Safety Invariant Verification that Meets Engineers' Expectations

Alexei Iliasov[1], Linas Laibinis[2], Dominic Taylor[3],
Ilya Lopatkin[1], Alexander Romanovsky[1,4]

[1] The Formal Route Ltd., UK
[2] Institute of Computer Science, Vilnius University, Lithuania
[3] Systra Scott Lister, UK
[4] Newcastle University, UK

Abstract. This industrial experience report discusses the problems we have been facing while using our formal verification technology, called SafeCap, in a substantial number of live signalling projects in the UK mainline rail, and the solutions we are now developing to counter these. Symbolic execution and safety invariant verification are well-understood subjects and yet their application to real life high assurance systems requires going a few steps beyond the conventional practice. In engineering practice it is not sufficient to simply know that a safety property fails: one needs to know why and hence where and what exactly fails; it is also crucial to positively demonstrate no safety failure is missing. In this industrial report we show how to derive a list of all potential errors by transforming safety invariant predicate using information from symbolic state transition system. The identified possible errors are verified by an automated symbolic prover, while a report generator presents findings in an engineer-friendly format to guide subsequent rework steps.

1 Introduction

There is a growing number of railway signalling companies that use verification techniques based on formal proofs for demonstrating and assuring system safety. The automated technologies, including AtelierB[1], Ovado/Rodin[2], Prover[3] and GNAT Ada[4], formally verify that a system satisfies a collection of identified safety properties. These tools report violations found during verification to the signalling engineers, who use that information to guide the rework process.

One of the earliest forms of computer-based interlocking was the Solid State Interlocking (SSI) [1], developed in the UK in the 1980s through an agreement between British Rail and two signalling supply companies, Westinghouse and GEC General Signal. SSI is the predominant technology used for computer-based interlockings on UK mainline railways. It also has applications overseas, including in India, Australia, New Zealand, France and Belgium.

[1] https://www.clearsy.com/en/tools/atelier-b/
[2] https://ovado.net/
[3] https://www.prover.com/
[4] https://www.adacore.com/gnatpro

In the last two years we have been applying a modern verification technology called SafeCap as part of a number of industrial signalling projects to verify the safety of SSI designs. Safety verification of 30 mainline interlockings, developed by different suppliers and design offices, has been successfully conducted for our industrial partners [4].

SafeCap was originally developed as an open toolkit for modelling railway capacity and verifying railway network safety in a number of public projects led by Newcastle University [2]. In the last 5 years the tool has been fully redesigned to deliver scalable and fully-automated verification of industrial interlockings [3], [4]. The resulting toolset has been proven in commercial applications to the verification of signalling projects that use SSI, and successor technologies. The two distinguishing features of SSI SafeCap are a fully automated verification and complete hiding of formalisation details. As a result, engineers receive the diagnostics reports describing the found safety problems in terms familiar to them, i.e., by explicitly referring to the applicable railway layouts (schemas) and SSI data.

Our approach to proving safety of signalling data is based on expressing signalling principles as a collection of predicates constituting *safety invariants*, translating the source data (the schema and the SSI data) into a formal model – a *state transition system*, and then generating and discharging proof obligations to establish that every system transition maintains the safety invariant. At the basic level, the output is a list of names of violated signalling safety principles. This is clearly inadequate and hence we provided further detail about identified violations in two ways:

- a particular transition leading to a failed proof obligation is used to define the source code location of a potential error;
- the state of an undischarged proof obligation is used to report the probable cause of proof failure and thus indicate the actual cause of an error in the source data.

The second way was an heuristic and was not guaranteed to succeed in producing a useful commentary. In particular, it did not attempt to list all errors in a sense that an error is understood by engineers as a specific mistake in source data. Such a rift between the tool output and engineers' expectations could not be bridged without changing the very approach to verification of safety invariants.

Initial industrial projects highlighted the deficiencies of the conventional safety invariant verification procedure, most critically that a single verification pass cannot produce the list of all possible errors due to masking of unreported errors by reported ones.

