Model Checking a Distributed Interlocking System Using k-induction with RT-Tester

Signe Geisler and Anne E. Haxthausen

DTU Compute, Technical University of Denmark, Kongens Lyngby, Denmark.
sgei@dtu.dk and aeha@dtu.dk

Abstract. This paper investigates the use of k-induction with RT-Tester for tackling the challenge of verifying the safety of a distributed railway interlocking system. For a real-world case study, it is described how a generic and reconfigurable model of the system is modelled in a new extension of the RAISE Specification Language (RSL). The generic model is instantiated with concrete data sets and subsequently model checked with respect to safety properties using the k-induction facilities in RT-Tester. The performance metrics of the verification with k-induction are additionally compared with the metrics of verifying the same system model with the SAL model checker.

Keywords: model checking, RAISE, railway interlocking systems, distributed systems, k-induction, RT-Tester

1 Introduction

This paper considers how to use k-induction to tackle the challenge of formally verifying the safety of a real-world geographically distributed interlocking system, RELIS 2000. This system was originally developed by INSY GmbH Berlin and first described in [14].

Railway interlocking systems are responsible for controlling the track side equipment and granting movement authorities to trains in a railway network such that derailments and collisions of trains are avoided. Traditionally, such systems are centralised. The interlocking system we consider in this paper, however, is geographically distributed: there are multiple control components which have been deployed at specific points along the tracks and in the trains, respectively. The control components must communicate and collaborate to control the track side equipment and grant movement authorities to trains in a safe manner.

We have previously investigated the formal specification and model checking of this interlocking system [13]. We used a stepwise development method where we incrementally added details to the model specification. For the verification of the interlocking system we used model checking, since the verification process is fully automated with this method.

We were using RSL$\star$, an extension to RSL-SAL, which itself is an extension of RSL – the *RAISE Specification Language* [20]. RSL is a formal specification language, allowing several different styles of formulating modular specifications.

RSL-SAL [19] extends the functionality of RSL with constructs for the specification of transition systems in a guarded command style. RSL-SAL specifications can be translated by the RSL tool set (*rsltc*) and subsequently model checked using the SAL model checker [2]. Therefore, we used the SAL model checker as a proof of concept, but found that the scalability was limited in terms of for how large railway networks the model could be verified.

In a previous study [23], k-induction with RT-Tester [18, 22] was used for proving the safety of a centralised interlocking system with great efficiency and scaling. Our goal in this paper is to investigate whether the scalability of the verification of the distributed interlocking system in [13] can also be improved by using k-induction.

Therefore, we have equipped *rsltc* with functionality for translating RSL$\star$ specifications into model representations in RT-Tester such that k-induction with RT-Tester can be used to prove the safety of models of the distributed interlocking system considered in [13].

1.1 Related Work

Formal verification of railway interlocking systems is a well-researched topic. An overview of recent trends can be found in [4, 9].

Centralised systems are the tradition for railway interlocking, and therefore a great part of the literature on formal verification of interlocking systems focuses on these. However, there are sources which cover the formal verification of distributed interlocking systems: The case study considered in this paper, was previously investigated in [14], but the behavioural model in [14] was expressed using the RSL process algebra instead of the RSL$\star$ guarded command language that we are using, and the verification used the RAISE theorem prover rather than model checking. Another example can be found in [11], where a geographically distributed railway interlocking system was formally modelled and verified using the UMC model checker [1, 6], although with a very different control protocol than the one used in this paper. A discussion of advantages and challenges of distributed interlocking can be found in [10].

The k-induction technique with RT-Tester that we are exploring is the same as used by Vu et al. in [23], but they used it for verifying a centralised interlocking system, and the modelling language was not a general-purpose language like RSL$\star$, but a domain-specific railway modelling language.

There are examples of using many other model checking tools such as Simulink and UPPAAL ([5]), SPIN and nuSMV ([12]), proB ([3]), mCRL2 ([7]), and FDR ([16]) for the verification of railway control systems. For a comparative study, see [15].

1.2 Paper Overview

In Sect. 2, we introduce the background for the verification scheme. In Sect. 3, we introduce the case study under consideration in this paper, and in Sect. 4, we outline the generic model specification we have created of the interlocking system from the case study. Sect. 5 presents our verification efforts and discusses the scalability of our verification approach. Finally, Sect. 6 gives a conclusion and ideas for future work.

2 RT-Tester and k-induction

In this section we introduce some mathematical foundations for the applied verification approach. We briefly explain how a system model can be expressed and then describe the k-induction verification scheme.

