Mutation testing for rule-based verification of railway signalling data

Linas Laibinis, Alexei Iliasov, and Alexander Romanovsky

Abstract—**Industry applications of formal verification to signalling control tables require formulation of a large number of mathematical conjectures expressing verification rules. It is paramount to establish the validity and completeness of these conjectures. The paper discusses a mutation based validation technique that guides domain experts in the construction of such verification rules. Furthermore, we use genetic programming to quickly generate millions of well-formed data mutations of control tables and to synthesise mutation programs. The technique is illustrated by a synthetic running example and a discussion of our experience in using it in the industrial setting.**

Index Terms—**Mutation testing, formal verification, verification conditions, railway, signalling**

I. Introduction

This paper discusses the process of checking the correctness of safety critical data sets, specifically, railway control tables, and a way the procedure of constructing verification rules for such data sets can be made more efficient and interactive. The presented work is an extension of the SafeCap platform – a toolkit for verifying railway network safety [16]. By a verification rule we understand a formal predicate capturing a safety argument. The process of checking such a rule in SafeCap is completely automatic and relies on a number of automated back-end tools. The following is an example of a simple safety argument:

$$\mathrm{dom}(\mathsf{point\_tracks}) = \mathsf{Point}$$

Here $\mathsf{point\_tracks}$ is a constant relation between types Point and Track. It might have a concrete value capturing, in this instance, how points are placed in some geographic location; but it might also remain undefined, be partially defined or defined indirectly via constraints imposed by others predicates, such as axioms and lemmata. Likewise, types Point and Track may remain abstract or be populated, completely or partially, with the names of tracks and points found in some area.

The condition above states that the relation is total in its domain - it maps every point to at least one track. Informal interpretation is that every points has a valid track location. This is a set-theoretic way of writing a safety argument. One may choose to state the same in a differing manner, e.g.,

$$\forall p \in \mathrm{dom}(\mathsf{point\_tracks}) \Rightarrow \varnothing \subset \mathsf{point\_tracks}(p)$$

The choice of style is often dictated by the practicalities of the automated verification and the need for providing human-friendly reports when a rule fails.

Enumerated types, constant relations and functions define a *data set under verification*. Simplistically. verification rules define an acceptance test for data set instances. Such a test could have been implemented, in principle, as a program printing yes or no after some computation. We choose formal rules as they can be directly justified by a respective formal semantics of a data set model, are orders of magnitude more compact and deliver a more efficient implementation harnessing symbolic theorem proving and constraint solving.

The overall number of verification rules may reach several hundreds for an industrial project. The primary responsibility for defining verification rules lies with domain experts. That provide an informal description and a formalisation step is carried out by a formal methods expert. Constructing a complete set of formal rules is a demanding process that may take several years while verification itself is automatic and takes at most few minutes in our tool [19].

Over the course of several industrial projects it became apparent that in order to be systematic, rule construction requires a tool-assisted method. Such a method, assisting a domain expert by demonstrating likely omissions in the formulated rule set, is discussed in this paper. At the highest abstraction level, the method attempts to estimate the *semantic* coverage achieved by one or all of verification rules. The coverage assessment, calculated via a form of mutation testing [22], is then interpreted to reveal the data set parts that are not sufficiently constrained by the verification rules.

In this paper, to measure the semantic coverage, we attempt to generate structurally and syntactically sound changes in a data set that do not trigger any of the rules. The proportion of such changes in data that go undetected by any validation rule indicate the degree of freedom permitted by the rules. The higher such degree is, the lower is the semantic coverage of achieved by rules with respect to a given data set.

One widely recommended strategy [23] to the construction of verification rules is a top-down elicitation process starting with one or few global goals. During such a process one refines top-level goals into the corresponding low-level verification rules while presenting arguments that all the necessary rules are defined and every defined rule addresses some meaningful goal. A well known example of that can be found in [37].

Linas Laibinis is with Institute of Computer Science, Vilnius University, Didlaukio 47, LT-08303 Vilnius, Lithuania (e-mail: linas.laibinis@mif.vu.lt).
Alexei Iliasov is with The Formal Route Limited, 32A Woodhouse Grove, E12 6SR, London, UK (e-mail: alexei.iliasov@formal-route.com).
Alexander Romanovsky is with The Formal Route Limited, 32A Woodhouse Grove, E12 6SR, London, UK, and also with School of Computing, Newcastle University, NE1 7RU, Newcastle upon Tyne, UK (e-mail: alexander.romanovsky@ncl.ac.uk).

The elicitation process may also be completely formal and carried out as a development of an hierarchy of formal specifications linked by a formal refinement or simulation relation [32]. This is a rigorous approach that requires a high level of expertise and a substantial time investment. Another, often overlooked aspect for real-life applications of formal models is the need to overcome relatively poor legibility of formal notations to be able to communicate and convince stakeholders in the adequacy of an elicitation exercise [38].

We have attempted the elicitation of railway signalling verification rules but failed both in the application of formal refinement and informal safety case construction [19]. Our experience suggests that formal models are too resistant to changes necessary to accommodate missing requirements. On the top of poor legibility, early technical decisions on data structures and a formalisation style, influenced by the provability and tool usability concerns, make it difficult to enact substantial changes. In other words, a formal model-based notation is a poor medium for persisting details of a live, ongoing elicitation process.

The main intuition behind the Goal Structuring Notation (GSN) [13] and many similar methods is that a statement can be decomposed into several more detailed sub-statements, which, combined together, would establish the validity of their parent. The reality proves much messier. First, it is common to be faced with several possible decomposition dimensions with unclear consequences of pursuing either. Second, it is normally impossible to formally check the logical entailment between decomposing sub-statements and their parent statement. Finally, it is difficult to maintain a neat tree structure with many connections going across different branches and layers.

In the search for an alternative, we turned to a bottom-up approach based on genetic programming and it has proven itself to be more suitable to go hand-in-hand with rule elicitation without requiring an extensive separate development stage. The technique successfully addresses the following main important concerns: *identification of missing* verification rules, *detection of incorrect* rules, and *planning of effort and estimate* of overall number of rules required.

We rely on genetic programming to solve the problem of finding the data changes that are more likely to remain undetected by verification rules. Among such changes we expect to find cases indicating issues with the rules themselves. Random data exploration is not an option here as the a number of potential changes is practically infinite while the ”interesting” subset that needs targeting is likely to be is extremely small.

The main contributions of the paper are as follows. First, we put forward a method for systematic and automatic assurance of the attained coverage of formal verification rules constraining a data set, illustrated with a railway signalling configuration data. Second, as the means to this end, we propose a novel statistical testing technique based on a combination of genetic programming and mutation testing.

The paper has seven sections. Section II gives a brief overview of various background domains and methods relevant to this work, and the process of rule-based verification of signalling data sets in our SafeCap platform [16]. It also introduces the running example that we use in the following sections and briefly discusses relevant related work. The technique of automatic data set mutation (adopted from genetic programming) is described in Section III. The next section presents our core contribution: the methodology for mutation-based rule validation. Section V illustrates the validation process on a case study from the railway domain. Finally, Section VI discusses open questions arising when applying our approach, while Section VII presents some concluding remarks and possible directions of future work.

II. Background

A. Validation of railway configuration data

The SafeCap Platform is a toolkit for modelling railway capacity and verifying railway network safety [16]. It helps signalling engineers to design stations and junctions relying on a domain specific language (the SafeCap DSL), as well as to check their safety properties and to evaluate potential improvements of capacity by using a combination of theorem proving, SMT solving and model checking [17]. The platform has been substantially extended by adding the support for representing a wide range of the existing signalling frameworks [19], [20].

The SafeCap verification and proof back-ends enable automated reasoning about static and dynamic properties of railways and their signalling. The two principal verification routes are the built-in prover backed by a SAT solver, a range of external provers provided via the Why3 framework [8], and the ProB model checker [27] (used just as a constraint solver).

We have been collaborating extensively with the railway industry on developing automated verification solutions for signalling designs. The main motivation is to establish safety standard compliance and to replace expensive and time consuming manual checking of control tables and signalling data.

Signalling is central to the safe and efficient operation of a railway. It enables higher network capacity through higher train speeds and shorter separation distances, and prevents unsafe train movements and equipment operations. Signalling controls the movable infrastructure to set and protect a train path during train movement. At the heart of any signalling system there are one or more *interlockings*. These safety-critical devices constrain the authorisation of train and infrastructure movements to prevent unsafe situations.

The increasing complexity of modern digital interlocking, both in terms of geographical coverage and functionality, poses a major challenge to ensuring railway safety. Even though formal methods have been successfully used in the railway domain (e.g. [6], [5]), their industrial application is scarce. SafeCap offers an industry-strength verification approach that does not require engineers to learn mathematical notations and can be applied to real-life stations providing user-friendly reports within seconds.

Two safety principles are in the core of all signalling operations. The first is the protection of movable equipment (points, diamond crossings) with the aim to avoid derailment and equipment damage. The second is avoidance of train collision. In practical applications, an engineer follows the existing standards prescribing how a certain signalling technology must be realised in order to uphold these principles. It is common in railways to introduce extra assurances to contain isolated violations of driving rules or malfunction of signalling equipment. The two examples are train flank protection (commanding of a point to divert any unauthorised moves away from a set route) and provision of overlaps so that a train can overrun past a stopping point without causing a collision. The already numerous rules establishing safety principles often come in a conflict with the rules introduced for achieving best performance. Hence, under certain circumstances one may remove flank protection of a route path or reduce the overlap length in order to free up a busy point. Formalising such rules is not an easy task. Our experience shows that one needs hundreds of distinct verification statements in order to achieve acceptable coverage. A smaller set rules would be sufficient to establish the principal safety concerns but it would also result in an unacceptably large number of false positives.

