be present in a centralised system. Hence, the distribution of control gives new challenges for the safety verification.

Using formal methods for the verification of distributed interlocking systems is a natural choice, as formal methods are strongly recommended by the CENELEC standard EN 50128 [2] for safety-critical railway control components and have proved useful for many applications. For instance, Haxthausen and Peleska demonstrated this in [8], where they modelled and verified the distributed interlocking system considered in this paper. For this they used the RAISE Specification Language, RSL [11], and the RAISE theorem prover, respectively.

Theorem proving, as used in [8], handles complex systems very well, but the proof derivation process is very time consuming, as it must be directed by a human. Furthermore, theorem provers are often unable to give counter-examples when a proof fails. With model checking, the verification process is fully automated, and if some asserted property is not satisfied in some state of the system, the model checking tool will produce a counter-example, usually showing the path to that state. The path can then be investigated in order to discover the unintended behaviour. Therefore, in this paper, we will investigate the use of *model checking* for verifying the considered interlocking system.

1.2 Contribution

The main contribution of the paper is a method for modelling and verifying a distributed system by stepwise specification and model checking, and the application of this method to a distributed railway interlocking system.

For the system specification the method uses an extension of RSL, called RSL-SAL [10], which allows to specify systems by state transition system models. In contrast to this, the work in [8] used the RSL process algebra to specify the final model of the system. The formal verification is now performed using the SAL symbolic model checker which is a backend to RSL-SAL. The challenge of capturing the system behaviour in appropriate detail was tackled by using *stepwise development* of state transition system models. This approach is novel in the context of RAISE.

1.3 Related Work

Formal verification of interlocking systems via *model checking* is an active research topic, investigated by several research groups, see e.g., [5,9,13,14] mostly focusing on *centralised* interlocking. In [6,7] RSL-SAL and SAL was

also used for modelling and verifying an interlocking system, but this was a centralised (relay) interlocking system, and in that work no stepwise development was used.

In [4], a geographically distributed railway interlocking system was formally modelled and verified using UMC instead of RSL-SAL and SAL. The control protocol presented in [4] radically differs from the one considered in our case study: in [4], full train routes are allocated before trains start moving. This is done using a two-phase commit protocol for determining agreement between the control components. The control protocol in our case study allows trains to allocate each section of their routes separately, which allows for greater flexibility, since train routes can be interleaved to a greater extent.

1.4 Paper Overview

First Sect. 2 gives a brief introduction to the case study: the engineering concept of the considered distributed interlocking system and an overview of the formal development. Then, the following sections (Sects. 3, 4 and 5) give an overview of the generic model specifications and the development steps between them. The verification of model instances is described in Sect. 6. Finally, Sect. 7 gives a conclusion and states ideas for future work.

2 Case Study

2.1 Engineering Concept

The *control strategy* of the system must ensure the safety of the system by preventing the derailment and collision of trains. In this engineering concept, safety is achieved by only allowing one train on each track segment at the same time and ensuring that points are locked in correct position while trains are passing them. To this end, trains must *reserve* track segments before entering them and *lock* points in correct position before passing them.

The *control components* of the system are responsible for implementing the control strategy. Each train is equipped with a *train control computer*. In the railway network, several *switchboxes* are distributed, each controlling a single point or an end point of the network. These components communicate with each other in order to collaboratively control the system. Each control component has its own, local state space for keeping track of the relevant information. As can be seen from Fig. 1, each of the train control computers has information about the train's route (a list of track segments) with its switch boxes, the train position, and the reservations and locks it has achieved. Each switchbox has information about its associated sensor (used to detect whether a train is passing the critical area close to the point), which segments are connected at its associated point (if any), for which train the point is locked (if any), and for which train each of the associated segments is reserved (if any).

