https://doi.org/10.1007/s00165-020-00507-2
BCS © 2020
Formal Aspects of Computing

Formal Aspects of Computing

Stepwise development and model checking of a distributed interlocking system using RAISE

S. Geisler and A. E. Haxthausen

DTU Compute, Technical University of Denmark, Kongens Lyngby, Denmark.

Abstract. This paper considers the challenge of designing and verifying control protocols for geographically distributed railway interlocking systems. It describes how this challenge can be tackled by stepwise development and model checking of state transition system models in a new extension of the RAISE Specification Language.

Railway interlocking systems are reconfigurable systems which can be configured by supplying data describing the network to be controlled and other details. Therefore, such systems are natural candidates for being modelled by generic state transition systems, which abstract away from the concrete configuration at the time of modelling, and can later be instantiated with concrete data.

For a real-world case study, a generic state transition system is developed in steps, starting with an abstract model of the essential system behaviour and incrementally adding details and restrictions. The stepwise development method allows different variants of the control protocol to be explored. The generic models are instantiated with concrete configuration data, after which desired properties, in particular safety properties, of the system models are verified using model checking.

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

1. Introduction

This paper considers the challenge of formally modelling and verifying the safety of the real-world geographically distributed railway interlocking system presented in [HP00]. The engineering concept of this was originally developed by INSY GmbH Berlin for their railway control system RELIS 2000 designed for local railway networks that are not highly frequented by trains and consist of a single line connecting a series of small, adjacent stations having one or two tracks.

Correspondence to: S. Geisler, E-mail:sgei@dtu.dk

Published online: 21 February 2020

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 [HP00, FH18], the control has been *geographically distributed* to communicating 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 such distributed interlocking systems is the lower cost for track side installation and maintenance as signals and many copper cables are replaced by wireless communication. Furthermore, system configuration and thereby also the certification of system configurations is considerably simplified (and thereby also cheaper) as the configuration data of each control component along the track layout only depends on local (adjacent) track elements and the configuration data for a train control computer only depends on its own route, while the configuration of a centralised system in the form of a global route control table is very complex, cf. the discussion in [FGH+16]. This in particular means that re-configuration due to network changes only requires small local changes in a few components for a distributed system, while the whole route control table for a centralised system must be completely recreated. These advantages make distributed interlocking systems an attractive solution, especially for small, local railway networks which can only operate if the costs for (re-)configuration, installation and maintenance of the system are low.

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 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 make safe decisions, so communication between the control computers must be introduced. This adds additional threats which would not be present in a centralised system. Hence, the distribution of control gives rise to 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 [ECfES11] for safety-critical railway control components and have proved useful for many applications. For instance, Haxthausen and Peleska demonstrated this in [HP00], where they modelled and verified the distributed interlocking system considered in this paper. For this they used the RAISE Specification Language (RSL) [RAI92], and the RAISE theorem prover, respectively.

Theorem proving, as used in [HP00], 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 safety of 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 a new extension of RSL, which we call RSL⋆. We have built RSL⋆ on top of another extension of RSL, called RSL-SAL [PG07], which allows to specify systems by state transition system models. RSL⋆ provides additional convenient language constructs, allowing the use of generic variables and named collections of transition rules. In contrast to this work, the work in [HP00] 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 [SAL01] which is a back-end to the RAISE tool set *rsltc* [Geo03] supporting RSL-SAL and now also RSL⋆. 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.

Compared to the previously published work by the same authors [GH18], the main extensions to this paper are as follows:

1. The language constructs of the new RSL⋆ language are presented for the first time.
2. New functionality that the RSL tools framework has been equipped with in order to support RSL⋆ is presented.
3. Models and assertions are now specified in RSL⋆ instead of RSL-SAL.
4. Additional details of the generic models are presented.
5. A complete list of properties to be verified has been presented and examples of formal specifications of some of these are shown.
6. All example configurations which have been verified are shown and it is explained how they have been selected.
7. Performance data for model checking is shown.
8. The tools, the formal specifications and the performance results have been made available on a GitHub repository.
9. It is shown by an example how functions used in the models have been tested.
10. The related work has been expanded and the number of references has been more than doubled.

1.3. Related work

Formal verification of interlocking systems is an active research topic, investigated by several research groups. An overview of recent trends can be found in [Fan14, BtBF+18].

The formal verification method used may be *theorem proving*, such as in [CDP+17, Sab16] and [HP00] (the work from which the case study of this paper originates). The theorem proving approach is generally characterised by being once-and-for-all, such that every possible instance of the specified system has been proven to be correct with respect to some defined properties of e.g. safety.

There are also several examples of using *model checking* as the formal verification method such as in [FMGF10, VHP17, JMN+14a, Win02, BtBFL19, RFT16]. This approach has the advantage over theorem proving of being fully automated. However, model checking of the desired properties must be repeated for each system instance one wishes to verify. There are examples of using many different model checking tools such as UPPAAL ([BtBFL19]) and nuSMV ([FMGF10]) for the verification of interlocking systems. Like in this paper, in [HBK10, Hax14] RSL-SAL and SAL was used for modelling and verifying an interlocking system, but in contrast to the work of this paper, the system was not distributed, but centralised, not computer-based, but relay-based, and no stepwise development of the state transition models was used.

We develop in this paper a system model by stepwise specification, starting out with an abstract model and gradually adding details and restrictions. Such stepwise development methods are well-known from other languages, for example from TLA/TLA+, where one can formally define refinement relations between models at different levels of abstraction [Mer08]. The same idea of refinement, progressing from an abstract model to a concrete one through a number of steps is incorporated in both classical B and Event-B [Abr18]. In this paper we suggest to use two refinement techniques. We use guard strengthening, where the admissible events of the system are restricted by strengthening guards of (some of) the guarded commands. The refinement in this case is based on trace inclusion, which is well known from other languages which can be used to specify state-based systems. For example, such refinements are used in Event-B [MFTFL18, Abr18]. The other refinement technique we suggest to use is event decomposition: we decompose each of the collaborative events into multiple sub-events of the original event. This development step uses the same idea as refinement of Event-B models using decomposition of atomic events as shown in [But09]. The idea in the method from Event-B is to first model the desired outcome as an atomic event, and then decompose the atomic event into multiple sub-atomic events.

Most of the works using formal methods for verification of interlocking systems mentioned above are focusing on *centralised* interlocking with *fixed block* interlocking, where movement authority is granted based for static blocks of track sections. The system in the case study of this paper also uses fixed block interlocking, but is distributed. *Moving block* interlocking is used in CBTC (communications-based train control) systems and in ERTMS Level 3 [HBR18]. With moving block interlocking, trains are instead surrounded by so-called envelopes

which must not overlap with each other. Such systems have for instance been explored and verified by [CDP+17, CLM+19].

As current computer-based interlocking systems usually have a *centralised* design, there are only a few examples, e.g. [HP00, FH18], of formal verification of *distributed* interlocking systems. In [HP00], the same distributed system as we consider was also formally modelled and verified. For the specification of the behavioural model, the RSL [RAI92] process algebra was used rather than the RSL⋆ guarded command language that we are using, and for the verification the RAISE theorem prover was used rather than the SAL model checker. The specification of data types representing the state spaces of components, and the specification of guards and updater functions were in both papers specified in pure RSL and are almost equivalent. There are some changes in the choice of datatype representations and formulation of functions as we need to be able to translate our specifications into SML and SAL. The guards used in the models in the first two steps of our development correspond to the guards used in the fifth step in the development in [HP00], while the guards used in the sixth step of [HP00] are more restrictive, requiring that a train reserves the segments of its route one by one in the order they are to be visited according to its route (but still only as far as the train wishes). The guards used in our third model are even more restrictive than that, only allowing a train to reserve the next segment in its route.
In [FHN17], a geographically distributed railway interlocking system was formally modelled and verified using UMC [UMC, tBFGM11] instead of RSL-SAL and SAL. The control protocol presented in [FHN17] radically differs from the one considered in our case study: in [FHN17], 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.

For validation of configuration data there are two major approaches: static checking [HØ16, LJ18] and model checking [PKHP19]. In this paper we use static checking.

1.4. Paper overview

First, Sect. 2 gives a brief introduction to the new language constructs of RSL⋆ and the new functionality of the RSL tools framework. The reader may skip this section. Then, Sect. 3 gives a brief introduction to the case study: the engineering concept of the considered distributed interlocking system and an overview of the formal development. The following sections (Sects. 4, 5 and 6) give an overview of the generic model specifications and the development steps between them. The verification of model instances is described in Sect. 7. Finally, Sect. 8 gives a conclusion and states ideas for future work.

2. Language constructs of RSL⋆ and added tool support

This section gives a short introduction to RSL⋆ which is used in this paper to create and instantiate generic specifications for system models of a distributed interlocking system.

RSL⋆ is an extension to RSL-SAL, which itself is an extension to RSL. RSL is a wide-spectrum, formal specification language that enables the formulation of modular specifications in several different styles. RSL-SAL provides the possibility of specifying state transition systems in a guarded command style and later model checking properties of the systems using the SAL model checker.

We have extended the existing RSL-SAL specification language with new language constructs better suited for the specification of generic models. In particular, this extended version of RSL, which we call RSL⋆, provides the possibility of declaring *generic variables*, *generic constants*, *initialisation constraints*, and *naming collections of transition rules*. Functionality supporting the use of these constructs in conjunction with the SAL model checker have been added to the RSL tools environment. We will elaborate on each these additions in the following.

2.1. Specifications in RSL⋆

The specification of a transition system model in RSL⋆ consists of

1. Type declarations.

2. Value declarations (i.e. constants and functions).
3. Axioms.
4. (State) variable declarations.
5. An (optional) initialisation constraint on the variables.
6. A transition relation specification.

Declared types can be used in the value declarations and the variable declarations, and the declared values and variables can be used in axioms and in the initialisation constraint (if included) and in the transition relation.

2.1.1. Generic specifications in RSL⋆

Interlocking systems are reconfigurable systems which can be instantiated by specifying configuration data, e.g., for the railway network which should be controlled. Therefore, it is natural to model an interlocking system as a *generic model*, that is, a model which abstracts away from the concrete configuration and which can later be instantiated with the desired configuration.

In RSL⋆ it is possible to abstract from concrete configuration data by using abstract types

type T1

and underspecified constant values

value v : T2

which can later be concretised by instantiating the types with concrete types, *ty*:

type T1 = ty

and by instantiating the values with concrete values, *val*, either by adding *val* as in:

value v : T2 = val

or by adding an axiom:

axiom v = val

Furthermore, it is possible to abstract away from the initial values of variables by simply not specifying these. Later, during instantiation, the initial values can be added by specifying initialisation constraints.

In the following sections, some language constructs which ease the specification of generic transition system models, and which are helpful for the reader to know about, are introduced. Some of these language constructs are from the existing RSL-SAL, and some of them are new language constructs added in RSL⋆. It will be noted in the section whether the language construct introduced is from RSL-SAL or RSL⋆.

2.2. Notation for generic variables

In RSL⋆ we allow the declaration and referencing of so-called *generic* variables. The generic variables provide an elegant way of specifying that there exists a variable for each value of a given type. The declaration of a generic variable is thus a shorthand for a set of variable declarations. This is particularly useful when creating generic models of systems which may contain a set of similar entities for each of which some state variable(s) must be maintained.

Generic variables in RSL⋆ are declared by statements in the following form:

variable generic_var_name[t_1 : $\mathrm{T1}_1$, ..., t_n : $\mathrm{T1}_n$] : T2

This specifies a collection of *concrete variables*, $generic\_var\_name[t_1, \ldots t_n]$, where the collection ranges over one or more parameters t_i of finite types $T1_i$. Each member, $generic\_var\_name[t_1, \ldots t_n]$, of the collection has the same variable type, $T2$.

It is possible to refer to an instance of a generic variable anywhere where a *variable reference* may normally occur. A *reference* to an instance of a generic variable has the following form:

generic_var_name[ta_1, ..., ta_n]

where ta_i is some actual value belonging to the type $T1_i$ over which the variable is defined.

We will show an example of the declaration of and reference to a generic variable. Suppose that we are specifying a system in which there is a configurable number of trains, each identified by a unique train identifier of the type *TrainID*. For each train we wish to keep track of its position. Suppose that a position is simply given by a single track segment of type *SegmentID*. We could then declare the following generic variable:

```
variable position[t : TrainID] : SegmentID
```

This states that there exists a variable, $position[ta]$, of type *SegmentID* for each value *ta* belonging to the type *TrainID*.

Now suppose that the type *TrainID* is an enumeration variant type declared as follows:

```
type TrainID == t1 | t2 | t3
```

A reference to the specific instance of the generic variable associated with $t1$ could then be written as follows:

```
position[t1]
```

In our previously published work [GH18], where we used RSL-SAL not providing generic variables, we had to make a work-around using the *map* RSL-construct, mapping component (e.g. train) identifiers to state values (such as a train's position), as follows:

```
variable position : TrainID ~m~> SegmentID
```

Using that work-around is less intuitive, as e.g., the train positions of all trains are stored in a single variable, and not in individual variables as it is the case for a real-world distributed system such as the one considered in this paper.