It is common to represent railway interlocking and signalling logic as static data, such as tables. This is a standard practice in the UK and most European countries (i.e., the UK Railway Group Standard GK/RT 0202 [34]). For deployment, the data is translated, preferably in a well-established correct-by-construction manner, into low-level commands used by the specific signalling equipment.

Signalling data is interpreted in a context of some track topology that captures concepts such as track connectivity, directionality, track gradient and length. These concepts are overlaid with abstract topology comprising routes, overlaps, sub routes and sub overlaps (paths in a graph) and sections (sub graphs of topology graph) as well as configuration information about the track-side equipment.

SafeCap can automatically verify signalling control tables by expressing a control table data as a collection of typed sets and relations. A similar approach is taken for the conceptual representation of the layout graph. Once expressed using a common mathematical basis, a layout and a control table can be checked against each other using the formulated verification rules. In SafeCap, a verification rule is written in a combination of the first order logic and set theory and is a formal embodiment of an informal safety argument.

To define formal semantics of control tables, we rely on a simple and versatile mathematical representation based on the Zermelo-Fraenkel set theory. Every entity of the theory is either an empty set $\varnothing$ or a set of the form $\{s_1, \ldots, s_n\}$, where s_i are some valid sets. An ordered pair of two elements a and b is called a *maplet* and denoted as $a \mapsto b$ or (a, b). A binary relation is defined as a set of such maplets (pairs). The standard relation operations, calculating its domain and range, are defined as $dom(r) = \{a \mid \exists b \cdot a \mapsto b \in r\}$ and $ran(r) = \{b \mid \exists a \cdot a \mapsto b \in r\}$. An image of a relation r over a set s is written as $r[s]$ and is defined as $r[s] = \{b \mid a \mapsto b \in r \wedge a \in s\}$. Finally, the inverse operator r^{-1} constructs a relation with all maplets flipped around, while the operator $f \dagger h$ is functional override of a relation f by a set of maplets h, defined as $f \dagger h = \{a \mapsto b \mid a \mapsto b \in f \wedge a \notin dom(h)\} \cup h$.

Formulating different constraints on relations allows us to introduce different kinds or types of relations such as total or partial relations, functional relations (functions), injective, surjective, or bijective functions and so on. For instance, a function is a relation satisfying the functionality constraint, i.e., each element from its domain is mapped to just one element from its range. An injective function is a function, the inverse of which is also a function, etc.

Sparse and dense sequences may be interpreted as functions from the index type into the associated value type. Trees and graphs are encoded as connectivity or parent-child relations.

A control table, a form of data set under verification, is represented as collection of named relations. The information about the kind or type of each constructed relation is defined and maintained in the system. That is, each relation is statically typed: at all times we know the sets containing relation domain and range, the totality of its domain and range, and functionality and injectivity of the defined mappings, etc.

B. Running example

To illustrate various railway concepts and to help in the discussion of the presented method, we use a synthetic running example. The example is a very small junction with two points and six signals. It is a simplified representation of the typical "fly-out" junction found as a part in most real-life junctions. Compared to a real-life system, it lacks non-main signals, overlaps, TPWS/AWS markers, distant signals and etc. However, it is sufficient system to demonstrate the essence of route-based interlocking and its validation.

The example layout is given in Fig. 6. In the presented layout, the traffic may flow from left (boundary node A) to right, splitting on point P101 to continue towards boundary nodes B and C. It may also flow from right to left but only from the top right corner of node C and towards node A. The layout is delimited into *routes* by *signals*. For our purposes, it suffices to treat a signal as a marker identifying route start and end points. A route name typically starts with the prefix `R` and followed by signal index (e.g., 10 for `S10`), path letter A-Z, route class ((M) for main, (S) for shunt, (C) for call-on and so on) and, optionally, route sub-class (permissive or non-permissive). Hence, a main route from `S10` towards `S12` would be called `R10A(M)`.

From signal `S12` there are two possible routes - one ending at S24 is called `R12A(M)` and the other ending at S14 is `R12B(M)`. Besides routes and signals, important elements of a layout are *track sections* – named sub-graphs equipped with the facility to report the presence (but not the number) of any trains, and *points* - tri-state (normal, reverse and middle) elements defining the currently available layout topology. Whenever a route goes over a point, the point state must be such that the route path is a sub-graph of the current layout graph. For instance, for point `P101`, the normal state enables connection to track section `TAE`, while also disabling one to TBE. Hence, the point must be *commanded* normal when setting route `R12B(M)`. The middle state means that the point is not *locked* in either direction (or is still moving) and a train over such point is likely to derail.

With route-based signalling, a train may proceed into a protected area only after a route is set and the signal proceed

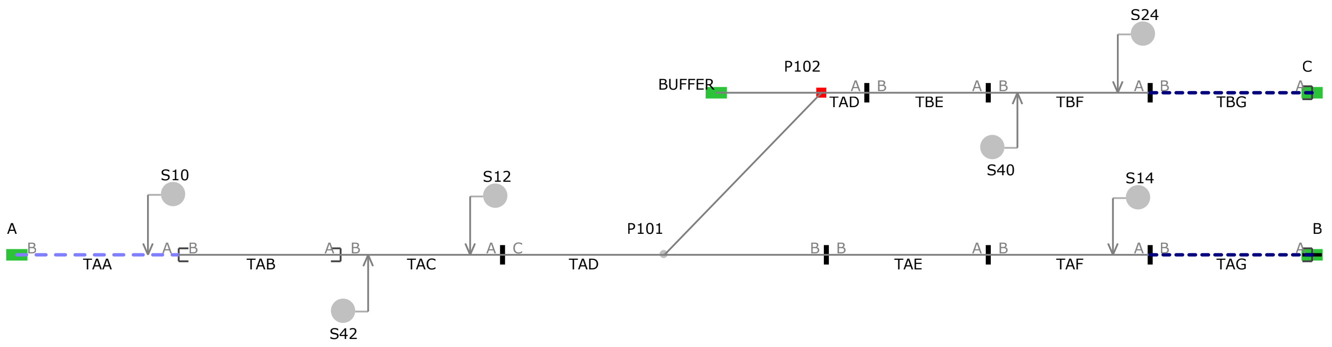


Fig. 1. Signalling diagram for a synthetic running example.

conditions are established. Route setting is typically requested by a system outside of the signalling scope. The signalling logic ensures that any such request is fulfilled only when it is safe to do so.

There is a dedicated table in signalling control tables defining the conditions for route setting. The following is a greatly simplified example of such a table.

Route	Points normal	Points reverse	Tracks clear
R10A(M)	-	P101, P102	TAB, TAC, TAD, TBE, TBF
R10B(M)	P101, P102	-	TAB, TAC, TAD, TAE, TAF
...			

To set, as an example, route R10B(M) starting from signal S10 and proceeding to signal S14, one needs to check that points P101 and P102 are commanded and detected normal; one must also check that the track sections TAB, TAC, TAD, TAE and TAB are clear. A full control table defines detailed controls for every route, signal and point as well as every sub-route and sub-overlap.

A control table must be compliant with the safety principles set out in relevant standards. One such principle, derived from the absence of derailment due to unlocked or moving points, states that the points over the route path must be commanded to align with this path. It is also necessary to command any flank protection points away from the route path and, possibly, command overlap points.

The example is not specific to any layout or control table – it is a generic principle applicable to all the cases that use a compatible technology.

To aid in formalisation of verification rules, we have developed a controlled natural language for the formulation of a condition to be verified. Such a statement emphasises the static nature of verification and enforces certain structure.

The semi-formal statement for the point setting rule takes the following form:

```
[for]
    every route
[it holds that]
    all route points are commanded
    to enable the route path
```

Note that we have not yet included the details about flank protection and overlaps. Although such oversight is obvious in the example, it may be less obvious in a real-life setting.

To state a formal counterpart of the above, we need to formalise a number of concepts. In this case, they are constants relations derived from layout and control tables:

1) Route and Point are enumerated sets of all routes and points;
2) $\mathsf{l\_normal}, \mathsf{l\_reverse} \in \mathsf{Route} \to \mathbb{P}(\mathsf{Point})$ are constant functions returning sets of points that must be set normal (l_normal) and reverse (l_reverse) to enable a route path;
3) $\mathsf{ct\_normal}, \mathsf{ct\_reverse} \in \mathsf{Route} \to \mathbb{P}(\mathsf{Point})$ are constant functions containing the points defined in the columns ”Points normal” and ”Points reverse”.

Here, l_normal and $l\_reverse$ are *constant* relations automatically derived from the layout graph in Fig. II-B. Hence, we contrast a layout topology with the information present in control tables.

Now it is straightforward to write a formal predicate capturing the semi-formal statement – all we need to do is check that sets l_normal and l_reverse are contained in ct_normal and ct_reverse, respectively:

$$\forall r \in \mathsf{Route} \;\Rightarrow\; \mathsf{l\_normal}(r) \subseteq \mathsf{ct\_normal}(r) \wedge \\ \mathsf{l\_reverse}(r) \subseteq \mathsf{ct\_reverse}(r)$$

The rule is syntactically valid and well-typed. Hence, it can be given to the SafeCap verification engine for automatic verification. In this case, verification completes with a definite result of no violations. It would appear that the task of formally verifying this safety condition is successfully achieved. Yet, as we shall later, the rule has critical flaws and does not, in fact, provide any sensible safety check.