Model representations in RT-Tester. In RSL⋆ a system model is specified by a set of variable declarations (defining the state space), an initialisation constraint and a collection of transition rules in guarded command style. Using *rsltc*, such a system model can be translated to RT-Tester's [22] internal representation, where the model is expressed as a Kripke structure. In the Kripke structure, the initial state and the transition relation are expressed in propositional form over the states of the system:

$\mathcal{I}(s_0)$ is a proposition which holds for a state s_0, if s_0 is an initial state.

$\Phi(s, s')$ is a proposition which holds for a pair of states (s, s'), if there is a transition from s to s' in the model.

***k*-induction.** k-induction [17, 21] is a verification scheme which can be used to prove a state invariant ϕ in two steps: a *base case* and an *induction step*, both of which can be formulated as bounded model checking problems. This makes it possible to use RT-Tester's integrated bounded model checker to perform k-induction.[1]

In the *base case*, it should be proved that the property ϕ holds in every state of every acyclic path of length $k > 0$ starting from every initial state. This is formulated as a bounded model checking problem by searching for violations of the base case, i.e. searching for a witness for the following formula:

$$\mathcal{I}(s_0) \wedge \pi^{=}(s_0, \ldots, s_{k-1}) \wedge \neg \bigwedge_{i=0}^{k-1} \phi(s_i) \tag{1}$$

where $\pi^{=}(s_0, \ldots, s_{k-1})$ is a proposition expressing that $s_0, \ldots, s_{k-1}$ is an acyclic path in the model, and $\phi(s_i)$ denotes that ϕ holds in the state s_i. Thus, if a solution is found, there is a state which is reachable within k steps from one of the initial states and which does not satisfy ϕ.

In the *induction step*, it should be proved that if ϕ holds in every state of every acyclic path of length $k > 0$ starting from an arbitrary state, then ϕ also

[1] The facilities in RT-Tester that enable k-induction were originally developed by Linh Hong Vu in collaboration with Jan Peleska and his team [23] based on the work presented in [17, 21] and using the bounded model checker of RT-Tester.

holds in any $k+1$th state. This is formulated as a bounded model checking problem by searching for violations of the induction step, i.e. searching for a witness for the following formula:

$$\pi^{=}(s_n, \ldots, s_{n+k}) \wedge \bigwedge_{i=0}^{k-1} \phi(s_{n+i}) \wedge \neg\phi(s_{n+k}) \tag{2}$$

If a solution is found, there is a path of length $k+1$ for which ϕ holds for the first k states, but not for the last state in the path.

If violations are neither found for the base case nor for the induction step (i.e. no witness was found for either of the above formulae), ϕ is an invariant.

If a violation is found in the base case, ϕ is not an invariant.

If a violation is found in the induction step, this might be a false negative, if the considered path starts in an unreachable state. To eliminate such *spurious* violations, one can (1) define a *strengthening invariant* ψ that is not satisfied in that unreachable start state, and then try to prove $\phi \wedge \psi$ instead of just ϕ, or (2) one can re-try the induction with a higher value of k (incremental k-induction).

Incremental k-induction works by first attempting to prove the base case starting at $k = 1$ and, if this is successful, attempting to prove the induction step for the same k. If the induction step fails, the value of k is incremented by one and the attempt to prove the base case and the induction step is repeated for the new value of k. The proof terminates unsuccessfully if the base case fails (or if the upper limit set for k is reached). The proof terminates successfully when the base case and induction step are proved for some value of k.

In principle, the incremental approach has the advantage that the process is automated. However, for some systems, with this approach k will reach such large values that it will take too much memory or unreasonably long time to explore the necessary subset of the state space. In such cases, it will be more efficient to invent and use strengthening invariants such that a lower value of k (e.g. $k = 1$) can be used.

3 Case Study

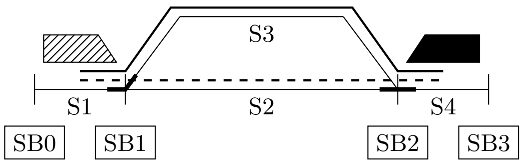
This section presents the considered interlocking system from [14]. This description of the case study is from [13].

The *control strategy* of the system must ensure the safety of the system by preventing 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 associated with a single point or an endpoint 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 switchboxes, 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).

