Quantitative Validation of Formal Domain Models

Alexei Iliasov, Alexander Romanovsky
Newcastle University
Newcastle Upon Tyne, UK
{alexei.iliasov, alexander.romanovsky}@ncl.ac.uk

Linas Laibinis
Institute of Computer Science, Vilnius University
Vilnius, Lithuania
linas.laibinis@mif.vu.lt

***Abstract*—Application of formal methods to verification of well-formedness and semantic correctness of data sets from a particular domain becomes increasingly practical with the advances in automated verification tools. However, it is difficult for domain experts to understand and formulate formal verification constraints (VCs), yet much trust is invested in their validity and completeness. The paper discusses a novel validation approach based on statistical testing of VCs against pre-validated data sets. We illustrate the proposed technique using a synthetic railway example and also relate our experience of integrating the approach within a large-scale industry-based project.**

I. Introduction

Formal verification offers a reliable and comprehensive way of checking validity of a program, a system, or a data set. Quality of such verification is predicated on the quality of formal statements expressing validation constraints. There are a number of potential problems to look out for: missing constraints, over-constrained statements resulting in false positives, over-relaxed statements yielding false negatives, and irrelevant statements that never flag any errors.

Data set verification aims to automatically check whether a given data set possesses required qualities, often linked to the safety of a system interpreting such data. Once formal constraints are available, the verification proceeds by testing whether a given data set instance meets every constraint. This process is normally automated by employing automatic theorem provers and constraint solvers.

A typical, if somewhat idealistic, departure point for such an approach is the construction of a closed, "whole system" formal domain model that provides formal semantics of a system interpreting a given data set. The definitions of verification constraints (VCs) would then arise naturally from the assumptions made in the constructed whole system model.

Construction and proof of such a model is challenging if not impossible in an industrial setting. A complex domain may require a prohibitively expensive formalisation stage before any benefits are seen. In addition, industrial practice does not always reach the level of formality required to build a comprehensive mathematical model. It might require years of background work before a complete formal model is realised.

In this paper we offer an alternative solution based on statistical testing of VCs against mutated data sets. It offers weaker assurance than a fully verified formal system model, however requires much less effort on the part of a designer.

The essence of the proposed technique is in exploring how the existing VCs react to changes in a data set known to be valid. While a single random data mutation may or may not result in a detected error (as defined by the VCs), when such data changes and checks are performed en masse, certain patterns may start to emerge. For instance, we may expect to see, from a priori knowledge of domain data properties, that some data changes cause errors and some do not. Therefore, the statistical characterisation of many data change experiments coupled with some understanding of the problem domain can offer insights into the deficiencies of given VCs.

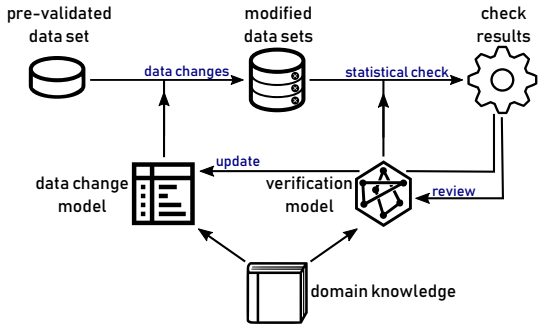


Fig. 1. Statistical testing process.

Our approach depends on the existence of data sets known to be correct. Such data sets may be already available in the form of historically verified artefacts. For instance, railway signalling data undergoe months of rigorous review by highly experienced and qualified engineers.

The approach also presumes that a data set under verification is fairly strongly constrained. That is, it is expected, for a given set of VCs and well-suited data change rules, that a data change injects an actual error with a however small but lower-bounded (non-zero) probability. To give a trivial example, a data set made of one value $x \in \mathbb{Z}$ is weakly constrained for constraint $x \neq 0$ and it is strongly constrained for $x = 0$.

There are two essential steps to setup our statistical checking procedure: the definition of data changes to be applied, and the interpretation of error detection statistics. Both steps rely on the domain expert knowledge and should be independent of the formulation of VCs.

The diagram in Fig. 1 shows the proposed validation flow. A collection of valid data sets is used to collect the statistics on how a given verification model reacts to the data changes. This statistics may consequently lead to a review of the verification model and also fine tuning of the data change model.

The remaining paper is structured as follows. Section II

describes the background of our research: our previous verification results in the railway domain, the involved mathematical basis, and the running example we use later on. In Section III we present our main contribution: the approach of quantitative validation focused on the developed domain models, while Section IV summarises our industrial experience. Finally, Section V and Section VI conclude the paper with discussion of related work and future plans.