In order to rectify this, we replaced the proof state analysis heuristics with a safety invariant transformation procedure that produces a collection of safety invariant satisfaction proof obligations tailored to a specific symbolic state transitions. The procedure was designed in such a manner that every constructed proof obligation aligns exactly with a potential engineering error in source data.

Thus, with the same technique, we give a formal definition of an engineering error and a method to produce all their instances for a given system.

2 Reporting Safety Invariant Violations

2.1 Establishing System Correctness

The central question in the verification of signalling correctness is what constitutes a safe signalling design. Certain basic principles are universally accepted, for instance, the absence of train collisions and derailment. However, it is almost hopeless to verify the absence of such hazards in the strictest possible sense. Moreover, interlockings, by themselves, provide only a modicum of protection against driver errors through the provision of overlaps and control of signal aspects. Interlockings also do little to protect against equipment failures, but are themselves designed to be resilient to such failures.

For these reasons, correctness is established not against the basic principles but rather against the lower level signalling principles derived from foundational safety principles and designed to enforce railway operation with an acceptable level of risk and failures. Such principles are carefully designed by domain experts but can vary between regions and do change over time.

2.2 A Running Example

In the following we shall use a simple, but real-life case, of a signalling safety principle to illustrate our approach. The principle states that

> *For every set route, it holds that all sub routes of the route that are within the interlocking control area are locked.*

In other words, when setting a route, all the sub routes of the route path must be commanded locked or detected locked. In the mathematical notation we employ, which is a combination of first order logic and set theory, the rule can be formally expressed as the following safety invariant:

$$route\_subrouteset[\mathsf{route\_s} \setminus \mathsf{route\_s'p}] \cap SubRoute.ixl \subseteq \mathsf{subroute\_l} \qquad (1)$$

where *route_subrouteset* is a constant defined by a *signalling plan document* (a formalised data set describing a geographical interlocking area), stating which sub routes comprise the path of a route; *SubRoute.ixl* is a signalling plan constant defining current interlocking sub routes; route_s and route_s'p are the current and a previous states of a model variable recording route locking status (that is, $r \in \mathsf{route\_s}$ implies that route r is currently locked).

2.3 Symbolic Verification of Signalling Safety Principles

For the purposes of verification, each signalling principle is rendered as an *inductive safety invariant* – a system property that must hold when a system boots up and must be maintained (or, equivalently, reestablished) after any state update. Verification is then understood as the problem of checking that any safety invariant is respected by every state update. Technically this is done be generating conjectures of the form "*if an invariant holds in a previous state and an certain state update happens, is it true that the invariant holds for the new state?*". Formally, a conjecture (also called a *proof obligation* (PO)) is represented as a logical sequent consisting of a number of hypotheses (H) and a goal (G), denoted as $H \vdash G$.

In general, a schematic proof obligation for the preservation of a safety invariant (for a state transition $j \in J$) takes the following form:

$$M(c) \wedge A(c, v) \wedge I(c,'v, v) \wedge P_j(c, v) \wedge Q_j(c, v, v') \Rightarrow I(c, v, v') \tag{2}$$

where $M(c)$ and $A(c, v)$ are constants and constraints from the formalised signalling plan, defined over the constants c and model state (variables) v, a state transition is characterised by a pre-condition predicate $P_j(c, v)$ and a post-condition $Q_j(c, v, v')$ relating a next state v' to the current state v and constants c, J represents the set of all such state transitions, and $I(c, v, v')$ stands for the invariant property to be preserved.

The number of such proof conjectures is $m * n$, where m is the number of safety invariant predicates (67 defined so far) and n is the number of possible state updates (for the industrial projects we have carried out this value varies between 4000 and 140000 with a mean 17641). This is a small number when contrasted against the number of potentially reachable states (more than 2^{2000}).

When a prover fails to discharge (i.e., to complete automated proof of) a proof obligation derived from a safety invariant, we know that a safety invariant is violated. Clearly this alone is not sufficient since a typical interlocking data has thousands of lines of code. However, a failed proof obligation itself can be traced, via the associated state transition, to the data source code and, more precisely, to one or more control flow threads of the verified data. This gives us an initial error localisation in terms of available signalling data.

