Formal Verification of Signalling Programs with SafeCap

Alexei Iliasov[1], Dominic Taylor[2], Linas Laibinis[3], Alexander Romanovsky[1]

[1] Newcastle University, UK
[2] Systra Scott Lister, UK
[3] Institute of Computer Science, Vilnius University, Lithuania
Corresponding author: alexei.iliasov@ncl.ac.uk

Abstract. SafeCap is a modern toolkit for modelling, simulation and formal verification of railway networks. This paper discusses the use of SafeCap for formal analysis and fully-automated scalable safety verification of solid state interlocking (SSI) programs – a technology at the heart of many railway signalling solutions. The focus of the work is on making it easy for signalling engineers to use the developed technology and thus to help with its smooth industrial deployment. In this paper we explain the formal foundations of the proposed method, its tool support, and their application to real life railway verification problems.

1 Introduction

Effective signalling is essential to the safe and efficient operation of a railway network. It enables trains to travel at high speeds, run close together, and serve multiple destinations. Whether by mechanical semaphores, colour lights or electronic messages, signalling only allows trains to move when it is safe for them to do so. Signalling locks moveable infrastructure, such as the points that form railway junctions, before trains travel over it. Furthermore, signalling often actively prevents trains travelling further or faster than it is safe and sometimes even drives the trains. At the heart of any signalling system there are one or more *interlockings.* These devices constrain authorisation of train movements as well as movements of the infrastructure to prevent unsafe situations arising.

The increasing complexity of modern digital interlockings, both in terms of the geographical coverage and that of their functionality, poses a major challenge to ensuring railway safety. Even though formal methods have been successfully used in the railway domain (e.g. [3], [2]), their industry application is scarce. In spite of a large body of academic studies addressing issues of formal verification of railway systems, they typically remain an academic exercise due to a prohibitive cost of initial investment for their industrial deployment. The following are some of the reasons. First, signalling engineers need to learn mathematical notations to apply them. Second, the tools often cannot be applied for analysing large real stations due to their poor scalability. Third, the companies need to drastically change the existing development processes in order to use them.

This paper proposes a formal tool-based approach that addresses these issues by (i) verifying the signalling programs and layouts developed by signalling engineers in the ways they are developed by industry, (ii) ensuring fully-automated verification of safety properties using a family of the state of the art verification techniques (in particular, automated theorem provers and solvers), and (iii) providing diagnostics in terms of the notations used by the engineers. All together, this affirms that the developed methods and tools can be easily deployed to augment the existing development process in order to provide extra guarantees of the railway safety.

The paper is structured as follows. Section 2 presents the work background by overviewing the SafeCap toolkit, the role of SSI programs in railway signalling, and the key safety principles that these programs must follow. In Section 3 we discuss the SafeCap verification core, including its underlying modelling language and essential verification techniques. The proposed verification method is illustrated by a case study of a real railway station in Section 4. Finally, Section 5 concludes the paper by summarising the achieved results.

2 Background

SafeCap Platform. The SafeCap platform is a toolkit for modelling railway capacity and verifying railway network safety [11]. It allows signalling engineers to design stations and junctions relying on the provided domain specific language (SafeCap DSL), as well as to check their safety properties and evaluate potential improvements of capacity by using a combination of theorem proving, SMT solving and model checking [12]. The platform has been substantially extended by adding new simulators, solvers and provers, as well as the support for representing a wide range of the existing signalling frameworks [14, 15].

This paper also takes our work on SafeCap further by developing a set of new tools for importing, analysing and proving safety of railway data in standard SSI and SSI-based technologies such as Smartlock[1] by Alstom and WESTLOCK[2] by Siemens. The overall SafeCap architecture is presented in Fig. 1. Verification of SSI is our first experience with constructing and verifying large (i.e., containing tens of thousands of state transitions) models of the system dynamic behaviour. Previously, our industrial experience was concerned solely with verification of static data. The current work extends the SafeCap framework with advanced capabilities for reasoning about dynamic (i.e, transition-based) systems.

The developed SafeCap verification and proof back-ends enable automated reasoning about static and dynamic properties of railways or their signalling data. Our two principal verification routes are the built-in symbolic prover backed by a SAT solver, a range of external provers provided via the Why3 framework [4], and the ProB model checker [17] (used just as a constraint solver).

The SafeCap DSL provides a formal, graph oriented way to capturing railway schemas and some aspects of signalling [13]. In the DSL, a railway schema is a mathematical object consisting of data structure definitions (namely, types and

[1] For more details, see https://www.mobility.siemens.com/mobility/global/en/interurban-mobility/rail-solutions/rail-automation/electronic-interlockings/pages/electronic-interlockings.aspx

[2] For more details, see http://www.alstom.com/products-services/product-catalogue/rail-systems/signalling/products/smartlock-interlocking-products/

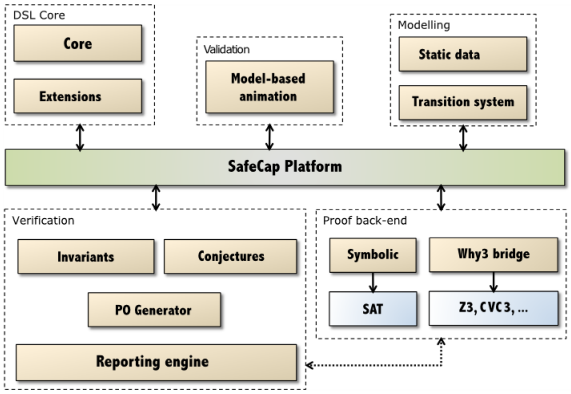


Fig. 1. SafeCap architecture

constants) as well as required logical constraints on the defined data (axioms and lemmata). We can distinguish two main parts of the SafeCap DSL – the Core and its various extensions. The Core provides means to mathematically describe the physical topology of a railway schema (or, in fact, any graph-based structures). Its first-class concepts are graphs and subgraphs. These typically represent track topology, track circuits, routes or axle counters. As such, a model of a railway schema in the SafeCap DSL Core is independent of any given signalling solution.