II. Background

The presented approach is based on our previous work on the SafeCap platform – a toolkit for modelling railway capacity and verifying railway network safety [8], [9]. SafeCap aims to help signalling engineers to discover better solutions with the help of state-of-the-art computer science techniques. SafeCap speciality is automated formal verification using symbolic theorem provers and SAT/SMT solvers [10], [11]. At the core of SafeCap verification is a formal domain specific language (DSL) and a generic verification framework (GVF).

The SafeCap DSL represents different aspects of railways using graph-based structures. From these structures, graph theoretical verification statements are automatically generated and verified, including isomorphism properties between constituent subgraphs, path validity, graph connectivity, etc.

The SafeCap GVF is a SafeCap extension bringing up a mathematical notation and tools based on set theory and first order logic. GVF enables automated reasoning about static and dynamic systems in a formal manner. The verification backend relies on a combination of a built-in symbolic prover and a SAT solver, a range of external provers provided via the Why3 framework [6], and model checker ProB [14].

To define data set semantics, 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. A map is a set of form $\{a, \{b\}\}$, written as $a \mapsto b$; it defines an ordered pair of two elements. A binary relation is defined as a set of pairs. An image of a relation r over set s is written as $r[s]$ and is defined as $r[s] = \{b \mid a \mapsto b \in r \wedge a \in s\}$. A function is a relation that, for each included pair, maps a singleton set into a singleton set. 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 data set is as a collection of named relations. Each relation is statically typed: a separate document carries information about the relation domain and range sets as well as about totality, functionality and injectivity of the contained mappings.

A verification constraint is written as an expression of first order logic and set theory. It states an expected relationship between various elements of a data set, for instance,

$$\forall\, r \in \mathrm{Route} \cdot \\ route : normalpoints \subseteq ct : route : normal \wedge \\ route : reversepoints \subseteq ct : route : reverse$$

This condition implements safety property **SAF1** of the running example, as defined in the next subsection.

A. Running Example

Our running example is verification of railway signalling data on the basis of a synthetic example defined in the SafeCap platform. Signalling is central to the safe and efficient operation of a railway. It controls the moveable infrastructure by setting and protecting a train path during train movement. At the heart of any signalling system there are one or more *interlockings*. These safety-critical devices constrain authorisation of train movements as well as movements of the infrastructure to prevent unsafe situations arising.

The increasing complexity of modern digital interlocking, both in terms of their 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. [4], [3]), their industry 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.

There are two safety principles at the foundation of all signalling operations. First, protection of movable equipment (points, diamond crossings) with the aim to avoid derailment and equipment damage. Second, avoidance of train collision. In practical applications, one follows the existing standards prescribing how a certain signalling technology must be realised in order to uphold these principles. However, it is common in railways to introduce extra assurances to contain isolated violations of driving rules or malfunction of signalling equipment. Examples of that 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 of rules would be sufficient to establish the principal safety concerns but would also result in an unacceptably large number of false positives.

The diagram and table presented in Fig. 2 are the schematic layout and control table of a junction studied in this example. The layout shows the topology of a small station with five platforms. For simplicity, the layout diagram only depicts signals, track sections and points omitting speed limits, stopping points, and overlap indicators. We also use very simplified versions of two (out of 16) control tables - *Signal, Route & Aspect Controls* and *Point controls*, merged into one table.

The presented control table defines route setting conditions. A topological route path for a route of form R110A(M) is uniquely decoded from its name. In this case, it is the first (clockwise) route originating from signal S110. The suffix (M) stands for the *main* route class: a route used by general traffic

and requiring all the standard safety measures. The next three table columns define the route entry signals, as well as the sets of points to be commanded normal and reverse in order to set the route path and, possibly, enable flank protection and reserve common overlap points.

The column Track sections clear lists the sections that must be checked clear before a route set. Here we diverge from the real-life behaviour: subroute availability (paths through a track section) is checked during route setting, while track section occupation is checked by displaying the signal proceed aspect. Finally, the Routes not set column contains a list of conflicting routes that must be not set when setting a given route.

To demonstrate verification process of the chosen layout and its control table, we define three safety properties that permit rather compact formulation and require limited context in terms of verification. Two of them are related to the route setting activity and the third one speaks about conditions when a point may be commanded into a new state:

SAF1 When a route is set, all the route path points are commanded into a correct position;

SAF2 When a route is set, all trap points adjacent to route path are set away from the route path;

SAF3 When a point is commanded, all of its track sections are clear.

Property **SAF1** demands that a route must be topologically correct by the time all points are detected in the required position. Property **SAF2** is an example of a flank protection rule. A trap point is a derailing (or diverting towards a buffer stop) device, placed to protect main line traffic from unauthorised moves. Finally, property **SAF3** expresses the point movement protection.