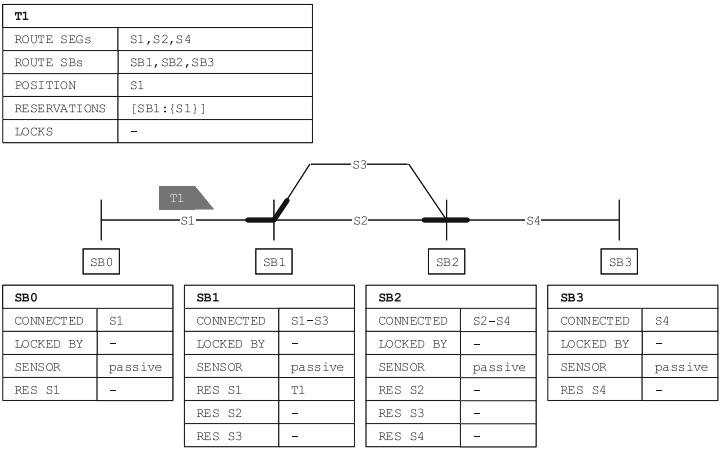

Fig. 1. An example system.

The basic idea of the control strategy is as follows:

1. *Permisson to enter a segment:* For a train control computer (TCC) to decide whether it is legal to enter the next segment of its route, the TCC must observe its local state space and check whether it has the needed reservations and locks. More precisely, the following must hold:
   - the next segment must have been reserved for the train at the two upcoming switchboxes, and
   - the point must have been switched in the direction for the train route and locked for the train at the next switchbox.

   In the scenario shown in Fig. 1, for the train $T1$, this means that it must have reservations for segment $s2$ at both the switchboxes $SB1$ and $SB2$, and a lock for the point at $SB1$, before it can be allowed to enter $s2$.
2. *Making reservations and locks:* Reservations and locks are made by the trains by issuing requests to the relevant switchboxes. Depending on its local state, a switchbox may or may not comply with a request from a train. The switchbox can only fulfil a segment reservation request if the segment is not already reserved at the switchbox. Similarly, a switchbox can only lock a point (after potentially having switched the point in the direction for the train route), if the point is not already locked. Additionally, a request for locking a point can only be made if the train has reservations for the two segments in its route on either side of the point to be locked. In the scenario shown in Fig. 1, for the train $T1$, this means that it must have a reservation for segments $s1$ and $s2$ at the switchbox $SB1$, before it can request to lock the point at $SB1$.

   If a switchbox can meet a request, it will update its state space accordingly. In any case, the switchbox will send a response to the train, based on which the train can determine whether the request has been met and, thereby, whether the train should update its state space as well.

3. *Release of reservations and locks:* When a train has passed the critical area of a switchbox, both the lock and reservations for that train at that switchbox are *released* in the state space of the train as well as in the state space of the switchbox.

2.2 Overview of Formal Development

The modelling process follows a *stepwise development* paradigm, where several different models are developed, going from a very abstract view of the real-world system to a more concrete view. In this way, three specifications of generic state transition system models were developed.

The first is an abstract model capturing the system behaviour, but abstracting away from the explicit communication between the control components. Hence, e.g. a *reservation* event is treated as an atomic event, abstracting away from the intermediate steps issuing requests and acknowledgements. However, it was known from the start that these intermediate steps should later be explicitly modelled. The starting point is thus a stage where there is already an idea of needing *event decomposition*. This affects the specification of the first model, where the auxiliary functions for checking and updating the state spaces of the control components are divided into functionality for train control computers and switchboxes, respectively.

The second model is developed using event decomposition for collaborative events (i.e. events involving communication between control components) of the first model in order to model the steps of the communication protocols for such events. At this modelling level, the transition rules are specified in a property-oriented manner, resulting in the least restrictive possible behaviour of the system. This allows for several different legal orders of events.

The third model is an example of restricting the second model to a more specific control protocol for each collaborative event inducing a specific order of events. This is achieved by restricting the guards of relevant transition rules, such that the corresponding transitions can only be executed in fewer cases. Thus the set of paths of the state transition system of the third model is a subset of that of the second model.

The specified system models are generic, i.e. without any configuration data describing the railway network and the control components with their data. To verify the models by model checking, they must be instantiated with configuration data. The instantiation and verification will be described in Sect. 6, while the generic models will be explained in Sects. 3, 4, and 5.

3 First Model

The specification of the first (generic) model can be divided into several different parts:

- Types and values for the static configuration data and dynamic data.
- Functions describing wellformedness and consistency of configuration data.

- Functions describing the safety of the system.
- Guard and state updater functions.
- State variables.
- Transition system rules.

Static Configuration Data. The static configuration data consist of the data for the railway network, which includes information about which segments and switchboxes are in the network, and data about which trains are in the network.