To discover why this rule is broken, we shall have to look at how the formulated predicate reacts to mutations in data sets representing control tables.

C. Related work

Formal railway data verification is a popular research area given the safety critical nature of the domain. Some of the most relevant examples are the Ovado tool that relies on a B-like notation and the ProB model checker as the verification back-end [3]. The work [26] illustrates its applications to the validation of data sets of railway assets rather than signalling. In contrast, our work emphasises automated verification of the safety critical part of signalling with an aim to offer certification without manual review. Simulation is a popular way to validate formal models and also widely used in the railway domain to validate control tables. We see our technique as complementary to possible simulation solutions.

In our work on a verification model we take much inspiration from D. Bjørner's "Domain Engineering" [7]. Much of the SafeCap internal DSL is built in this style, although we clearly could not apply all the suggested validation steps: it would quite literally take many years for a formal method practitioner to learn and properly represent this domain.

In [25], the authors demonstrate how the ProB model checker can be use to validate large data sets against the given properties in the railway domain. In contrast to the presented method, our approach focuses on validation of the verification conditions themselves.

There is a wide variety of techniques aimed at model validation based on error or fault injection techniques, see, e.g., the survey [14]. Hardware and software fault injection has been successfully used to evaluate the dependability of computer systems [14]. Faults representing typical abnormal situations that a system could face in run-time are injected either at the hardware or software level to check the behaviour of the evaluated target system.

A number of fault injection tools have been developed and successfully applied in industry to evaluate dependability of systems [10], [40]. The main difference between the developed techniques and our approach is that we mostly rely on formal verification, using statistical validation via mutation testing as additional assurance that our formal basis (domain model) is sound and complete.

The way we mutate data sets to statistically validate our domain models is very similar to the techniques employed by genetic algorithms, see, e.g., [4], [30], [42]. In the work [30], authors use evolving genetic algorithms to simulate fault injection attacks. However, genetic algorithms often rely on the predefined and fixed verdict functions to estimate the algorithm progress, while in our approach the domain model itself serves both as a formal basis used as a verdict verifier and a model to be checked and possibly changed.

Mutation testing of software pursues a similar goal [11], [12], [21], [22], [29] of identifying program parts not adequately covered by tests. A software mutation introduces some random change in a program text often designed to mimic a programming blunder. The decisive difference with our work is that a data set carries few a priori defined semantic constraints and hence, rather than to use heuristics for targeted error injection, we have to rely on the accumulated statistics over a large number of mutations.

The problem of validation of system configuration data is also very important in managing system components or packages constrained by their given dependencies. Over the years, many different automated techniques have been developed relying on SAT or SMT solving, theorem proving, and model checking [1], [2], [15], [28], [33]. In contrast to our approach, the focus of such methods is primarily on conflict finding and resolution, because the configuration data are typically already over-constrained.

A variation of mutation testing, called mutation proving, was developed for analyzing verification projects in the Coq proof assistant [9]. It applies a set of mutation operators to Coq definitions of functions and datatypes and then checks proofs of lemmas affected by operator application. As a result, failed proofs helped to uncover many incomplete specifications as well as a bug in Coq itself. Despite similarities, our approach additionally employs genetic programming (to generate mutation programs) and statistical evaluation to find "weak spots" in the analysed data.

In [35], a new mutation testing based process to assess formal verification tools (in the automotive and aerospace industry domains) is proposed. In particular, the applied method helped to detect defects (induced mutations) in Simulink/Stateflow based software specifications as well as assess whether the analysed tools help testers identify defects effectively. In our approach, we are mostly interested in validation of the constructed domain model in the form of a collection of formulated verification conditions.

Our developed framework relies on a combination of formal verification by theorem proving and less formal quantitative validation by mutation testing. Such a combination is also quite closely connected to recent numerous attempts to combine theorem proving and model checking, see, e.g., [36]. Most of general purpose theorem provers are nowadays using model checking techniques to test potential goals (theorem candidates) before attempting costly theorem proving. Such theorem "testing" is based on trying different concrete variable values attempting to falsify a theorem in question. In our case, we rely on a kind of statistical mutation testing, focusing on validation of the underlying formal model basis itself.

III. Data set mutation

In the next two sections we present our method for formal validation of verification rules for railway signalling data, based on a combination of mutation testing and genetic programming. Section III focuses on data set mutation and genetic program synthesis, while Section IV demonstrates how these techniques can be used to facilitate analysis and validation of the given verification rules.

In a presented scenario of verifying a data set using formal predicates, or verification rules, one of the essential problems is testing of the coverage achieved by a given set of such rules. The rule coverage may be inadequate due a combination of missing and incorrect rules. To assess this coverage, one needs to generate many data changes, verifying the whole rule set after each change. The number of all possible changes for a non-trivial data set is extremely large and there is no practical possibility of exploring any significant subset of it. Instead we try explore an "interesting" subset containing changes that we believe, due to the method of their construction, are more likely pass through the net of verification rules.

Such a subset is constructed by executing tiny programs, called *mutators*. The point of having such programs is to find a balance between the mutation depth (the number of atomic operations used to compute a single change) and the mutation width (variability in arguments of individual changes). It

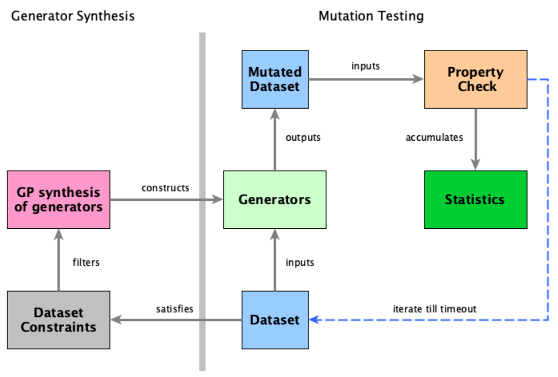


Fig. 2. The principal steps of mutation testing of the verification rules and its relation to generator synthesis via genetic programming.

would be a rather tedious exercise to manually construct mutator programs so, to facilitate this process, we use a automated synthesis procedure based on genetic programming. Formally, mutators operate over a relational data set model and hence define mutations (or translation programs) over the defined relations. The fitness function driving the evolution process depends here on a concrete set of verification rules and this makes synthesis output problem- or domain specific.

The proposed technique presumes that a considered data set is strongly constrained and the vast majority of random changes result in an error. This property must be established by domain experts before the technique can be applied.

The objective of mutation testing is to identify a discrepancy between "the vast majority" and the proportion of changes flagged as errors by the verification rules. In the absence of a better metric, "the vast majority" is taken to be 100% and the detection rate is estimated via mutation testing. The number of possible changes for a non-trivial data set is extremely large, hence we apply stochastic exploration of the overall set of data sets with the aim to obtain a representative sample of population.

The diagram in Fig. 2 shows the interaction between the mutator synthesis and mutation testing and the principal assets (depicted as boxes) and steps (displayed as arrows) of our method. The inputs are "Dataset" and "Dataset Constraints", while the output is "Statistics".

It is important to stress that the proposed technique is not designed and generally cannot be used to demonstrate complete absence of "bugs", especially "corner case bugs" in verification rules.

A. Data transformers

A mutation must ensure that a resultant data set is still a collection of relations belonging to the predefined specific relation types and kinds (e.g., functional, total, injective, surjective and etc). Note that we do not check the validity of the involved data set types, assuming them to be correct. Finding errors in the data set structure and typing is outside of the scope of this work.

We have previously explored a Monte Carlo based technique generating mutation candidates [18]. Its weakness is a high candidate rejection rate due to violations of such relation types by the generated changes. Such filtering also skews the distribution properties and makes the estimation of deviation and convergence rather tricky. It is also a very slow technique that can struggle to keep up with the speed of solvers verifying rules for the given railway data.

To address the shortcomings of Monte Carlo, we use the genetic programming to synthesise pieces of code for generating mutations. A fitness function is designed to favour mutations satisfying the relation type and kind constraints. If the resulting mutations do not satisfy the constraints, they are discarded because the fitness function gives the value 0. We also use available data sets to evaluate the synthesised code performance, thus allowing the evolution process to adapt to the data structure and constraints present in real-life control tables.

At the foundation of the data mutation method we employ there are five primitive mutation commands applicable to constant relations. Four of these commands add, remove, change and swap values in the relation range. Additionally, there is also one operation on the relation domain, which removes a value from the domain. We do not consider changing and adding values to a relation domain as such operations would require a fairly expensive solving procedure to satisfy relation typing requirements. This means that a certain class of errors cannot be detected by our technique. These are the errors of omission and duplication of an entity description. An entity here is a value that only occurs in the relation domains and this would correspond to a top-level object on a model.

Let now consider some relation $c \in Q \star R$, where Q and R are respectively the domain and range sets of relation c, while $\star$ denotes the relation kind: plain relation, function, total, surjective, bijective, injective, or some combination of these. Also, let q be some element in the domain of c, i.e., $q \in \mathrm{dom}(c)$. For ordered maps, the range R takes the form of $\mathbb{Z} \times T$ – the integer component is used to realise total order.