The railway signalling information, such as control tables, should be verified against the given layout. The safety or operational principles establish the maximal set of safe signalling and verification procedures to ensure that a given signalling implementation is acceptable for a given layout. What exactly is acceptable is expressed via the verification conditions typically written as first order logic statements over the layout and signalling data. Hence, in order to conduct verification, we shall consider together the railway schema layout, control tables, and logical property formalisations.

III. Quantitative Model Validation

Statistical testing operates on the basis of repeated data changes, similarly as in error injection or genetic algorithms, followed by testing VCs against the modified data. Cumulatively, such testing provides us with a certain statistical characterisation of VCs. The data sets used for this statistical checking are presumed to be free of errors (i.e., satisfying the verification model) so that the pre-existing errors do not mask the injected ones.

The technique relies on the following hypothesis: *a data set under consideration is well constrained by the verification statements*. That is, we presume that the given VC set would react to a significant number of data changes by flagging up the changed data as erroneous.

Central to the proposed technique is the concept of model *slack* – the degree of insensitivity of a verification model to data changes. One way to quantify slack is to calculate the expected number of data changes before an error is triggered. The stronger VCs are, the less slack we expect to see. Since VCs normally formalise the pre-existing knowledge of informal data semantics, we found that there is a good understanding of expected *relative* slack (i.e., the slack in one part of data compared to another).

Let k be the expected number of data changes between error detections. Then slack, denoted as C, is $C = k - 1$.

Slack of VCs has to be interpreted by a domain expert. It turned out to be the notion that is relatively easy to communicate about: small slack values indicate "data set is well constrained, almost every change is an error", large slack values mean "data set seems unconstrained, there might be missing constraints". In our industrial applications we found that experts have strong intuition on what the slack should be for a given constraint.

By a data change we regard here a smallest elementary, semantically consistent data set mutation. Slack measurement is sensitive to the number of such elementary data changes done prior to rechecking the VCs. We refer to this as the "depth" parameter D. Our experience indicates that VCs do not exhibit slack at all when D is too small and flat-line once D is sufficiently large.

Next we will overview the steps of statistical validation and then discuss how to perform automatic data changes in an attempt to introduce errors. We then proceed with illustration of our approach using the running example, discuss how the proposed data changing process can be tailored to a problem at hand, and finally relate our experience of applying the technique in an industrial project.

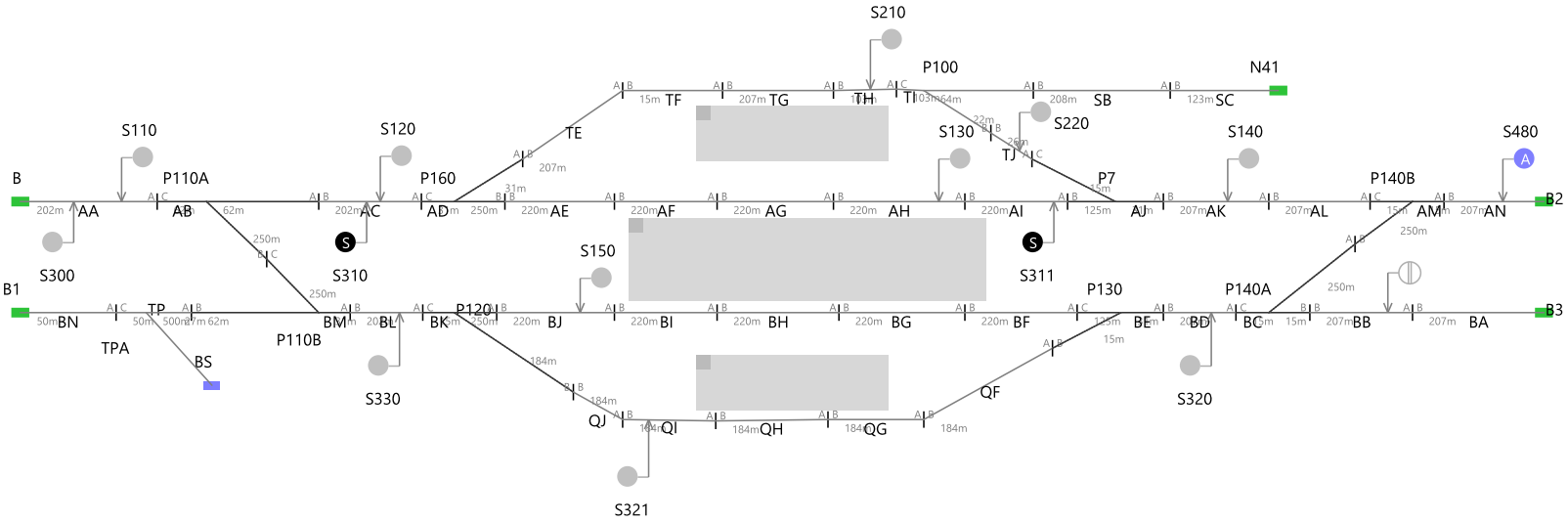
A. General Methodology

The proposed method of statistical checking consists of the following steps. One starts by assessing the overall model slack as well as the slacks related to the data set projections (e.g., columns or filtered data).