The addition in RSL⋆ of generic variables offers an elegant way of specifying state variables at the generic level. Additionally, using a dedicated construct makes it clearer for any readers which variables are generic.

Using generic variables (which can be unfolded to regular variables, cf. Sect. 2.8) is an advantage when translating to model checker input languages, as many of them do not support data structures such as maps.

2.3. Notation for initialisation constraints

In RSL⋆, it is possible to initialise variables using an *initialisation constraint*. An initialisation constraint is a conjunction of basic initialisation constraints and quantified initialisation constraints. A *basic initialisation constraint* has the following form:

```
init_constraint
  v = value_expression
```

where *v* is a variable reference. (This variable reference may be a simple variable reference or a reference to an instance of a generic variable.)

A *quantified initialisation constraint* representing a conjunction over a set of constraints only differing by an index parameter *t* of finite type *T*, can be expressed as follows (here for a generic variable):

```
init_constraint
  (∀t : T • v[t] = value_expression)
```

where *v* is a generic variable defined over the type T and *t* may occur in *value_expression*. It is a shorthand for writing the conjunction of all constraints that can be obtained by substituting a value val of type T for t in the constraint, $v[t] = value\_expression$.

2.4. Notation for generic constants

RSL⋆ supports *generic constants* similar to generic variables. *Generic constants* in RSL⋆ are declared by statements in the following form:

```
value generic_const_name[t1 : T1_1, ..., t_n : T1_n] : T2
```

This specifies a collection of *concrete constants*, $generic\_const\_name[t_1, \ldots t_n]$, where the collection ranges over one or more parameters t_i of finite types $T1_i$. Each member, $generic\_const\_name[t_1, \ldots t_n]$, of the collection has the same type, $T2$.

It is possible to refer to an instance of a generic constant anywhere where a *value expression* may normally occur. A *reference* to an instance of a generic constant has the following form:

generic_const_name[ta$_1$, ..., ta$_n$]

where ta_i is some actual value belonging to the type $T1_i$ over which the constant is defined.

For instance, one can write

axiom generic_const_name[ta$_1$, ..., ta$_n$] = val

where val is some value of type $T2$.

2.5. Notation for transition rules

In RSL-SAL, and thereby in RSL⋆, transition rules are specified in a *guarded command* style. A *basic transition rule* has the following form:

value_expression /∗ *must be Boolean* ∗/
⟶
v$_1$′ = value_expression$_1$, ..., v$_n$′ = value_expression$_n$

where v_i' is the primed version of a variable reference, v_i. The rule consists of a guard and an effect (separated by ⟶), 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 (□). Suppose two rules are combined by the non-deterministic choice operator:

guard$_1$ ⟶ v$_1$′ = value_expression$_1$
□
guard$_2$ ⟶ v$_2$′ = value_expression$_2$

Then the non-deterministic choice operator allows the effect of the first rule to take place if the guard, $guard_1$ evaluates to true. Similarly, the operator allows the effect of the second rule to take place if the guard, $guard_2$ evaluates to true. If both $guard_1$ and $guard_2$ evaluate to true, then a non-deterministic or "random" choice is made between the effect of the first rule and the effect of the second rule.

A non-deterministic choice over a set of rules of the same form, only differing by a parameter t of finite type T, can be expressed as a *quantified transition rule*:

(□ t : T •
  value_expression
  ⟶
  v$_1$′ = value_expression$_1$, ..., v$_n$′ = value_expression$_n$)

where *t* may occur in the *value_expression*s and/or as an access parameter in the variable references v_i, if v_i is an instance of a generic variable.

Such a rule is a shorthand for writing a non-deterministic choice between all rules that can be obtained by substituting a value val of type T for t in the rule, $value\_expression \longrightarrow v_1' = value\_expression_1, \ldots, v_n' = value\_expression_n$.

In particular, a non-deterministic choice between updating different instances of a generic variable v defined over the type *T*, can be expressed as follows:

(□ t : T •
  value_expression
  ⟶
  v′[t] = value_expression)

2.6. Named collections of transition rules

RSL⋆ allows the naming of collections of transition rules. Named collections of transition rules provide a simple way of compartmentalising the specification into logical parts. A named collection of transition rules has the following form:

```
[Name] = (
  rule_1
  []
  ...
  []
  rule_n
)
```

where each $rule_i$ is a transition rule (or the name of another named collection of transition rules[1]). A collection of transition rules must consist of at least one (possibly quantified) rule.

The names of collections of transitions can be referred to in a system composition, which has the following form:

```
transition_rules
  C_1 [] ... [] C_n
  where
  [C_1] = ( ... ),
  ...
  [C_n] = ( ... )
end
```

where each C_i is the name of a collection of transition rules. Alternatively, it is possible for the system composition to combine a mix of names of collection of transition rules and just transition rules.

We now show an example of the declaration and usage of a named collection of transition rules. Suppose that we are specifying a system with trains, where an overall logical partition of the possible events in the system could be (1) rules related to *moving* the trains, (2) rules related to the trains making *reservations* for track sections, and (3) rules related to the *locking* of points in the tracks.

For this system, the collection of transition rules related to the moving of trains could be specified as follows (where the ellipsis indicate additional rules):

```
[Move] = (
  ([] t : TrainID •
    allowed_move(t) ⟶ position′[t] = next_position(t)
  )
  []
  ...
)
```

Suppose that the collections of transition rules related to reservation and locking, respectively, were declared as named collections in a similar manner. Then, a system composition of the named collections of transitions could be declared as follows:

```
transition_rules
  Move [] Reservation [] Locking
  where
  [Move] = ( ... ),
  [Reservation] = ( ... ),
  [Locking] = ( ... )
end
```

2.7. Notation for LTL properties

LTL properties in RSL-SAL (and RSL⋆) are logical value expressions where the temporal operators of LTL, *G*, *F* and *X*, are allowed as function symbols:

[1] Since nesting of named collections of transition rules inside other collections is allowed, cyclic references are disallowed.

- G expresses “always” or “globally”.
- F expresses “eventually” or “in the future”.
- X expresses “in the next state”.

A quantified property representing a set of properties only differing by a parameter t of finite type T, can be expressed as follows:

(∀t : T • value_expression)

where t may occur in the *value_expression*.

It is a shorthand for writing the conjunction of all properties that can be obtained by substituting a value val of type T for t in *value_expression*.

2.8. Tool support

We have added tool support to the existing RAISE *rsltc* tool [Geo03] for the extensions of RSL⋆ introduced in Sects. 2.2, 2.3, 2.4, and 2.6. We have added syntax and type checking support for the constructs, and moreover, we have added an *unfolder* to the RSL framework. The extended *rsltc* tool for the RSL⋆ is free to use, and can be found online.[2]

The unfolder takes as input a model specification in RSL⋆ and outputs a so-called *unfolded* model specification. In the unfolded model specification, the generic constructs have each been replaced with what they are shorthand for. Thus, the unfolded model specification is within the subset of RSL-SAL supported by the RSL to SAL translator (i.e. RSL-SAL). This allows us to continue using the translator, and thereby the SAL model checker as back-end.

The input model specification of the unfolder must be non-generic, i.e. it must be instantiated with concrete types and values, as these are used in the creation of the unfolded model specification.

3. Case study

3.1. Engineering concept

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).
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:

[2] https://github.com/raisetools/rslstar

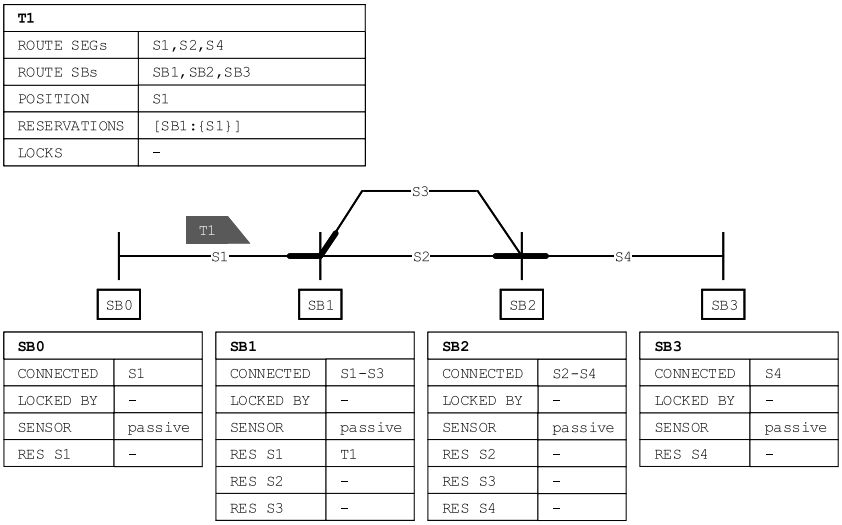

Fig. 1. An example system.

- 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 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.

3.2. 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 explicit 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, enforcing 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. 7, while the generic models will be explained in Sects. 4, 5, and 6. The instantiated models for all the configurations we consider in later sections can be found online.[3]

4. First generic model

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

- Types and values for the static configuration data.
- Types and state variables for the dynamic control component data.
- Functions for describing wellformedness of static configuration data.
- Functions for describing state invariants for dynamic control component data.
- Functions for describing the safety of the system.
- Guard and state updater functions.
- Transition system rules.

In addition there are verification obligations, which are specified as test cases and as LTL assertions, which use the functions for describing wellformedness of the static configuration data and the functions for describing state invariants for dynamic control component data and for safety. The following sections will elaborate on the different parts of the generic model and show examples of the specification of test cases and LTL assertions using the aforementioned functions. Further details on the verification of the test cases and LTL assertions can be found in Sects. 7.1.3 and 7.1.4, respectively.

4.1. Static configuration data

The static configuration data consist of data describing the railway network under consideration as well as route information for each train running in the network.

4.1.1. Network data

The data for the railway network includes information about which segments and switchboxes appear in the network and how the layout (relative placement) of these is. The data also includes information about which trains run in the network. The network layout data is included in the models solely for the purpose of being able to verify the consistency of the control component data (i.e. routes, reservations, etc.) with respect to the network layout.

[3] https://github.com/raisetools/rslstar/tree/master/spec-examples/dracos/faoc

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* and switchbox identifiers must at least include the special value *sb_none*. We specify the subtypes *TrainID′* and *SwitchboxID′* of the train identifier and switchbox identifier types, respectively, which do not contain the special values.

```
type
  SegmentID,
  TrainID == t_none | _,
  TrainID′ = {| t : TrainID • t ≠ t_none |},
  SwitchboxID == sb_none | _,
  SwitchboxID′ = {| sb : SwitchboxID • sb ≠ sb_none |}
```

The network layout describing how the segments of the system are connected is given by a value *network* of an explicitly defined type *Network*.

```
type
  Network = Connection-set,
  Connection == conn(fst : SegmentID, snd : SegmentID) | border(e : SegmentID)
value
  network : 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. The network should be represented as the set of all connections (of the form $conn(S1, S2)$) between neighbouring segments $S1$, $S2$ of the network and all border segments (of the form $border(S)$) of the network. The network is oriented according to two distinguished destinations, *DOWN* and *UP*, such that at each segment there is a uniquely defined direction to reach *DOWN* and *UP*, respectively. Driving in the direction from *DOWN* to *UP* will be referred to as the *up*-direction. The opposite direction (i.e. from *UP* to *DOWN*) is referred to as the *down*-direction. We assume that the connections of the network that have the form $conn(S1, S2)$ are all given such that they describe the connection in the up-direction.

The wellformedness requirements for the *network* value will be elaborated on in Sect. 4.3.

We must also describe how the switchboxes are placed in the network. This is given by a value *netSwitchboxes* of an explicitly defined type *SwitchboxDesc*.

