2016 IEEE 17th International Symposium on High Assurance Systems Engineering

Formal analysis of railway signalling data

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

***Abstract*—This paper presents an approach to a tool-assisted analysis and verification of safety-critical interlocking and signalling data. We show how to combine multiple, non-homogeneous sources of data and render such data as a collection of mathematical entities, namely sets, relations and functions. We then demonstrate how top-level railway safety concerns can be broken down, via a combination of informal structured argumentation and formal inference, into formal verification goals suitable for automatic verification tools.**

I. Introduction

Modern railway operation is an archetypal example of a large scale information system. There is a great number of diverse but still intricately interlinked subsystems ranging from the prediction of service demand to wiring diagrams for a connection between some two pieces of equipment. New information is constantly generated while existing is altered or removed. In a way, railway operation exists in two parallel dimensions: a completely digital representation used for the analysis, modelling and prediction; and the physical reality containing actual tracks and trains. For the virtual snapshot to remain useful and reliable one has to ensure that it accurately captures the reality and provides the right tools to understand a railway operation at varying levels of abstraction. It is not possible to build such an information system by simply digitizing every possible aspect of a railway. One has to plan ahead to ensure that 'data rot' does not set in: the information database does not get polluted with outdated, conflicting or mis-attributed data.

In the paper we address a fairly narrow aspect of the problem. We look in some detail into the data defining signalling and interlocking rules of some non-trivial railway section (i.e., a station) and show how static verification, specifically constraint solving, may be used to perform rich rule-based automated verification of fairly extensive data sets. The technicalities of verification - synthesis of a formal conjecture from some schematic rule and specific data and then processing using an automated theorem prover - turned out to be the easier part of the exercise. What is far more difficult is to bridge the level at which one can state meaningful verification concerns, and the level at which concepts appear in the source data. The former are high-level properties whereas the latter are relatively fine-grained and reflect certain implementation decisions. It happens that interesting high-level properties, such as the absence of derailment, cannot be stated formally in a succinct way at the abstraction level of the analysed data. To address this we take an approach where low-level formal verification conditions are connected to high-level concerns via a mixture of formal and informal argumentation.

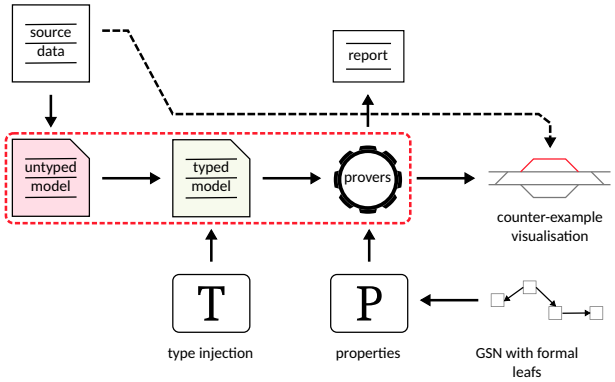


Fig. 1. The overall workflow of the proposed approach. Highlighted in red is a sequence of tool-automated activities.

In this work we use an approach inspired by the Goal Structuring Notation [18] to capture such argumentation. In essence, for each verification concern, in particular operational safety, we strive to provide a safety case that offers sufficient arguments that formal verification rules appearing in the leaves of a safety case are cumulatively strong enough to support the root goal of the case. For the most part, the construction and analysis of such a safety case relies on the domain knowledge of an expert. Around leaf nodes that encode concrete verification rules we also apply structuring based on the laws of logical inference. This allows a verification engineer to write a fairly concise if not practical verification rule that can be transcribed into a compact natural language counterpart and thus be matched with an informal formulation from which it originates. Afterwards, such a statement may be decomposed and reformulated to suit the needs of verification back-ends. At this stage argumentation is formal: every reformulation and decomposition step must be proven, via an automated theorem prover, to be equivalent or stronger than its original.

The approach has evolved in collaboration with Siemens Rail Automation UK who provided input raw data for the case study. The data defines the topology and interlocking rules of a mid-size station in UK (Reading). It is split between two data formats and defines a wide range of concepts spanning from track gradient to route signalling to equipment control. As the basis of our work we used the SafeCap Platform - an integrated modelling environment aimed at a railway signalling engineer [1], [16]. Its focus is the microscopic analysis of a railway

1530-2059/16 $31.00 © 2016 IEEE
DOI 10.1109/HASE.2016.44

node (station, marshalling yard, complex junction) with the purpose of investigation, with the maximum level of detail, the performance of a node under various service patterns, train types and stability margins.

The diagram in Fig. 1 shows the overall workflow of the approach. The source data is first transformed in a set theoretic model form, it is then typed with the assistance of user-provided typing constraints (Section IV). A set of properties developed via an extension of GSN are combined with the typed and given to automated verification tools (Section V). As a feedback, a user gets a verification report or an illustration on a schema layout of a counter-example (Section VI).

II. Railway operational safety concepts

The section presents the high-level safety concerns of a railway operation. These are universal safety goals shared by all route-based signalling operations.