We then define the atomic transformers over relations as follows:

- **Replace**: replacement transformer t_m changes a value in the relation range and is defined as $t_m(c, q, r, r') = c \setminus \{q \mapsto r\} \cup \{q \mapsto r'\}$, where $r \in c[\{q\}], r' \in R, r \neq r'$ (however, it possible that $r' \in c[\{q\}]$);
- **Add**: mutation transformer t_a adds a new value to the relation range and is defined as $t_a(c, q, r') = c \cup \{q \mapsto r'\}$, where $r' \in R \setminus c[\{q\}]$;
- **Remove**: mutation transformer t_r removes a value from the relation range and is defined as $t_r(c, q, r) = c \setminus \{q \mapsto r\}$, where $r \in c[\{q\}]$;
- **Swap**: mutation transformer t_s swaps the element order when type R is a sequence and c is a function and is defined as $t_s(c, q, r, r') = c \dagger \{q \mapsto s'\}$,[1] where $s' = c(q) \dagger \{s^{-1}(r) \mapsto r', s^{-1}(r') \mapsto r\}, s = c(q)$;
- **eXclude**: mutation transformer t_x removes a value from the relation domain and is defined as $t_x(c, q) = \{a \mapsto b \mid a \mapsto b \in c \wedge a \neq q\}$.

[1]Notation $f \dagger h$ defines relational override of f with h, that is, $f \dagger h$ behaves as f if an input value is not in the domain of h and as h otherwise. Formally, $f \dagger h = \{q \mapsto r \mid q \mapsto r \in f \wedge q \notin \mathsf{dom}(h)\} \cup h$.

```
pick dom
if (total)
    add l-hop l-hop img
else exclude
```

```
pick img
swap 0 2
```

```
pick dom
if (card img > 2)
    remove 1
    remove 0
else skip
```

Fig. 3. Pretty-printed examples of trivial synthesised generators.

B. Genetic program synthesis

Genetic program synthesis works on the basis of mutation and cross-cutting of the available abstract syntax tree (AST) fragments. AST defined below represents an imperative program with limited syntax. The syntax includes the *if* block, a command sequence, and the syntax for basic expressions and predicates. To contain the degree of freedom available (that is, the number and the value ranges of parameters that can be evolved during synthesis) to the evolution algorithm, the choice of literals is constrained by using generators that yield a random number or random element from a set.

One uncommon and domain-specific part is the inclusion of commands computing a set of elements related to a given set using the topological proximity of a layout (l-hop) or the conceptual proximity of control tables (c-hop). This allows synthesis of programs that go beyond mere random change but without exposing all the complexity of underlying computations.

The topological proximity defines how far away graph elements of same type are located on a layout graph and is computed as the length of a shortest path from one element to another (see graph in Figure 4). The conceptual proximity between some two elements is zero if these are mentioned in the same column of a control table, and the shortest path on the graph (where control tables are nodes and column values are edges) otherwise.

The definitions table in Table I describe the abstract syntax tree (not concrete syntax) of a mutation program. Since such programs are synthesised by a computer, there is no real need for a concrete textual grammar also we do show pretty-printer examples later one.

The language has only two control flow constructs – sequential composition and conditional execution. There is also a simple expression language to be used as command arguments and condition predicates. The transformers (commands) are the atomic mutation transformers defined above.

The purpose of a data mutation is to define such a combination of atomic transformers that has a high chance of going undetected by the formulated verification conditions. As parameters, the atomic transformers typically take sets of values from which they choose one value randomly. Sets are computed by starting with the relation domain, range or image over a given set and then transforming such an initial set by "hopping" to topologically or conceptually adjacent values. For numeric values, the options are a small constant value, a random number generator (its range is a hyper-parameter for genetic programming) or the size of some given set.

In principle, the size of programs and individual expressions can grow indefinitely but there are mechanisms penalising "the bloat" where an extra size does not present any competitive advantage over other generated specimen [41].

generator	$\triangleq$	pick *set-literal block*
block	$\triangleq$	*if* | *sequence* | *transformer* | skip
if	$\triangleq$	*condition block block*
sequence	$\triangleq$	*block block*
transformer	$\triangleq$	mutate *set-literal set-literal* | add *set-literal* | remove *set-literal* | swap *num-expression num-expression* | exclude
condition	$\triangleq$	total | func | surj | *relational* | *condition* and *condition* | *condition* or *condition*
relational	$\triangleq$	*num-expression* op_1 *num-expression* | *set-literal* op_2 *set-expression*
set-literal	$\triangleq$	some *set-expression*
num-expression	$\triangleq$	0 | 1 | 2 | rnd | card *set-expression*
set-expression	$\triangleq$	dom | ran | img | c-hop *set-expression* | l-hop *set-expression*
op_1	$\triangleq$	$<$ | $=$
op_2	$\triangleq$	$\in$ | $\notin$

TABLE I
MUTATION PROGRAM ABSTRACT SYNTAX TREE,

Initial transformer population is seeded with `pick` dom and `pick` img programs. Command pick assigns an implicit value holding the domain element (q in the mutator definitions) used by the atomic transformers. Most of the syntax is dedicated to construction of expressions defining transformer arguments and predicates for *if*-blocks.

The language is not complete (i.e., does not allow one to express every possible generator) and has evolved considerably over a year of experiments. The two "hop" operators introduce an explicit bias towards graph-like structures so one should regard the presented syntax as an illustration of a possibility rather than a concrete recipe.

We do not attempt to remove or avoid equivalent mutators for the following two reasons. First, detecting the mutator equivalence is computationally expensive hence it cannot be integrated into the fitness function. Second, the heuristics aimed at removing equivalent mutators outside of the genetic programming framework are likely to interfere with the randomised nature of the algorithm while it would be still very difficult to ascertain their effectiveness.

To assess the quality of a synthesised generator, we generate n ($n \in 10 \ldots 200$) data changes and then apply the following fitness function to evaluate the performance:

$$f(g) = w_1 \begin{cases} 0, & \text{when a constraints is violated;} \\ w_2 \mathrm{r_l} + w_3 \mathrm{r_g} - b, & \text{otherwise.} \end{cases}$$

where $\mathrm{r_l}$ is the ratio of locally unique changes (non repeating for the same mutator), $\mathrm{r_g}$ is the ratio of globally unique changes (not repeating for the whole mutator population), $w_1 > 0$, $w_2 > 0$ are weight coefficients, and b is the

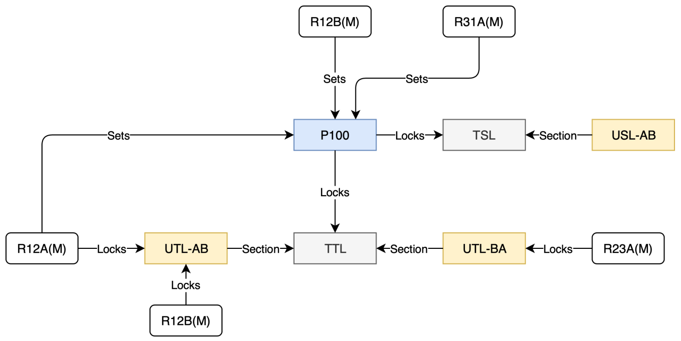


Fig. 4. Conceptual proximity graph. From R12A(M) one can reach, among other, routes R23A(M), R12B(M) and R31A(M) via paths of length 3, 1 and 1. Thus expression c-hop dom executed on a table defining R12A(M) might return R12B(M) or R31A(M).

bloat penalty discussed below. The fitness is zero whenever a relation constraint is violated for any generated change. To penalise overly complex candidates, we set execution timeout of $n * 10^{-2}$ milliseconds.

A generator program is a tree satisfying the generating grammar given above. To evolve interesting population of such trees, we use grammatically constrained graph-based mutation and crossover operations as described in [24]. While this yields syntactically correct programs that normally can be executed, it is exceedingly rare for population to contain a generator with a non-zero fitness score.

To increase the likelihood of evolving a useful generator, we employ the standard technique of probabilistic selection of mutation and crossover candidates based on a distribution defined by a mixture of manually defined heuristics and performing record derived from historic runs (the PIPE algorithm [39]). This narrows the search space to an extent where it becomes possible to observe several non-zero fitness generation in a randomly generated population of a few thousands generators.

To combat ”the bloat” (i.e., population-wide increase in program size in the course of evolution rounds), we apply a variation of covariant parsimony pressure technique [31] that penalises candidates deviating too far from the mean program size. Moreover, an extra penalty is applied for using more than two transformer operations.

Out of many rounds of generator synthesis we pick the best performing ones for each of the columns of a control table. Since there is one pick command per generator, a generator always focuses on a single column. We strive to obtain at least two generators per each column and atomic transform type Generators are synthesised once per each revision of generator syntax. It takes between 9 to 30 hours to obtain a collection achieving sufficient coverage when statistical metrics stabilise.

The diagram in Fig. 2 shows the mutation testing process flow. It starts with the synthesis of generators, as described above, and the main loop is iterative change/check statistics accumulation. The process runs until there is a certain predefined number (circa 1000-5000) distinct changes per column. The simplest form of available statistics is a measure sensitivity of a given rule to various transform kinds, e.g., sensitivity to remove transform is high if most or all remove changes are flagged as errors by the rule.

C. Running example, continued

We now apply mutation testing to the running example. The testing injects data changes solely in the control table part of the data. The relevant functions are ct_normal and ct_reverse. First, we obtain the measures of rule sensitivity with respect to these columns for each available pair of a layout and a control table. The results are then averaged over to obtain overall sensitivity. The table below summarises the findings.