Should VCs exhibit unexplained slack, one attempts to narrow down possible causes with additional testing. To achieve this, the testing is redone with the restricted (e.g., remove elements only) or fine tuned (mutate in a certain way) data change rules to determine likely causes of such slack value. The obtained results should be validated across multiple data sets to rule out data set bias.

The findings are passed on to a domain expert who gives a verdict whether there is likely a deficiency in VCs and whether some VCs are missing or wrong. If any changes in VCs are deemed necessary than the statistical check is redone.

It might be the case that naive, random data changes are not enough to provide a meaningful report within reasonable time due to a large number of potential changes. Then one can fine tune the way changes are performed by defining custom data change rules and probability mass functions responsible for determining data change rule parameters.



Route	Entry signal	Points normal	Points reverse	Track sections clear	Routes not set
R110A(M)	S110	P110		AB, AC	R110B(M), R110C(M)
R110B(M)	S110	P120	P100	AB, BM, BL, BK, BJ	R110A(M), R110C(M)
R110C(M)	S110		P110, P120, P130, P140	AB, BM, BL, BK, QJ, QI, QH, QG, QF, BE, BD, BC, AM, AN	R110A(M), R110B(M)
Point				Track sections clear	
P110				AB, BM	
P120				BK	
P130				BE	
P140				BC, AM	
TP				TPA	

Fig. 2. Example railway schema and a excerpt of its control table.

B. Computing Data Changes

A data change (or a *transformer*) can be defined as a non-idempotent (i.e., producing a result distinct from a given input) function of the form

$$g \in \mathcal{D} \to \mathcal{D}, \qquad \mathrm{id}(\mathcal{D}) \not\subseteq g,$$

where $\mathcal{D}$ is a generic data type. Since a data set is a typed set theoretical model, such transformers must preserve typing constraints. Ignoring typing would result in data changes producing trivially incorrect and thus rejected data sets.

Consistent with the set-theoretic viewpoint, by a data set we understand a collection of named relations, each relation being a named set of mappings. Relation names are syntactically referenced in the VCs and hence should not be changed. The typing constraints also cover checking that a relation belongs to a particular relation class (e.g., a partial function or a surjection). A transformer must be change mappings in a data set without affecting its relation types and classes.

If we denote by Q the information about relation types and classes, we have that Q is an invariant property with respect to a transformer g:

$$\forall d \cdot d \in \mathcal{D} \wedge Q(d) \Rightarrow Q(g(d)) \tag{1}$$

Without loss of generality, one can perform data changes independently for each relation. Consider some relation $r \in Q \leftrightarrow R$. Then transformer t is a relation over relations: $t \in (Q \leftrightarrow R) \leftrightarrow (Q \leftrightarrow R)$. Such t must satisfy the above Condition (1), although it applies only to the part related to r. In our modelling language, there are just four potential constraints in Q pertaining to some r: domain totality, range totality, functionality and injectivity.

To perform a data change, we seek some relation t such that it preserves Q. In general, this is a difficult constraint solving problem. It is, however, much easier to find a single data change t_1 and then approximate t through iteration t_1^D, where D is the "depth" parameter defined above.

To find an instance of t_1, we use a combination of two techniques: a naive Monte-Carlo (MC) based algorithm that explores a small (typically between 100 to 1000) number of potential data change candidates. If this fails, the problem is delegated to a capable SMT solver such as Z3.

One way to define all possible t_1 is to consider some $t{+}_1$ and $t{-}_1$ that add and remove a mapping from a relation:

- *addition transformer* $t{+}_1$ adds a new mapping to some relation r: $t{+}_1 = \{r \mapsto r' \mid \exists a, b \cdot a \mapsto b \notin r \wedge r' = r \cup \{a \mapsto b\} \wedge Q(r')\}$.
- *removal transformer* $t{-}_1$ removes an existing mapping from some relation r: $t{-}_1 = \{r \mapsto r' \mid \exists a, b \cdot a \mapsto b \in r \wedge r' = r \setminus \{a \mapsto b\} \wedge Q(r')\}$.

A combination of these two may be used to incrementally build an arbitrary transformer.

Data change is realised by first picking an available transformer and then computing a changed data set by applying the transformer. The computation of a transformer instance, for instance $t{+}_1$, entails finding some a and b such that $Q(r \cup \{a \mapsto b\})$ is satisfied. As mentioned above, this is accomplished via a combination of MC and a SMT solver. The generality of the described technique makes it possible to use the SafeCap mathematical notation to define any number of custom transformers. Specifically, one may define a transformer set as a list of polymorphic relations:

```
T = [{r -> r \/ {a -> b} | a -> b /: r},
     {r -> r \ {a -> b} | a -> b : r}]
```

The condition $Q(q)$ is generated and verified automatically when a transformer is applied to some relation q.