Once a railway schema model is created (or imported from an external format) in the Core, it is checked for its validity or well-definedness. For that, a number of graph theoretical statements are automatically generated and verified, including isomorphism properties between constituent subgraphs, path validity within a given graph, connectivity, acyclicity, node degree and so on.

Various concepts of a railway schema such as signals and signalling solutions, speed limits, stopping points and so on can be incorporated into via DSL extension plug-ins. Such plug-ins introduce new data (as custom annotations) and supporting logic (as additional logical constraints or relationships). Such a tool architecture allows us not to commit to any regional technology and thus to offer a broadly similar approach for a range of legacy and current technologies.

In this paper we focus on the fixed-block signalling prevalent in the UK and common to many other countries. In doing that, we rely on a dedicated SafeCap DSL plug-in for incorporating this type of signalling data.

Computerised signalling and SSI. The first signalling interlockings were mechanical devices that constrained the movement of levers, were connected to points and semaphore arms, and were contained in mechanical signalling boxes. During the twentieth century these devices were superseded by electrical relay-based interlockings that switched electrical current to motorised points, colour light signals and other lineside devices. The safety conditions applied by relay interlockings included tests of track circuits or axle counters – the devices that automatically determine whether a section of line has a train on it. The advent of computer technology brought the opportunity to replicate and build on the relay interlocking functionality within a computer based interlocking.

```
*QR117B(M)                     / route request block for route R117B(M)
  if R117B(M) a                / route R117B(M) is available
             USD-CA f,OSC-BA f,OSV-BA f   / sub-route and sub-overlaps are free
          then if OSL-AC l,      / sub-overlap is OSL-AC locked
                  P223 fr , P224 fr   / points P223, P224 free to move reverse
               then @P223QR \  / call subroutine P223QR
             if OSD-BC f       / sub-overlap is OSD-BC is free
                LTR04 xs       / latch (boolean flag) not set (false)
                P224 crf       / point P224 commanded reverse or free to move reverse
             then R117B(M) s   / set route set flag for R117B(M)
                USD-AC l , USC-AB l , USB-AB l , OSA-AB l  / set sub-routes/overlaps
                P224 cr        / command point P224 reverse
                LARR xs        / clear latch LARR
                S117 clear bpull   / clear signal button pull flag
                if P223 xcr , P223 rf then / check point states
                 @P223QR   / point command subroutine
                    EP230 = 0 \ / reset timer EP230
```

Fig. 2. SSI example: route request code for route R117B(M) in the PRR module

One of the earliest forms of computer based interlocking was the Solid State Interlocking (SSI), developed in the UK in the 1980s through an agreement between British Rail (the then nationalised railway operator) and two signalling supply companies. Running on bespoke hardware, SSI software consists of a core application (common to all signalling schemes) and site specific geographic data. The original SSI has now been superseded by more powerful, modern hardware platforms running software developed in accordance with modern standards for safety critical software. Nonetheless, the functionalities of the core application and the Geographic Data Language (GDL) remain largely unchanged.

SSI GDL data configures a signalling area by defining site specific rules, concerning the signalling equipment as well as internal latches and timers that the interlocking must obey. Despite being referred to as data, a GDL configuration resembles a program in a procedural programming language. The configuration is iteratively executed in three major stages: reception of input state messages from signalling equipment, followed by execution of rules, followed by construction and transmission of output command messages to the signalling equipment.

There are two main modules defining the signalling behaviour – the route and point request (the PRR module) and formation of output telegrams (the OPT module). An example of route request code (a part of logic that reacts to an external route request) is given in Fig. 2. Notice that the parts between `if` and `then` are atomic predicates combined with implicit conjunction, while everything between `then` and a slash character is a command (made of a sequence of atomic commands). For instance, `USD-CA f` stands for *test that subroute* `USD-CA f` *is free*, while `USD-CA l` commands the subroute to be freed.

Safety principles. There are two main safety principles shared by all signalling operations that employ the SSI technology. From these a large number of operational constraints can be derived, that consequently become verification conditions to check against given signalling data.

A schema must be free from collisions. A collision happens, potentially, when two trains may occupy the same part of a track at the same time. In route-based as well as speed-based signalling, the principal mechanism to address this property is that of route locking and holding. A train is given permission to enter an area of a railway, once there is a continuous and safe path through the area assigned exclusively to this train. Such a path is normally called a *route* and is delineated by *signals* – either physical track-side signals with lamps or conceptual signals displayed to a driver on a computer screen. The extent between successive signals defines the smallest train separation.

For a route to be locked, all the movable equipment such as *points* or level crossings must be set and detected in a position that would let a train safely travel on its desired route. They must remain locked in such a state and their position must be positively confirmed before a train enters the route.

A schema must be free from derailments. A derailment may happen when a train moves over a point that is not set in any specific direction and thus may move under a train. To avoid this, a point must be positively confirmed to be *locked* before a train may travel over it. Typically, a signaller must define conditions under which point reconfiguration is considered safe.