```
type
  SwitchboxDesc = SwitchboxID m→ SwitchboxConnection,
  SwitchboxConnection ::
    stem : SegmentID
    branches : SegmentID-set
value
  netSwitchboxes : SwitchboxDesc
```

As for the *network* value, the *netSwitchboxes* value is not specified further in the generic model, but should be defined by a constant when the model is instantiated. This constant should map each switchbox identifier of the network to a description of what its *stem* and *branch* segments are. Recall that a switchbox is associated with a point or an endpoint. For a point, the stem segment refers to the fixed part of the point (for $SB1$ in Fig. 1 this corresponds to segment $S1$), and the branch segments refer to those segments which the point can switch between (for $SB1$ in Fig. 1 this corresponds to the segments $S2$ and $S3$). A special case is if a switchbox is placed at the joint of just two segments, i.e. there is a fixed connection between two segments (a one-one-link). In this case, there will only be a single "branch" segment, and the switchbox at that point cannot switch direction. An endpoint has only a stem, but no branch segments (for $SB0$ in Fig. 1 this means that its stem is the segment $S1$ and that its set of branch segments is empty).

The wellformedness requirements for the *netSwitchboxes* value will be elaborated on in Sect. 4.3.

4.1.2. Static control component data

In Sect. 3.1, we introduced the information which train control computers and switchboxes, respectively, must keep track of. Each control component has its own data with fields corresponding to the information shown in Fig. 1 for the train control computers and the switchboxes, respectively.

In this section we show the specification of suitable types for the static fields in the control component data and show the specification of generic constants of these types. Wellformedness requirements for the constants will be described in Sect. 4.3. The specification of suitable types for the dynamic fields in the control component data and specification of generic variables of these types will be shown in Sect. 4.2.

We have the following types for the static information stored in the train control computers:

- The *route*, which is a sequence of adjacent segments, is modelled as a value in the type, *Route*, which is a mapping from segment identifiers to segment identifiers, such that, e.g., the route ($S1$, $S2$, $S4$) (for the train $T1$ in Fig. 1) is described by the value [$S1 \mapsto S2$, $S2 \mapsto S4$]. This way of modelling the route was chosen because RSL-SAL does not support lists, which would otherwise have been the natural choice for the type.

  type Route = SegmentID $\overrightarrow{m}$ SegmentID

- The *switchboxes*, which is a sequence of adjacent switchboxes along the route, is modelled as a value in the type, *Switchboxes*, which is a mapping from switchbox identifiers to switchbox identifiers, similarly to the *Route* type. This means that the switchbox sequence ($SB1$, $SB2$, $SB3$) (for the train $T1$ in Fig. 1) is described by the value [$SB1 \mapsto SB2$, $SB2 \mapsto SB3$].

  type Switchboxes = SwitchboxID′ $\overrightarrow{m}$ SwitchboxID′

We declare static values using the types shown above. In the generic model, the values are underspecified, so they must be given values when the model is instantiated.

We use generic constants for modelling the static data of the train control computers. The generic constants have the form *value_name*[t : *TrainID*′] : *type*. Note that we are using the marked train identifier type, i.e., the subtype which does not contain the special *t_none* value.

For the train control computers, we have the following generic constants, describing the route and the switchboxes along the route, respectively:

```
value
  route[t : TrainID′] : Route,
  switchboxes[t : TrainID′] : Switchboxes,
```

4.2. Dynamic control component data

We show the specification of suitable types for the dynamic fields in the control component data and show the specification of generic variables of these types.

We have the following types for the dynamic information stored in the train control computers:

- The train *position*, which is described by one or two segments (depending on whether the train is currently crossing a boundary between two segments or not),[4] is modelled as a value in the variant type *Position* which can either be a *single* position consisting of a single segment identifier or a *double* position consisting of two segments.

  type Position == single(seg : SegmentID) | double(tlseg : SegmentID, hdseg : SegmentID)

- The *reservations* stored in a train is modelled as a value in the type *TReservation*, which is a mapping from switchbox identifiers to sets of segment identifiers, such that multiple segments can be reserved by a train at the same switchbox. For each train, the domain of this map will be the set of switchboxes at which the train currently has reservations. This means that the reservation for the train $T1$ in Fig. 1 is described by the value [$SB1 \mapsto \{S1\}$].

  type TReservation = SwitchboxID′ $\overrightarrow{m}$ SegmentID-**set**

- The *locks* stored in a train is modelled as a value in the type *TLock*, which is a set of switchbox identifiers such that if a switchbox identifier is included in the set, the train has a lock at that particular switchbox. This means that the reservation for the train $T1$ in Fig. 1 is described by the value {} (since the train has not acquired any locks yet).

  type TLock = SwitchboxID′-**set**

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

Similarly, we have types for the information stored in switchboxes:

- The associated *sensor*'s status is modelled as a value in the variant type, *SensorState*.

  type SensorState == active | passive

- The current *connection* of segments at the point of the switchbox is modelled as a value in the variant type *Connection* already shown above. The current connection for the switchbox $SB1$ in Fig. 1 is described by the value $conn(S1, S3)$.

  type Connection == conn(fst : SegmentID, snd : SegmentID) | border(e : SegmentID)

- The *reservations* stored in a switchbox is modelled as a value in the type *SbReservation*, which is a mapping from segment identifiers to train identifiers. For each switchbox, the domain of this map will be exactly the set of segments which that switchbox is connected to. The value corresponding to a segment will be the special value *t_none* when no reservation is made. This means that the reservations for the switchbox $SB1$ in Fig. 1 is described by the value $[S1 \mapsto T1, S2 \mapsto t\_none, S3 \mapsto t\_none]$.

  type SbReservation = SegmentID $\overrightarrow{m}$ TrainID

- The *locks* stored in a switchbox is not modelled as a value in any additional type, but just a value in the type *TrainID*. When no lock is present, the special value *t_none* will be used.

Several state variables are declared in the transition system model using the types shown above. 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.

For modelling the dynamic data of each control component, we use generic variables. The generic variables have the form *variable_name*[*t* : *TrainID*′] : *type* and *variable_name*[*sb* : *SwitchboxID*′] : *type*. Note that we are using the marked train and switchbox identifier types, i.e., the subtypes which do not contain the special *none* values.

For the train control computers, we have the following generic variables, describing the position, the next switchbox[5] (to be passed next or which is currently being passed), the reservations, and the locks, respectively:

variable
pos[t : TrainID′] : Position,
nextSb[t : TrainID′] : SwitchboxID,
tReservations[t : TrainID′] : TReservation,
tLocks[t : TrainID′] : TLock

Similarly, for the switchboxes, we have the following generic variables, describing the sensor value, the connection, the reservations, and the locks, respectively:

variable
sbSensor[sb : SwitchboxID′] : SensorState,
sbConnection[sb : SwitchboxID′] : Connection,
sbReservations[sb : SwitchboxID′] : SbReservation,
sbLock[s : SwitchboxID′] : TrainID

4.3. Wellformedness of static configuration data

The static configuration data values for a model instance must adhere to specific requirements to ensure that the data is *wellformed*. There may be requirements for a value itself, or the requirements may relate the value to the value of other static configuration data.

Because the requirements only deal with static data, we can use static checking (rather than model checking) to verify that the requirements hold for a model instance. The static verification is performed by expressing the wellformedness requirements as *test cases* and executing those. We will give further details on the static verification in Sect. 7.

[5] Needed for keeping track of the progression through the map of switchboxes.

Below we will informally describe the wellformedness requirements. In two cases, we will also show how they are expressed formally. For the rest, we will show the name of the test case, such that the reader can find it in the specification files. The specification of test cases is generic such that it can be re-used for different model instances.

Network wellformedness. The *network* value must be wellformed. Recall that the network is represented as a set of values of the form $conn(S1, S2)$ and $border(S1)$, where a value of the form $conn(S1, S2)$ indicates that the segments $S1$ and $S2$ are neighbours in the up-direction.

For the network it must hold that (1) the network is non-empty; (2) the network has no loops;[6] (3) each border segment must have at least one neighbour segment; (4) all neighbour segments of a border segment must be on the same side of the border segment; and (5) each non-border segment in the network must have at least one neighbour in each direction. This can be expressed by the test case:

```
test_case
  [network_wf] network_wf(network)
```

where the function *network_wf* is specified as follows:

```
network_wf : Network → Bool
network_wf(net) ≡
  net ≠ {} ∧ /* (1) */
  no_loops(conns_of_net(net)) ∧ /* (2) */
  (∀c : Connection • c ∈ net ⇒
    case c of
      border(segm) → /* (3) and (4) */
        (¬ is_fst_in_conn (segm, net) ∧ is_snd_in_conn(segm, net)) ∨
        ( is_fst_in_conn (segm, net) ∧ ¬is_snd_in_conn(segm, net )),
      conn(segm1,segm2) → /* (5) */
        (∃c2 : Connection • c2 ∈ net ∧ c2 ≠ c ∧
          segm1 ∈ segments_of_conn(c2)) ∧
        (∃c2 : Connection • c2 ∈ net ∧ c2 ≠ c ∧
          segm2 ∈ segments_of_conn(c2))
    end)
```

using several auxiliary functions: *no_loops* checks for loops in the network; *conns_of_net* extracts all connections which are not border segments from the network; *is_fst_in_conn* and *is_snd_in_conn* check whether a segment is the first, respectively second, component of some connection which is not a border segment; and *segments_of_conn* extracts the segments from a *Connection* value.

Consistency between net switchboxes and network. The *netSwitchboxes* value must be consistent with the *network* value: It must hold that (1) for each switchbox it describes, each branch segment is a neighbour to the stem in the network; and (2) for each pair of adjacent segments and each border segment in the network, there is one and only one matching switchbox in *netSwitchboxes* in terms of stem and branch segments.

In a way similar to above, this is expressed by a test case named *cons_sb_desc_net* using a function *cons_sb_desc_net* : $SwitchboxDesc \times Network \rightarrow \mathbf{Bool}$.

Consistency between train routes and network. For each train t, its train route (map) in the $route[t]$ value must represent a sequence of adjacent segments in the *network* value: as the route is represented as a map, the invariant can be checked by checking that (1) the domain of the map has exactly one more segment s_1 than the range and the range of the map has exactly one more segment s_n than the domain (such that the map represents a linear sequence of distinct segments $s_1, \ldots, s_n$, where $s_{i+1} = route[t](s_i)$ for i = 1, …, n-1) and that (2) each segment is mapped to a neighbour segment in the network. This can be expressed by the test case:

```
[cons_route_network]
  (∀t : TrainID′ • cons_route_network(route[t], network))
```

where the function *cons_route_network* is specified as follows:

[6] Anti-symmetry is a consequence of this requirement.

```
cons_route_network : Route × Network → Bool
cons_route_network(route, net) ≡
  card(dom(route) \ rng(route)) = 1 ∧ card(rng(route) \ dom(route)) = 1 /* (1) */ ∧
  (∀seg : SegmentID • seg ∈ dom(route) ⇒ are_neighbors_in_net(seg, route(seg), net) /* (2) */
```

and the auxiliary function *are_neighbors_in_net* is specified as follows:

```
are_neighbors_in_net : SegmentID × SegmentID × Network → Bool
are_neighbors_in_net(seg1, seg2, net) ≡
  conn(seg1, seg2) ∈ net ∨ conn(seg2, seg1) ∈ net,
```

Consistency between train switchboxes and network. For each train t, its switchboxes map in the *switchboxes*[t] value must represent a sequence of switchboxes which are adjacent in the *netSwitchboxes* value. This is expressed by a test case named *cons_switchboxes_netswitchboxes*.

Consistency between train switchboxes and routes. For each train t, its sequence of switchboxes, *switchboxes*[t], and its route, *route*[t], must be mutually consistent: each (key, $value$)-pair of segments in a train's route can be paired with a switchbox from the sequence of switchboxes, and vice versa. This is expressed by a test case named *cons_switchboxes_route*.

4.4. State invariants

There are several state invariants which should hold for the values of the control components' state variables in each model instance. The values in the state variables must be mutually consistent (i.e. in agreement), and the values must be consistent with the static configuration data as well. We will give further details on the verification of invariants in Sect. 7.

Below we will informally describe the state invariants we have considered. For one of the invariants, we will also show how it is expressed formally as an LTL formula. For the rest, we will show the name of the invariant, such that the reader can find it in the specification files. The specification of state invariants is generic such that it can be re-used for different model instances. The first seven invariants express internal consistency between different variables and between variables and constants of each train control computer, the next invariant expresses that the state of each train control computer must be consistent with the network data, the following two invariants express that the state of each switchbox must be consistent with the network data, and the three last invariants express mutual consistency between the states of train control computers and the states of switchboxes.

Consistency between train position and reservations. For each train, t, its reservations, *tReservations*[t], must be consistent with its position, *pos*[t], such that the train has the necessary reservations for legally being in its current position: a train in a single position should always at least have a reservation for its current segment at its next switchbox, and a train in a double position should always at least have a reservation for each of its current segments at the next switchbox (which is the one it is passing) along with a reservation for its front-most segment at the switchbox after its next switchbox. This can be expressed by the formula:

```
[cons_res_pos]
  (∀t : TrainID′ • G(cons_res_pos(tReservations[t], pos[t], nextSb[t], switchboxes[t])))
```

where *G* is the global temporal operator from LTL and the function *cons_res_pos* is specified as follows:

```
cons_res_pos : TReservation × Position × SwitchboxID × Switchboxes→ Bool
cons_res_pos(res, pos, nextsb, switchboxes) ≡
  case pos of
    single(seg) → seg ∈ res(nextsb),
    double(tlseg,hdseg) → hdseg ∈ res(nextsb) ∧ tlseg ∈ res(nextsb)
      ∧ nextsb ∈ dom(switchboxes) ∧ hdSeg ∈ res(switchboxes(nextsb))
  end
```

Consistency between train position and locks. For each train, t, its locks, *tLocks*[t], must be consistent with its position, *pos*[t], such that the train at least has the necessary lock for legally being in its current position: a train in a single position does not need to have any locks, but a train in a double position should always at least

have a lock for its next switchbox (which is the one it is passing). This is expressed in the LTL assertion named *cons_locks_pos*.

Consistency between train reservations and route. For each train, t, its reservations, *tReservations*[t], must be consistent with its route, *route*[t]: the reservations should only be for segments in the route. This is expressed in the LTL assertion: *cons_res_route*.

Consistency between train locks and switchboxes. For each train, t, its locks, *tLocks*[t], must be consistent with its sequence of switchboxes, *switchboxes*[t]: the locks should only be for switchboxes in the sequence of switchboxes, but should never contain a lock for the last switchbox in the switchbox sequence (as the train should never pass this switchbox). This is expressed in the LTL assertion named *cons_locks_switchboxes*.

Consistency between train position and route. For each train, t, its position, *pos*[t], must be consistent with its route *route*[t]: a single train position must use a segment which is in the route and a double position must use adjacent segments of the route. This is expressed in the LTL assertion named *cons_pos_route*.

Consistency between a train's next switchbox and its switchboxes. For each train, t, its next switchbox, *nextSb*[t], must be consistent with its sequence of switchboxes *switchboxes*[t]: the next switchbox must be in the train's sequence of switchboxes. This is expressed in the LTL assertion named *cons_nextsb_switchboxes*.

Consistency between a train's next switchbox and its position. For each train, t, its next switchbox, *nextSb*[t], must be consistent with its position, *pos*[t]: the next switchbox must be associated with all segments of the train's position (the *netSwitchboxes* value is used for retrieving the associated segments of the next switchbox). This is expressed in the LTL assertion named *cons_nextsb_pos*.

Consistency between train position and network. For each train, t, its position, *pos*[t], must be consistent with the *network* value: a single train position must use a segment in the network and a double position must use two segments which are adjacent in the network. This is expressed in the LTL assertion named *cons_pos_network*.

Consistency between point connection and network. For each switchbox, sb, its current recorded connection of its associated point, *sbConnection*[sb], must be consistent with the *netSwitchboxes* value: if a switchbox' current connection connects two segments, that connection of segments must be possible according to the *netSwitchboxes* value. If a switchbox' current connection does not connect two segments (i.e. is describing a border segment), the corresponding part of the *netSwitchboxes* value must have no branch segments. This is expressed in the LTL assertion named *cons_connection_netswitchboxes*.

Consistency between switchbox reservations and network. For each switchbox, sb, its reservations, *sbReservations*[sb], must be consistent with the *netSwitchboxes* value: the set of segments that are given a reservation in *sbReservations*[sb] must consist of the stem segment and any branch segments belonging to the switchbox according to the *netSwitchboxes* value. This is expressed in the LTL assertion named *cons_ reservations _netswitchboxes*.

Consistency between train reservations and switchbox reservations. For each train, t, its reservations, *tReservations*[t], must be consistent with the reservations saved in each of the switchboxes' state spaces, *sbReservations*[sb]: if a train has recorded a reservation for one or more segments at a switchbox, then that switchbox' reservations must also contain the reservation of those segments for that train. This invariant is expressed in the LTL assertion named *cons_t_sb_res*.

Similarly, the reservations saved in each of the switchboxes' state spaces must be consistent with the reservations saved in each of the trains' state spaces.[7] This is expressed in the LTL assertion named *cons_sb_t_res*.

[7] Note that this consistency should only be required to hold when the system is not in the middle of making a new reservation. This is only relevant in the second and third models where the events have been decomposed.

Consistency between train locks and switchbox locks. For each train, t, its locks, $tLocks[t]$, must be mutually consistent with the locks saved in each of the switchboxes' state spaces, $sbLock[sb]$: if a train has a lock for a switchbox, then the switchbox must be locked for that train; and if a switchbox is locked for a train, then that train has a lock for that switchbox.[8] This is expressed in the LTL assertion named *cons_locks*.

Consistency between train position and switchbox sensor. For each train, t, the sensor, $sbSensor[sb]$, of its next switchbox, $sb = nextSb[t]$, must be consistent with the train position, $pos[t]$: if a train's position is a double position, then the sensor associated with the train's next switchbox should be active. This is expressed in the LTL assertion named *cons_sensor_pos*.

4.5. Safety invariants

The safety invariants for the system are specified in a similar manner to the other state invariants. We will give further details on the verification of the safety invariants in Sect. 7.

No collision of trains. It is a requirement for safety that trains never collide. Two trains, $t1$ and $t2$, in the system, should never occupy the same segment(s): the intersection of the sets of segments in their positions, $pos[t1]$ and $pos[t2]$, should be empty. The invariant can be expressed by the formula:

```
[ no_collide ]
  (∀t1 , t2 : TrainID′ • G(no_collide(t1 , t2 , pos[t1] , pos[t2] )))
```

where the function *no_collide* is specified as follows:

```
no_collide : TrainID × TrainID × Position × Position → Bool
no_collide ( tid1 , tid2 , pos1, pos2) ≡
  tid1 ≠ tid2 ⇒ segments_of_pos(pos1) ∩ segments_of_pos(pos2) = {}
```

where the function *segments_of_pos* retrieves the segments of a *Position* value as a set.

No derailment of trains. It is a requirement for safety that trains are never derailed. For each train, t, in the system, its position, $pos[t]$, should, while passing a point (i.e. is not on a single segment), fit the position of the point: the train's position should correspond to the current connection of the switchbox it is passing. The invariant can be expressed by the formula:

```
[ no_derail ]
  (∀t : TrainID′, sb : SwitchboxID′ • G(sb = nextSb[t] ⇒ no_derail(pos[t] , sbConnection[sb])))
```

where the function *no_derail* is specified as follows:

```
no_derail : Position × Connection → Bool
no_derail (pos, c) ≡
  ¬ is_single_pos (pos) ⇒ (c = conn(hdseg(pos), tlseg (pos)) ∨ c = conn(tlseg(pos ), hdseg(pos )))
```

4.6. Guard and state updater functions

The guard and state updater functions are used when specifying 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 *t_can_reserve*, which is used to determine, from the point of view of a train, whether a reservation of a specific segment at a specific switchbox can be made. This should be the case if the train is in a single position; the train has not already made the same reservation; the switchbox is one of the switchboxes along the train's route; and the segment to be reserved is a segment in the train's route.

[8] Note that this consistency should only be required to hold when the system is not in the middle of making a new lock. This is only relevant in the second and third models where the events have been decomposed.

```
t_can_reserve : Position × Switchboxes × Route × TReservation × SegmentID × SwitchboxID → Bool
t_can_reserve(pos, switchboxes, route, res, seg, sb) ≡
  is_single_pos(pos) ∧
  ¬(sb ∈ dom(res) ∧ seg ∈ res(sb)) ∧
  (sb ∈ dom(switchboxes) ∨ sb ∈ rng(switchboxes)) ∧
  seg ∈ segments_of_route(route)
```

The parameters of the guard function are the position of the train (*pos*), the train's switchboxes (*switchboxes*), the train's route (*route*), the train's reservations (*res*), the segment to be reserved (*seg*) and the switchbox at which to make the reservation (*sb*).

Another example of a guard function is *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.

```
sb_can_reserve : SbReservation × SegmentID → Bool
sb_can_reserve(res, seg) ≡ seg ∈ dom(res) ∧ res(seg) = 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 *reserve_sb*, which updates the reservations of a switchbox:

```
reserve_sb : SbReservation × TrainID × SegmentID → SbReservation
reserve_sb(res, t, seg) ≡ res † [seg ↦ t]
```

where † is the override operator, overriding the first map (*res*) with the second map ($[seg \mapsto t]$), when the domain value matches *seg*. 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 (t) for which the segment (*seg*) should be reserved.

There is another updater function, *reserve_t*, which updates the reservations of a train:

```
reserve_t : TReservation × SwitchboxID × SegmentID → TReservation
reserve_t(res, sb, seg) ≡
  if sb ∈ dom(res) then
    res † [sb ↦ add_to_segset(res(sb), seg)]
  else
    res † [sb ↦ {seg}]
  end
```

where *add_to_segset(set, elem)* is an auxiliary function which adds a segment *elem* to the set of segments *set*. The parameters for the updater function consist of the data component, i.e. the reservations of the train (*res*), to which changes should be made, and the data necessary for the change, i.e. the switchbox (*sb*) at which the segment (*seg*) should be reserved.

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

4.7. Transition system rules

The rules of the transition system define the possible events (state transitions) of the system.

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

```
([] sb : SwitchboxID′, t : TrainID′, ... •
  [rule_name]
  t_can_e(...) ∧ sb_can_e(...)
  ⟶
  tData′[t] = t_new_value(...),
  sbData′[t] = sb_new_value(...))
```

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); $t\_new\_value(\ldots)$ and $sb\_new\_value(\ldots)$ are place-holders for functions calculating the new values for the variables.

The transition rules are divided into three collections of transition rules: *Move*, *Reserve* and *Lock*. This means that the system composition looks as follows:

```
Reserve [] Lock [] Move
```

The *Reserve* collection contains a single rule for reserving a segment seg for a train t at a switchbox sb.

```
[Reserve] =
(([] sb : SwitchboxID′, t : TrainID′, seg : SegmentID •
  [reserve]
   t_can_reserve(pos[t], switchboxes[t], route[t], tReservations[t], seg, sb) ∧
   sb_can_reserve(sbReservations[sb], seg)
  ⟶
  tReservations′[t] = reserve_t(tReservations[t], sb, seg),
  sbReservations′[sb] = reserve_sb(sbReservations[sb], t, seg)))
```

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.

The *Lock* collection contains a single rule for switching and locking a point associated with a switchbox sb in a connection con for a train t.

```
[Lock] =
(([] sb : SwitchboxID′, t : TrainID′, con : Connection •
  [lock]
  t_can_lock(pos[t], con, route[t], switchboxes[t], tReservations[t], sb, tLocks[t]) ∧
  sb_can_lock(con, sbLock[sb], sbSensor[sb])
  ⟶
  sbConnection′[sb] = con,
  tLocks′[t] = lock_t(tLocks[t], sb),
  sbLock′[sb] = t))
```

The rule uses two guard functions to determine whether the point can be switched to the desired connection and subsequently locked in this position. As for reservations, the switchbox and train must agree on the legality of the event. The effect of the rule uses two updater functions to update the locks of the train and switchbox, respectively. In addition, the current connection of the switchbox (*sbConnection*) is updated to the desired connection.

The *Move* collection contains two rules: one for moving a train when it is in a single position and one for moving the train when it is in a double position. Move is not a collaborative event, so it does not follow the same pattern as the rules for reserving and locking.

```
[Move] =
(([] t : TrainID′, sb : SwitchboxID′ •
  [move_single_to_double]
   is_single_pos (pos[t]) ∧
  sb = nextSb[t] ∧
  t_can_move(seg(pos[t]), nextSb[t], route[t], switchboxes[t], tLocks[t], tReservations[t])
  ⟶
  pos′[t] = move_pos_single(pos[t], route[t]),
  sbSensor′[sb] = active)

[]

([] t : TrainID′, sb : SwitchboxID′ •
  [move_double_to_single]
  ¬ is_single_pos (pos[t]) ∧
  sb = nextSb[t]
  ⟶
  pos′[t] = move_pos_double(pos[t]),
  nextSb′[t] = move_nextsb(sb, switchboxes[t]),
  tReservations′[t] = move_t_reservations(sb, tReservations[t]),
  tLocks′[t] = move_t_locks(sb, tLocks[t]),
  sbSensor′[sb] = passive,
  sbReservations′[sb] = move_sb_reservations(pos[t], sbReservations[sb]),
  sbLock′[sb] = t_none))
```

In the first rule, *move_single_to_double*, we use a guard function to determine whether the train is allowed to move. Note that there is only a guard function for the train, as movement legality is determined solely from a train's local state space. It is also checked that the switchbox from the quantification (*sb*) is equal to the train's next switchbox, as the effects of the rule should only take place in this case. If the guard holds, we can make the updates to the train's position and to the sensor, which will become *active* (i.e., detecting something).

In the second rule we do not call any guard function, as a train will always have a correct reservation when it has been allowed to move partly onto the next segment (corresponding to the front segment of the train). The guard for the rule has the same check for the train's next switchbox as in the guard for the first rule.

In the effects of the rule, several variables are updated, changing the trains position and next switchbox, making the sensor *passive* (i.e., not detecting anything), and releasing reservations and locks in the train state space and the switchbox state space (cf. the explanation of the Engineering Concept in Sect. 3.1).