T1	
Route segments	S1, S2, S4
Route switchboxes	SB1, SB2, SB3
Position	S1
Reservations	SB1:{S1}
Locks	–

T2	
Route segments	S4, S3, S1
Route switchboxes	SB2, SB1, SB0
Position	S4
Reservations	SB2:{S4}
Locks	–



SB0	
Connected	S1
Res S1	–
Locked by	–
Sensor	passive

SB1	
Connected	S1-S3
Res S1	T1
Res S2	–
Res S3	–
Locked by	–
Sensor	passive

SB2	
Connected	S2-S4
Res S4	T2
Res S2	–
Res S3	–
Locked by	–
Sensor	passive

SB3	
Connected	S4
Res S4	–
Locked by	–
Sensor	passive

Fig. 1. An example system, adapted from [13].

The basic idea of the control strategy is as follows:

1. *Permission 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 $SB1$, and $S1$ must be connected to $S2$ 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 switch and 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 enters the critical area of a switchbox, the sensor associated with the switchbox will become *active*, and when the train later leaves the critical area of the switchbox, the sensor will become *passive* which in turn causes both the lock and reservations for that train at that switchbox to be *released* in the state space of the switchbox. When the train leaves the critical area of the switchbox, also the lock and reservations at that switchbox will be *released* in the state space of the train.

4 Generic Model and Verification Obligations

In this section, we will outline the model specification of the distributed interlocking system and the specification of proof obligations. The interlocking system is modelled as a *generic model*, which can later be instantiated with configuration data. An example instantiation of the full model corresponding to Fig. 1 can be found online.[2]

The specification can be divided into several different parts:

– Types for the network configuration data.
– Types and values for the static control component data.
– Types and state variables for the dynamic control component data.
– Interface and communication variables.
– Transition system rules.
– Functions for describing invariants of the system.

[2]`https://github.com/raisetools/rslstar/tree/master/spec-examples/dracos/discorail2020`

The verification obligations (expressing the safety properties and other desired invariants) are specified as LTL assertions, which use the functions describing invariants of the system.

In Sect. 4.1 we give a brief overview of the adaptations we have made to the model from [13]. Then, in the following sections, we will elaborate on the different parts of the adapted model and the verification obligations.

Note that the original transition system model in [13] was developed in a stepwise manner and with translation to SAL in mind. More details about this can be found in our paper [13]. The final model of the stepwise development process resulted in the specification of a control system adhering to a *just-in-time* allocation principle, where each train can only make the immediately necessary reservations and locks at any point in time. This means that the train must only make reservations of its next segment (at the two upcoming switchboxes along its route) and that it must only lock the point at the next upcoming switchbox. It is this variant of the control algorithm that we are considering in this paper.

4.1 Overview of Adaptations

Since the subset of RSL⋆ translatable to SAL differs from the subset translatable to RT-Tester, it has been necessary to adapt the original specification from [13] in order to be able to translate it into RT-tester. For instance, the original specification uses RSL⋆ maps and sets which are translatable to SAL, but not to RT-Tester. These data structures have been replaced with RSL⋆ arrays, which are translatable to RT-Tester. As arrays can also be translated to SAL, it has actually been possible to create a specification which is translatable both to RT-Tester and to SAL, allowing us to compare the performance results from both model checkers on the same system model.

In order to use less space for data, we also made some additional adaptions of the model.

4.2 Network Configuration Data

The network configuration consists of data about the segments, switchboxes and trains in the network. Each segment, switchbox and train is given a unique identifier by specifying a type for each of them. In the generic model, the types are not further specified, but the intention is that the types enumerate the concrete identifiers for each component when the model is instantiated. Each identifier type must at least include a special *none* value: *seg_none* for segments, *sb_none* for switchboxes and *t_none* for trains. For each of the types, we also specify subtypes which do not include the special *none* values.

type
  SegmentID == seg_none | _,
  SegmentID_prime = {| s : SegmentID • s $\neq$ seg_none|}
  SwitchboxID == sb_none | _,
  SwitchboxID_prime = {| sb : SwitchboxID • sb $\neq$ sb_none |}

TrainID == t_none | _,
TrainID_prime = {| t : TrainID • t $\neq$ t_none |},

4.3 Static Control Component Data

In Sect. 3, we introduced the information that each switchbox and each train must keep track of. Some of this data is static: the route information for each train in the network and segments adjacency information for each switchbox in the network. In the generic model, this static data is modelled by generic constants. For instance, the following declaration states that for each train t in the type *TrainID_prime* (i.e. for each train in the network), there is a constant *route*[*t*] storing the train's route (of type *Route*).

value
route[t : TrainID_prime] : Route

The type *Route* is defined as follows:

type
Route = **array** RouteIndex **of** SegmentID,
RouteIndex = {| i : **Int** • i $\geq$ 0 $\wedge$ i < max_route_length + 1 |}

As it can be seen, a route is modelled as an array of segment identifiers. The index type of the array, the *RouteIndex* type, is an integer range from zero to the maximum route length (the value of which depends on the initialisation of the model).