Related work. There have been a number of studies focusing on formal verification of SSI programs. The majority of works (e.g., [20, 16, 10]) use various forms of model-checking in an attempt to verify safety of *train run scenarios*, with interlocking rules derived manually or via an automated translation from SSI data. With few exceptions, the proposed techniques actually scale up to only toy examples, or cover a small subset of functionalities, or both. For instance, the approach presented in [10] uses NuSMV to model check a small subset of safety properties for a selected subset of SSI data based on real-life signalling data. In the face of sheer number of train run scenarios, one way to avoid the state explosion problem might be statistical simulation of train runs [6]. However, this approach has non-trivial implications on result interpretation.

We see a fundamental flaw in all such scenario exploration techniques: by introducing train runs and assuming certain traffic patterns they cannot find, even if they were to scale up, serious signalling mistakes that do exist in real-life implementations and only manifest themselves when a combination of several rare conditions happen [8]. Our approach does not suffer from this limitation as we do not need consider train runs (and thus limit verification to few assumed possibilities). Instead we check the worst case safety implications for all possible train run scenarios. Another problem with these solutions is their poor diagnostics, where the feedback on safety violations is not given in terms that signalling engineers could understand (i.e., SSI and the schema language).

In [5] the authors build a model of railway operation constrained by imported signalling data. A model checker automatically explores train movement scenarios (i.e., model states) and reports on violation of safety properties. The technique does not support generic safety properties (which have to be written separately for a specific layout) and the reported result indicate it is unlikely to scale to the industrial scale. In cases where a track graph can be cut with a very spectral ratio (i.e., two stations connected by a straight graph), it is sound to conduct verification of subparts separately [18, 19]. This is not often found in practice as SSI is traditionally limited to 64 or 256 controlled pieces of equipment and it is impractical to wire equipment at significant distance from a control box.

Verification and validation of a fragment of *safety logic* for European Railways Train Management System (ERTMS), ensuring also interoperabilty of different signalling solutions, is described in [7]. ERTMS specifications (written in a structured programming language) are automatically translated into formats of the employed external verification tools. Paper [9] presents an ongoing work on automatic model generation and verification of Railway Markup Language (RailML) formatted data, which also include route tables and interlocking information. Interlocking programs are defined in RailML using route scheduling and route automata. Neither of these approaches however could be applied for verification of SSI programs. Moreover, contrary to our work, they heavily rely on model checking techniques and tools for verification of railway safety properties.

3 Modelling and Verification in SafeCap

In this section we present our main contribution – the integrated generic framework for modelling and verification in SafeCap that we rely on to verify safety properties of railway signalling.

3.1 SafeCap Data Analytics

In our earlier works [12, 14, 15] we have proposed a formal model to capture and verify concrete signalling constraints by enforcing a certain standard of input data representation. However, industrial applications do not easily fit into the proposed view and there is a wide variation in the kind, rigour and comprehensiveness of the data defining existing signalling designs.

To be able to deal with varying forms of signalling input data, we complement SafeCap DSL with a generic modelling framework. The framework, called SafeCap Data Analytics (SDA), offers modelling concepts similar to that of a state-based modelling language. It is not meant to be used by an end user but rather as an intermediary tool bridging signalling input data and generated verification conditions. The SDA approach allows us to incorporate any extensions that require non-trivial reasoning (in particular, specific signalling solutions) with the DSL Core in an uniform and mathematically consistent way.

An SDA model comprises the static part, defining model constants, axioms, as well as verification statements (called conjectures), and the dynamic part, defining state transitions over model variables and thus expressing possible state evolution. Overall, it can be seen as a characterisation of a state transition system. There is no time model (SSI timers are seen purely as integer counters enabling causation reasoning) and state transitions are assumed to fire in an atomic fashion. As the focus is exclusively on static proof, we only define the proof semantics and do not consider construction of model states or traces.

We employ first-order logic equipped with the Zermelo-Fraenkel version of set theory and arithmetics to write predicates defining system axioms, conjectures as well as pre- and post-conditions of state transitions. Relational and functional model structures are expressed as special kinds of sets (i.e., sets of mappings between associated elements) and variable values can be drawn from finite or infinite sets. There are also the predefined sets of integers, booleans and reals.

The notation and underlying formal semantics of a transition system (a variation of the weakest precondition semantics) are adopted from the B Method [1].

In relation to the railway domain, for each format of input data representing signalling data, there is a dedicated importing plug-in translating it into an SDA model: a collection of constants, axioms and state transitions. The number of such formal elements for a real life example is quite large – from several thousands to tens of thousands. The resulting formal model is a solid foundation for formal reasoning about the properties, in particular operational safety, of a chosen signalling design. At the moment SafeCap supports two schema formats – LDL (proprietary) and RailML – and two signalling data formats – SSI and XML-based (a proprietary schema).

There are three main classes of signalling models distinguished by the mix of axioms (static constraints) and state transitions (system dynamics):

- a purely static model reasoning about data with no model variables or state transitions. An example is a model derived from a set of *control tables* – signalling design data represented in a tabular form. For a verification tool, its is a collection of conjectures (lemmata) expressing data properties;
- a purely dynamic model where signalling is defined by state transitions. An example is SSI signalling data that we see as piece of code to be transformed into a state transition system. Such models are verified via safety invariants;
- a mixture of the two. An example is verification of equivalence between a control table and its implementation SSI data.