Algorithm 1: Statistical checking algorithm

$i, e \leftarrow 0, 0$
$\mathrm{card}(cbuf) \leftarrow WS$
while $i < PMAX$ *and* $(i < \mathrm{card}(cbuf)$ *or* $|\bar{x}^w - \bar{x}| > \varepsilon)$ **do**
  $z \leftarrow d$ ▷ make a copy of data set d
  $j \leftarrow 0$
  while $j < D$ **do** ▷ iterated transformer
    $g \leftarrow \mathrm{TSFR}(T, \mathbf{w})$ ▷ pick transformer
    $z \leftarrow g(z)$ ▷ do data change on z
    $j \leftarrow j + 1$
  $c \leftarrow \mathrm{VERIFY}(z)$ ▷ run verification on z
  if $c = \mathrm{FALSE}$ **then** $e \leftarrow e + 1$; ▷ count errors injected
  $cbuf.\mathrm{add}(e/i)$ ▷ add mean to circular buffer
  $\bar{x}, \bar{x}^w \leftarrow e/i, \frac{\sum cbuf}{\mathrm{card}(cbuf)}$ ▷ update current and window mean
  $i \leftarrow i + 1$
return $\bar{x}^{-1} - 1$

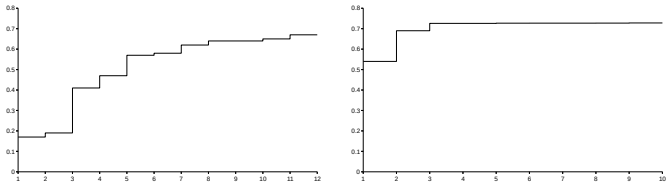
C. Algorithm of Statistical Checking

We now formulate the overall statistical checking algorithm. The algorithm is a form of the Monte-Carlo procedure and is parameterised by the overall number of potential error injections, the maximum number of points PMAX (i.e., a point is when D value changes), the stabilisation threshold ε, the depth parameter D, and the weights vector $\mathbf{w}$ determining the relative rate of transformer selection from the transformer list T. Note that the dimensions of T and $\mathbf{w}$ must be same.

The pseudo-code in Algorithm 1 illustrates the logic of the statistical checking procedure. The function TSFR uses weight vector $\mathbf{w}$ to pick a transformer from the set T. The function VERIFY runs VCs against the transformed data set z and returns TRUE if all VCs hold and FALSE otherwise. The generators $tm_1, ta_1, \ldots$ pick values using discrete uniform distributions over the corresponding enumerated types. For integer types, their uniform distribution is defined over a predefined finite interval. The procedure stops either when the maximum number of points PMAX is explored or the mean value $\bar{x}$ has stabilised, i.e., the "window" mean $\bar{x}^w$ computed over the past few values is close to the current mean $\bar{x}$. The algorithm outputs slack value $\bar{x}^{-1} - 1$.

D. Example

In order to apply Algorithm 1 to our example, we must determine several parameter values. We define the transformer list to be $T = [t{+}_1, t{-}_1]$, and the convergence window parameters to be $WS = 4$, $PMAX = 3000$, and $\mathbf{w} = (0.5, 0.5)$ ($t{+}_1$ and $t{-}_1$ are to be used at the same rate).



(a) Transformers $\{t{+}1, t{-}1\}$ (b) Transformers T_c

Fig. 3. $\bar{x}$ values computed for two transformer sets by Algorithm 1; x-axis gives depth values D, y-axis gives slack.

An optimal value of D can be determined by iteratively running the algorithm until there is no further change in $\bar{x}$. Such D can then used for all experiments with a given verification model.

Two plots in Fig. 3 shows the overall value of $\bar{x}$ plotted against D values for two different transformer sets. The custom set T_c, defined below, shows better results. Since, with T_c, there is little improvement beyond $D = 3$, this is the slack value of $\bar{x}$ we are going to use.