To store the segments adjacent to each switchbox, we declare the following generic variable.

value
sbSegments[sb : SwitchboxID_prime] : SbSegments

The type *SbSegments* is defined as follows:

type
SbSegments = **array** SbIndex **of** SegmentID,
SbIndex = {| i : **Int** • i $\geq$ 0 $\wedge$ i < 3 |}

The adjacent segments are modelled as an array of segment identifiers. The index type of the array, the *SbIndex* type, is an integer range from zero to two (such that there is an entry for each of the up to three segments with which a switchbox may be associated).

When the model is instantiated, for each t in *TrainID_prime*, a concrete value for *route*[*t*] must be specified, and for each *sb* in *SwitchboxID_prime*, a concrete value for *sbSegments*[*sb*] must be specified as configuration data.

4.4 Dynamic Control Component Data

Most of the data that the trains and switchboxes must keep track of is dynamic. The trains store their position, their reservations and locks. The switchboxes similarly store their reservations and locks along with data about the associated sensor and information about which of its adjacent segments are currently connected.

In the generic model, this dynamic data is modelled by generic variables in a similar way as static data was modelled by generic constants. For instance, the following generic variable is used for storing reservations at each of the switchboxes *sb*.

```
variable
  sbReservations[sb : SwitchboxID_prime] : SbReservation
```

```
type
  SbReservation = array SbIndex of TrainID,
  SbIndex = {| i : Int • i ≥ 0 ∧ i < 3 |}
```

As it can be seen, a switchbox reservation is modelled as an array of *TrainID*s with an index in the same range as used for the *SbSegments* arrays (such that there is an entry for each of the up to three segments with which a switchbox may be associated). Hence, $sbReservations[sb][i] = t$ models that switchbox *sb* has recorded a reservation for train *t* of the segment which can be found at index *i* in the *sbSegments*[*sb*] array.

When the generic model is instantiated, initial values for all the resulting variables must be specified as configuration data.

4.5 Interface and Communication Variables

The control components must collaborate on making reservations and locks, so they must be able to communicate. The communication between the components is modelled as a simple request-acknowledge protocol, where the train is the initiating party. A train will consider its own state and can send a request to a switchbox if its state fulfils the requirements. If a switchbox receives a request, it will consider its own state to decide whether the request can be fulfilled. Depending on this, the switchbox sends a positive or negative acknowledgement to the train. If the switchbox can accommodate the request it will also update its state accordingly, as will the train when it receives a positive acknowledgement. If the switchbox sends a negative acknowledgement, neither the state of the switchbox nor the train is updated. We model the communication, i.e. the requests, acknowledgements and data sent as part of a request, by shared variables.

We declare three different generic variables to keep track of the requests and acknowledgements. For example, if a train *t* has sent a request to a switchbox *sb*, then $req[t] = sb$.

```
variable
  req[t : TrainID_prime] : SwitchboxID,
  ack[sb : SwitchboxID_prime] : TrainID,
  nack[sb : SwitchboxID_prime] : TrainID
```

We declare data variables for storing data sent as part of a request – for example, the segment to be reserved:

```
variable
  reqSeg : SegmentID
```

In addition, we declare event variables which describe which kind of event is taking place (*reserve at the next switchbox*, *reserve at the switchbox after the next switchbox* and *lock at the next switchbox*). For example, we declare a variable:

```
variable
  reserveNextEvent : Bool
```

This variable is set to true as soon as a train sends a request for reserving a segment at its next switchbox. The variable is set to false as soon as the train has received an acknowledgement (negative or positive).

4.6 Transition System Rules

In RSL*, the possible behaviour of a system is specified by state transition system rules. Each rule consists of a guard and an effect. The guard is a boolean expression over the state variables. It determines in which states the effect of the rule may occur. The effect of the rule is a collection of variable updates of the form $x' = value$, where the primed variable name x' refers to the variable x in the post state. Rules may be combined by using the non-deterministic choice operator $[]$.