Next we consider how verification of signalling data differs depending on the class of a considered SDA model.

3.2 SDA Verification: Static model

For a static SDA model, its verification involves proving a set of logical conjectures expressing the required data consistency properties. By a conjecture we understand a predicate (logical condition) constraining the model constants. Such a conjecture must be proven in the context of model axioms, i.e.:

$$\mathsf{ctx}(c) \vdash \mathsf{conj}(c),$$

where c are model constants. Here $\mathsf{ctx}(x)$ represents a set of axioms from the DSL Core such as the railway topology data definitions as well as all incorporated extensions. Predicate $\mathsf{conj}(c)$ stands for an expected data property or a constraint to be implied by such core definitions.

As one example, we might wish to check that a route setting control table includes all the points necessary to be set reverse to enable the given path of the route. This statements translates into the following conjecture:

$$\begin{array}{l} \forall r \in \mathrm{Route} \;\cdot\; r \in \mathrm{dom}(\mathsf{Routes.Point}) \Rightarrow \\ \quad \mathsf{Node.base}[\mathsf{schema.reversepoints}[\{r\}]] \subseteq \\ \qquad (\mathsf{Points.base}[(\mathsf{Points.base}^{-1}[\{r\}] \cap \mathrm{ran}(\mathsf{Routes.Point}[\{r\}]))]) \end{array}$$

Here $\mathsf{schema.reversepoints}[\{r\}]$ is the topology derived set of points to be set reverse to enable the route r, while $\mathsf{Routes.Point}[\{r\}]$ defines an ordered list of required points. The (topology-derived) constant relations $\mathsf{Node.base}$ and $\mathsf{Points.base}$ map between the physical and logical points and between the point names and the point states respectively. Finally, $[\cdot]$ and $(\cdot)^{-1}$ are relational image and inverse operators. For more details on the used notation, see [1].

Depending on the kind and form of $\mathsf{ctx}(c)$, a property $\mathsf{ctx}(c) \vdash \mathsf{conj}(c)$ can be handled by a constraint solver, a symbolic prover (SMT solver), or a satisfiability (SAT) solver. There are several provers available within SafeCap, such as Why3, ProB, or Minisat. In the production version of the tool, a conjecture is always checked by at least two distinct provers, one of which must be external.

There are a number of requirements to satisfy for a conjecture to be deemed logically meaningful and well-formed. Overall, a conjecture must not be a contradiction or tautology. The reason for these checks is to avoid conjectures that are logically inconsistent or those that are true or false irrespectively of a verified schema or signalling data. The latter cases, while logically consistent, in practice indicate serious mistakes in the formulation of a conjecture predicate.

3.3 SDA Verification: Dynamic model

For a dynamic SDA model, its verification boils down to proving an inductive system invariant expressed as set of predicates. For simplicity, we refer to each such a predicate as a safety invariant. A safety invariant represents a property on the system state (variables) to be maintained during the system functioning. Safety invariants formalise the established principles of interlocking operation. They are formulated manually with the help of domain experts and translated into a formal notation. This is done once for any given technology (e.g., SSI).

To show that an invariant property is indeed preserved by the system, one must prove a logical sequent (theorem) of the following form:

$$\mathsf{ctx}(c) \land \mathsf{inv}(c, s) \land \tau(c, s, s') \vdash \mathsf{inv}(c, s'),$$

where $\mathsf{ctx}(c)$ are all the defined axioms constraining the model constants c, $\mathsf{inv}(c, s)$ is a safety invariant over the constants c and the current state (variables) s, and $\tau(c, s, s')$ is some state transition producing a new state s'. $\tau(c, s, s')$ is usually defined by a conjunction of transition pre- and post-conditions: $\mathsf{pre}(c, s) \land \mathsf{post}(c, s, s')$. Finally, $\mathsf{inv}(c, s')$ is an invariant over the new state s'.

It is convenient to generalise the above statement to also account for some historic (previous) model state. We refer to such a state as s_h and understand it as the state observed prior to the current state s :

$$\mathsf{ctx}(c) \land \mathsf{inv}(c, s, s\_h) \land \tau(c, s, s') \vdash \mathsf{inv}(c, s', s)$$

Historic states are not manipulated in state transitions. The only source of information about a historic state is the invariant inv. Conceptually, when a transition happens, the old state (s) takes the place of the historic state (s_h) and the new state s' replaces the old state s. Since we are doing symbolic proof, this is all we need to know about historic states. The principle can be generalised to arbitrary deep historic trace although we did not encounter a need for this.

As an example, the following concrete invariant checks that the minimal conditions of point switching are met:

$$\forall p \in \text{Node} \;\cdot\; \mathit{point\_c}(p) \neq \mathit{point\_c\_h}(p) \Rightarrow \\ \quad \mathsf{schema.pointcleartracks}[\mathsf{Node.base}^{-1}[\mathsf{Node.base}(p)]] \cap \mathit{track\_o} = \varnothing$$

Here model variables are given in italic, while all the other identifiers are constants originating from the underlying model railway schema. The model variable *point_c_h* is a historic version of the current-state variable *point_c*.

The verification conditions for such a model are generated by instantiating model invariants. Namely, a separate verification condition (proof obligation) is created for each pair of invariant $\mathsf{inv}(c, s, p)$ and state transition $\tau(c, s, s')$. Thus, for 10 invariants and 2000 transitions, there are up to 20000 conditions to prove.