Unique identifiers for segments, switchboxes, and trains of the system are given by types. These are not further specified in the generic model, but are intended to be defined by variant types enumerating the concrete identifiers when the model is instantiated. Train identifiers must at least include the special value *t_none*.

```
type
  Segment,
  TrainID == t_none | _,
  SwitchboxID
```

The network layout describing how the segments of the system are connected is given by a value *network* of an explicitly defined type *Network*. The *network* value is not specified further in the generic model, but is intended to be explicitly defined by a constant when the model is instantiated.

```
type Network = ...
value
  network : Network
```

Types for Dynamic Data. Besides configuring the system with static configuration data, the system must also be configured with initial values for dynamic data which changes e.g. when trains move. The types specified are the ones needed for each of the fields in the state spaces of the control components. For example, the type for the reservations of a switchbox is called *SbResMap* and is a mapping from segment identifiers to train identifiers:

type SbResMap = Segment $\overrightarrow{m}$ TrainID

There is a similar map for reservations stored in a train:

type TResMap = SwitchboxID $\overrightarrow{m}$ Segment-**set**

For modelling the *state spaces* of each control component, types of the form *TrainID* $\overrightarrow{m}$ [*value type*] and *SwitchboxID* $\overrightarrow{m}$ [*value type*] have been defined. For example, the reservations for each of the switchboxes are saved in a variable of the type

type SbResState = SwitchboxID $\overrightarrow{m}$ SbResMap,

and the reservations for the trains are saved in a variable of the type

type TResState = TrainID $\overrightarrow{m}$ TResMap

Using maps from component identifiers to state values allows for the specification of the local state of each component.[1]

Wellformedness, Consistency and Safety Functions. The functions describing wellformedness and consistency of configuration data and the functions describing the safety of the system are used when formulating the transition system assertions, i.e. the properties which the instantiations of the models are checked against.

Guard and State Updater Functions. The guard and state updater functions are also used when formulating the transition system. They are used in the transition system rules, where each rule consists of a guard and a collection of effects, i.e. state updates. An example of a guard function is the following, *sb_can_reserve*, which is used to determine, from the point of view of a switchbox, whether a reservation can be made. This should be the case if the switchbox is associated with the segment to be reserved and the segment is not reserved already by any train at the switchbox.

$$\text{sb\_can\_reserve} : \text{SbResMap} \times \text{Segment} \to \mathbf{Bool}$$
$$\text{sb\_can\_reserve}(\text{res}, \text{seg}) \equiv \text{seg} \in \mathbf{dom}(\text{res}) \wedge \text{res}(\text{seg}) = \text{t\_none}$$

The parameters of the guard function are the segment which should be reserved (*seg*) and the reservations of the switchbox itself (*res*).

An example of an updater function is the following *sb_res*, which updates the reservations of a switchbox:

$$\text{sb\_res} : \text{SbResMap} \times \text{TrainID} \times \text{Segment} \to \text{SbResMap}$$
$$\text{sb\_res}(\text{res}, \text{tid}, \text{seg}) \equiv \text{res} \dagger [\text{seg} \mapsto \text{tid}]$$

The parameters for the updater function consist of the data component, i.e. the reservations of the switchbox, (*res*) to which changes should be made, and the data necessary for the change, i.e. the train (*tid*) for which the segment (*seg*) should be reserved.

For the reservation event, there is a similar guard function *t_can_reserve* which is used to determine, from the point of view of a train control computer, whether a reservation can be made and there is an updater function *t_res* to be used to update the train state.

For other events e there are similar guard functions and updater functions.

State Variables. Several local variables are declared in the transition system. The initial values of these determine the initial state of the transition system. In the generic model, the variables are uninitialised, so they must be given values when the model is instantiated for model checking.

The variables are specified using the types for dynamic data mentioned earlier. There is a variable for each field of the control component state spaces. For

[1] As the model is generic, the number of components is not yet known, so we can't specify a variable for each component holding its local state. Instead we use maps as shown above.

example, the variables for the switchbox reservations and for the train reservations are specified as follows:

local sbRes : SbResState, tRes : TResState

Transition System Rules. The rules of the transition system define the possible events (state transitions) of the system. A transition rule consists of a guard and an effect, where the guard is a predicate over the state variables determining for which states the effect of the rule can be applied, and the effect of the rule is a collection of state variable updates. In the state variable updates, primed versions of the variables refer to the variables in the resulting post state. Transition rules can be combined by non-deterministic choice ($[]$). Furthermore, a non-deterministic choice over a set of rules of the same form, only differing by a parameter x of finite type T, can be expressed as $([]\ x : T \bullet rule)$, where x occurs in the rule *rule*. It is a shorthand for writing a non-deterministic choice between all rules that can be obtained by substituting a value $v : T$ for x in *rule*.