Mutation kind	Sensitivity
Mutate	very low
Add	none
Remove	very low
Swap	none
Exclude	very low

The exact meaning of values ”very low”, ”low”, and ”none” shall be explained in the next section. Intuitively, we expect the absence or low sensitivity to indicate issues with a rule.

Looking at the original formulation of the rule, one cannot fail to find the conclusions surprising. The insensitivity to changing value order (Swap) is expected as the order points does not matter for a predicate that compares two sets. At the same time, no sensitivity to adding values may be worrying as it suggests the rule cannot detect any extraneous values. However, this is easily explainable by the form of the actual verification predicate – adding values on the right-hand side of a subset operator does not change the term validity.

The low sensitivity to Change and Exclude does not afford any simple explanation and suggests the rule is not achieving what it is designed to. And finally, the low sensitivity to removing values does not seemingly make sense as removing points being set ought to trigger errors. We have to conclude that, despite its simple formulation, the rule does not enforce the expected constraints on control tables. We shall analyse these results more closely in the next section.

IV. Mutation based rule validation

In this section we discuss how to apply the information collected during mutation testing to improve a set of verification rules. One aggregated result of our mutation testing procedure is the ratio of data changes flagged as errors to the overall number of changes. The two extremes of ratio values are quite intuitive. The first one, the value 0 or close to 0, signifies no reaction to any changes enacted. In this case we say that a tested rule lacks sensitivity to a given class of mutations. Often it is an indicator that a tested rule is not doing what it is supposed to do. The opposite case of the value 1 or close to 1 signifies strong sensitivity and it is what is normally desired of a verification rule.

In practice, it is rare for a single rule to address all the mutation cases and we often assess a collection of rules related by their intent. For instance, for control tables, we only insist that a conjunct of all rules referencing a given column delivers a strong sensitivity measure.

Typically, the observed sensitivity is neither 0 or 1. And since the technique is a statistical one, there is no inherent interpretation of such a ratio value. To assist result interpretation, in this work we define a simple mapping from ratio values into categorical values (so called rule ratings) and the mapping is based on our experience in the railway domain. Mutations are run till we detect the convergence of measured sensitivity. Specifically, we compare the calculated mean and variance values of a random sub-sample comprising 50% of all tests with the overall sample and test for the equality of two normal distributions.

A. Rule rating

The *rating* of a rule is an interpretation of the ratio of changes flagged as errors. Its purpose is to present a simple picture of how a given verification rule guards against certain kinds of random changes in a verified control table. We use letter codes ranging from dash mark (-), signifying none or very low sensitivity, to double capital letter (e.g., MM), denoting high or complete sensitivity. The following table gives one such mapping between the respective ratio values and ratings. This table reflects our intuition on the expected quality and coverage of control tables. For a different problem the value ranges could differ.

Ratio value	Rating
< 0.01	-
≥ 0.01 and < 0.1	m, a, r, s, x
≥ 0.1 and < 0.7	mm, aa, rr, ss, xx
≥ 0.7 and < 0.97	M, A, R, S, X
otherwise	MM, AA, RR, SS, XX

A rule rating is given with respect to a certain concept or a column. When a rule syntactically references several columns, a rating is given for each column.

From the methodological perspective, it is most advantageous to have the verification rules that achieve the highest possible rating for at least one transformer kind of some column. This gives a simple justification for the rule presence: it fully constrains one degree of freedom of a data set column. It is, however, common to have verification rules with at best a mid-level rating; then one has to expect that the remaining freedom is accounted for by some other rule.

The diagrams in Fig. 5 show a part of mutation testing feedback offered to a developer. The bar char shows the overall number mutations (1000) and the number of valid (when a rule evaluates to truth) and invalid (when it evaluates to false) cases. The radial diagram shows the number of transforms of a certain kind (gray colour) and the proportion of them detected as invalid (red). In this case, the rule is insensitive to addition and swap but is sensitive to mutation, removal and exclusion, when its rating here is `mm - r - x` or simply `mm r x` as it is clear what ratings are omitted.

B. Semi-formal contract

The ability to automatically estimate the semantic coverage of a verification rule can be used to strengthen the contract between the informal and formal parts of rule definitions.

In our experience while applying the proposed technique in a large transport solution company, we found that the domain experts could easily offer strong intuition on the expected rule rating, especially if it falls into the range of extreme values. The technique was applied in several real railway signalling projects with a number of experienced signalling engineers providing their feedback and offering their intuition on the rule rating. It is interesting to note here that this a form of the domain expertise that is very rarely captured during formal system modelling and verification.

To encode such intuition, we specify the expected rating in a semi-formal statement, as shown below.

```
[for]
    ...
[it holds that]
    ...
[sensitivity]
    {column 1}: rating
    {column 2}: rating
    ...
```

Checking of the expected rating against the actual one is a verifiable constraint linking the semi-formal and formal parts of a rule. One rating step down (e.g., `mm` instead `M`) results in a warning, two or more is an error. A higher than expected result is treated the same as a perfect match. This allows one to specify incomplete rating constraints in semi-formal statements and focus on some type transformer kinds (add, change, ...), usually the strongly constrained ones.

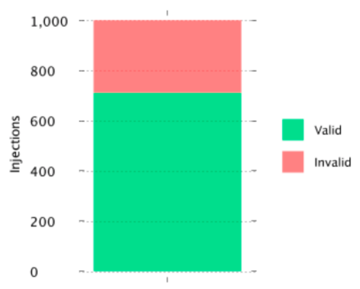
We believe such a rating contract can be a valuable methodological tool as it brings a way for a domain expert to express, at a high level of abstraction, executable tests for a formal rule.

C. Coverage analysis

To understand the current status of a rule set, it is typically not sufficient to analyse the sensitivity of each individual rule. As an illustration, say one observes sensitivity mm for a column c in a rule p_1 and, for the same column, sensitivity M in a rule p_2. There is no way to infer from this whether the column c is sufficiently constrained with respect to mutation transform types. One cannot simply add mm and M to obtain say MM: indeed, the constraint afforded by p_1 might be completely contained in p_2. Thus, the only sound way to infer the overall column rating from individual rule ratings is to use the $\max$ operator. This gives us the lower bound rating assessment.

A more relevant rating is obtained by testing a conjunct of all the rules constraining a given column and then conducting mutation testing to compute the rule rating for the column. Computing such ratings of all the columns gives the developers what we call semantic (as opposed to syntactic) coverage of verification rules for a given data set.

The global coverage shows what kind of rules are defined, suggests what rules are still missing or fail to realise a constraint intended by a domain expert. As we show later, it can be used to estimate the effort required to achieve a certain desired rating (typically `M A R - X`) by estimating the number of missing rules. This is an essential planning technique in the development of verification rules and, as far as we are aware, the only such technique available.



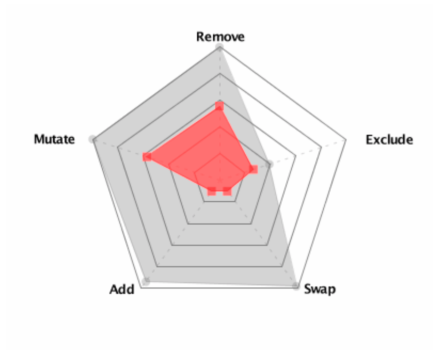


Fig. 5. A report of mutation testing of a single rule. Approximately 70% of changes went undetected (rules returning 'valid' verdict on changed data). The plot on the right shows the prevalence of mutation and removal atomic transforms in generated mutators. This is because it is easier for such mutators to pass fitness test in this particular data set.

D. *Running example*

Let us analyse the findings from the previous section. The insensitivity to adding new values suggests a missing requirement that should flag up unnecessary entries listed in the two columns. The domain knowledge suggests that very low sensitivity to Remove, Change and Exclude is quite contrary to expectations. Hence, we need to identify and fix an error in the rule or its formal predicate. The following table summarises the plan of action:

Mutation kind	Sensitivity	Action
Swap	none	add new requirement
Add	none	add new requirement
Remove	very low	fix the current rule
Change	very low	fix the current rule
Exclude	very low	fix the current rule

The first new requirement is addressed by a rule stating that the points are listed in the order they occur during route traversal from its entrance to the exit signal. No conditions are imposed on possible flank protection and overlap points.

```
[for]
    every route
[it holds that]
    route path points are given
    in the route traversal order
[sensitivity]
    {Points normal}: S
    {Points reverse}: S
```

The S rating (high sensitivity to Swap) is used for both columns. While the order is not important for some entries, we expect the number of path points (for which order is important) to form the vast majority of all entries in most cases and thus reordering should be detected in most cases.

The second new requirement tells what cannot be present in the columns Points normal and Points reverse.

```
[for]
    every route
[it holds that]
    no route points are commanded
    except those necessary to set
    the route path, common overlap
    points and flank protection points
[sensitivity]
    {Points normal}: AA
    {Points reverse}: AA
```

Note how the AA rating (almost every addition flags up as an error) aligns with the intent to address extraneous entries for these two columns.

The semi-formal statement of the original rule is now extended with a contract requiring high sensitivity to mutation, removal and exclusion.

```
[for]
    every route
[it holds that]
    all route points are commanded
    to enable the route path
[sensitivity]
    {Points normal}: M R E
    {Points reverse}: M R E
```

The intuition here is that every entry in either column is essential and uniquely correct. Hence any modification or removal would normally lead to an error. We cannot list strong sensitivity for either of these cases for the following reasons: mutation and removal may occur in the flank or overlap parts and hence not detected by the rule, while exclusion may happen for a route that does not need any path points. This suggests that extra requirements are necessary to obtain perfect overall MM RR XX rating for these two columns.