Transition system proofs are not as well suited for constraint solving as conjectures about control tables. The primary reason is the abundance of complex abstract relations. Off the shelf provers, such as E, SPASS, Z3, have proven themselves capable but are unable to return any valuable feedback on failed proofs and are generally quite slow (typically about 20 seconds per proof obligation) and memory demanding (some proofs require up to 64GB memory). To address this, we have developed a custom symbolic prover, which is described next.

The built-in SafeCap symbolic prover is used as the primary means for verifying safety invariants. Unlike conjectures for a static SDA model that are typically proven for concrete constant values, generated safety invariant proof obligations are stated over all permissible state values. The complexity of constraints and, to much less degree, the scale of state space make it an inordinately difficult verification task for a constraint solver.

The symbolic prover starts with the invariant statement as the top goal and tries to simplify, split or rewrite this goal until it becomes trivially true. It relies on a number of *tactics* – functions that implement goal transformations (like splitting a conjunctive goal into several subgoals).

Since the prover is supposed to be used fully automatically, a special attention in its implementation is given to the cases when it fails to prove its goal. The prover is designed to stop in a state best suited to the subsequent interpretation. To achieve this, once the prover detects a failed proof branch (due to a time out, an absence of applicable tactics, etc.), it backtracks while looking for a historic undischarged goal matching one of the predefined templates. For each of these templates, there is its defined interpretation in a natural language to be shown to an engineer to assist with the understanding of the nature of an error.

The symbolic prover is not designed to be ever used interactively or even outside of SafeCap. However, it is possible to write dedicated tactic scripts for each safety invariant. Such scripts can reorder, remove and parametrise proof tactics as well as define different proof branches using applicability tests. In particular, the form and nature of SSI data allows us to easily recognise repeating patterns in the usage of SSI commands and, even without seeing a specific SSI data instance, we are able define efficient tactics to support a safety invariant.

We use two complementary techniques to demonstrate that the built-in prover is sound. First, all the rewrite rules are known to be valid lemmata in first order logic and set theory[3]. Second, for each instance of a rewrite rule, SafeCap can

[3] With the unfortunate exception of arithmetics that is handled as a black-box rewrite

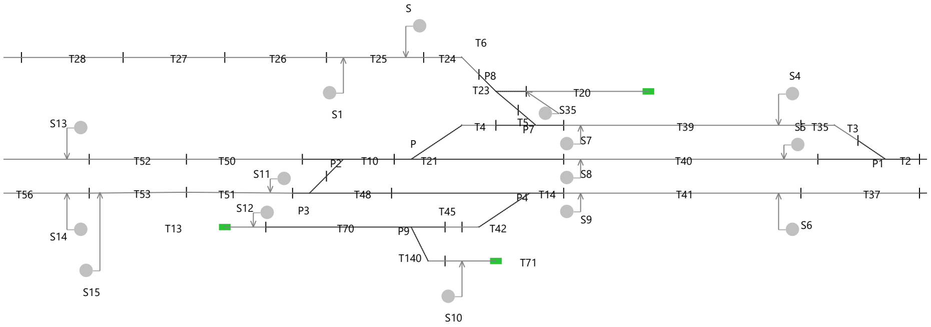


Fig. 3. Case study railway schema (an excerpt).

generate many thousands of theorems originating from successful or failed proof obligation and check them in an external prover. This technique is a form of automated mass testing to guard against programming mistakes. It can also be used to automatically recheck all proof scripts for extra reassurance.

Much attention is given to the presentation of verification results. It is imperative that the formal verification core operates autonomously and presents its findings in a clear and useful manner. To achieve this, together with every static or invariant verification condition one must write one or more reporting templates that define the mapping of verification output into a report coherent to a domain expert. The examples of such templates are given in Section 4.

4 Case study

As a case study, we consider a railway schema and the accompanying SSI signalling program for a real medium-size station. The SSI data for the case study were developed by an industrial company using the existing process. Verification was conducted against the properties mandated by the national railway authority (Network Rail).

A small part of a diagrammatic (not to scale) representation of its layout is given in Fig. 3. To give a sense of its actual size, the area consists of 117 train detection track circuits, 12 points and 42 routes. The labels in the diagram had to be obfuscated for the purposes of this publication. The signalling data (following the SSI standard) are defined in 14 separate modules, summing up to 274KB of disk space. The modules contain plain text source of SSI signalling in the SSI format described above.

The case study data were loaded into SafeCap in two stages. First, the railway schema was imported and represented in the SafeCap DSL Core. Second, the SSI signalling program was added (using the dedicated plug-in) as a DSL Core extension based on the SafeCap SDA. When a digital version of a railway schema exists, it can be imported directly into SafeCap. If it is only available as a paper or digital scan representation, it has to be manually drawn in the SafeCap schema editor. This takes about half a day for an experienced railway engineer.

A railway schema typically contains the track topology, track joints and signals. From this, the platform generates the necessary derived information (such as track circuits, points, routes, subroutes, overlaps, etc.) that is represented and

pre
$\text{LTR04} \notin \mathit{latch\_s} \ \wedge\ \text{OSD-BC} \notin \mathit{overlap\_l} \ \wedge\ \text{R117B(M)} \in \mathit{route\_a}$
$\text{OSC-BA} \notin \mathit{overlap\_l} \ \wedge\ \text{OSV-BA} \notin \mathit{overlap\_l} \ \wedge\ \text{USD-CA} \notin \mathit{subroute\_l}$
$\text{LTR117} \notin \mathit{latch\_s} \ \wedge\ \text{LTR119} \notin \mathit{latch\_s}$
$(\text{REVERSE} = \mathit{point\_c}(\text{P224})) \vee ((\text{TSD} \notin \mathit{track\_o}) \wedge (\text{USD-BC} \notin \mathit{subroute\_l}) \wedge \ldots$
$\mathit{request} = \text{QR117B(M)}$
post
$\mathit{route\_s}' = \mathit{route\_s} \cup \{\text{R117B(M)}\}$

Fig. 4. An example of a translated SSI transition: route locking

stored in the SafeCap DSL. SafeCap also attempts to automatically decode route names to match paths on the schema. For instance, R117B(M) would normally refer to a route starting from the signal S117 and taking the path B.