Note that there is an indirect communication from the train to the switchbox via the sensor associated with the switchbox: When a train passes a sensor, the sensor changes state, and as there is a wired connection between the sensor and its associated switchbox, a change in the sensor state is considered as being registered immediately in the switchbox' sensor variable. And immediately when the switchbox' sensor becomes passive (i.e. in the moment when a train has fully passed the sensor), the switchbox' reservation and lock variables are updated. Hence, the two *move* rules above include updates of the $sbSensor[sb]$ variable, and the second rule additionally includes updates of the $sbReservations[sb]$ variable and the $sbLock[sb]$ variable.

5. Second generic 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 (reservation and locking) 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 contrast to the reservation and locking events, the move event does not include an explicit communication scheme between the control components, where requests and acknowledgements are sent from and to the control components. As previously explained, a train's movement indirectly affects a switchbox' sensor variable, which in turn affects the switchbox' reservations and locks. The change of the sensor variable is viewed as instant since there is a wired connection between the physical sensor and the switchbox. After the change to the sensor variable, the variables for the switchbox' reservations and lock are updated almost immediately (within one hundred milliseconds which is at a significantly higher speed than the train's velocity). It would be possible to decompose the second move rule, *move_double_to_single* into two separate rules (representing two consecutive sub-events): rule (1) which only updates the sensor variable and the train's variables, and rule (2) which, based on the change in the sensor variable, updates the reservations and lock variables of the switchbox. However, since the second event happens so quickly, we consider it as happening instantly and have therefore chosen not to decompose the *move_double_to_single* rule.

In the communication protocols for the collaborative events, 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:

- *train _request _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.
- *switchbox_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.
- *train _e_ack*, 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.
- *switchbox_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.
- *train _e_nack* 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:

$$\begin{array}{l}\mathrm{req}[\mathrm{t} : \mathrm{TrainID}'] : \mathrm{SwitchboxID},\\ \mathrm{ack}[\mathrm{sb} : \mathrm{SwitchboxID}'] : \mathrm{TrainID},\\ \mathrm{nack}[\mathrm{sb} : \mathrm{SwitchboxID}'] : \mathrm{TrainID}\end{array}$$

For instance, $ack[sb] = t$ models a *positive acknowledgement* from a switchbox *sb* to a train *t*. $ack[sb] = t\_none$ models that no positive acknowledgement has been sent 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[9] is used to store the segment to be reserved:

$$\mathrm{reqSeg} : \mathrm{SegmentID}$$

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:[10]

$$\mathrm{reserveEvent} : \mathbf{Bool}$$

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 *Reserve* rules for requesting, acknowledging and concluding the *reservation* event:

$$\begin{array}{l}\mathrm{Reserve} =\\ (([]\ \mathrm{sb} : \mathrm{SwitchboxID}', \mathrm{t} : \mathrm{TrainID}', \mathrm{seg} : \mathrm{SegmentID} \bullet\\ \quad [\ \mathrm{train\_request\_reservation}\ ]\\ \quad \neg \mathrm{reserveEvent} \wedge \neg \mathrm{switchLockEvent} \wedge /{*}\ \mathit{idle}\ {*}/\\ \quad \mathrm{req}[\mathrm{t}] = \mathrm{sb\_none} \wedge\\ \quad \mathrm{t\_can\_reserve}\,(\mathrm{pos}[\mathrm{t}],\ \mathrm{switchboxes}[\mathrm{t}],\ \mathrm{route}[\mathrm{t}],\ \mathrm{tReservations}[\mathrm{t}],\ \mathrm{seg},\ \mathrm{sb})\\ \quad \longrightarrow\\ \quad \mathrm{req}'[\mathrm{t}] = \mathrm{sb},\\ \quad \mathrm{reqSeg}' = \mathrm{seg},\\ \quad \mathrm{reserveEvent}' = \mathbf{true}\\ )\end{array}$$

[9] Since only one event should be allowed at the same time in this model, it is sufficient to store a segment rather than having a generic variable over the train identifier type with type *SegmentID*, where for each train t, $reqSeg[t]$ could hold data sent by t.

[10] For this variable there is a similar comment as for *reqSeg*.

```
[]

([] sb : SwitchboxID′, t : TrainID′ •
  [switchbox_ack_reservation]
  reserveEvent ∧
  req[t] = sb ∧
  sb_can_reserve(sbReservations[sb], reqSeg)
  ⟶
  ack′[sb] = t,
  req′[t] = sb_none,
  sbReservations′[sb] = reserve_sb(sbReservations[sb], t, reqSeg)
)

[]

([] sb : SwitchboxID′, t : TrainID′ •
  [ train_reserve_ack ]
  reserveEvent ∧
  ack[sb] = t
  ⟶
  tReservations′[t] = reserve_t(tReservations[t], sb, reqSeg),
  ack′[sb] = t_none,
  reserveEvent′ = false
)

[] ...

[] ...

)
```

The *train_request_reservation* rule can be applied when the system is idle, i.e. when no events are ongoing,[11] when the train control computer has not already sent a request and when the reservation is legal from the train control computer's point of view. As its effect, the rule sets the request variable for the train identifier and switchbox identifier in question, sets a data variable to the segment to be reserved and enables the *reservation event variable*.

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

Finally, the *train_reserve_ack* rule can be applied when the *reservation event variable* is enabled and a positive acknowledgement has been received. 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.

For the *lock* event there is a similar set of rules for the communication between the train control computers and the switchboxes.

As the *move* event does not include explicit communication between the control components and should not be decomposed, as explained in the start of the section, the rules for this event are identical to those of the first model.

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.

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

6. Third generic 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

t_can_e_restricted : ... → **Bool**
t_can_e_restricted (...) ≡
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 *t_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 $segm$ at a switchbox sb, if (1) sb is one of the two upcoming switchboxes of the route of t and (2) the segment $segm$ is the next segment with respect to the train's position and route.

Hence, the restricted guard function is specified as follows:

t_can_reserve_restricted : Position × SwitchboxID × Switchboxes × Route × TReservation × SegmentID ×
SwitchboxID → **Bool**
t_can_reserve_restricted (pos, nextsb, switchboxes, route, res, segm, sb) ≡
is_single_pos (pos) ∧
¬(sb ∈ **dom**(res) ∧ segm ∈ res(sb)) ∧
(nextsb ∈ **dom**(switchboxes) ∧ (sb = nextsb ∨ sb = switchboxes(nextsb))) ∧ /* *new restriction (1)* */
seg(pos) ∈ **dom**(route) ∧ segm = route(seg(pos)) /* *new restriction (2)* */

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

The transition rule for *train_request_reservation* is obtained from the second model by replacing *t_can_reserve*(*pos*[*t*], *switchboxes*[*t*], *route*[*t*], *tReservations*[*t*], *seg, sb*) with *t_can_reserve_restricted*(*pos*[*t*], *nextSb*[*t*], *switchboxes*[*t*], *route*[*t*], *tReservations*[*t*], *seg, sb*).

Likewise, the transition rule for *train_request_lock* is obtained by replacing *t_can_lock*(*pos*[*t*], *c, route*[*t*], *switchboxes*[*t*], *tReservations*[*t*], *sb, tLocks*[*t*]) with *t_can_lock_restricted*(*pos*[*t*], *nextSb*[*t*], *c, route*[*t*], *switchboxes*[*t*], *tReservations*[*t*], *sb, tLocks*[*t*]).

Relation to second model.

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

7. Verification

This section describes our verification method, consisting of (1) the verification activities performed during the stepwise development of a generic model and (2) the verification activities performed after the generic model was fully developed.

7.1. Verification activities during development

A part of the stepwise development method for the generic models is to verify the models at each step. The main goal of this verification is to gain *confidence* in the correctness of the generic models. The models must be instantiated to allow for verification. The steps for verifying a model at one of the development steps are as follows:

1. Instantiate the model with configuration data.
2. Verify wellformedness of the configuration data using static verification.[12]
3. Verify properties, including the safety invariants, of the model instance using model checking.

In Sect. 7.1.2, we show how a generic model can be instantiated with concrete data. Then, in Sect. 7.1.3, we give details on the verification of the wellformedness of the configuration data using the test cases which were introduced in Sect. 4.3. In Sect. 7.1.4, we give details on the verification of properties using model checking.

Since the intention is that the final generic model can be instantiated with different configurations which are not known at the time of specification, during the development we perform the verification on instantiations of fragments of typical configurations where wrong behaviour of the control system could lead to unsafe states.

7.1.1. Reasons for verifying at each development step

It should be noted 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.

Therefore, we perform the verification of properties for each instance of the model at each development step in order to gain confidence in the correctness of the generic models. 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. 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 check the properties for the model instance of the third model.

Performing the verification of properties of the model at each development step has the advantage that the first generic model is much less complex than the second and third generic model. Therefore, it is beneficial to verify invariants already at this first development step, because the model checking process is quicker, which means that we can discover possible errors already at this early point of the development.

7.1.2. Example of instantiation

In Fig. 2 we illustrate a small configuration, which corresponds to a fragment of a typical configuration where two trains driving in opposite directions must pass through the same station. The two trains are shown in their initial position and the lines show their routes (the striped train's route is shown as a dashed line, whereas the black train's route is shown as a solid line).

[12] Note that for each set of configuration data, this step needs only to be done once.

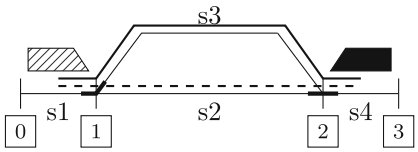

Fig. 2. An example system configuration with two trains and their routes.

The initial connections of the two points are shown by thick lines. We will now show how to instantiate the generic models with data corresponding to this fragment. Instantiations of the three generic models with the configuration data for the configuration shown in Fig. 2 are available online.[13]

The network configuration data for this instance is as follows:

```
type
  SegmentID == s1 | s2 | s3 | s4,
  SwitchboxID == sb0 | sb1 | sb2 | sb3 | sb_none,
  TrainID == t1 | t2 | t_none,
value
  network : Network =
    {conn(s1, s2), conn(s1, s3), conn(s2, s4), conn(s3, s4), border(s1), border(s4)},
  netSwitchboxes : SwitchboxDesc =
    [sb0 ↦ mk_SwitchboxConnection(s1, {}),
     sb1 ↦ mk_SwitchboxConnection(s1, {s2, s3}),
     sb2 ↦ mk_SwitchboxConnection(s4, {s2, s3}),
     sb3 ↦ mk_SwitchboxConnection(s4, {})]
```

The static control component configuration data is specified by axioms defining the value of each instance of each of the generic constants declared in Sect. 4.1.2. For example, the route constants for the two trains are defined as follows:

```
axiom
  route[t1] = [s1↦s2, s2↦s4],
  route[t2] = [s4↦s3, s3↦s1]
```

Furthermore, instances of the generic variables declared in Sect. 4.2 are initialised with their initial values. For example, the positions for the two trains are initialised as follows:

```
init_constraint
  pos[t1] = single(s1) ∧
  pos[t2] = single(s4)
```

For some of the generic variables, all instances must be initialised to the same value. For example, since both trains start in a single position, all sensors' initial status are *passive*. This can be specified by using a quantified constraint as follows:

```
init_constraint
  (∀sb : SwitchboxID′ • sbSensor[sb] = passive)
```

7.1.3. Configuration data checking

In Sect. 4.3, we introduced wellformedness and consistency requirements for the network configuration data and static data of the train control computers. Because the data is static, the requirements on the data can be statically verified.[14]

Static verification is carried out by specifying (generic) test cases once-and-for-all as explained in Sect. 4.3, and then for each configuration example unfolding and translating the test cases to SML, and running SML to get the results of each test case.

[13] https://github.com/raisetools/rslstar/tree/master/spec-examples/dracos/faoc/station-opposite-dir

[14] It would be possible also to statically verify that the initial state of the model instances satisfy the state invariants in Sect. 4.4. However, this is not strictly necessary as the state invariants will be checked to hold for the initial state as part of the subsequent model checking step.

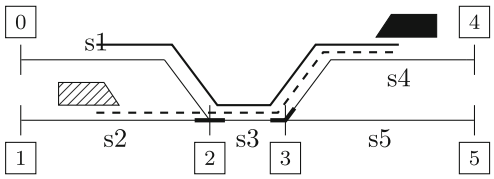

Fig. 3. An example system configuration with two trains with conflicting routes for some schedules.

Running SML on the unfolded and translated test cases results in a truth-value for each test case describing whether the specified property holds for the given configuration data. For example, assuming that the *network* value is indeed wellformed, the result of the test *network_wf* (shown in Sect. 4.3) would be shown as follows:

[network_wf] **true**

7.1.4. Model checking

In Sects. 4.4 and 4.5 we introduced (generic) consistency invariants and safety invariants, respectively, for the dynamic control component data. In addition to these invariants, we have also expressed (generic) progress properties that will be explained below. For a model instance, these invariant properties and progress properties can be verified by model checking. For model checking we use the SAL symbolic model checking tool [SAL01].

Safety properties. The safety properties state the absence of derailments and collisions of trains in all reachable states. (See Sect. 4.5.)

Consistency properties. The consistency properties state the consistency of the dynamic distributed data. (See Sect. 4.4.)