Such localisation alone is still not enough since one error can, and typically does, manifest itself in many failed proof obligations. Therefore, ascertaining the actual cause of each failed proof obligation is extremely laborious; in cases of hundreds or thousands of failed proof obligations it becomes simply impracticable.

Previously we processed every failed proof obligation at the report generation stage to identify its likely cause. This is much harder than simply looking at an *open goal*, i.e., a current proving conjecture that the prover failed to discharge. First, there can be a number of open goals. Second, an open goal could be stated, due to certain internal rewritings and simplifications, in terms that are of no relationship (i.e., having no common free identifiers) to the original transition and safety invariant predicate.

One technique that we employed to overcome this shortcoming was automatic backtracking of a failed sub goal to the state most suitable to reporting. Such a state had a goal matching certain predefined expression templates; backtracking combined pruning (that is, reversion) of proof steps with its own set of rewrite rules; as with this proving, there is no guarantee of arriving at a satisfactory result. However, in our applications it was consistently successful.

Knowing the cause of failed proof obligations allowed us to collate identical errors and produce usable reports. The compression factor here was quite significant: there could be between 10 to 50 failed proof obligations for every reported violation.

2.4 Running Example, Continued

The formal definition of our running example safety principle, as given in (1), is a typical set-theoretic statement of a safety predicate; however, any arising failed goals are difficult to backtrack and analyse. Expression *route_subrouteset*[route_s\ route_s'p] relates routes being set to their sub routes, therefore, when the proof fails, all we can hope to know is that one or more of the routes being set is not locking of one or more sub routes. However, we can rewrite the property into an equivalent predicate form:

$$
\begin{array}{l}
\forall \mathsf{r} \in \mathit{Route} \\
\quad \mathsf{r} \in \mathsf{route\_s} \setminus \mathsf{route\_s'p} \\
\quad \Rightarrow \\
\quad \forall \mathsf{sr} \in \mathit{SubRoute} \\
\qquad \mathsf{sr} \in \mathit{route\_subrouteset}[\{r\}] \cup \mathit{SubRoute.ixl} \\
\qquad \Rightarrow \\
\qquad \mathsf{sr} \in \mathsf{subroute\_l}
\end{array}
$$

In the rewritten proof obligation, the expression route_s\route_s'p collapses to a constant set of routes being set in a current transition. This allows the prover to eliminate the outer quantifier and introduce a new free identifier with the constraint $\mathsf{r} = R_1 \vee \mathsf{r} = R_2 \vee \ldots$. The proof then proceeds by analysing cases of disjunction. This, in turn, allows the prover to collapse *route_subrouteset*$[\{r\}]$ to a constant set, and to continue in the same manner. Although, as discussed before, useful identifiers like r describing proof context might be translated to other names or filtered out in actual proofs, the backtracking process can normally recover the proof state.

3 Positive Demonstration of the Absence of Violations

Decoding the proof state to infer an error has proven to be insufficient for finding all its causes and all the circumstances of its occurrence. It is possible for a failed proof obligation to reveal one error and, at the same, time mask the presence of another. Logically, nothing wrong happens here: analysis of a violated safety invariant, involving the backtracking process, reveals one likely cause of the

violation. Yet reporting all such causes associated with specific SSI source data errors is one of the principal requirements for an automated safety verification process.

In our example, if some route were missing the *locking* of two sub routes on its path, a failed proof obligation would point to only one of them. This is related to the fact that the prover is designed to identify a stuck goal as early as possible. Moreover, it is prohibitively expensive to try and explore every open (undischarged) sub-goal.

The simplest solution would be to fix all the identified problems and re-run the verification process, which would reveal previously masked errors. Unfortunately, this is not a practicable scenario for various reasons: 1) changes to signalling data change can takes months and conducting multiple re-verification cycles might be impossible due to delivery constraints; 2) generic signalling principles can, at times and subject to risk assessment, be deliberately violated to meet site-specific operational needs; 3) we cannot rule out the presence of false positives, partly to the complexity and constant evolution of real-life signalling data, and partly due to automated theorem proving potentially being undecidable for our modelling logic. A false positive can turn from being a benign issue to a critical one when it masks another error.