We treat SSI signalling data as a program made of large number of independent units (essentially event handlers). Furthermore, every such unit can be translated into a number of state transitions (one per each command). The result is a completely flat structure made of thousands of individual state transitions.

Fig. 4 illustrates one such translated state transition. The transition describe route setting resulting from the route request presented (in SSI GDL) in annotated Fig. 2. The route set update is specified in the transition postcondition. All the transition preconditions (apart from the last two) can be traced directly to the conditions of respective **if** blocks. The penultimate precondition is the result of expanding the SSI GDL expression `P224 crf`, testing whether point `P224` is already commanded reverse or is free to move into the reverse position. The last precondition associates this transition with the specific request type.

To conduct verification of an SSI data set, the underlying schema model and the generated dynamic SDA model (transition system) are integrated together. Additionally, the overall system model also includes safety invariants to be verified against the schema data and system state transitions.

The following is one example of a safety invariant. The invariant is concerned with route setting protocol. In particular, it ensures that the conflicting routes going in opposite directions cannot be set at the same time, which can formulated as specific conditions on the current free and locking subroutes.

$$\begin{array}{l}
\forall ra \in \mathsf{Route} \cdot\ ra \in \mathit{route\_s} \wedge ra \notin \mathit{route\_s\_h} \Rightarrow \\
\quad \forall rb \in \mathsf{Route} \cdot\ rb \in \mathsf{routeopposing}[\{ra\}] \wedge \mathsf{routedir}(ra) \neq \mathsf{routedir}(rb) \ \wedge \\
\qquad \mathsf{routelast}(rb) \in \mathrm{ran}(\mathsf{routetracks}[\{ra\}]) \Rightarrow \\
\qquad\quad \mathit{subroute\_l} \cap \mathrm{LastSubRoute}[\{rb\}] = \varnothing
\end{array}$$

In the above, $\mathit{route\_s}$ is a model variable of type $\mathbb{P}(\mathsf{Route})$, representing a set of routes. $\mathit{route\_s\_h}$ is its historic counterpart. The identifiers $\mathsf{routeopposing}$, $\mathsf{routedir}$, $\mathsf{routelast}$, $\mathsf{routetracks}$ are schema-derived constant relations. Specifically, the above invariant requires that, for any route ra and its opposing route rb such that the rb exit is within the ra extent ($\mathsf{routelast}(rb) \in \mathrm{ran}(\mathsf{routetracks}[\{ra\}])$), the last shared sub-route of rb must be checked free.

Overall, for route setting alone, we define 14 different invariants corresponding to 6 distinct safety principles. There are more invariants addressing telegram formation, flag operations and point commanding. Currently, we do not check timeliness conditions (on system reactions within a certain number of cycles).

(*the model constants and axioms (implied)*)
(*the safety invariant INV6*)
$\forall ra \in \mathsf{Route} \cdot \; ra \in route\_s \wedge ra \notin route\_s\_h \Rightarrow \ldots$
(*the transition preconditions*)
$\mathrm{LTR04} \notin latch\_s \;\wedge\; \text{OSD-BC} \notin overlap\_l$
$(\mathrm{REVERSE} = point\_c(\mathrm{P224})) \vee ((\mathrm{TSD} \notin track\_o) \wedge (\text{USD-BC} \notin subroute\_l) \wedge \ldots$
$\mathrm{R117B(M)} \in route\_a$
(*and the remaining preconditions*)
$\ldots$
(*the transition postcondition defining a new state*)
$route\_s' = route\_s \cup \{\mathrm{R117B(M)}\}$
$\vdash$
(*the safety invariant over the new state*)
$\forall ra \in \mathsf{Route} \cdot \; ra \in route\_s' \wedge ra \notin route\_s \Rightarrow \ldots$

Fig. 5. An example of an invariant preservation proof obligation

We also do not consider any liveness conditions as progress and the absence of livelocks and deadlocks is not part of interlocking safety requirements.

The verification process consists of generating verification goals to be proved (proof obligations) and attempting to dispatch them. The hypothesis list of the generated proof obligation combines declarations of the model constants and axioms, the current state version of the verified invariant as well as the pre- and post-conditions of the verified transition, while its goal states that the invariant in question must be preserved in any resulting transition state. Fig.5 shows an abbreviated example of the invariant preservation proof obligation, generated for the route locking state transition (see Fig.4) and the invariant presented above.