Progress properties. The progress properties state that the system progresses, i.e. that something good eventually happens.

We have expressed properties (only relevant for the second and third model) stating that decomposed collaborative events are always completed. For example, the fact that the reservation event is always completed is stated as follows:

[finish_reserve] G(resEvent ⇒ F(¬resEvent))

Note that model checking results for properties of the above form are only sound if there are no deadlocks, so before checking these properties, the SAL deadlock checker should be run.

We also consider the progress property that there is at least one possible schedule where all trains reach their destination. This property can be verified by contradiction: by model checking properties stating that the trains do *not* all eventually arrive at their destination:

[not_all_trains_arrive]
G(¬(∀t : TrainID′ • t_is_at_end (route[t], pos[t])))

where $t\_is\_at\_end(route[t], pos[t])$ expresses that the train's position is a single position with the segment corresponding to the last segment in its route. This property is expected to be false for *admissible* configurations and should generate a counter example showing a trace where all trains arrive at their destination.

This property is interesting as it is possible to create model instances where any system run will lead to a livelock preventing some trains from reaching their destination, e.g. for the very simple configuration example shown in Fig. 4.[15] For such a configuration, it would not be possible to create any schedule, as there are no system runs where both trains are able to reach their respective destinations. Such cases cannot be found by the deadlock checker (as there is no deadlock), but can be found by checking the *not_all_trains_arrive* property. If this produces a counterexample, then there does indeed exist a possible schedule where all the trains arrive as desired. With no counterexample, however, there does not exist any trace allowing all the trains to reach their destination, i.e. there are no possible schedules.

[15] For instance, for the model instances of the second and third generic models for that configuration, there is no deadlock, but a livelock as both trains are allowed to send reservation requests to their next switchbox for reserving their next segment. Such requests will subsequently be denied, i.e. the trains will receive negative acknowledgements since the reservations are not possible as there is already a train occupying (and thereby having a reservation for) the next segment.

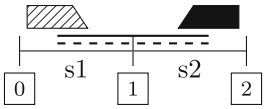

Fig. 4. An example system configuration with no possible schedule allowing the trains to arrive at their destination.

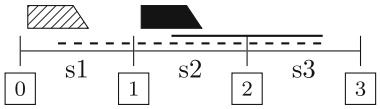

Fig. 5. Another example system configuration with no possible schedule allowing the trains to arrive at their destination.

Even if a configuration is admissible, one should not expect that all trains arrive at their destination in any possible trace. This is for instance the case for the configurations shown in Figs. 2 and 3.[16] The reason is that the control algorithms specified in the models do not ensure *fairness* for the trains (in the sense that every train will repeatedly, until it reaches its destination, receive permission to move onto the next segment after a bounded number of tries of making the needed reservations and locks). Ensuring such fairness is instead a task for the train scheduler.[17] During the train scheduling, it is checked that the timetables constructed by the scheduler are feasible, allowing trains to reach their destination without blocking each other's possibility to advance as in the livelock scenario described above, cf. [HH19].

7.1.5. Choice of examples

In this section we will present the example configurations for which we during the model development have performed the verification activities presented in Sects. 7.1.3 and 7.1.4 in order to gain confidence in the correctness of the developed models.

The network layouts and train routes of the examples were primarily chosen such that they include critical cases for which there is a potential risk of train collisions or derailments. More specifically, the network layouts and train routes were chosen such that they include (1) cases where two trains have overlapping routes such that there is a potential risk of a collision between the two trains on a common track segment, (2) cases where a train should pass a point which is in a wrong position so there is a risk of derailment (if it is not able to switch and lock the point in correct position before it moves over the point) and (3) cases where two trains require the same point to be switched in different positions to pass the point such that there is a potential risk of a derailment. In the railway domain, it is generally accepted that it is sufficient to consider two trains to check for collisions as there is no possibility of a three train collision without a two-train collision first, and it is enough to consider one train passing a point to check for derailments over that point cf. [Fan12] and [JMN+14b]. However, we also check for derailments when there are two trains requiring conflicting point positions to be sure that a train lock made by one train is not overruled by another train.

We only considered configurations where trains driving in opposite directions are using distinct track sections through the stations, i.e. where trains driving in the up-direction at all stations use the tracks on one side and trains driving in the down-direction use the tracks on the other side, because these are the characteristics of the real-world systems for which the final generic model should later be configured.

Risk of collision. For a collision to take place, two trains must enter the same segment(s) in the network. This can only happen if the two trains were previously on adjacent segments. First we consider two cases where the adjacent segments have a fixed connection (a one-one-link) and then four cases where there is a switchable connection (a point) between the adjacent segments: Figs. 4, 5, 6, 7, 8 and 9, respectively, show different cases where there is a potential risk of collision between the two trains if they are allowed to move to their next segments (in their route) as follows:

1. For the case shown in Fig. 4: if any of the two trains begin entering their next segment.

[16] For instance, for the configuration shown in Fig. 3, there is a system run where the striped train first enters segment $s3$, thus preventing the black train from progressing further. This, in turn, prevents the striped train from progressing further, but each train can still send requests (and receive negative acknowledgements for these requests).

[17] Note that in the railway domain it is standard to keep the responsibility for safety and liveness distinct. The former is ensured by the interlocking system and the latter by a train scheduler.

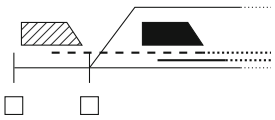

Fig. 6. Two trains driving in the same direction where the striped train may collide with the black train on the station.

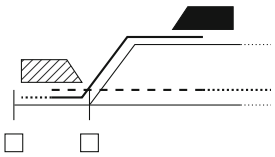

Fig. 7. Two trains driving in opposite directions where the striped train may collide with the black train on the open line.

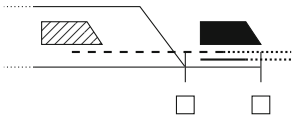

Fig. 8. Two trains driving in the same direction where the striped may collide with the black train on the open line.

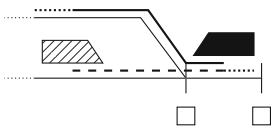

Fig. 9. Two trains driving in opposite directions where the striped train may collide with the black train on the open line.

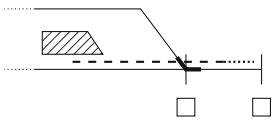

Fig. 10. A single train which may drive over a point which is not in the correct position.

2. For the case shown in Fig. 5: if the striped train begins entering the next segment before the black train has fully left it.
3. For the case shown in Fig. 6: if the striped train begins entering a station where a train driving in the same direction already resides.
4. For the case shown in Fig. 7: if the black train begins entering the open line before the striped train has fully left it.
5. For the case shown in Fig. 8: if the striped train begins entering the open line where a train driving in the same direction already resides.
6. For the case shown in Fig. 9: if the striped train begins entering the open line where a train driving in the opposite direction already resides.

Risk of derailment. For a derailment to take place in a one-train scenario a train must drive over a point which is switched in the wrong direction, or a point must be switched while the train is passing that point. The first kind of derailment can only happen if the train was previously driving towards a point on one of the point's branching segments and the point was switched to connect the stem with the other branch segment, as illustrated in Fig. 10. The second kind of derailment can only happen if the train was previously passing a point in correct position, as illustrated in Fig. 11.

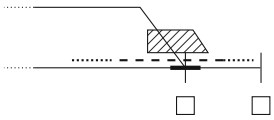

Fig. 11. A single train passing a point.

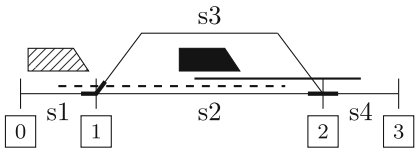

Fig. 12. Same direction, following each other through a station.

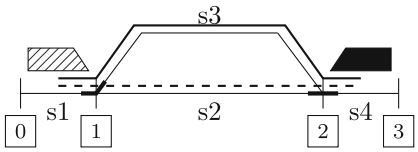

Fig. 13. Opposite directions, passing each other at a station.

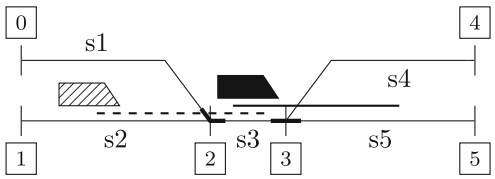

Fig. 14. Same direction, following each other on the open line.

Risk of not reaching destination. The two previously considered (undesirable/non-admissible) configurations in Figs. 4 and 5 also constitute examples where not all trains are able to reach their final destination. For instances of these configurations, it is expected that the *not_all_trains_arrive* property should not to give rise to a counter example.

Selected configurations. With these critical cases in mind, we have chosen the configurations illustrated in Figs. 4, 5, 12, 13, 14, 15 and 16. Here the configurations illustrated in Figs. 12 and 14 obviously cover the cases shown in Figs. 6 and 8, respectively, while the configurations illustrated in Figs. 13 and 15 cover the cases shown in Figs. 7 and 9, respectively, as the associated model instances include some system runs that reach the two desired configurations, respectively.[18] Lastly, Fig. 16 covers the cases shown in Figs. 10 and 11. It should be noted that the configuration in Fig. 16 can only be used to check for freedom of derailments of a train over a point which is not affected by other trains, while the configurations in Figs. 13 and 15 can be used to check for freedom of derailments in the case where two trains need the same point in different positions.

While the selected configuration examples include the critical cases that we wish to investigate, Fig. 13 in particular also includes the very common situation of two trains driving in opposite directions passing each other at a station. Similarly, Fig. 15 includes the situation of two trains driving in opposite directions "competing" to pass an open line between two stations. For these realistic configurations it is interesting not only to check for safety, but also to check that there is a schedule where the trains can pass each other and reach their destination.

[18] It is possible to use the SAL simulator tool to step through the model instances and see that one can reach the desired configurations.

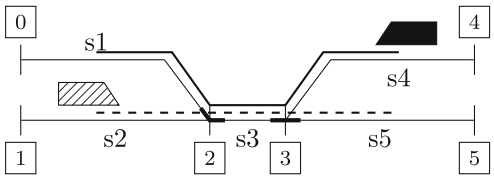

Fig. 15. Opposite directions, "competing" for the open line.

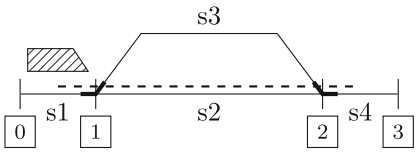

Fig. 16. One train passing a station where both points are switched in the wrong direction.

Figure 16 also includes a common situation where a train must pass a station with no other trains near it. For this configuration it is similarly interesting to check that the train is able to reach its destination.

Model instances for the seven selected configurations can be found online.[19]

7.1.6. Verification results

We have successfully statically verified all the desired wellformedness and consistency requirements (as explained in Sect. 7.1.3) for each of the seven configuration examples selected in Sect. 7.1.5 and model checked all the desired state invariants and progress properties for the corresponding model instances (as explained in Sect. 7.1.4). In particular, the property *not_all_trains_arrive* gave for the five configurations in Figs. 12, 13, 14, 15 and 16 as desired, rise to a counter example demonstrating that there exists at least one schedule, where all trains arrive at their destination, while it, as expected, did not give rise to a counter example for the configurations in Figs. 4 and 5.

Below we present verification metrics for the model instances (of each the three generic models) for each of the two configurations shown in Figs. 13 and 15, while the verification metrics for the remaining examples we have considered can be found online.[20] These two configurations were chosen to be presented in the paper as they are representative and were those that took the most time and memory to model check. One of the reasons for the latter is the fact that the train routes in these are longer than the train routes in the other configurations.

Table 1 shows metrics for the size of the configuration data: the number of segments, points, switchboxes, and trains in the network, as well as the route lengths of the trains. These metrics determine, as shown in Table 2, how many variables and transition rules the unfolded model instances have. For each of the configurations, the number of variables and number transition rules in the corresponding instance of the first model differ from those of the corresponding instances of the second and third models because of the extra variables introduced due to the decomposition of events. The number of variables might not appear to be very high, but recall that many of the variables have complex types (e.g. map types) allowing for quite many possible values.

Table 3 shows for each of the selected model instances, intervals for the time and memory consumption for model checking any of the properties with the SAL model checker, version 3.3. The results were measured using GNU Time[21] on a machine with a Intel(R) Core(TM) i7-8650U CPU @ 1.90 GHz and 31GiB of memory. We present the time and memory usage as an interval for proving any property rather than presenting the exact numbers for each property separately as these numbers were very similar.[22] The interested reader can find all the raw data online[23].

[19] https://github.com/raisetools/rslstar/tree/master/spec-examples/dracos/faoc
[20] https://github.com/raisetools/rslstar/tree/master/spec-examples/dracos/faoc
[21] https://www.gnu.org/software/time/
[22] Except for the properties *cons_t_sb_res* and *cons_sb_t_res* which were in some cases divided into two or four parts, in order to have a similar time for verification as the other properties.
[23] https://github.com/raisetools/rslstar/tree/master/spec-examples/dracos/faoc

Table 1. Configuration data size.