The original rule requirement has not changed apart from adding the sensitivity contract and we believe that the problem is in the formal predicate. To aid in finding the problem, we instruct the tool to present a number of mutated control tables that pass verification rule constraint. These are displayed in the following form, showing the before (struck out) and after column values:

Route	Points normal	Points reverse	Track clear
R10A(M)	~~P123~~ P126	~~P124~~ P114	TAB, TAC

After a consultation with a domain expert it become clear that the issue is incorrect treatment of *merged* points.

Merged points are several (normally two) point machines combined into one logical unit and commanded and sensed as one point. These are common as their usage reduces potential for inconsistent point states while also simplifying equipment wiring and installation.

Unlike in our running example layout in Fig. 1, there is no valid configuration requiring P101 and P102 to be set in differing states. When both are normal, the bottom line is active and is also protected from any movement from the branch (with P102 set normal any traffic is diverted towards BUFFER). Likewise, when set reverse, continuation via the top line is enabled albeit without any special protection from any traffic passing signal S14.

Now consider the two scenarios we would like to rule out. With P101 set reverse and P102 normal, no traffic can pass through the junction. When P101 is normal and P102 reverse, the bottom line traffic can flow but no flank protection is afforded from illegal movement from the branch.

By merging P101 and P102 into one logical unit where both P101 and P102 are either normal or reverse, signalling engineers rule out such undesirable configurations.

The common source of confusion is that a layout represents a topological view and one might not be unaware of merged point formations used by signalling. Control tables do define a mapping between point machines and points and we can apply this mapping to fix the rule.

The following predicate fixes the previous version by mapping between topological and logical points with a constant relation ct_pmap defined specifically for this purpose. The new version correctly accounts for merged points.

$$\forall r \in \mathsf{Route} \;\Rightarrow\; \mathsf{l\_normal}(r) \subseteq \mathsf{ct\_pmap}[\mathsf{ct\_normal}(r)] \wedge \\ \mathsf{l\_reverse}(r) \subseteq \mathsf{ct\_pmap}[\mathsf{ct\_reverse}(r)]$$

As a result, the revised version of the rule has the sensitivity ratings matching those specified in the semi-formal statement.

V. Practical experience

We have applied the presented technique to the validation of verification rules expressing the consistency and safety constraints of signalling control tables. The case study deals with real-life, industry-produced layouts and control tables and required verification rules were defined with the help of railway engineers.

A. Project description

We apply the mutation testing technique to a substantial (>100) number of verification rules validating control tables against the given railway layouts. One set of such rules implements a certain national standard and can be applied to a large number of specific cases of layouts and control table pairs. Control tables form a part of the production cycle of a SIL (Safety Integrity Level) 4 product and hence must go via thorough validation and testing stages. Since the data we use in the case study is no longer a part of an ongoing development project we expected to find only a small number of errors, with few, if any, critical ones.

A typical data set under verification is a station or a junction with around 100 routes, 60 points, and 50 signals. Control tables formatting must comply with a national standard, while control table data are given in a digital, structured format. We have analysed 12 control table types, comprising 204 columns. Some columns are split into sub-cells, so there are overall 260 constant relations capturing the control table data. This number would be the same for all control table instances as the structure remains the same throughout the analysis.

A layout is also represented as a collection of constant typed relations derived mechanically from the edge/node layout model, as illustrated in the running example.

Two data sources combined yield 370 constant relations as well as a number of shared data types such as routes, signals, tracks and etc. During the basic consistency check, SafeCap checks that there are no typing conflicts (e.g., one entity defined both as a signal and a route). Furthermore, there are 40 formal consistency checks encoding the assumptions about suitability of a combined data set. These help to uncover mismatches in versions of layout and control tables.

All together, there are 132 formal verification rules derived from 82 informal requirements. This set of rules was developed over the course of two years and is a very valuable asset. Checking its status in terms of the quality of individual rules and the attained global coverage is clearly quite an appealing goal. Since the project data sets are independently validated prior to attempted verification, there is a solid foundation for the mutation testing of rules.

B. Missing rules

The verification rule construction is guided by the railway safety principles, requirements for control tables, and domain expert knowledge. The process is informed by a certain methodology but cannot be made completely systematic and hence does not guarantee to deliver all necessary rules. Hence, it is vital to establish whether any rules were missing and what they might be before the approach based on automatic verification can itself be SIL qualified.

We have carried out investigation of missing rules for the columns presumed to be already well addressed by the existing verification rules. In particular, a domain expert could not suggest any extra rules related to these columns.

In this study, the first step was to understand the cumulative sensitivity of the rules constraining any given column. As described above, the column data is iteratively and randomly changed using the synthesised mutation scripts and then all the rules are re-evaluated on the changed data. This delivers per-column rating of a collection (that is, a conjunct) of the verification rules constraining the column data. Ideally, we expect to see maximum rating $\mathrm{MM\ AA\ RR\ SS\ XX}$.

A snapshot of the real-life mid-size project coverage is given in Fig. 7. The table in the figure depicts the measured column rating for one control table (truncated for presentation purposes) at a stage where no more rules could be suggested by the domain expert involved in the study. In other words, this is a stage of a perceived maturity where coverage analysis could be used reveal systemic problems in a formalisation.

The displayed coverage is rather poor. There are two possible explanations:

- the hypothesis about strongly constrained data set is incorrect and thus our technique reports overly pessimistic results. Conversely, the probability of idempotent mutation is significant and that is what we observe in experiments;

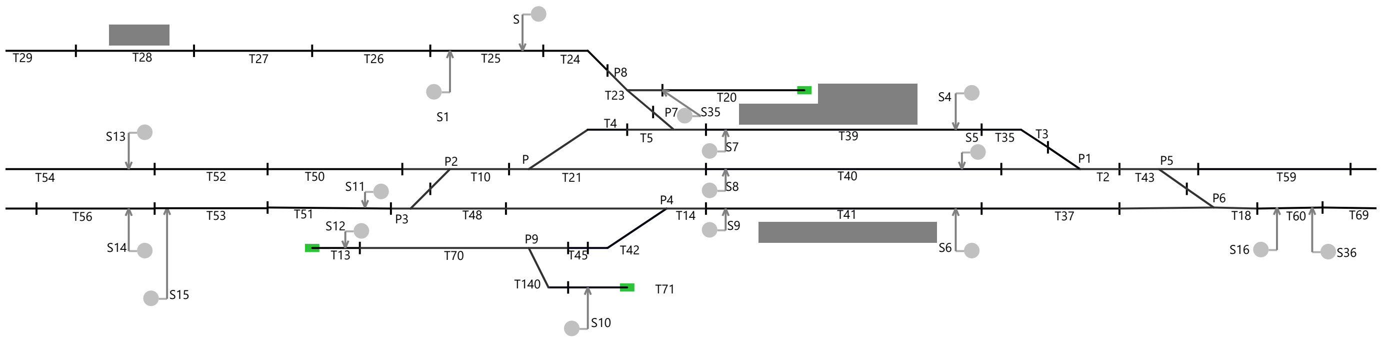
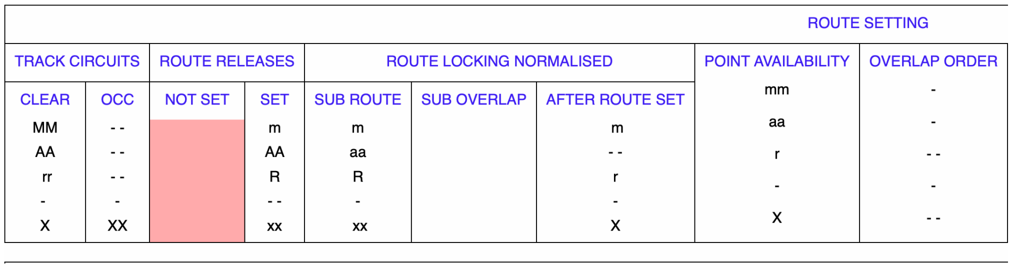


Fig. 6. Layout example, a small excerpt.



ROUTE SETTING								
TRACK CIRCUITS		ROUTE RELEASES		ROUTE LOCKING NORMALISED			POINT AVAILABILITY	OVERLAP ORDER
CLEAR	OCC	NOT SET	SET	SUB ROUTE	SUB OVERLAP	AFTER ROUTE SET		
MM	- -	🟥	m	m		m	mm	-
AA	- -	🟥	AA	aa		- -	aa	-
rr	- -	🟥	R	R		r	r	- -
-	-	🟥	- -	-		-	-	-
X	XX	🟥	xx	xx		X	X	- -

Fig. 7. Coverage diagram, an excerpt.

- there is a significant number of missing or ineffective verification rules.

To plan the next steps, it is imperative to rule out the possibility of weakly constrained control tables. For this, we have instructed our tool to print out control tables with the injected changes that are not flagged as errors by the verification rules.

A number (approximately 50) of such tables were manually analysed to determine their correctness. Almost all of them (49) were found to be incorrect. This led to the following conclusions:

1) there is a number of rules missing to address the situations that seem obviously nonsensical to a human (e.g., an entrance signal is missing for a route). However, these still must be a part of automated check;
2) most of the defined rules were asymmetric in their formulation and missing their converse counterpart. For instance, one would often check an implication of the form *when something holds for an element in a layout then a rule holds for the element in a control table*. A converse version would reverse the argument direction. One example is checking both that all the required route points are set and that no unnecessary points are listed as being set;
3) the exclusion and swap mutations were largely undetected. The former one requires extra conditions for domain checks (e.g., is there a route for which there is no table at all), while the latter one requires formulation of new requirements (e.g., are route points listed in the order they are traversed on a line of a route).