To rule out the masking of one error by another, we now approach safety invariant verification from a slightly different perspective. Instead of a verification process as such, we focus initially on the enumeration of all potential non-trivial errors. This is achieved by altering the method of proof obligation generation, aiming to obtain a dedicated proof obligation for each non-trivial potential error. In the following sections we define the meaning of *a non-trivial potential error* via a process that constructs bespoke proof obligations for each state transition.

3.1 Synthesising Focused Safety Invariant

Most of the attributes of SSI objects have two states (e.g., a route is set or unset, a sub route is locked or free and so on). In line with the set theory underpinning our formalisation, such attributes are modelled as set membership tests for an object in the model variable corresponding to the attribute. For instance, the fact that some sub route $UAA\text{-}AB$ is locked is expressed as $UAA\text{-}AB \in \mathsf{subroute\_l}$; the same sub route being free is, conversely, $UAA\text{-}AB \notin \mathsf{subroute\_l}$.

In our formalisation, a state transition is represented as a pair of predicates $Pre(c, v)$ and $Post(c, v, v')$. In a general case, that is all we can say. However, in the case of a state transition derived from signalling data, the restrictions of the SSI language allows us to make stronger assumptions. First, since SSI is a deterministic imperative language, state post conditions are limited to conjuncts of equalities of the form $v' = v \cup E_1 \setminus E_2$, where E_1 and E_2 are constant sets of the values to be added or removed. Second, the predicate language of SSI is also quite limited and, as a result, its translation yields a precondition that is a conjunct of just few forms of clauses:

- membership clause $v \in S$, $v \notin S$ or $v \subseteq S$;

– equality clause $f(v) = c$, $v = c$, $v \neq c$, ...;
– disjunctive clauses, quantifiers and implications.

By looking at a post condition, it is possible to deduce, via simple pattern matching, sets of objects that are being added or removed from the corresponding set variables (i.e., constant sets E_1 and E_2). That is, from the predicate of a post condition, we can unambiguously infer which sub routes are locked or freed and so on.

We shall *describe* various model variables of interest as indexed set Z_i. For instance, Z_0 could describe the model variable subroute_l, Z_1 – track_o, and so on. The description of a variable is different from the variable itself – it imprecisely characterises a variable state in the context of a given state transition.

Focusing on the post condition part of the variable description, we refer to the added and removed sets of of Z_i as Z_i^+ and Z_i^-. There is a simple relationship between Z_i and its variable counterpart v_i:

$$v_i = \overline{v}_i \cup Z_i^+ \setminus Z_i^- \tag{3}$$

Identifier $\overline{v}_i$ stands for the previous variable state. For the moment, we only know that it must satisfy the safety invariant. In our running example, for post condition subroute_l' = subroute_l' $\cup$ {*UAA-AB*} $\setminus$ {*UAA-BA*}, we have $Z_i^+ =$ {*UAA-AB*} and $Z_i^- =$ {*UAA-BA*}.

Next let us consider a precondition of a state transition. The situation is less certain here as we can rely only on the first two clause forms (presented above) to deduce the current state variable description, leaving some clauses not analysed.

Consider the first (membership) clause case. For every model variable, one can once again build two sets: values tested to be in a set variable and values tested not to be in a set variable. These define Z_i via added, $\overline{Z}_i^+$, and, removed, $\overline{Z}_i^-$, sets. Again, we can relate these two sets to the previous state of a model variable:

$$\overline{v}_i = x_i \cup \overline{Z}_i^+ \setminus \overline{Z}_i^- \tag{4}$$

Where x_i is a previous (unknown) variable state. Putting (3) and (4) together we have the following:

$$v_i = x_i \cup (\overline{Z}_i^+ \cup Z_i^+ \setminus Z_i^-) \setminus (\overline{Z}_i^- \cup Z_i^- \setminus Z_i^+)$$

Intuitively, $\overline{Z}_i^+ \cup Z_i^+ \setminus Z_i^-$ is a set of objects that are known to be added (locked or set), while $\overline{Z}_i^- \cup Z_i^- \setminus Z_i^+$ is the known set of removed (unlocked, freed) objects.