In this first model, for each event e, there is a rule of the following form:

```
([] sbid : SwitchboxID, tid : TrainID, ... •
  [rule_name]
  t_can_e (...)  ∧ sb_can_e (...)  ==>
    tData′(tid) = t_e (...),  sbData′(sbid) = sb_e (...))
```

the ellipsis in the first line represents any extra values needed for that particular event; *tData* and *sbData* are place-holders for variables in the train control computer state space and the switchbox state space, respectively, changed by the transition (multiple variables from each state space may be changed by one transition); *t_e* and *sb_e* are place-holders for updater functions returning the new value for the variables.

In case an event is not collaborative, but e.g. a pure train event like *move*, the format of the rule is reduced by removing the quantification over *tid* or *sbid*, respectively, and the guard and updates for the *tid* or *sbid* component, respectively.

As an example of a transition system rule, the rule for the *reservation* event is specified as follows:

```
([] sbid : SwitchboxID, tid : TrainID, seg : Segment •
  [res]
  t_can_reserve(tSboxes(tid),sbid,seg,tRoute(tid),tRes(tid)) ∧
  sb_can_reserve(sbRes(sbid),seg)  ==>
    tRes′(tid) = t_res(tRes(tid),sbid,seg),
    sbRes′(sbid) = sb_res(sbRes(sbid),tid,seg))
```

where *tRoute(tid)* and *tSboxes(tid)* give the segments and switchboxes of the route of train *tid*, respectively, and *tRes(tid)* and *sbRes(sbid)* give the reservations of train *tid* and switchbox *sbid*, respectively.

As can be seen, two guard functions are used to determine whether the reservation can be made: only if both the train and the switchbox agree, the event can take place. The effect of the rule is specified using two updater functions to update the reservations of both the train and the switchbox in question.

4 Second Model

In the second step, the model has been refined to explicitly model a communication scheme between the control components of the system. The collaborative events of the system are decomposed into multiple sub-events, such that a simple request-acknowledge protocol scheme is modelled. The event refinement has been chosen to be atomic (i.e. all the sub-events of an event have to be completed before a new event can happen) in order to keep the state space as small as possible. It can be shown that removing the atomicity requirements from the resulting model M_2 leads to a model $M_2^{'}$ which is behaviourally equivalent to M_2 with respect to the externally (physical) observable state, i.e. train positions and point positions. This is because the internal protocol states of different communication events are disjoint, so that every set of interleaved communication transactions has an outcome which is equivalent to that of a serialised execution of the same transactions in some specific order. Hence, any safety conditions proved for M_2 will also hold for $M_2^{'}$.

In the communication protocols, the train control computers are the initiating party, issuing requests to the switchboxes. When a switchbox receives a request, it decides whether it is able to comply with the request and, depending on this, sends either a positive or negative acknowledgement to the train. If the switchbox can comply with the request, it will also update its state space accordingly. Similarly, when a train control computer receives a positive acknowledgement, it will update its state space accordingly. If the switchbox cannot comply with the request, neither the state space of the switchbox nor of the train control computer will be updated.

To model the communication between the control components, the collaborative events of the system have been decomposed in the following manner. For each collaborative event e, the single transition rule in the first model is now replaced with several separate sub-rules:

- *req_e*, which is the initiation of the event. This corresponds to a train control computer issuing a request to a specific switchbox with any relevant information for the event in question.
- *ack_e*, which is the positive acknowledgement rule for the switchbox. This corresponds to the switchbox accepting the request, changing its own state space accordingly and issuing the positive acknowledgement to the train control computer in question.
- *end_e*, which concludes the event. This corresponds to the train control computer receiving the positive acknowledgement signal from the switchbox and updating its own state space accordingly.
- *nack_e*, which is the negative acknowledgement rule for the switchbox. This corresponds to the switchbox not being able to comply with the request, and therefore issuing a negative acknowledgement to the train control computer in question.