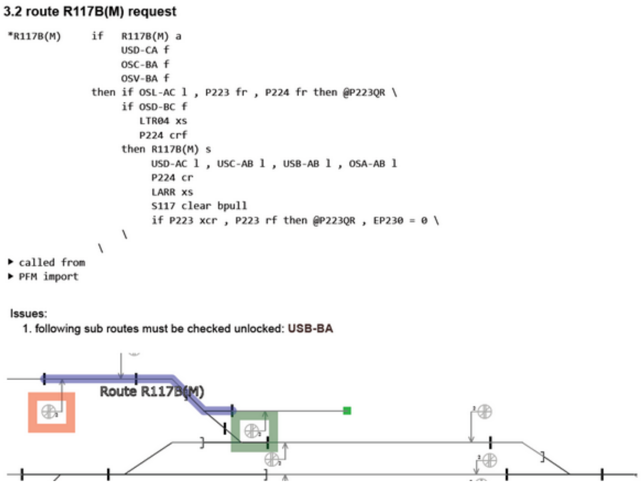
Once all obligations are generated, the built-in symbolic prover attempts to discharge every one of them. Each failed case is reported as a potential error in signalling data. By design, there is no provision to assist with automatic proof.

Table 1 gives SSI data verification summary of the conducted case study. Here transitions are all the state transitions derived from the data, while invariants are formalisations of various safety principles. Non-trivial p.o.'s (proof obligations) is the overall number of proof obligations after ignoring trivially correct ones (e.g., when a transition does not involve the variables mentioned in an invariant). Failed proof obligations indicate potential problems. Note that we do not attempt to distinguish between properties that are too hard to prove and those genuinely incorrect – they are all reported as potential errors. Finally, *Rejected* is the number of error reports rejected as false positives after manual inspection of a generated error report.

Table 1. Verification statistics

Transitions	Invariants	All p.o.'s[4]	Failed p.o.'s	Unique errors	Rejected
2248	9	1451	46	12	0

In addition to the built-in symbolic prover, the framework supports discharging a proof obligation via a number of different external provers. In practice they turned out to be much slower and not as capable overall. Below Table 2 gives the performance times for discharging all the proof obligations of the case study for different external tools. The external prover Why3 [4] is relying on the integrated



3.2 route R117B(M) request

```
*R117B(M)     if   R117B(M) a
                   USD-CA f
                   OSC-BA f
                   OSV-BA f
              then if OSL-AC l , P223 fr , P224 fr then @P223QR \
                   if OSD-BC f
                      LTR04 xs
                      P224 crf
                   then R117B(M) s
                        USD-AC l , USC-AB l , USB-AB l , OSA-AB l
                        P224 cr
                        LARR xs
                        S117 clear bpull
                        if P223 xcr , P223 rf then @P223QR , EP230 = 0 \
                   \
              \
▸ called from
▸ PFM import
```

Issues:
  1. following sub routes must be checked unlocked: USB-BA

Fig. 6. Verification report sample

Alt-Ergo[5] and CVC3[6] SMT solvers and eventually arrives at exactly the same result as the built-in prover albeit it takes several hours. Moreover, it turns out to be very sensitive to the available amount of RAM, e.g., restricting RAM to only 8GB leads to 52 undischarged proof obligations. The built-in SAT solver is unable to discharge a number of proof obligations proven by the symbolic prover but agrees on the set of proven conditions. It is fast and can be used to confirm the result of the symbolic prover in a production setting. Finally, ProB [17] (run in the constraint solver mode) leaves a number of of additional proof obligations undischarged and takes rather long time to complete the proof.

Table 2. Prover comparison

Prover	Run time	Undischarged proof obligations
Built-in symbolic	12s	46
Built-in SAT	12m	207
Why3 + Alt-ergo + CVC3	4h 15m	46
ProB	2h 46m	101

The experiments were conducted on Intel I7-4790K @ 4.0Ghz with 64Gb RAM. The built-in prover has used a number of custom tactic scripts tuned to the invariants defined. The end result of a verification exercise in SafeCap is an automatically generated verification report in a PDF format. A sample subsection of such a report is given in Fig.6. A report briefly describes the nature of the failed conditions, points to the problem source code location, and, if applicable, generates a part of the schema diagram with the key elements related to the failed proof. In the case of the displayed report in Fig.6 the problem is a missing sub-route test and the diagram shows the offending route location.

From the user perspective, the whole process consists of only two steps – providing input data and analysing the generated output. The actual model

[5] http://alt-ergo.lri.fr/

[6] https://cs.nyu.edu/acsys/cvc3/

construction, generation of proof obligations and proving of them: all this happen behind the scenes. Invariant construction is perhaps the most intricate and demanding part of the process that we are going to discuss in our future papers.

5 Conclusions

In this paper we presented the SafeCap approach to verifying railway signalling, in particular, signalling data with program-like representation called SSI. As a number of attempted case studies have demonstrated, the approach proved to scale well. Moreover, although only a subset of safety principles is currently encoded, we are confident that the approach is capable to effectively capture and formalise different formats of signalling data as well as required safety properties.

While SSI is a rather simple notation, it is still liable to state explosion. With all possible modules defining controllers for equipment such as signals and points connected, the state space grows to about 10^{1204} states. Also, one should notice that in industry safety principles are not designed or discussed in terms of train movements – something we commonly see in research papers applying simulation or state exploration techniques – but rather as constraints on signalling rules.

A combination of set theory and first order logic as the underlying mathematical language is the result of experiments over the course of several years. It appears to deliver the optimal combination of a terse, efficient notation for expressing conjectures and safety invariants, while, at the same time, also enabling effective symbolic automated proofs. Two other alternatives we have also explored are pure predicate logic and first order logic with functions and equality.