Due to a large number of missing rules required to address cases 1 and 2 above, we have added an automatic rule template generation for a given column and its rating. A template is designed to address specific weaknesses in column rating and, at the moment, we cover the cases of low mutation, addition or removal rating.

The generation templates are as follows:

- To address weak mutation rating:

$$\forall\, d, v \cdot d \mapsto v \in c \;\Rightarrow\; \text{"put your predicate here"}(d, v)$$

  where c is the column relation. The template asks to provide a rule for each element $v \in c(d)$. The intuition is that, should the value v change to an incorrect one, the predicate would flag it up as an error;
- To address weak addition rating:

$$\forall\, d, v \cdot d \in \mathrm{dom}(c) \;\Rightarrow\; c[\{d\}] \subseteq \text{"define upper bounding set"}$$

  To guard against an invalid extra value, the template asks to provide a bounding set containing all concept elements. It most cases, such a set would be indirectly qualified by a predicate and written as a set comprehension;
- To address weak removal rating:

$$\forall\, d, v \cdot d \in \mathrm{dom}(c) \;\Rightarrow\; \text{"define lower bounding set"} \subseteq c[\{d\}]$$

  This template is a mirror version of the addition template asking for a lower bounding set.

C. Rule debugging

It is fairly common to make a mistake in a verification rule. One typical case is a problem carried over from informal and semi-formal statements that cover the "normal" case but forget to cover less common but often numerous exceptional cases. This makes a rule overly restrictive, which is caught by examining verification reports. Using mutation testing, we aim to debug the rules that do not trigger verification errors. Apart from perfectly correct ones, such rules fall into a number of categories: a blunder in a formal statement making check vacuous, a mismatch between formal and semi-formal statements where the formal one covers only a part of semi-formal constraints (an artifact of incremental development), or a conceptual error that can be traced back to semi-formal and informal statements.

There were plenty of examples of either case in the project but due to space constraints we illustrate here only the first one. In this case, a simple rule is incorrectly formalised and could not react to injected changes it was expected to. The semi-formal statement is as follows:

```
[for]
    every permissive ixl route
[it holds that]
    {TRACK CIRCUITS>OCC} not empty.
```

The statement requires that a permissive route has at least one occupied track section declared in a control table.

The initial, incorrect, formal rule was given in a set theoretic style in the following manner:

$$\begin{array}{l}\mathrm{SettingTrackOccupied}[\mathsf{ixlRoutes} \cap \\ \quad (\mathrm{route.class}^{-1}[\{\mathsf{C}\}] \cup \mathrm{route.class}^{-1}[\{\mathsf{S}\}]) \cap \\ \qquad \mathrm{route.subclass}\ [\{\mathsf{P}\}]] \neq \varnothing\end{array}$$

where route.class is a constant function mapping a route name into a route class. Consequently, $\mathrm{route.class}^{-1}[\{\mathsf{C}\}]$ is a set of all routes of class C (call-on). S is another route class – a shunt class. The partial function route.subclass defines a route sub-class. In this property we are interested in shunt routes with subclass P (permissive).

Mutation testing shows that the rule is completely insensitive to any changes in column {`SIG & ASP>ROUTE SETTING>TRACK CIRCUITS>OCC`} represented by concept SettingTrackOccupied. The problem is that the formal statement translated back would correspond to a rather different semi-formal statement:

```
[for]
    any permissive ixl route
[it holds that] ...
```

The rule would only trigger an error if all the routes (a half hundred of them) had the said column empty. This is a state that is never, in practice, explored by mutation testing. The correct formal statement checks the non-emptiness condition for every route.

$$\begin{array}{l}\forall r \cdot r \in \mathsf{ixlRoutes} \wedge \\ \quad r \in \mathrm{route.class}^{-1}[\{\mathsf{C}\}] \cup \mathrm{route.class}^{-1}[\{\mathsf{S}\}])\cap \\ \qquad \mathrm{route.subclass}\ [\{\mathsf{P}\}] \\ \quad \Rightarrow \\ \quad \mathrm{SettingTrackOccupied}[\{r\}] \neq \varnothing\end{array}$$

The revised rule has `r XX` rating that is in line with the expectations.

D. *Effort planning*

Without a top down elicitation process in place, there is no immediate progress indication for the verification rule construction process. Despite this, effort planning is essential for industrial projects.

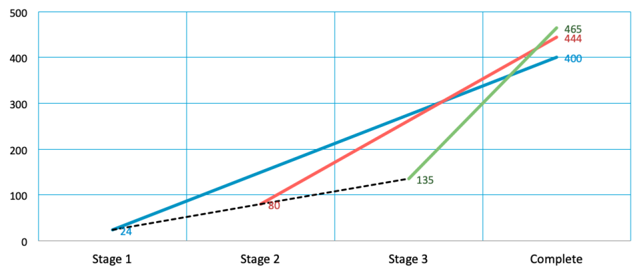


Fig. 8. Extrapolation of rule construction from three different points. The predicted overall number of rules required to achieve the complete coverage is about 500. The blue, red and green lines are extrapolations built at different project stages. By plotting the trend line against time, one can get an estimate of the overall time to define properties required to achieve complete coverage.

The challenge of effort planning can be attempted at different levels of specificity:

- Establishing, early in a project stage, whether a verification project is feasible in principle. For instance, if 20-30 already constructed rules yield no measurable coverage, it should ring alarm bells;
- Estimating the amount of effort required to attain a certain coverage threshold, e.g., 300 rules to attain the 80% coverage requiring circa 400 man/hours;
- Planning the effort within a column to attain the maximum rating.

The principle behind effort planning is extrapolation of a record of the rule construction effort using the data obtained from the coverage analysis. Via sub-sampling of a set of constructed rules, we can estimate the deviation and plot confidence intervals for further predictions. The diagram in Fig. 8 depicts three extrapolations constructed at different project stages. Stage 1 is the point when the project reached 24 rules, Stage 2 – 80 rules and Stage 3 – 135 rules. Each successive stage includes all the rules of the previous stage plus a number of additional rules.

In this projection the prognosis is circa 500 rules to attain the 80% coverage. It is prudent to presume it is an optimistic estimate as commonly the simpler principles are formulated first, while the ”trickier” ones are left for later.

VI. Discussion

As railway signalling is typically required to satisfy SIL4 requirements, there is a number of exhaustively tested and verified data sets, which can be used to establish the ’ground truth’ for rule testing. As a result, any deficiency in the coverage is vastly more likely attributed to missing or inadequate rules. In other words, for an already validated data set, it is extremely unlikely that a data set mutation would correct an existing error and it is relatively likely that a mutation would make a data set incorrect.

The manner in which a mutation is performed is controlled by the given verification rules, e.g., a mutation tries to generate mutants that trigger the verification errors. Such errors in turn lead to possible corrections made in a set of rules (with the approval of domain experts). Thus, in our approach mutation testing itself is the main technique guiding rule development.

A question arises whether one may end up in a degenerate situation where the described process results in mutants that guide towards a ”worse” version of verification rules. In our work, ”worse” has concrete, quantitative meaning expressed in the terms of lesser coverage and constrainment. Therefore, a degenerate situation is the one where mutation testing suggests a change that would make the verification rules somehow less differentiating or target a smaller part of a data set. However, in our method such an outcome is rather impossible as any changes are prompted by low rule sensitivity (i.e., with a goal of increasing rule sensitivity), thus leading to increasing the semantic rule coverage.

It is entirely possible to have a combination of the verification rules that cannot be improved by our technique. One reason is the high semantic complexity of a data set that makes difficult generation of useful mutants. Another one is that, with the increase in the semantic coverage attained by the verification rules, it becomes more difficult to find an (ever smaller) subset of incorrect values missed by these rules.

If we were to encode a typical railway signalling data set in a list of Boolean values, the list would have approximately 40000 entries. Clearly, 2^{40000} is well beyond any Monte Carlo style algorithm. Our technique works because a data set is naturally constrained and respecting these constraints hugely improves the odds. This however is still not enough to be able to rely on Monte Carlo. The conducted computational experiments show that respecting the data set constraints reduces the exploration space to circa $2^{40} - 2^{200}$. Within this space many point pairs are equivalent (i.e., all rules react in the same way to all such points) and many points are not interesting (all the rules report correct result). Mutant generation attempts to reduce the search space and hit interesting points more by tuning the given mutation rules using a fitness function defined over the verification rules.

One critical issue is whether the method can be safely transferred to a different setting. We believe we have a reliable applicability test that would indicate whether useful results may be obtained for some pair of data set and constraining rules. The applicability test entails computing the data set coverage, followed by picking and removing any rule intuitively deemed important, and then recomputing the coverage metric. The metric should indicate lesser coverage. If the metric has not decreased then the method should not be applied.

As the main limitation of the technique, we see the prerequisite of a strongly constrained data set. It is an open question how one can establish this efficiently when it is not a given fact. Mutator synthesis is also something that is not guaranteed to be successful for every application. It is not impossible that it will fail for data sets with highly complex structure. Finally, the interpretation of sensitivity values and their mapping might have to be changed in a different application setting.