– *end_nack_e* is an auxiliary action for "consuming" the negative acknowledgement from a switchbox and not changing the state space of the train control computer.

To keep track of the messages sent between the control components, several variables have been added to the model:

Interface variables are used to record whether a message is a *request*, an *acknowledgement* or a *negative acknowledgement*, and to record who the sender and receiver are:

req : TrainID $\overrightarrow{m}$ SwitchboxID, −− *request variable*
ack : SwitchboxID $\overrightarrow{m}$ TrainID, −− *positive acknowledge variable*
nack : SwitchboxID $\overrightarrow{m}$ TrainID, −− *negative acknowledge variable*

For instance, *ack(sb)* = *t* models a *positive acknowlegment* from a switchbox *sb* to a train *t*.

Data variables are used for storing data sent as part of a request. For example, for a *reservation* request, the following variable[2] is used to store the segment to be reserved:

tmpSeg : Segment

Event variables are used to keep track of which type of the collaborative events is currently ongoing (if any). There is a Boolean variable for each kind of collaborative event. For example, for the *reservation* event, the following variable is used:[3]

resEvent : **Bool**

The variable is set to true whenever a train control computer requests a reservation of a segment at some switchbox, and set to false when the train control computer has received an acknowledgement (either positive or negative).

As an example of how the new rules of the transition system are specified and how the additional variables are used, consider the rules for requesting, (positive) acknowledging and concluding the *reservation* event:

([] sbid : SwitchboxID, tid : TrainID, seg : Segment •
[req_res]
  ¬resEvent ∧ ¬switchLockEvent ∧
  t_can_reserve (tSboxes(tid), sbid, seg, tRoute(tid), tRes(tid)) ∧
  tid ∉ **dom**(req) ==>
    req′ = req † [tid ↦ sbid],
        resEvent′ = **true**,

[2] Since only one event should be allowed at the same time in this model, it is sufficient to store a segment rather than a map from trains to segments, where for each train *t*, *tmpSeg*(*t*) could hold data sent by *t*.
[3] For this variable there is a similar comment as for *tmpSeg*.

```
        tmpSeg′ = seg)
[]
([] sbid : SwitchboxID, tid : TrainID •
[ack_res]
  tid ∈ dom(req) ∧ req(tid) = sbid ∧ resEvent ∧
  sb_can_reserve(sbRes(sbid),tmpSeg) ==>
    req′ = req \ {tid},
    ack′ = ack † [sbid ↦ tid],
    sbRes′(sbid) = sb_res(sbRes(sbid),tid,tmpSeg))
[]
([] sbid : SwitchboxID, tid : TrainID •
[end_res]
  sbid ∈ dom(ack) ∧ ack(sbid) = tid ∧ resEvent ==>
    tRes′(tid) = t_res(tRes(tid),sbid,tmpSeg),
    ack′ = ack \ {sbid},
    resEvent′ = false)
```

The *req_res* rule can be applied when the system is idle, i.e. when no events are ongoing[4], when the reservation is legal from the train control computer's point of view and the train control computer has not already sent a request. As its effect, the rule sets the request variable for the train identifier and switchbox identifier in question, enables the *reservation event variable* and sets a data variable to the segment to be reserved.

The *ack_res* rule can be applied when a request has been issued, the reservation event variable is enabled and the *reservation* event is legal from the point of view of the switchbox. As its effect, the rule removes the issued request, issues a positive acknowledgement and updates the state space of the switchbox with the reservation (here, the segment data variable from before is used).

Finally, the *end_res* rule can be applied when a positive acknowledgement has been received and the *reservation event variable* is enabled. As its effect, the rule updates the state space of the train control computer (and again uses the segment data variable), removes the acknowledgement and disables the *reservation event variable*.

There are two additional rules (not shown here) for expressing the sending of a negative acknowledgement from a switchbox to a train and for the train receiving it, respectively.

Relation to the First Model. Instances of this model are clearly able to simulate all possible events of the corresponding instances of the first generic model, which was the intention with this step in which no behaviour should be lost. Furthermore, instances of the first model are able to simulate all atomic events of the corresponding instances of this second generic model.

[4] It is this condition which enforces the atomic event refinement.