A custom made symbolic prover might seem a dangerous direction to take for an industry-oriented tool. Indeed, the prover we have developed is not anywhere as powerful or comprehensive as many state-of-the-art provers. However, it has a decisive advantage of being highly customisable via per-invariant tactic scripts. At such a level of fine-tuning it showed to be able to outrun any competition. The prover is also carefully designed to backtrack and terminate in a state facilitating helpful end user feedback.

The approach developed offers immediate industry benefits as it can be used within the existing SSI GDL production processes. The rapid, automated verification that it offers enables errors to be identified earlier in these processes, thereby reducing time consuming and expensive re-work. Furthermore, the SafeCap formal approach to verification provides additional assurance over the scenario based testing that is traditionally used in railway signalling. As the safety case underpinning SafeCap develops, and the range of safety properties that it verifies expands, further industry benefits become possible as the manual testing and checking activities are replaced by automated verification by SafeCap.

References

1. J.-R. Abrial. *The B-book: Assigning Programs to Meanings.* Cambridge University Press, New York, NY, USA, 1996.
2. F. Badeau and A. Amelot. Using B as a High Level Programming Language in an Industrial Project: Roissy VAL. In *Proc. of ZB 2005: Formal Specification and Development in Z and B*, volume 3455 of *LNCS*, pages 334–354. Springer, 2005.

3. P. Behm, P. Benoit, A. Faivre, and J.-M. Meynadier. Météor: A successful application of B in a large project. In *Proceedings of FM'99 – World Congress on Formal Methods*, volume 1708 of *LNCS*, pages 369–387. Springer, 1999.
4. F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3: Shepherd your herd of provers. In *Proccedings of Boogie 2011*, pages 53–64, 2011.
5. S. Busard, Q. Cappart, C. Limbrée, C. Pecheur, and P. Schaus. Verification of railway interlocking systems. In *Proceedings of ESSS 2015*, pages 19–31, 2015.
6. Q. Cappart, C. Limbrée, P. Schaus, J. Quilbeuf, L.-M. Traonouez, and A. Legay. Verification of Interlocking Systems Using Statistical Model Checking. In *Proceedings of HASE – High Assurance Systems Engineering*, pages 61–68, 2017.
7. A. Cimatti, R. Corvino, A. Lazzaro, I. Narasamdya, T. Rizzo, M. Roveri, A. Sanseviero, and A. Tchaltsev. Formal Verification and Validation of ERTMS Industrial Railway Train Spacing System. In *CAV*, pages 378–393. Springer, 2012.
8. Department for Transport. RAIB review of the railway industry's investigation of an irregular signal sequence at Milton Keynes. 2008. Available at https://www.gov.uk/raib-reports/review-of-the-railway-industry-s-formal-investigation-of-an-irregular-signal-sequence-at-milton-keynes.
9. T. Gonschorek, L. Bedau, and F. Ortmeier. Automatic Model-based Verification of Railway Interlocking Systems using Model Checking. In *Proc. of ESREL*, 2018.
10. M. Huber and S. King. Towards an Integrated Model Checker for Railway Signalling Data. In *Proceedings of FME 2002: Formal Methods Europe*, volume 2391 of *LNCS*, pages 204–223. Springer, 2002.
11. A. Iliasov, I. Lopatkin, and A. Romanovsky. The SafeCap Platform for Modelling Railway Safety and Capacity. In *Proceedings of SAFECOMP - Computer Safety, Reliability and Security. LNCS 8135. Springer*, 2013.
12. A. Iliasov, I. Lopatkin, and A. Romanovsky. Practical Formal Methods in Railways – The SafeCap Approach. In *Proceedings of Reliable Software Technologies - Ada-Europe, LNCS 8454*, pages 177–192. Springer, 2014.
13. A. Iliasov and A. Romanovsky. SafeCap Domain Language for Reasoning about Safety and Capacity. In *Proceedings of PRDC - Pacific-Rim Dependable Computing*, pages 1–10. IEEE, 2012.
14. A. Iliasov and A. B. Romanovsky. Formal Analysis of Railway Signalling Data. In *Proceedings of HASE – High Assurance Systems Engineering*, pages 70–77, 2016.
15. A. Iliasov, P. Stankaitis, and D. Adjepon-Yamoah. Static Verification of Railway Schema and Interlocking Design Data. In *Proceedings of RSSRail: Reliability, Safety, and Security of Railway Systems*, pages 123–133, 2016.
16. P. James, A. Lawrence, F. Moller, M. Roggenbach, M. Seisenberger, A. Setzer, K. Kanso, and S. Chadwick. Verification of Solid State Interlocking Programs. In *SEFM 2013 Workshops*, volume 8368 of *LNCS*, pages 253–268. Springer, 2014.
17. M. Leuschel and M. Butler. ProB: A Model Checker for B. In Araki Keijiro, Stefania Gnesi, and Mandrio Dino, editors, *Formal Methods Europe 2003*, volume 2805, pages 855–874. Springer-Verlag, LNCS, 2003.
18. C. Limbrée, Q. Cappart, C. Pecheur, and S. Tonetta. Verification of Railway Interlocking - Compositional Approach with OCRA. In *Proc. of RSSRail – Reliability, Safety, and Security of Railway Systems*, pages 134–149. Springer, 2016.
19. H. D. Macedo, A. Fantechi, and A. E. Haxthausen. Compositional Verification of Multi-station Interlocking Systems. In *Leveraging Applications of Formal Methods, Verification and Validation*, pages 279–293. Springer, 2016.
20. M. J. Morley. Safety Assurance in Interlocking Design. PhD thesis, University of Edinburgh. 1996.