Configuration	Segments	Points	Switchboxes	Trains	Route lengths
Figure 13	4	2	4	2	3
Figure 15	5	2	6	2	3

Table 2. Number of variables and transition rules in the unfolded model instances.

Configuration	Model	Variables	Transition rules
Figure 13	Model 1	24	178
	Model 2/3	38	242
Figure 15	Model 1	32	386
	Model 2/3	50	482

Table 3. Time and memory usage for verifying properties. Given as intervals since the numbers are similar for each property.

Configuration	Model	Time (mm:ss)	Memory (MB)
Figure 13	Model 1	[00:02; 00:02]	[94; 100]
	Model 2	[00:12; 00:15]	[133; 143]
	Model 3	[00:06; 00:09]	[103; 337]
Figure 15	Model 1	[00:19; 00:21]	[133; 137]
	Model 2	[05:44; 06:16]	[177; 230]
	Model 3	[08:51; 10:28]	[182; 207]

The bar charts in Fig. 17 show the time taken to model check the *no_collide* property for the model instances of the first, second, and third model ($M1$, $M2$, and $M3$, respectively) for each of the two selected configurations. It is a general pattern for all the instances we have verified that it takes less time to verify properties of instances of the first model than of instances of the other models (for the same configuration). This is not surprising, as the instances of the first model have both fewer variables and fewer transition rules. There is no general pattern for whether the instances of the third model take less time to verify than the instances of the second model (for the same configuration)—for some configurations it takes less time and for others it takes a longer time, as can be seen in the two bar-charts of Fig. 17.

The bar charts in Fig. 18 show the memory consumption during the model checking of the *no_collide* property for the model instances of the first, second and third model for each of the two selected configurations. Similar to the time taken to verify the property, it is a general pattern that the verification of instances of the first model consume less memory than instances of the second and third model. Comparing the memory consumption for verifying instances of the second model and instances of the third model does not give rise to a general pattern. For the instances of the configurations shown in Figs. 13 and 15, the third model requires less memory in both cases, but for other instances (e.g. for the configurations shown in Figs. 12 and 14) the second model requires less memory.

7.2. Verification of real-world systems

After the final (third) generic model has been developed, it can be instantiated with configuration data representing real-world railway systems to be verified.

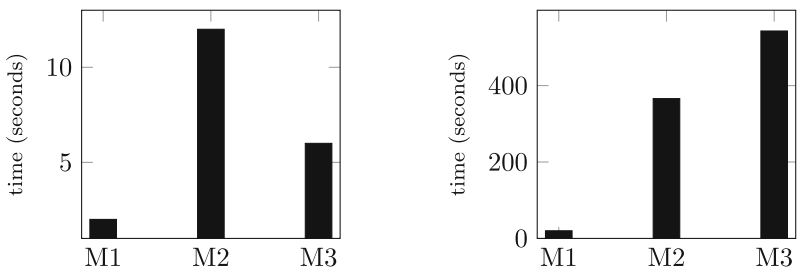

Fig. 17. Time usage for verifying *no_collide* for the three model instances for the configurations shown in Figs. 13 and 15, respectively.

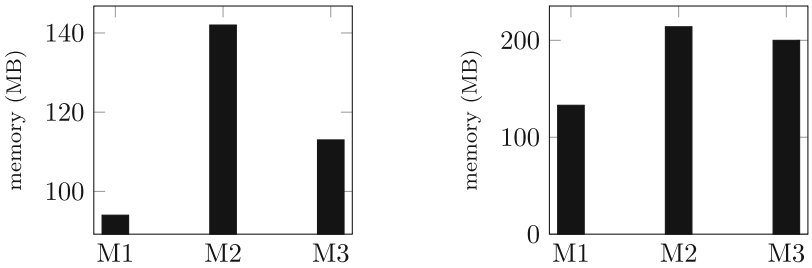

Fig. 18. Memory usage for verifying *no_collide* for the three model instances of the configurations shown in Figs. 13 and 15, respectively.

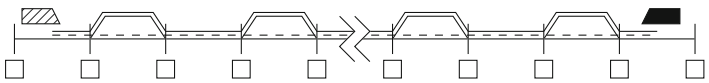

Fig. 19. An example of a typical local railway network.

Safety verification of such instances can be done in a similar manner to the verification done during the development. First, the wellformedness of the static configuration data should be statically verified. Then, ideally, the safety properties should be verified by performing model checking on the associated model instance. However, instantiating the third generic model with data corresponding to a real-world system will often lead to state explosion problems, since adding stations and trains to the line may result in exponential growth of the state space to be investigated by the model checker, even though the network topology is quite trivial. In this case, a possibility may be to use a combination of model checking of smaller fragments of a system and compositional reasoning [FHM17, LCPT16].

The RELIS 2000 system was intended for local railways with 10–20 stations with 1–2 track segments each connected by single lines and operated by 2–3 trains. Instantiating the third generic model with data corresponding to systems of this size resulted in memory exhaustion. Therefore, as will be explained below, we model checked smaller fragments common to all configurations of the considered class of local railways and used compositional reasoning for arguing for the safety of system instances (of the third model) for such local railways. As can be seen from the case analysis below, compositional reasoning is quite simple for the type of railway network layouts considered in this article.

7.2.1. Verification of two-train configurations

For the considered local railways, a typical initial state is the one illustrated in Fig. 19, where two trains are driving in opposite directions from either end of a network with routes that use distinct track sections through the stations (such that they can pass there). We will now argue by case analysis for the safety of systems having these characteristics.

As the third model is using the just-in-time allocation principle, the two trains will initially not communicate with the same switchboxes. We will now consider two cases: firstly, the case where the two trains still are in positions where they do not communicate with the same switchboxes, and then, the case where the trains are close enough to each other to communicate with the same switchboxes.

For the first case, we have already shown for the configuration in Fig. 16, that a train is able to safely pass a station even if the points are initially in the wrong position compared to the train's route. Therefore, it can also safely pass several stations until it reaches the zone of influence of another train.

We now analyse the system to find configurations corresponding to the earliest possible time where the two trains may interact with the same switchboxes, and thereby the earliest point that the trains' reservations and locks may interfere with each other.

Figures 20 and 21 show for two different positions of the black train, the earliest point of the striped train at which the two trains can interact with the same switchbox. The switchboxes that each train may communicate with are marked in the train's corresponding line type (i.e. dashed for the striped train and solid for the black train). Since the two situations are included in the system runs of the model instance for the configuration illustrated in Fig. 22, it is sufficient to verify that configuration.

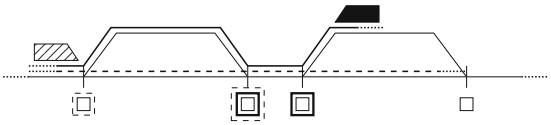

Fig. 20. One of the two cases where two trains driving in opposite directions can communicate with the same switchbox: the black train is fully at a station.

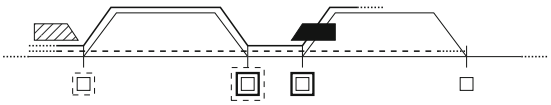

Fig. 21. The other of the two cases where two trains driving in opposite directions can communicate with the same switchbox: the black train is partially at a station.

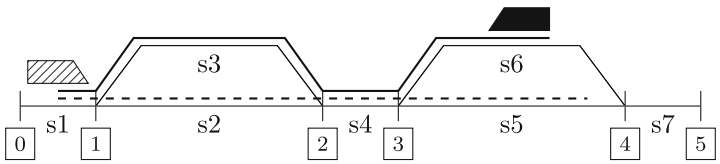

Fig. 22. A configuration which includes the cases where two trains driving in opposite directions can communicate with the same switchbox.

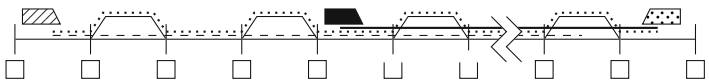

Fig. 23. An example of a typical local railway network with an extra train.

Note that the configuration where the striped train has moved one segment further (i.e. is inside the left-most station) and the black train has not yet (fully) entered the right-most station (i.e. is fully or partially on the line to the right of the right station) is equivalent to the two cases shown in Figs. 20 and 21, so we do not need to also verify another configuration capturing these cases. Note that the situations where the trains are closer to each other than shown in Fig. 21 (and thus still may communicate with some of the same switchboxes) are also included in the configuration illustrated in Fig. 22. The safety properties of an instance of the third model for the configuration shown in Fig. 22 were successfully verified, and the *not_all_trains_arrive* property was (as expected) false. Therefore, we can conclude that two trains can safely pass each other. This concludes the case analysis for this kind of two-train configurations.

7.2.2. Verification of three-train configurations

In some local railways, a third train may be placed roughly in the middle between the two end-destinations as shown in Fig. 23. This means that two trains may be going in the same direction, so we need additionally to analyse a third case. (Even though the trains in the considered local railways are supposed not to be so close that they can interfere and thereby constitute a risk of collisions, we still wish to verify the safety for a case where the rear train is catching up on the front train and gets into the zone where it can interfere with the front train as this could happen in the case of irregularities, e.g. if the front train had a break down). Figures 24 and 25 shows for four different positions of the (black) front train, the earliest point of the (striped) rear train at which the two trains can interact with the same switchbox. Since the four situations (and any situations where the trains are closer to each other) are included in the system runs of the model instance for the configuration illustrated in Fig. 26, it is sufficient to verify that configuration. The safety properties of an instance of the third model for this configuration were successfully verified, and the *not_all_trains_arrive* property was false as expected. Therefore, we can conclude that two trains can safely drive behind each other when they are in each other's influence zone.

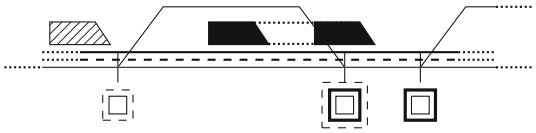

Fig. 24. Two cases where two trains driving in the same direction can communicate with the same switchbox: the black train is either fully or partially at the station.

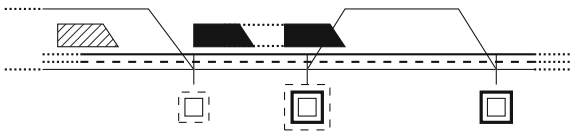

Fig. 25. Two cases where two trains driving in the same direction can communicate with the same switchbox: the black train is either fully or partially on the open line.

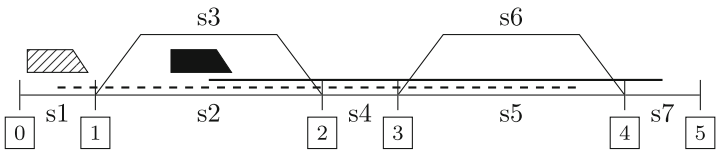

Fig. 26. A configuration which includes the cases where two trains driving in the same direction can communicate with the same switchbox.

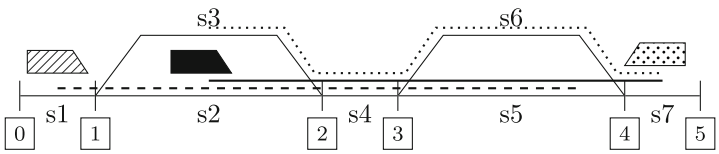

Fig. 27. A configuration which includes the cases where three trains (two of them driving in the same direction) can communicate with the same switchbox.

In addition, having a third train in the network means that there are cases where all three trains are close enough that they can communicate with the same switchbox. The configuration illustrated in Fig. 27 includes the two main situations where the three trains (two of them driving in the same direction) can all communicate with the same switchbox. The first of these situations is the one where the dotted train has moved fully onto segment $s6$. In this case, the three trains can all communicate with switchbox $SB2$. The second situation is the one where the black train and the striped train have each moved one segment forward, i.e. onto segment $s4$ and segment $s2$, respectively. In this case, the three trains can all communicate with switchbox $SB3$. The safety properties of an instance of the third model for this configuration were successfully verified, and the *not_all_trains_arrive* property was false as expected. Therefore, we can conclude that three trains can operate safely when they are in each other's influence zone.

7.2.3. Verification metrics

In Table 4, we show the numbers for the configuration data size of the configurations shown in Figs. 22, 26 and 27. In Table 5 the resulting number of variables and transition rules of the unfolded model instances of the third model are shown. Finally, in Table 6, we present the time and memory consumption for the verification of the safety properties for the instances of the third model corresponding to the configurations shown in Figs. 22, 26 and 27, respectively.

Table 4. Configuration data size.

Configuration	Segments	Points	Switchboxes	Trains	Route lengths
Figures 22 and 26	7	4	6	2	4
Figure 27	7	4	6	3	4

Table 5. Number of variables and transition rules in the unfolded model instances.