5 Third Model

The third model has been restricted to model a *just-in-time* allocation principle. In the previous models, any order of legal events was possible. This means, for example, that nothing was preventing a train from reserving the last segment of its route as the first event (other than if the segment was already reserved, of course). This third model should now specify a control strategy, stating that a train must only make reservations of the next upcoming segment in its route (at the two upcoming switchboxes of its route), and must only lock the point at the next upcoming switchbox. This strategy is just one of many choices, and is used to demonstrate the possibility and technique of restricting the protocol of the second model to enforce events to happen in a more specific order.

As mentioned, the train control computers are the initiating party for collaborative events. Therefore, the desired restriction can be achieved by strengthening the guard functions used by the train control computers. This limits the amount of possible events such that they match the steps of the control strategy.

The restriction of the guard functions is accomplished by using the following pattern. If the guard function was previously of the form

t_can_e : ... → **Bool**
t_can_e (...) ≡ ...

then the new, restricted guard function is of the form

restricted_t_can_e : ... → **Bool**
restricted_t_can_e (...) ≡
  t_can_e (...) ∧ new_restriction_1 ∧ ... ∧ new_restriction_n

The extra conjunct(s) can, in some cases, lead to the possibility of the properties of *t_can_e* to be reduced. This is the case when one of the new restrictions implies (parts of) the properties found in the *can_e* guard function.

For the *reservation* event, the restrictions to be included in the updated guard function consist of only allowing a train t to reserve a segment *seg* at a switchbox *sb*, if (1) *sb* is one of the two upcoming switchboxes of the route of t and (2) the segment *seg* is the next segment with respect to the train's position and route.

Hence, the restricted guard function is specified as follows:

restricted_t_can_reserve : SboxMap × SwitchboxID × SwitchboxID × Segment ×
  Route × Position × ResMap → **Bool**
restricted_t_can_reserve (sboxes,sbid,nextsb,segment,route,pos,res) ≡
  t_can_reserve(sboxes,sbid,segment,route,res) ∧
  (sbid = nextsb ∨ (nextsb ∈ **dom**(sboxes) ∧ sbid = sboxes(nextsb))) ∧
  is_single_pos (pos) ∧ seg(pos) ∈ **dom**(route) ∧ segment = route(seg(pos))

In this case it turned out that some of the added sub-properties imply some of the sub-properties in *t_can_reserve(sboxes,sbid,segment,route,res)*, so we simplified the conjunction.

The transition rule for *req_res* is obtained from the second model by replacing *t_can_reserve(tSboxes(tid),sbid,seg,tRoute(tid),tRes(tid))* with *restricted_t_can_reserve(tSboxes(tid),sbid,tNextsb(tid),seg,tRoute(tid), tPos(tid),tRes(tid))*.

Relation to Second Model. Instances of the second model can clearly simulate all possible behaviours of the corresponding instances of this third generic model.

6 Verification

At each of the three specification steps, model instances of the generic model at that level have been verified and tested in several different ways, as explained below, in order to get confidence in the correctness of the generic models. Later, if new network and train configurations are considered, the idea is that the final generic model should be instantiated with that data and model checked.

Note that we have not formally verified a formal refinement/simulation relation between the models, which would require considerably higher verification effort, but only discussed this informally in the previous sections.

6.1 Model Checking

Each of the three generic models have been instantiated with several typical network layouts and a collection of trains. The network layouts and train routes should be chosen such that they include cases where trains need access to the same shared resources (e.g. track segments). In this paper we consider the configuration shown in Fig. 2. In this network two trains are shown in their initial position and the coloured lines show their routes. As it can be seen, the two trains have routes passing the same station in opposite direction. Another typical case we have considered is one, where two trains have routes passing the same line between two stations in opposite direction.

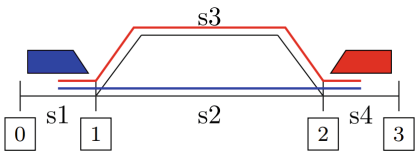

Fig. 2. An example system configuration with two trains and their routes.

After having instantiated the three generic models with configuration data for our example, the three resulting model instances were model checked against several assertions expressed in Linear Temporal Logic (LTL) using the symbolic model checking tool of *Symbolic Analysis Laboratory* (SAL) [1]. The properties asserted were as follows.