The prevailing practice in validation of verification rules in railway is based on a manual review of such rules by domain experts. Our method is not aiming to replace such a validation process but rather to extend or improve it by adding steps of automated validation based on mutation testing, the goal of which is to identify or suggest to domain experts the potentially weak or incorrect rules. On the other hand, using the traditional completely formal techniques for proving that the set of verification rules is sound and complete is rather expensive in terms of time and effort required as they need a full and verified formal railway domain model.

The proposed approach can be seen as a compromise between the two extremes, which can give the developers a tool and experiment supported argument on the adequacy and completeness of the compiled verification rules, even if this argument can never reach absolute terms afforded by formal techniques, where derivation of such rules relies on a complete formal model of the problem domain. Crucially, our technique gives an unambiguous direction towards improvement of the given rule set, especially when the coverage is still fairly low. In particular, a tautology or a rule, which is subsumed by the other existing rules, would not improve the coverage and thus can be removed. A contradictory rule is easy to detect with all the metrics maxing out, while an overly restrictive rule would affect many sensitivity metrics and produce a big jump in the coverage. With the coverage increasing, it takes longer to collect meaningful statistics on the relative coverage change and hence there will be a point where it becomes impractical to run such a computational experiment. However, this very difficulty to improve the coverage would indicate a high quality of the resulting set of verification rules with respect to both their consistency (soundness) as well as completeness (coverage).

VII. Conclusions

This paper proposes a mutation based validation technique that guides domain experts in the construction or modification of the required verification rules. While the paper is based on a study in the railway domain, we do not see why the same technique cannot be applied in another setting of configuration data verification. Data-rich systems are common and assurance of correct data model instances is essential for the certification of safety critical applications.

In our previous work [19] we have developed a similar approach based on the Monte Carlo method. However, the application of of this method did not scale up to the complexity of real industrial systems. The work presented in this paper significantly improves on the earlier approach by exploring an alternative avenue based on genetic programming. This made it both more practical and scalable. The presented case study serves as a good illustration of that.

To the best of our knowledge, our work is the first approach aiming to help the developers to evaluate the semantic coverage of formal verification rules. Unlike code mutation testing, our proposed technique mutates models under verification represented as a collection of formalised typed relations. To improve the efficiency of generating the well-formed data mutations, genetic programming methods are used to synthesise the mutation programs themselves.

The automatic check of sensitivity rating acts as a form of an enforceable contract between semi-formal and formal parts of a rule and this opens novel development opportunities. We would like to collect usage statistics to see whether the introduction of rating in semi-formal statements has a measurable impact on productivity and quality of formalisation.

From our experience of applying the technique to an industrial case study, the categorical sensitivity values are well understood and accepted by domain experts. More case studies should help to assess if there is a more effective way to communicate the mutation testing results to various stakeholders (signalling engineers and managers, certification experts, formal method experts).

In our future work we will continue to improve the tool support and the feedback offered on the verification rule coverage as well as to seek to deploy the approach in industry and in differing problem domains. In a longer term, an exciting and challenging objective will be to develop an approach to automated synthesis of missing verification rules.

Acknowledgements. The work was partially supported by the EPSRC/UK STRATA platform grant. We would also like to thank the reviewers for their very valuable comments and suggestions that helped us to significantly improve the paper.

References

[1] P. Abate, R. Di Cosmo, J. Boender, and S. Zacchiroli. Strong Dependencies between Software Components. In *Symposium on Empirical Software Engineering and Measurement*, ESEM 09, pages 89–99. IEEE Computer Society, 2009.

[2] P. Abate, R. Di Cosmo, R. Treinen, and S. Zacchiroli. Dependency solving: A separate concern in component evolution management. *Journal of Systems and Software*, 85:2228–2240, 10 2012.

[3] R. Abo and L. Voisin. Formal Implementation of Data Validation for Railway Safety-Related Systems with OVADO. In *SEFM Collocated Workshops on Software Engineering and Formal Methods, LNCS 8368*, pages 221–236. Springer, 2014.

[4] M. Affenzeller, S. Winkler, and A. Beham. *Genetic algorithms and genetic programming: modern concepts and practical applications*. New York: CRC Press, 2009.

[5] 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.

[6] 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.

[7] D. Bjørner. *Domain Engineering: Technology Management, Research and Engineering*. JAIST, Japan, 2009.

[8] 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.

[9] A. Celik, K. Palmskog, M. Parovic, E. Jesus Gallego Arias, and M. Gligoric. Mutation Analysis for Coq. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 539–551. IEEE Computer Society, 2019.

[10] S. K. Cha, M. Woo, and D. Brumley. Program-Adaptive Mutational Fuzzing. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2015.

[11] T.T. Chekam, M. Papadakis, Y. Le Traon, and M. Harman. An Empirical Study on Mutation, Statement and Branch Coverage Fault Revelation That Avoids the Unreliable Clean Program Assumption. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 597–608, 2017.

[12] G. Fraser and A. Zeller. Mutation-Driven Generation of Unit Tests and Oracles. *IEEE Transactions on Software Engineering*, 38(2):278–292, 2012.

[13] GSN. Goal Structuring Notation. The GSN Working Group Online. Online at http://www.goalstructuringnotation.info.

[14] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer. Fault Injection Techniques and Tools. *IEEE Computer*, 30(4):75–82, 1997.

[15] A. Ignatiev, M. Janota, and J. Marques-Silva. Towards Efficient Optimization in Package Management Systems. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 745–755. Association for Computing Machinery (ACM), 2014.

[16] 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.

[17] 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.

[18] A. Iliasov, A. Romanovsky, and L. Laibinis. Quantitative Validation of Formal Domain Models. In *19th IEEE Int. Symposium on High Assurance Systems Engineering, HASE 2019, Hangzhou, China*, pages 17–24. IEEE, 2019.

[19] A. Iliasov and A. B. Romanovsky. Formal Analysis of Railway Signalling Data. In *Proceedings of HASE – High Assurance Systems Engineering*, pages 70–77, 2016.

[20] 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.

[21] Y. Jia and M. Harman. Higher Order Mutation Testing. *Information and Software Technology*, 51(10):1379 – 1393, 2009.

[22] Y. Jia and M. Harman. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering*, pages 649–678, 2011.

[23] T. P. Kelly and J. A. McDermid. Safety Case Construction and Reuse Using Patterns. In P. Daniel, editor, *SafeComp 97*, pages 55–69. Springer London, 1997.

[24] J. R. Koza. *Genetic Programming*. MIT Press, 1994.

[25] T. Lecomte, L. Burdy, and M. Leuschel. Formally Checking Large Data Sets in the Railways. *CoRR*, abs/1210.6815, 2012.

[26] T. Lecomte and E. Mottin. Formal Data Validation in the Railways. In *Safety-critical Systems Symposium 2016*, February 2016.

[27] M. Leuschel and M. Butler. ProB: A Model Checker for B. In *Formal Methods Europe 2003*, volume 2805, pages 855–874. Springer-Verlag, LNCS, 2003.

[28] F. Mancinelli, J. Boender, R. di Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. Treinen. Managing the Complexity of Large Free and Open Source Package-Based Software Distributions. In *Int. Conference on Automated Software Engineering*, ASE 06, pages 199–208. IEEE, 2006.

[29] M. Papadakis and N. Malevris. An Empirical Evaluation of the First and Second Order Mutation Testing Strategies. In *Int. Conference on Software Testing, Verification, and Validation Workshops*, pages 90–99, 2010.

[30] S. Picek, L. Batina, D. Jakobovic, and R. Boix Carpi. Evolving genetic algorithms for fault injection attacks. In *37th Int. Convention on Information and Communication Technology, Electronics and Microelectronics*, pages 1106–1111, 2014.

[31] R. Poli and N. F. McPhee. Parsimony Pressure Made Easy. In *Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation*, GECCO '08, New York, NY, USA, 2008. ACM.

[32] C. Ponsard and E. Dieul. From Requirements Models to Formal Specifications in B. In *Proceedings of ReMo2V*, 2006.

[33] G. Primiero and J. Boender. Negative trust for conflict resolution in software management. *Web Intelligence*, 16:251–271, 10 2018.

[34] Railway Group Standards. Signalling Design: Control Tables. Available at http://www.rgsonline.co.uk/.

[35] A. C. Rao, A. Raouf, G. Dhadyalla, and V. Pasupuleti. Mutation Testing Based Evaluation of Formal Verification Tools. In *Int. Conference on Dependable Systems and Their Applications (DSA)*, pages 1–7, 2017.

[36] S. Ray and R. Sumners. Combining Theorem Proving with Model Checking Through Predicate Abstraction. *IEEE Design & Test of Computers*, 24(2):132–139, 2007.

[37] Tornado IPT Safety Case Report. Annex C: Tornado MAR Safety Case, Version 1, UK, 2004.

[38] A. Romanovsky and M. Thomas. *Industrial Deployment of System Engineering Methods*. Springer Publishing Company, 2013.

[39] R. Salustowicz and J. Schmidhuber. Probabilistic Incremental Program Evolution. *Evol. Comput.*, 5(2), June 1997.

[40] K. Sen, D. Marinov, and G. Agha. CUTE: A Concolic Unit Testing Engine for C. In *10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT Int. Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, 2005.

[41] S. Silva and J. Almeida. Dynamic Maximum Tree Depth. A Simple Technique for Avoiding Bloat in Tree-based GP. *GECCO 2003, Lecture Notes in Computer Science*, 07 2003.

[42] P. Srivastava and T.-H. Kim. Application of Genetic Algorithm in Software Testing. In *International Journal of Software Engineering and Its Applications*, volume 3, 2009.