Configuration and model	Variables	Transition rules
Figures 22 and 26, model 3	50	794
Figure 27, model 3	55	1191

Table 6. Time and memory usage for verifying the safety properties for the instances of the third model.

Configuration and model	Property	Time (mm:ss)	Memory (MB)
Figure 22, model 3	*no_collide*	19:36	284
	no_derail	20:16	283
Figure 26, model 3	*no_collide*	12:55	285
	no_derail	12:59	285
Figure 27, model 3	*no_collide*	06:22	406
	no_derail	06:06	389

7.3. Other verification activities

Before beginning the process of symbolic model checking 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:* As one of the first activities during the model development, important functions (e.g. for expressing safety and consistency properties, which are used in the transition system assertions) were tested once 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.
  For example, we tested the function for checking the consistency between train reservations and train position, *cons_res_pos(tReservations*[*t*], *pos*[*t*], *nextSb*[*t*], *switchboxes*[*t*]*)* (shown in Sect. 4.4). The validation of the function was achieved by testing different cases, for example the following, where *sb1*, *sb2*, *sb3*, *s1* and *s2* are concrete values:
  - Train reservations and position which are *expected to be consistent*.

    [tc1] cons_res_pos([sb1 ↦ {s1,s2}, sb2 ↦ {s2}], double(s1,s2), sb1, {sb1 ↦ sb2, sb2 ↦ sb3])
  - Train reservations and position which are *expected to be inconsistent*.

    [tc2] cons_res_pos([sb1 ↦ {s1,s2}], double(s1,s2), sb1, {sb1 ↦ sb2, sb2 ↦ sb3])

- *Bounded model checking:* 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, than when using the global model checker.

For a given model instance, one can decide on a suitable depth for discovering bugs by counting the (expected) number of transitions at least needed for all the trains to reach their final destination. For example, for the instance of the third model obtained by using the configuration shown in Fig. 2, the striped train must make a reservation of segment $s2$ at $sb1$ and $sb2$ (6 transitions), then lock $sb1$ (3 transitions), and then move to segment $s2$ (2 transitions). For the black train, a similar set of transitions must be made (possibly interleaved with the above mentioned transitions of the striped train) for it to enter segment $s3$. Both the striped train and the black train must then again make a similar set of transitions for reaching their respective next segments, $s4$ and $s1$. This gives in total 44 transitions that must at least be made for the trains to be able to reach their final destination for the configuration in Fig. 2, and thus, in this case we would choose 44 as the depth for the bounded model checker.

8. 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 properties and other properties of instances of these models by model checking. This method could also carry over to other, similar applications. Although stepwise refinement of state transition systems is well known from other languages, it is novel for RSL.

The models are expressed in a new extension to RSL and RSL-SAL [PG07]: RSL$\star$, which has also been presented. This extension provides language constructs that better facilitate the specification of generic systems.

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. 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.

The RSL-SAL model checker back-end could handle small networks, but not large networks. Therefore, 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 [Ver13], in order to find an efficient verification technique that scales better up such that compositional reasoning reasoning would not be needed. In another case study [VHP17], 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. So we believe that this could also be the case for the system considered in this paper.

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 [HP00]. We would also like to thank him and the reviewers for very useful comments to a draft of this paper.

References

[Abr18] Abrial J-R (2018) On B and Event-B: principles, success and challenges. In: Butler M, Raschke A, Hoang TS, Reichl K (eds) Abstract State Machines, Alloy, B, TLA, VDM, and Z, pp 31–35. Springer, Cham

[BtBF+18] Basile D, ter Beek MH, Fantechi A, Gnesi S, Mazzanti F, Piattino A, Trentini D, Ferrari A (2018) On the industrial uptake of formal methods in the railway domain—a survey with stakeholders. In: Furia CA, Winter K (eds) Integrated formal methods, volume 11023 of Lecture notes in computer science, pp 20–29. Springer, Cham

[BtBFL19] Basile D, ter Beek MH, Ferrari A, Legay A (2019) Modelling and analysing ERTMS L3 moving block railway signalling with Simulink and Uppaal SMC. In: Larsen KG, Willemse T (eds) Formal methods for industrial critical systems, volume 11687 of Lecture notes in computer science, pp 1–21. Springer, Cham

[But09] Butler M (2009) Decomposition structures for Event-B. In: Leuschel M, Wehrheim H (eds) Integrated formal methods, volume 5423 of Lecture notes in computer science, pp 20–38. Springer, Berlin

[CDP+17] Comptier M, Deharbe D, Perez JM, Mussat L, Pierre T, Sabatier D (2017) 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, volume 10598 of Lecture notes in computer science, pp 148–159. Springer, Cham

[CLM+19] Comptier M, Leuschel M, Mejia LF, Perez JM, Mutz M (2019) Property-based modelling and validation of a CBTC zone controller in Event-B. In: Collart-Dutilleul S, Lecomte T, Romanovsky A (eds) Reliability, safety, and security of railway systems. Modelling, analysis, verification, and certification, volume 11495 of Lecture notes in computer science, pp 202–212. Springer, Cham

[ECfES11] CENELEC European Committee for Electrotechnical Standardization (2011) EN 50128:2011—railway applications—communications, signalling and processing systems—software for railway control and protection systems

[Fan12] Fantechi A (2012) Distributing the challenge of model checking interlocking control tables. In: Margaria T, Steffen B (eds) Leveraging applications of formal methods, verification and validation. Applications and case studies, volume 7610 of Lecture notes in computer science, pp 276–289. Springer, Cham

[Fan14] Fantechi A (2014) Twenty-five years of formal methods and railways: what next? In: Counsell S, Núñez M (eds) Software engineering and formal methods, volume 8368 of Lecture notes in computer science, pp 167–183. Springer, Cham

[FGH+16] Fantechi A, Gnesi S, Haxthausen A, van de Pol J, Roveri M, Treharne H (2016) SaRDIn—a safe reconfigurable distributed interlocking. In: Proceedings of the 11th world congress on railway research (WCRR 2016). Milano, Ferrovie dello Stato Italiane

[FH18] Fantechi A, Haxthausen AE (2018) Safety interlocking as a distributed mutual exclusion problem. In: Howar F, Barnat J (eds) Formal methods for industrial critical systems, volume 11119 of Lecture notes in computer science, pp 52–66. Springer, Cham

[FHM17] Fantechi A, Haxthausen AE, Macedo HD (2017) Compositional verification of interlocking systems for large stations. In: Cimatti A, Sirjani M (eds) International conference on software engineering and formal methods, volume of 10469 of Lecture notes in computer science, pp 236–252. Springer, Cham

[FHN17] Fantechi A, Haxthausen AE, Nielsen MBR (2017) 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

[FMGF10] Ferrari A, Magnani G, Grasso D, Fantechi A (2010) 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, Cham

[Geo03] George C (2003) The development of the RAISE tools. In: Aichernig BK, Maibaum T (eds) Formal methods at the crossroads. From Panacea to foundational support: 10th anniversary colloquium of UNU/IIST, the International Institute for Software Technology of The United Nations University, Lisbon, Portugal, March 18–20, 2002. Revised papers, pp 49–64. Springer, Berlin

[GH18] Geisler S, Haxthausen AE (2018) Stepwise development and model checking of a distributed interlocking system—using RAISE. In: Havelund K, Peleska J, Roscoe B, de Vink E (eds) Formal methods, volume 10951 of Lecture notes in computer science, pp 277–293. Springer, Cham

[Hax14] Haxthausen AE (2014) Automated generation of formal safety conditions from railway interlocking tables. Int J Softw Tools Technol Transf (STTT), Spec Issue Form Methods Railw Control Syst 16(6):713–726

[HBK10] Haxthausen AE, Bliguet ML,Kjær AA (2010) Modelling and verification of relay interlocking systems. In: Choppy C, Sokolsky O (eds) 15th monterey workshop: foundations of computer software, future trends and techniques for development, volume 6028 of Lecture notes in computer science, pp 141–153. Springer

[HBR18] Hoang TS, Butler M, Reichl K (2018) The hybrid ERTMS/ETCS level 3 case study. In: Butler M, Raschke A, Hoang TS, Reichl K (eds) Abstract State Machines, Alloy, B, TLA, VDM, and Z, volume 10817 of Lecture notes in computer science, pp 251–261. Springer Verlag

[HH19] Haxthausen AE, Hede K (2019) Formal verification of railway timetables—using the UPPAAL model checker. In: ter Beek MH, Fantechi A, Semini L (eds) From software engineering to formal methods and tools, and back: essays dedicated to Stefania Gnesi on the occasion of Her 65th Birthday, volume 11865 of Lecture notes in computer science, pp 433–448. Springer International Publishing, Cham

[HØ16] Haxthausen AE, Østergaard PH (2016) On the use of static checking in the verification of interlocking systems. In: Margaria T, Steffen B (eds) Leveraging applications of formal methods, verification and validation, volume 9953 of Lecture notes in computer science. Springer

[HP00] Haxthausen AE, Peleska J (2000) Formal development and verification of a distributed railway control systems. IEEE Trans Softw Eng 26(8):687–701

[JMN+14a] James P, Möller F, Nguyen HN, Roggenbach M, Schneider S, Treharne H, Trumble M, Williams D (2014) Verification of scheme plans using CSP||B. In: Counsell S, Núñez M (eds) Software engineering and formal methods, volume 8368 of Lecture notes in computer science, pp 189–204. Springer

[JMN+14b] James P, Möller F, Nguyen HN, Roggenbach M (2014) Steve Schneider, and Helen Treharne. Techniques for modelling and verifying railway interlockings. Int J Softw Tools Technol Transf 16(6):685–711

[LCPT16] Limbrée C, Cappart Q, Pecheur C, Tonetta S (2016) Verification of Railway Interlocking - Compositional Approach with OCRA. In: Lecomte T, Pinger R, Romanovsky A (eds) Reliability, safety, and security of railway Systems. Modelling, analysis, verification, and certification. RSSRail 2016. Lecture Notes in Computer Science, vol 9707, pp 134–149. Springer, Cham

[LJ18] Luteberget B, Johansen C (2018) Efficient verification of railway infrastructure designs against standard regulations. Form Methods Syst Des 52(1):1–32

[Mer08] Merz S (2008) The specification language TLA+, pp 401–451. Springer, Berlin
[MFTFL18] Mammar A, Frappier M, Tueno Fotso SJ, Laleau R (2018) An Event-B model of the hybrid ERTMS/ETCS level 3 standard. In: Butler M, Raschke A, Hoang TS, Reichl K (eds) Abstract State Machines, Alloy, B, TLA, VDM, and Z, pp 353–366. Springer, Cham
[PG07] Perna JI, George C (2007) 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
[PKHP19] Peleska J, Krafczyk N, Haxthausen AE, Pinger R (2019) Efficient data validation for geographical interlocking systems. In: Reliability, safety, and security of railway systems. Modelling, analysis, verification, and certification, pp 142–158
[RAI92] The RAISE Language Group, George C, Haff P, Havelund K, Haxthausen AE, Milne R, Bendix Nielsen C, Prehn S, Wagner KR (1992) The RAISE Specification Language. The BCS Practitioners Series. Prentice Hall Int.
[RFT16] Reichl K, Fischer T, Tummeltshammer P (2016) Using formal methods for verification and validation in railway. In: Aichernig BK, Furia CA (eds) Tests and proofs, volume 9762 of Lecture notes in computer science, pp 3–13. Springer Verlag
[Sab16] Sabatier D (2016) Using formal proof and B method at system level for industrial projects. In: Lecomte T, Pinger R, Romanovsky A (eds) Reliability, safety, and security of railway systems. Modelling, analysis, verification, and certification, volume 9707 of Lecture notes in computer science, pp 20–31. Springer Verlag
[SAL01] Symbolic Analysis Laboratory, SAL, Home page (2001). http://sal.csl.sri.com. Accessed 6 Feb 2020
[tBFGM11] ter Beek MH, Fantechi A, Gnesi S, Mazzanti F (2011) A state/event-based model-checking approach for the analysis of abstract system properties. Sci Comput Program 76(2):119–135
[UMC] UMC homepage. http://fmt.isti.cnr.it/umc/V4.2/umc.html. Accessed 6 Feb 2020
[Ver13] Verified Systems International GmbH (2013) RT-tester model-based test case and test data generator—RTT-MBT—user manual. Available on request from http://www.verified.de. Accessed 6 Feb 2020
[VHP17] Vu LH, Haxthausen AE, Peleska J (2017) Formal modelling and verification of interlocking systems featuring sequential release. Sci Comput Program 133(Part 2):91–115. https://doi.org/10.1016/j.scico.2016.05.010.
[Win02] Winter K (2002) Model checking railway interlocking systems. In: Proceedings of the twenty-fifth australasian computer science conference (ACSC2002), pp 303–310

Received 9 July 2019
Accepted in revised form 22 January 2020 by Erik de Vink, Ana Cavalcanti, Jan Peleska, Bill Roscoe, and Cliff Jones