To demonstrate definition of custom transformers, we define a specific transformer set that targets the data changes that we hypothesised are more likely to reveal verification model coverage issues in our example:

- *mutation transformer* tm_1, which picks one mapping in a relation and then changes its right-hand side value. As an example, changing the first mapping in relation $er = \{2 \mapsto 3, 3 \mapsto 4\}$ to $2 \mapsto 2$ creates a new relation $er = \{2 \mapsto 2, 3 \mapsto 4\}$. The formal definition of tm_1: $tm_1 = \{r \mapsto r' \mid \exists a, b \cdot a \in \mathrm{dom}(r) \wedge r' = r \lhd\!\!+ \{a \mapsto b\} \wedge Q(r')\}$.
- *addition transformer* ta_1, which adds an extra mapping such that the first element is already in the domain of the transformed relation. To continue with the example above, $ta_1(er)$ becomes $\{2 \mapsto 3, 3 \mapsto 4, 2 \mapsto 4\}$. ta_1 is formally defined as $ta_1 = \{r \mapsto r' \mid \exists a, b \cdot b \notin \mathrm{ran}(r) \wedge r' = r \cup \{a \mapsto b\} \wedge Q(r')\}$.
- *removal transformer* tr_1, which removes a mapping from a relation, e.g., $tr_1(er) = \{3 \mapsto 4\}$: $tr_1 = \{r \mapsto r' \mid \exists a, b \cdot a \mapsto b \in r \wedge r' = r \setminus \{a \mapsto b\} \wedge Q(r')\}$.
- *swapping* transformer ts_1, which picks some two mappings and swaps their right-hand values: $ts_1 = \{r \mapsto r' \mid \exists a, b, c, d \cdot a \neq c \wedge b \neq d \wedge \{a \mapsto b, c \mapsto d\} \subseteq r \wedge r' = r \setminus \{a \mapsto b, c \mapsto d\} \cup \{a \mapsto d, c \mapsto b\} \wedge Q(r')\}$.
- *exclusion transformer* tx_1, which removes one element from the domain of a relation: $tx_1 = \{r \mapsto \{a\} \lhd r \mid \exists a \cdot a \in \mathrm{dom}(r) \wedge Q(r')\}$.

The transformer set $T_c = [tm_1, ta_1, tr_1, ts_1, tx_1]$ is neither complete nor minimal. However, using T_c in place of T leads to quicker convergence of Algorithm 1.

The plot in Fig. 4 shows $\bar{x}$ plotted with $D = 3$. This suggests there is plenty of slack in the example VCs. To better understand the sources of the model slack, we look at the per-concept slack values for control table columns.

The chart in Fig. 5 gives error rates ($\bar{x}$ or slack + 1) for four selected columns. The overall error rate is depicted in dark grey, while the yellow and blue colours show the error rates

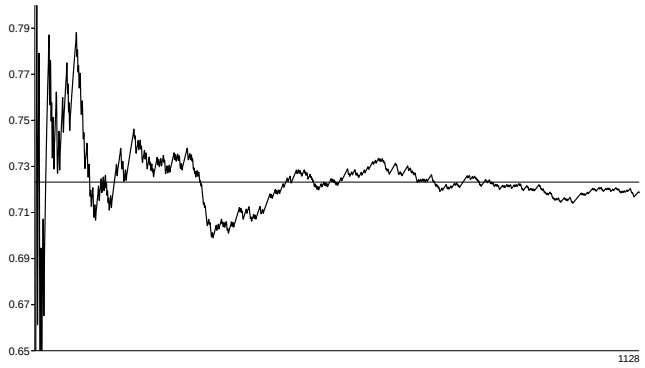


Fig. 4. Value of current $\bar{x}$ plotted against simulation step. The value converges to 0.75 giving the slack value of 0.33.

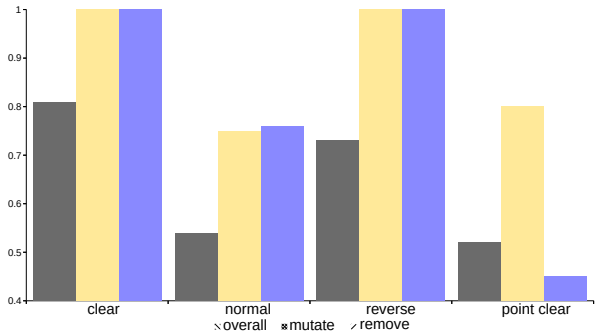


Fig. 5. Per-column and per-transformer analysis of the slack.

for transformer sets made of a single transformer – mutation and removal, correspondingly. We have omitted other transformers as they have same values for all the properties. The addition transformer never triggers errors, while the exclusion transformer always triggers errors. The chart makes clear that the *route point normal* and *point track clear* columns exhibit large amounts of the slack.

We proceed with a review of the VCs in order to improve their performance. From informal understanding of the design principles behind control tables, we conclude that the current VCs lack statements to cover the *route point normal* and *point track clear* columns. A missing statement is formulated by a domain expert who defined four additional properties focusing on the safety issues related to the identified columns:

SAF4 When a route is set, all route common overlap points are commanded in a correct position;

SAF5 When a route is set, all route flank points are commanded in a position away from the route path;

SAF6 When a point is commanded, all of its track sections are clear;

SAF7 When a route is set, all swinging overlap points conflicting with the points set by an opposing route are commanded in a correct position.