In the specification of our generic model, we use 15 generic transition system rules. A generic rule is a shorthand for a non-deterministic choice over a set of rules of the same form, only differing by one or more parameters. For example, we have the following generic rule expressing for any switchbox *sb*, the sending of a positive acknowledgement to a train `t` that it has reserved the segment at index *i* for `t`:

```
([] sb : SwitchboxID_prime, t : TrainID_prime, i : SbIndex •
  [switchbox_ack_reservation]
  (reserveNextEvent ∨ reserveNextNextEvent) ∧ req[t] = sb ∧
  sbSegments[sb][i] = reqSeg ∧
  sb_can_reserve(sbReservations[sb][i])
  ⟶
  ack′[sb] = t,
  req′[t] = sb_none,
  sbReservations′[sb][i] = t
)
```

This rule represents the non-deterministic choice of the set of rules that are obtained by replacing the parameters *sb*, *t* and *i* with each of the possible combinations of the possible concrete values for each parameter.

In the rule effects, an acknowledgement is sent to the train *t* from the switchbox *sb*, the request from the same train is consumed (by setting the variable to the special value *sb_none*), and the reservations in the same switchbox is updated to have a reservation for t of the segment at its index i. The guard expresses that this can only happen if (1) there is a reservation event going on and the train t has sent a request to the switchbox sb, (2) the segment $reqSeg$ to be reserved is the one at index i of the switchbox, and (3) the segment is not already reserved. The latter is expressed by the following auxiliary function:

$$\begin{array}{l}\text{sb\_can\_reserve} : \text{TrainID} \rightarrow \textbf{Bool} \\ \text{sb\_can\_reserve}(\text{res}) \equiv \text{res} = \text{t\_none},\end{array}$$

4.7 Safety Invariants

We have specified two safety invariants to be proved. They describe that trains do not collide and trains do not derail, respectively. The specification of the former is as follows:

$$\begin{array}{l}(\forall\ \textbf{distinct}\ \text{tid1},\ \text{tid2} : \text{TrainID\_prime} \bullet \\ \quad \text{G}(\text{no\_collide}(\text{hdPos}[\text{tid1}],\ \text{tlPos}[\text{tid1}],\ \text{hdPos}[\text{tid2}],\ \text{tlPos}[\text{tid2}])))\end{array}$$

where $hdPos[t]$ and $tlPos[t]$ is the head, respectively tail, position of a train t[3] and *no_collide* is a function defined as follows:

$$\begin{array}{l}\text{no\_collide} : \text{SegmentID} \times \text{SegmentID} \times \text{SegmentID} \times \text{SegmentID} \rightarrow \textbf{Bool} \\ \text{no\_collide}(\text{hdPos1},\ \text{tlPos1},\ \text{hdPos2},\ \text{tlPos2}) \equiv \\ \quad \text{hdPos1} \neq \text{hdPos2} \wedge \text{hdPos1} \neq \text{tlPos2} \wedge \text{tlPos1} \neq \text{hdPos2} \wedge \text{tlPos1} \neq \text{tlPos2}\end{array}$$

Supplied with the head position and tail position of two *distinct*[4] trains, the function checks that there are no overlaps in the trains' position.

4.8 Strengthening Invariants

As a part of the verification process, we have added several strengthening invariants in order to prove the safety invariants by 1-induction.

Many of the strengthening invariants we have added are adaptions of the consistency properties that we have previously proved for earlier variants of the system [13]. These invariants express agreement between the train data and the switchbox data. For example, one of the strengthening invariants express that each reservation or lock possessed by a train is also found in the reservations/locks of the corresponding switchbox[5].

[3]For the considered local railway, trains are shorter than any segment, so they will at most occupy two segments at a time.

[4]The quantification with the *distinct* keyword generates only distinct combinations of the parameters – in this case distinct pairs of train identifiers.

[5]Note that this property is indeed invariant, even though the updates to the train and switchbox reservations/locks happen in separate transitions, because the update to switchbox reservations/locks always happens before the update to the train reservations/locks. Note therefore also that the converse of the property is *not* invariant and only holds if the system is not in the middle of making a reservation/lock.

We also added several strengthening invariants relating to the interface variables used for the communication between train control computers and switchboxes. For example, one of the strengthening invariants expresses that a switchbox never sends both a positive and negative acknowledgement to a train simultaneously:

$(\forall \text{ sb} : \text{SwitchboxID\_prime} \bullet$
$\quad (\text{ack[sb]} \neq \text{t\_none} \vee \text{nack[sb]} \neq \text{t\_none}) \Rightarrow \text{ack[sb]} \neq \text{nack[sb]})$

5 Verification

In this section, we present our experiences using k-induction with RT-Tester to verify the safety of instances of our generic model. In particular we show the performance metrics (time and memory usage) for verifying the safety properties of models having configurations of increasing size (in terms of number of stations). Furthermore, we compare these results with the performance measures for the same configurations when verifying with the SAL model checker.