3.2 Computing Potential Errors

Model variable descriptions inferred for a state transition allow us to transform a safety invariant predicate into a program, computing the set of generally more

numerous but individually simpler proof obligations. This is due to the fact that most invariant statements are written in a predicate form with quantifiers to facilitate reporting (via introduction of bound variables that may provide error context) and such quantifiers can be eliminated, in the vast majority of cases, using the information contained in the description of model variables. The end result is a proof obligation without quantifiers, where bound variables are instantiated to specific constants.

Every such derived proof obligation is a test for a presence of an error; a unique name generated for a proof obligation is the name of a corresponding potential error.

The balance at play here is the granularity of errors. At one extreme we can take an original safety invariant and declare a violation of this invariant to be one error. Another, more desirable, extreme, is to deduce a unique combination of schema entities that may give rise to an error, and reflecting this combination in the *name of an error*.

We require the technique to be sound – it is fine to list errors that cannot possibly arise (and, we hope, will be filtered out by the prover), however it is not acceptable to omit errors that can potentially arise.

As a starting point, the name of an error is a combination of safety invariant predicate (referenced either in proxy by its given name or directly as a predicate) and state transition (we provide a unique name for each state transition). For instance, `inv1/transition5` or, with predicate, $I_n(c, \overline{v}, v)$`/transition5`.

When we drill down into what can go wrong for a specific transition with respect to a given invariant, we might discover that the original coarse-grained error name is refined into a number of more specific errors: `inv1/transition5/a/b/.../goal`. Here `a` and `b` are some schema objects. Another addition here is a goal predicate – for `inv1/transition5/a/b` to reference an error distinct from, say, `inv1/transition5/a/c`, each error needs its own distinct proof obligation. To summarise, an error name is a combination of:

- invariant;
- transition;
- identifying schema entities;
- verification goal.

There are two obvious restrictions on the set of error names for a given pair of invariant and state transition. In the worst case scenario, we have a one-to-one mapping between safety invariant predicate and an error name, that is, one potential error for every case of safety principle encoded in a predicate. The opposite extreme is combining names of all schema elements (that is, Cartesian product of sets of routes, tracks and so on) to name individual errors.

Let us consider again the example invariant of sub route locking on route setting:

$$\begin{array}{l} \forall \mathsf{r} \in \mathit{Route} \\ \quad \mathsf{r} \in \mathsf{route\_s} \setminus \mathsf{route\_s'p} \\ \quad \Rightarrow \ldots \end{array}$$

Predicate $\mathsf{r} \in \mathsf{route\_s} \setminus \mathsf{route\_s'p}$ filters out routes not commanded in the current state transition. Let some Z_p correspond to model variable route_s, then Z_p^+ and Z_p^- correspond to sets of routes being set and unset respectively. We can now reformulate the invariant, for the particular state transition and without any loss of precision, as

$$\forall \mathsf{p} \in Z_p^+ \setminus Z_p^- \quad \Rightarrow \ldots$$

Crucially, set $Z_p^+ \setminus Z_p^-$ is known exactly from the description of route_s, which allows us to soundly replace the external quantification with iteration. We shall employ a distinct syntax to define *programs* that compute focused proof obligations; for instance, the fragment above can be translated into the following imperative notation:

$$\texttt{WITH}\ p\ \texttt{FROM}\ Z_p^+ \setminus Z_p^- \quad \texttt{GOAL}\ (\forall \mathsf{sr} \in \mathit{SubRoute} \ldots)$$

here p is iterated over a set of values (derived from a given transition) and for each value of p, the programs constructs a dedicated proof obligation, as defined by the GOAL clause. The derived error names then would take form, for instance: `inv1/<transition name>/R123/<goal>`.