These properties are translated into a formal notation and incorporated into overal VCs. The model is rechecked using $D = 3$ and transformer set T_c. The results are shown in the charts in Fig. 6. Given the much smaller observed slack, we

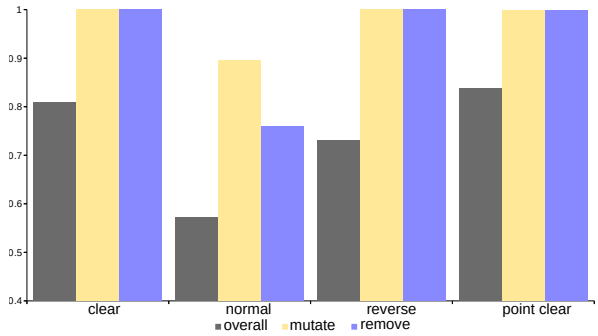


Fig. 6. Revised verification model analysis.

can now declare that the verification model demonstrates better coverage with respect to the columns we have considered.

In practice, it is more expedient to add one verification property at a time and see how statistical checking reacts to it. Adding a property that has no influence whatsoever should raise questions whether it is necessary and correct, although it might be explained by incorrect setup of statistical checking.

IV. Industrial Experience

We have applied the proposed approach in an industrial project concerned with the verification of control tables prepared according to the national UK standard. The standard mandates 16 different table types (with some degree of customisation permissible) and also includes natural language formulation of principal safety properties and control design rules. In a typical example there would be about 400 separate tables concerning *interlocking* – an area of signalling covering a station or a junction. To conduct their verification, it is necessary to combine the control table and railway schema information. Due to the way control tables are produced and retained, there are almost no natural examples of incorrect data sets. This made it difficult to assess how good the constructed VCs are, as we could only rely on a small number of artificially produced error data sets.

The original motivation for developing the proposed technique was merely to confirm the coverage of the existing VCs and then track the coverage progress while extending VCs. When the technique was first tried out, the corresponding VCs consisted of 87 formal statements. The model was created in a close cooperation with the domain experts. Its results were reviewed and confirmed by the domain experts as well.

During the first trial of statistical checking it became clear that the model coverage was far from expected. Moreover, VCs counter-intuitively did not react at all to any changes to certain columns (despite the fact they were directly referenced by VCs). They also did not react to adding new values to some tables and columns, while at the same time showed a surprisingly high slack for others. Some of these observations have led to almost immediate discovery of vacuously true statements of the form $\forall P \Rightarrow Q$, where P was over-constrained to the extent of being a contradiction, or the statements that effectively (when considered in the context of any reasonable data set) represented a contradiction.

To some extent, the errors were due to mistakes in natural language formulations. Others were incorrect translations into a formal notation. Moreover, a whole class of missing VCs were identified for various tables. Out of 87 such conditions, 21 were modified as the result of attempted statistical checks. The need for further 70 to 90 missing conditions was mapped out and the work of updating VCs is still in progress.

To apply the proposed technique effectively, one requires close assistance from an industrial partner. At the same time, it gives formal method practitioners a systematic and a well-argued way to question and discover potential gaps in VCs under consideration.

V. Related Work

In our work on verification model for railway signalling we take much inspiration from D. Bjørner's "Domain Engineering" [5]. 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.

Railway data verification is quite common given the safety critical nature of the domain. One of the most notable examples is the use of the Ovado tool that uses a B-like notation and ProB as the verification back end [1]. The work [13] illustrates applications to the validation of data sets of railway assets. 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.

Hardware and software fault injection has been successfully used to evaluate the dependability of computer systems [7]. Faults representing typical abnormal situations that a system could face in runtime 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. The main difference between the developed techniques and our approach is that we mostly rely on formal verification, using statistical validation via error injection 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., [2], [15], [17]. In the work [15], authors also use evolving genetic algorithms to simulate fault injection attacks. However, genetic algorithms often rely on the pre-defined 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 has a similar goal [12] of identifying the program parts not adequately covered by tests. A software mutation introduces a small random change in a

program text often designed to mimic a programming blunder. The decisive difference is that a data set carries a few of 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.

Our developed framework relies on a combination of formal verification by theorem proving and less formal quantitative validation by statistical checking. Such a combination is also quite closely connected to recent numerous attempts to combine theorem proving and model checking, see, e.g., [16]. 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 model checking, focusing on validation of the underlying formal model basis itself.

VI. Conclusions

The essence of the proposed idea is quite simple – to mutate a given data set to see how the considered verification constraints react. Bringing the idea to its realisation required extensive research and tooling work. In our view, the results are very encouraging: the technique is cheap to set up and produce initial results and can configured in different ways to adapt to a particular domain. The deployment experience is positive, although unfortunately we are unable to share many interesting technical details.