The RELIS 2000 system was intended for local railways having stations with 1-2 track segments, each connected by single lines and operated by 2-3 trains. Therefore, in our verification we consider configurations of the model which are representative for this class of local railways.

5.1 Bounded Model Checking

Before performing the k-induction, we used bounded model checking for quick detection of bugs in the considered model instances.

Bounded model checking can be used to find bugs in the specification within the first n transition steps. Given a bound n, the bounded model checker only explores the paths of the transition system of lengths up to the bound. Therefore, bounded model checking can be much quicker than global model checking or k-induction (with a large k) to find bugs.

Bounded model checking (with a bound ≥ 1) can in particular be used to check that the initial state (specified in the configuration data) of a certain model instance adheres to the safety and strengthening invariants. This is especially useful for large networks, where there is some risk of making errors in the configuration data.

The bounded model checker can also be used to show that a system progresses. We used it for proving that there exists at least one path where all trains can reach their destination, (but there might still be other paths for which a livelock or a deadlock may occur).[6] Selecting a path without deadlocks and livelocks is a task for the scheduler, rather than for the interlocking system. Our aim is to ensure that there exists a possible schedule (i.e. at least one path where

[6]This was done by asserting that it is never the case that all trains have reached their destination. This is expected to produce a counter-example showing the transition steps to a state where the trains indeed did reach their destinations.

all trains reach their destinations). For the bound for these experiments, we calculated the minimum number of transition steps needed for all trains to reach their final destination (for example, it takes three transition steps to finalise a reservation).

5.2 *k*-induction

First we tried to use incremental k-induction to verify the safety invariants without any strengthening invariants. Even for a small system instantiation with just one station (as in Fig. 1), the value of k eventually had to be increased to such a degree that the base case took an unreasonable amount of time to prove: After a total of fifty-three hours, the value of k had been increased to fifty-four and we manually terminated the execution.

Therefore, instead, we decided to add strengthening invariants until we obtained a property that is provable by k-induction with $k = 1$. As mentioned in Sect. 2, the strengthening invariants are conjuncted with the safety properties such that the k-induction should now attempt to prove the safety properties *and* the added strengthening invariants. The strengthening invariants were found by inspecting the counter-examples that resulted from failed induction steps and taking inspiration from the consistency properties of an earlier variant of the model [13].

5.3 System Instantiations

As our main goal is to investigate the scalability of using k-induction with RT-Tester, we have instantiated[7] the generic model with configurations of increasing size (in terms of the number of segments and stations in the network). The configurations follow the typical patterns of real-world systems in the class of railway networks we are considering.

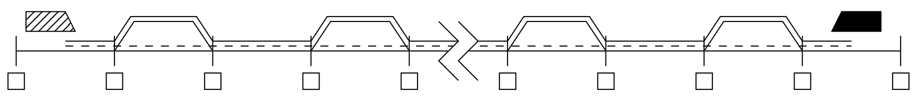


Fig. 2. An example of a typical local railway network (from [13]).

For the class of networks we are considering, the configuration pattern illustrated in Fig. 2 is very typical. Two trains start at either end of the network driving in opposite directions and using distinct track segments at the stations such that they are able to pass each other there (the train routes are illustrated with lines, where the dashed line is the route of the striped train and the solid line is the route of the black train). We use instantiations of the system model

[7]To instantiate a model, configuration data for types, values and variables must be specified as explained in Sect. 4.2, Sect. 4.3, and Sect. 4.4, respectively.

with configurations of increasing size (with one, two, five, ten, fifteen, and twenty stations) for comparing the verification performance of SAL and RT-Tester.

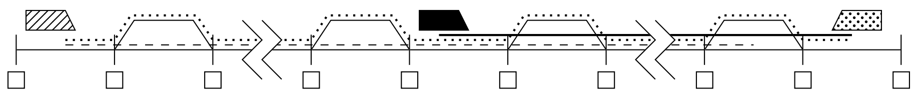


Fig. 3. An example of a typical local railway network with an extra train (from [13]).

Another typical configuration pattern is illustrated in Fig. 3, where an additional train has been added somewhere between the other two trains which are starting at each respective end of the network. We also use instantiations of the system model with configurations following this pattern with ten, fifteen and twenty stations, respectively.

5.4 Results and Verification Metrics

As mentioned previously, the model can be translated to both RT-Tester and SAL. This allows us to compare the performance of the two model checkers when verifying the safety properties of the same model. Note, however, that the k-induction proves the added strengthening invariants along with the safety properties, whereas SAL solely proves the two safety properties.