The improvement over the base case is that any found violation can be readily attributed to some route p_i even without analysing the proof state.

We can continue in the same manner focusing on the inner universal quantifier. This would deliver an extra level of refinement (granularity) reflected in error name at the price of producing one proof obligation per every sub route of a route.

$$\begin{array}{l}\texttt{WITH}\ p\ \texttt{FROM}\ Z_p^+ \setminus Z_p^- \\ \quad \texttt{WITH}\ sr\ \texttt{FROM}\ \mathit{route\_subrouteset}[\{r\}] \cup \mathit{SubRoute.ixl} \\ \quad\quad \texttt{GOAL}\ sr \in \mathsf{subroute\_l}\end{array}$$

The key here is that set $\mathit{route\_subrouteset}[\{r\}] \cup \mathit{SubRoute.ixl}$ is known at this point (as the p value was already computed) so we can once again render quantification as an iteration over a new, smaller goal.

We apply this program to achieve two goals: to compute the list of all possible errors, and to compute proof obligations for all such errors. By iterating over all combinations of state transitions and invariant predicates, we obtain the overall list of errors and their proof obligations.

There is one further potential refinement of the described procedure: we can use the variable description to deduce the cases where there is a definite error even before attempting a proof. This saves us from relying on the prover to fail to prove such cases or can be used to cross check the prover itself.

4 Discussion and Conclusion

The process outlined in the paper extends the safety invariant technique to produce error names and proof obligations at a finer level of granularity; previously,

a failed proof obligation could indicate one or more errors; with the new approach a failed proof obligation is exactly one error.

On one extreme, it can filter out all proof obligations for a given invariant. On another, it can inflate their number by orders of magnitude. On the balance, the conducted experiments and initial application in the life signalling projects show a significant but manageable (about ten fold) increase of the number of proof obligations. These proof obligations are individually much simply and hence the overall proof time increase is generally between 2 and 5 times. It is an acceptable price to pay for an increased assurance in the verification process.

The one to one mapping between "engineering" errors (deficiencies in source data) and failed proof obligations permits a fairly straightforward implementation of a tracking of historic improvements in data.

The process upon an already industrially successful application of formal methods; we routinely verify interlocking with hundreds of routes where control logic defines several thousand variables (Boolean, integer and categorical). We have seen cases of 24 deep nestings of conditional operators and blocks of codes with nearly 200 conditional statements.

The verification process is completely automatic: due to the sheer scale of a system under verification it is impracticable to require manual intervention at any stage of the process. Needless to say, such systems cannot be verified to any level of assurance via simulation or state space exploration techniques.

With the new approach the number of proof obligations is much higher: a typical interlocking verification results in between 20K and 100K proof obligations (derived from about 60 safety invariant predicates); this is after application of all the known reduction technique to symbolic state transition system. The proof takes longer, however, it is comfortably under 2-4 minutes for most complex projects, for the majority of the projects it takes less than 10 seconds.

We expect this effort to contribute to a substantial improvement of the SafeCap diagnostics reports, ensuring that the tool is fit for purpose and the eventual safety certification of our verification process as an alternative to manual checking.

References

1. D H Stratton. Solid State Interlocking. First edition, IRSE Booklet, 28. Institution of Railway Signal Engineers (IRSE). 20 pages. 1988.
2. 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*, pages 130–137, 2013.
3. Alexei Iliasov, Dominic Taylor, Linas Laibinis, and Alexander Romanovsky. Formal Verification of Signalling Programs with SafeCap. In *Proceedings of 37th International Conference, SAFECOMP 2018, Västerös, Sweden, September 19-21. LNCS 11093, Springer*, pages 91–106, 2018.
4. Alexei Iliasov, Dominic Taylor, Linas Laibinis, and Alexander B. Romanovsky. Formal verification of railway interlocking and its safety case. In *Proceedings of Safety-Critical Systems Symposium (SSS 2022), February 8-10, 2022*. Safety-Critical Systems Club, UK, 2022.