Statistical checking does not need to be exacting to offer useful insights. The underlying (data transformer driven) data changes can be quite coarse grained and still able to address the coverage and adequacy of VCs. In principle, a thorough specification of transformer set could completely validate the defined verification properties if one would demonstrate that such transformers generate all important data points. In practice, statistical checking would however always remain a fairly coarse check of VCs.

The proposal has its limitations. It depends on the availability of pre-validated data sets. To be successful, a practical application of the technique also requires involvement of a domain expert to interpret statistical results. On average, Algorithm 1 converges rather slowly ($O(\sqrt{\text{PMAX}})$) and there is a possibility for improvements. Finally, development and validation of the technique is based on a single domain – railway signalling – and this possibly has introduced some bias. In our future work we are going to apply our technique to validate verification of medical device configuration data.

One potential obstacle to statistical checking is the difficulty in finding a sufficient number of data changes that lead to the data satisfying a VCs. This happens when VCs are already nearly complete and thus the chances of random non-error data change are small. It may also happen if the data model is quite complex and random exploration cannot yield satisfactory coverage. As we have shown, one may construct a function to define arbitrary suitable transformer t. In practice, this is a tedious task as precise error injection amounts to creating a negated version of a complete verification model. In our view the balance lies in maintaining the probabilistic nature of t, yet providing ways to constrain its behaviour by characterising the random variables driving data changes. In future work we are going to explore how to fine tune our data transformers with custom probability mass functions driving selection of replacement/removal candidates.

Acknowledgment. This work is supported by the EPSRC STARTA project.

References

[1] Robert Abo and Laurent Voisin. Formal Implementation of Data Validation for Railway Safety-Related Systems with OVADO. In *SEFM 2013 Collocated Workshops on Software Engineering and Formal Methods, LNCS 8368*, pages 221–236. Springer-Verlag, 2014.

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

[3] Frédéric Badeau and Arnaud Amelot. Using B as a High Level Programming Language in an Industrial Project: Roissy VAL. In H.Treharne, S. King, M. C. Henson, and S. A. Schneider, editors, *Proceedings of ZB 2005: Formal Specification and Development in Z and B*, volume 3455 of *LNCS*, pages 334–354. Springer, 2005.

[4] Patrick Behm, Paul Benoit, Alain Faivre, and Jean-Marc Meynadier. Météor: A Successful Application of B in a Large Project. In J. M. Wing, J. Woodcock, and J. Davies, editors, *Proc. of FM'99 – World Congress on Formal Methods in the Development of Computing Systems*, volume 1708 of *LNCS*, pages 369–387. Springer, 1999.

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

[6] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, August 2011.

[7] Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. Fault injection techniques and tools. *IEEE Computer*, 30(4):75–82, 1997.

[8] Alexei Iliasov, Ilya Lopatkin, and Alexander Romanovsky. The SafeCap Platform for Modelling Railway Safety and Capacity. In *Proceedings of the 32nd International Conference on Computer Safety, Reliability and Security. LNCS 8135, Springer*, 2013.

[9] Alexei Iliasov, Ilya Lopatkin, and Alexander B. Romanovsky. Practical Formal Methods in Railways – The SafeCap Approach. In *Proceedings of Ada-Europe 2014, 19th Ada-Europe International Conference on Reliable Software Technologies*, pages 177–192, 2014.

[10] Alexei Iliasov and Alexander B. Romanovsky. Formal analysis of railway signalling data. In *17th IEEE International Symposium on High Assurance Systems Engineering, HASE 2016, Orlando, FL, USA, January 7-9, 2016*, pages 70–77, 2016.

[11] Alexei Iliasov, Paulius Stankaitis, and David Adjepon-Yamoah. Static Verification of Railway Schema and Interlocking Design Data. In *Proceedings of RSSRail 2016: Reliability, Safety, and Security of Railway Systems*, pages 123–133, 2016.

[12] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, pages 649–678, 2011.

[13] Thierry Lecomte and Erwan Mottin Clearsy. Formal Data Validation in the Railways. In *Safety-critical Systems Symposium 2016*, February 2016.

[14] M. Leuschel and M. Butler. ProB: A Model Checker for B. In Araki Keijiro, Stefania Gnesi, and Mandrio Dino, editors, *Formal Methods Europe 2003, LNCS 2805*, pages 855–874. Springer-Verlag, 2003.

[15] Stjepan Picek, Lejla Batina, Domagoj Jakobovic, and Rafael Boix Carpi. Evolving genetic algorithms for fault injection attacks. In *Proc. of 37th Int. Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1106–1111, 2014.

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

[17] Praveen Srivastava and Tai-Hoon Kim. Application of genetic algorithm in software testing. In *International Journal of Software Engineering and Its Applications*, volume 3, 11 2009.