Below we present the time and memory consumption for verifying the safety properties by k-induction with RT-Tester and with the SAL model checker, respectively. The results were measured using GNU Time[8] on a machine with a Intel(R) Core(TM) i7-8650U CPU @ 1.90GHz and 31GiB of memory.

We firstly present the verification metrics for the instances following the configuration pattern shown in Fig. 2, i.e. with two trains driving in opposite directions from either end of the network. Table 1 shows the time and memory usage for verifying both of the safety properties with RT-Tester and SAL, respectively. For the smaller instantiations with one and two stations, SAL and RT-Tester have approximately the same time and memory usage. However, for the instantiation with five stations RT-Tester uses less than three minutes to verify the properties, whereas SAL uses more than an hour and a half. The memory consumption of RT-Tester is also much lower than that of SAL. For the instantiation with ten stations, the SAL execution was terminated by the operating system after almost seventeen hours due to memory exhaustion. Therefore we did not attempt to verify the safety of larger instantiations with SAL. In contrast to that, we successfully verified the properties with RT-Tester for instantiations with ten, fifteen, and twenty stations.

We now present the verification metrics for the instances following the configuration pattern shown in Fig. 3, i.e. with two trains driving in opposite directions

[8] `https://www.gnu.org/software/time/`

Table 1. Time (hh:mm:ss) and memory (MB) usage for verifying the safety properties for the different instances of the system model with the initial state shown in Fig. 2.

	Time		Memory	
# of stations	RT-Tester	SAL	RT-Tester	SAL
1	00:00:06	00:00:06	118	113
2	00:00:18	00:00:17	187	165
5	00:02:20	01:42:57	678	5,076
10	00:33:02	*	2,860	*
15	02:10:53	†	7,493	†
20	10:04:09	†	16,232	†

* The execution was terminated by the operating system due to memory exhaustion.
† This experiment was not run.

from either end of the network and a third train starting roughly midway between the other two trains. Table 2 shows the time and memory consumption for verifying both of the safety properties with RT-Tester.

As it can be seen, the addition of a third train increased the time and memory usage. We again successfully verified the properties for instantiations with ten and fifteen stations, but for twenty stations the execution was this time terminated by the operating system due to memory exhaustion. SAL was not able to verify any of these instantiations.

Table 2. Time (hh:mm:ss) and memory (MB) usage for verifying the safety properties for the different instances of the system model with the initial state shown in Fig. 3.

	RT-Tester	
# of stations	Time	Memory
10	8:32:11	7,905
15	41:25:47	22,813
20	*	*

* The execution was terminated by the operating system due to memory exhaustion.

6 Conclusion and Future Work

In this paper we have shown the specification of a generic system model of a real-world geographically distributed interlocking system and successfully verified the safety of model instances representative for the class of local railway networks for which this interlocking system is intended.

We performed the verification using k-induction with RT-Tester. In order to make the proof within reasonable time, we found it was necessary to add several strengthening invariants and make the induction for $k = 1$ rather than use incremental k-induction. We successfully verified model instances for typical, real-world network configurations for the considered local railways.

We also verified the system model with the SAL model checker, and in general found k-induction with RT-Tester to be much more efficient – both in terms of time and memory consumption. The scalability of verifying the distributed interlocking system with RT-Tester was found to be substantially better compared to the SAL model checker, as it was possible to verify much larger networks with RT-Tester before reaching memory exhaustion. A main reason for this is that k-induction is based on bounded model checking. In contrast to this, SAL is performing global symbolic model checking and thus must explore the full state space in order to verify properties.

For future work, it would be interesting to automate the addition of strengthening invariants, for example using some of the strategies presented in [8]. We would also like to further optimise the system model in terms of how the data of the control components is stored and investigate whether the method can scale to even larger systems. In addition, it would be interesting to investigate k-induction with RT-Tester for other system models specified in RSL$\star$ – of railway interlocking systems or even other safety-critical systems.

Acknowledgements

The authors would express their gratitude to Jan Peleska for wonderful inspiration and discussions, and for hosting the first author on several research stays and granting access to the RT-Tester source code, without which this paper would not have been possible. The authors would also like to acknowledge the work of Linh Hong Vu, in collaboration with Jan Peleska and his team, who originally implemented the k-induction facilities in RT-Tester, and whose work has been very valuable to us. In addition, Jan Peleska, Niklas Krafczyk, and Linh Hong Vu deserve many thanks for their technical help with the RT-Tester source code. The authors would also like to thank Chris George for valuable input during the development of the RSL$\star$ extension to RSL. Finally, thanks go to the anonymous reviewers for helpful comments concerning the presentation.