- *Safety* properties, stating the absence of derailments and collisions of trains in all reachable states. The absence of collisions is stated as follows, using an auxiliary function named *no_collide*:

  [no_collide] TS ⊢ (∀tid1,tid2 : TrainID •
  G(tid1 ≠ tid2 ∧ tid1 ≠ t_none ∧ tid2 ≠ t_none ⇒ no_collide(tid1,tid2,tPos)))

  where $no\_collide(tid1, tid2, tPos)$ expresses that the intersection of the segments of the positions of $tid1$ and $tid2$ is empty, i.e. the trains are not both occupying the same section. $tPos$ is a state variable storing the positions of all the trains. Similarly, the absence of derailments is stated as follows:

  [no_derail] TS ⊢ (∀tid : TrainID •
  G(¬is_single_pos(tPos(tid)) ⇒ no_derail(tPos(tid),sbConn))),

  where $\neg\ is\_single\_pos(tPos(tid))$ expresses that the train is passing a point (is not on a single segment) and $no\_derail(tPos(tid), sbConn)$ expresses that the train's position $tPos(tid)$ fits the position of the point. $sbConn$ is a state variable storing the point positions at all the switchboxes.
- *Consistency* properties, stating the consistency of distributed data, e.g. that reservations saved in the train control computer state spaces are in agreement with those from the switchbox state spaces, in all reachable states.
- *Wellformedness* properties, stating the wellformedness of configuration data wrt. the static configuration data in all reachable states.
- *Liveness* properties, stating that events are always completed. This only applies to the second and third model. For example, the fact that the reservation event is always completed is stated as follows:

  [finish_res] TS ⊢ G(resEvent ⇒ F(¬resEvent))

  Note that the result for such properties is only sound if there are no deadlocks.
- *Reachability* properties, expressing that there is at least one possible schedule where all trains reach their destination. These have been verified by contradiction: by model checking properties stating that the trains do *not* all eventually arrive at their destination:

  [not_all_trains_arrive] TS ⊢ G(¬(∀tid : TrainID •tPos(tid) = dest(tid)))

  where $dest(tid)$ is the destination position of train tid. This property is expected to be false and should generate a counter example showing a trace where all trains arrive at their destination.

All the desired properties were successfully verified for the three model instances. (In particular, the property [$not\_all\_trains\_arrive$] gave in each case, as desired, rise to a counter example demonstrating that there exists at least one schedule, where all trains arrive at their destination.)

Furthermore, we applied successfully the SAL deadlock checker to the three model instances to ensure *absence of deadlocks*.

Note that even if invariant properties for a model instance of the first generic model has been model checked, we need to model check them again for the

corresponding instance of the second generic model as there are new intermediate states we want to be sure are safe.

Note also that in principle, the model checking of invariant properties for a model instance of the third generic model should not be necessary when they have been model checked for the corresponding instance of the second generic model (as all behaviours of the third model are simulated by behaviours in the second model), but since we made some simplifications of the guards in the third generic model, we also model checked the properties for the model instance of the third model.

6.2 Other Verification Activities

Before beginning the process of symbolic model checking different model instances against the desired properties, other tools were used to gain confidence in the correctness of the function and transition system rule specifications.

- *Testing of functions:* Important functions (e.g. for expressing safety and consistency properties, which are used in the transition system assertions) were tested using the RSL test case construct. The functions were validated to ensure that the assertions to be verified in the model checking process are correct. This testing activity was only needed in the first specification step, as no new functions were used in the later steps.
- *Bounded model checking:* The transition rules of the model instances were tested using the SAL bounded model checker, which only explores the paths in the transition system to a certain, given depth. Therefore, attempting to verify the properties stated above with the bounded model checker reveals bugs much faster.

7 Conclusion and Future Work

In this paper we have shown a method to stepwise develop a generic state transition system model of a real-world distributed railway interlocking system and verify safety and consistency properties of instances of these models by model checking. This method could also carry over to other, similar applications.

The models are expressed in an extension to RSL: RSL-SAL [10]. Although stepwise development of state transition systems is well known from other languages, it is novel for RSL. The stepwise development has shown to be very useful: Firstly, it allows the initial specification to abstract away from details and complexity which can be added later in a development step. This means that a simpler model expressing essential system behaviour can be developed first without worrying about concrete details. This eases the modelling process. It also has the advantage that essential system behaviour can be verified already at this stage, allowing the developer to gain confidence in the specification, before adding details that would most likely increase the time and memory usage of the verification. Secondly, the idea of letting the second model be so general

(e.g. without having a restriction on the ordering of reservations that a train should send) that it can be refined to several different concrete behaviours (e.g. with specific orderings of reservations) by restricting the guards is useful as the invariant properties which are shown to be satisfied by the general model will also be satisfied by any restrictions. In this way one can create a *library* of different families of models, and variants of different control protocols can be explored and compared.