A schema must be free from collisions. A collision happens (potentially) when two trains occupy the same part of a track at the same time. Reasoning about collisions must take into account concrete topology, requires an explicit train notion, the definition of laws of train movement and assumptions about train driver (either human or automatic) behaviour. Note that if train drivers choose to ignore whatever means of indication of track occupation states are available to them (e.g., track side signals) there is nothing preventing two trains from colliding. Hence, the absence of collisions is ensured by demonstrating the compatibility of specific topology, signalling and certain driving rules.

In route-based signalling, the principal safety mechanism is that of route locking and holding. A train is given permission to enter an area of a railway once there is a continuous and safe path through the area assigned exclusively to this train. Such a path is normally called a *route* and is delineated by *signals* — either physical track-side signals with lamps or conceptual signals displayed to a driver via a computer screen. Two-aspect signals (red/green or stop/proceed) are positioned at the maximum braking distance from each other and this defines the smallest train separation. Signals with 3, 4 or a higher number of aspects allows trains to come closer by advising drivers on the safe speed and the extent of free track available in front.

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

In the example in Fig. 2, signals S12 and S24 define the boundaries of route S12_S24 on a path between boundary nodes A and C. The route comprises three sections (train detection circuits): AD, BE, BF. It also contains points P and T that must be set and proven into a certain direction. Finally, the proceed rule of signal S12 protecting the route entry checks that *overlap* section BG is free. This is an extra safety measure in case a driver fails to stop in time and is often complemented by a system that automatically engages emergency braking [6].

A schema must be free from derailments. A derailment may happen when a train moves over a point that is not set in any specific direction and thus may move under a train. To avoid this, a point must be positively confirmed to be *locked* before a train may travel over it. In a control table one writes a condition defining when a point reconfiguration may happen. For the example in Fig. 2 such a rule requires that sections AD and BE are clear.

Another reason for derailment is driving a train through a curve at an unsafe speed. As a train goes over a curve, the combination of gravitational, centripetal and centrifugal forces exerts a rolling force on train carriages and a substantial lateral force on rails. This effect can be mitigated by track canting although no single canting is a perfect fit for all train types. Hence, enforcing a safe speed limit before a train enters a curved track area is an essential safety consideration. There are several ways of doing this. One is a static speed limit (sign (30) in Fig. 2). It is realised as a sign board warning a driver or an electronic signal sent to an on-board computer. A speed restriction may be also enforced by signalling: a signal does not switch into a permissive aspect until a train is detected to occupy some preceding detection circuit for a duration time. A combination of such time duration and track length gives an upper train speed limit. For instance, control rule occupied(AC : 15) for route S12_S24 requires occupation of section AC before a proceed aspect may be indicated. This equates to approximately 60mph - a slightly more conservative restriction than the 30m/s static speed limit used for the same purpose.

III. Railway data verification

It is common to represent railway interlocking and signalling logic as static data, such as tables. This is the standard practice in the UK and most European countries (i.e., UK Railway Group Standard GK/RT 0202 [15]). At deployment, the data is translated, preferably in a well-established correct-by-construction manner, into low-level commands appropriate for 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 extent, overlaid abstract topology (possible routes and extents of train detection circuits) as well as configuration information describing kind, positioning and capabilities of track-side equipment.

To simplify data query, a typical case contains a large amount of duplicate information where some part of data is completely defined by another. At times there is a one to one mapping between two structures (i.e., a schema made of track with circuit isolation points and a schema made up of detection circuit), or one part may be constructed via a semantic compression of some other data (i.e., paths of schema, routes of a schema with signals).

A proportion of such data may be rendered as a familiar visual depiction of railway track schema made of tracks, routes and signals. Another core part is a set of tables capturing point and route setting conditions (a control table). There may be further data that does not easily fit into either representation or is specific to a certain signalling solution. Railway computerisation means that of more and more of such data becomes available in a structured digital form such as XML.

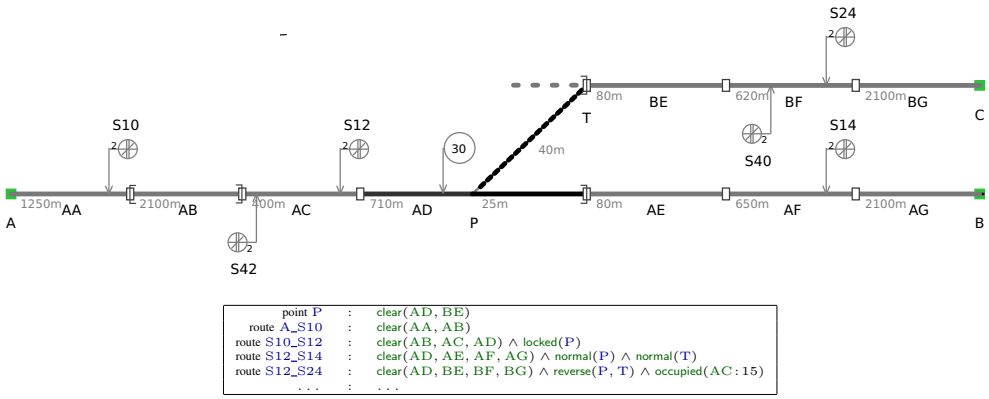


Fig. 2. A simple junction and an excerpt from its control table.