References

1. UMC homepage, `http://fmt.isti.cnr.it/umc/V4.2/umc.html`
2. Symbolic Analysis Laboratory, SAL, home page: `http://sal.csl.sri.com` (2001)
3. de Almeida Pereira, D.I., Deharbe, D., Perin, M., Bon, P.: B-specification of relay-based railway interlocking systems based on the propositional logic of the system state evolution. In: Collart-Dutilleul, S., Lecomte, T., Romanovsky, A. (eds.) Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification. Lecture Notes in Computer Science, vol. 11495, pp. 242–258. Springer International Publishing, Cham (2019)
4. 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. Lecture Notes in Computer Science, vol. 11023, pp. 20–29. Springer International Publishing (2018)

5. Basile, D., ter Beek, M.H., Ferrari, A., Legay, A.: 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
6. 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). https://doi.org/http://dx.doi.org/10.1016/j.scico.2010.07.002
7. Bouwman, M., Janssen, B., Luttik, B.: Formal modelling and verification of an interlocking using mCRL2. In: Larsen, K.G., Willemse, T.A.C. (eds.) Formal Methods for Industrial Critical Systems - 24th International Conference, FMICS 2019, Proceedings. Lecture Notes in Computer Science, vol. 11687, pp. 22–39. Springer (2019). https://doi.org/10.1007/978-3-030-27008-7_2
8. Bradley, A.R., Manna, Z.: Property-directed incremental invariant generation. Formal Aspects of Computing **20**(4-5), 379–405 (2008)
9. 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)
10. 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)
11. 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). https://doi.org/10.1109/PDP.2017.66
12. Ferrari, A., Magnani, G., Grasso, D., Fantechi, A.: Model Checking Interlocking Control Tables. In: Schnieder, E., Tarnai, G. (eds.) FORMS/FORMAT 2010 – Formal Methods for Automation and Safety in Railway and Automotive Systems. pp. 107–115. Springer (2010)
13. Geisler, S., Haxthausen, A.: 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
14. Haxthausen, A.E., Peleska, J.: Formal Development and Verification of a Distributed Railway Control Systems. IEEE Transaction on Software Engineering **26**(8), 687–701 (2000)
15. 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`
16. Möller, F., Nguyen, H.N., Roggenbach, M., Schneider, S., Treharne, H.: Defining and Model Checking Abstractions of Complex Railway Models Using CSP$\|$B. In: Hardware and Software: Verification and Testing, pp. 193–208. Springer (2013)
17. de Moura, L.M., Rueß, H., Sorea, M.: Bounded model checking and induction: From refutation to verification. In: Jr., W.A.H., Somenzi, F. (eds.) 15th International Conference on Computer Aided Verification, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings. Lecture Notes in Computer Science, vol. 2725,

pp. 14–26. Springer (2003). https://doi.org/10.1007/978-3-540-45069-6_2, http://dx.doi.org/10.1007/978-3-540-45069-6_2

18. Peleska, J.: Industrial-Strength Model-Based Testing - State of the Art and Current Challenges. In: Petrenko, A.K., Schlingloff, H. (eds.) Proceedings 8th Workshop on Model-Based Testing, Rome, Italy. Electronic Proceedings in Theoretical Computer Science, vol. 111, pp. 3–28. Open Publishing Association (2013)
19. Perna, J.I., George, C.: Model Checking RAISE Applicative Specifications. In: Proceedings of the Fifth IEEE International Conference on Software Engineering and Formal Methods, 2007. pp. 257–268. IEEE Computer Society Press (2007)
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. Sheeran, M., Singh, S., Stålmarck, G.: Checking safety properties using induction and a sat-solver. In: Jr., W.A.H., Johnson, S.D. (eds.) 3rd International Conference on Formal Methods in Computer-Aided Design, FMCAD 2000, Austin, Texas, USA, November 1-3, 2000, Proceedings. Lecture Notes in Computer Science, vol. 1954, pp. 108–125. Springer (2000). https://doi.org/10.1007/3-540-40922-X_8, http://dx.doi.org/10.1007/3-540-40922-X_8
22. Verified Systems International GmbH: RT-Tester Model-Based Test Case and Test Data Generator - RTT-MBT - User Manual (2013), available on request from http://www.verified.de
23. Vu, L.H., Haxthausen, A.E., Peleska, J.: Formal modelling and verification of interlocking systems featuring sequential release. Science of Computer Programming **133, Part 2**, 91–115 (2017). https://doi.org/http://dx.doi.org/10.1016/j.scico.2016.05.010, http://www.sciencedirect.com/science/article/pii/S0167642316300570