For the model checking, the SAL symbolic model checker was used, just for a proof of concept of our method, but other back-ends can be used as well.

In future work we plan to experiment with other model checking techniques, e.g. SAT-based k-induction, and other back-ends, e.g. RT-Tester [12], in order to find the most efficient verification technique and apply these also to larger networks. In another case study [13], RT-Tester was used to perform k-induction in order to prove a centralised interlocking system and turned out to be very efficient and scale up to big networks. We also plan to extend the models with additional operations for cancelling reservations and for changing the direction of a train.

Acknowledgements. The authors would like to express their gratitude to Jan Peleska from whom the case study originates and together with whom the second author had the great pleasure to verify the same case study by theorem proving [8]. We would also like to thank him and the reviewers for very useful comments to drafts of this paper.

References

1. Symbolic Analysis Laboratory, SAL (2001). http://sal.csl.sri.com
2. CENELEC European Committee for Electrotechnical Standardization. EN 50128:2011 - Railway applications - Communications, signalling and processing systems - Software for railway control and protection systems (2011)
3. Fantechi, A., Gnesi, S., Haxthausen, A., van de Pol, J., Roveri, M., Treharne, H.: SaRDIn - a safe reconfigurable distributed interlocking. In: Proceedings 11th World Congress on Railway Research (WCRR 2016). Ferrovie dello Stato Italiane, Milano (2016)
4. Fantechi, A., Haxthausen, A.E., Nielsen, M.B.R.: Model checking geographically distributed interlocking systems using UMC. In: 2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP), pp. 278–286 (2017)
5. Ferrari, A., Magnani, G., Grasso, D., Fantechi, A.: Model checking interlocking control tables. In: Schnieder, E., Tarnai, G. (eds.) FORMS/FORMAT 2010, pp. 107–115. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14261-1_11
6. Haxthausen, A.E., Automated generation of formal safety conditions from railway interlocking tables. Int. J. Softw. Tools Technol. Transf. (STTT) **16**(6), 713–726 (2014). Special Issue on Formal Methods for Railway Control Systems
7. Haxthausen, A.E., Le Bliguet, M., Kjær, A.A.: Modelling and verification of relay interlocking systems. In: Choppy, C., Sokolsky, O. (eds.) Monterey Workshop 2008. LNCS, vol. 6028, pp. 141–153. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-12566-9_8

8. Haxthausen, A.E., Peleska, J.: Formal development and verification of a distributed railway control system. IEEE Trans. Softw. Eng. **26**, 687–701 (2000)
9. James, P., et al.: Verification of scheme plans using CSP||B. In: Counsell, Steve, Núñez, Manuel (eds.) SEFM 2013. LNCS, vol. 8368, pp. 189–204. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-05032-4_15
10. Perna, J.I., George, C.: Model checking RAISE applicative specifications. In: Proceedings of the Fifth IEEE International Conference on Software Engineering and Formal Methods, pp. 257–268. IEEE Computer Society Press (2007)
11. The RAISE Language Group: George, C., Haff, P., Havelund, K., Haxthausen, A.E., Milne, R., Nielsen, C.B., Prehn, S., Wagner, K.R.: The RAISE Specification Language. The BCS Practitioners Series. Prentice Hall Int., Englewood Cliffs (1992)
12. Verified Systems International GmbH. RT-Tester Model-Based Test Case and Test Data Generator - RTT-MBT - User Manual (2013). http://www.verified.de
13. Vu, L.H., Haxthausen, A.E., Peleska, J.: Formal modelling and verification of interlocking systems featuring sequential release. Sci. Comput. Programm. **133**(2), 91–115 (2017). https://doi.org/10.1016/j.scico.2016.05.010
14. Winter, K.: Model checking railway interlocking systems. In: Proceedings of Twenty-Fifth Australasian Computer Science Conference (ACSC 2002), pp. 303–310 (2002)