The data describing a railway operation is not created in one instance as a final depiction of reality. Even for an existing and well-tried signalling solution, the collection and systematization of relevant data is a challenging and error-prone activity. For an ongoing project, data is constantly evolving with changes committed by multiple parties. Practice standards, cross-reviews and simulations ensure a level of consistency. However, as the amount of data grows and the dependencies and inter-connections become more subtle, maintaining consistent evolution of critical data becomes increasingly challenging.

Recall the issue of data overlap and redundancy. The first indication of a problem would be a mismatch between data entries known to semantically overlap in a certain manner. Instead of constant manual cross examination and verification with a simulation tools (that may prove too lenient), one may define mechanisable verification rules that would apply automatically every time a new piece of data is added to a global database. At the simplest level it is a piece of code (say, a stored routine for a database engine) which role is to go through entries and check for certain signs of mismatch. This will undeniably save a great amount of time. However, given the scale of the challenge - a real-life example contains hundreds of distinct concepts - a new challenge arises almost immediately. Since the responsibility is now delegated to verification code, such code has to be developed, verified and maintained to a highest standard.

What could be the most promising way of constructing such code? To obtain a degree of assurance about code behaviour one needs to have a clear model of railway data semantics, a concise definition of verification rules and some means of establishing the fact that the code faithfully reproduces a given verification rule. Then provided a railway expert is able to read and critique verification rules and data semantics, the overall verification activity may be split among a software engineering performing low-level verification of code against rules and railway expert validating domain-specific concerns.

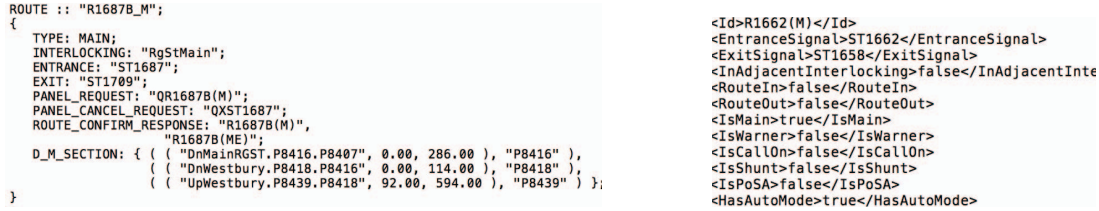
A clear downside of such approach is its costs and inflexibility: code verification is challenging on its own and once a piece of code is shown to be correct, it becomes too expensive to introduce any changes in the data semantics and verification rules.

Fortunately, a substantial subset of practically relevant rules may be ”executed” without translation into a program text. That is, a formal verification statement together with the source data may be passed to a tool to establish the validity of the statement. Automated theorems provers, such as Z3 [10], Vampire [11] and E [17], have advanced automated verification to such a degree as to making it realistic to have a black box verification solution based on a combination of automatic provers, solvers and model checkers. One historical difficulty is that such tools, given a verification problem, may return, in addition to truth and falsity, a third, ”unknown” status. Since most practically relevant logics are undecidable it is impossible in principle to build a tool that always delivers a definite answer. In this case, however, it has more to do with the practical limitations of internal proof routines, available memory and computation time. However, as we argue, we have reached the stage where even for very large models and most intricate verification rules that we encounter in the railway domain, the ”unknown” status is sufficiently rare that it becomes appropriate to treat it as falsity and resolve by adjusting the form of verification rules or face the cost of a false positive[1].

A code-free solution is already a big step in the direction of scalability. However, the amount of low-level data makes it challenging to come up with appropriate conditions. More fundamentally, we have to face the question of completeness of verification, that is, how do we know that all the relevant properties are codified in the multiplicity of formal verification rules. It turns out that no single low-level rule addresses what a railway engineer could regard as a genuine safety concern. This is not surprising as the rules are formulated at the level of ”implementation” or solution data where the strategic view is already buried and under the heap of implementation details.

To link high-level safety concerns of collisions and derailment avoidance and flank protection (Section II) with low level

[1]Which means that certain patterns of data representation would have to be ruled out to make data amenable to verification. So far we have not encountered such a situation.



```
ROUTE :: "R1687B_M";
{
   TYPE: MAIN;
   INTERLOCKING: "RgStMain";
   ENTRANCE: "ST1687";
   EXIT: "ST1709";
   PANEL_REQUEST: "QR1687B(M)";
   PANEL_CANCEL_REQUEST: "QXST1687";
   ROUTE_CONFIRM_RESPONSE: "R1687B(M)",
                           "R1687B(ME)";
   D_M_SECTION: { ( ( "DnMainRGST.P8416.P8407", 0.00, 286.00 ), "P8416" ),
                  ( ( "DnWestbury.P8418.P8416", 0.00, 114.00 ), "P8418" ),
                  ( ( "UpWestbury.P8439.P8418", 92.00, 594.00 ), "P8439" ) };
}
```

```
<Id>R1662(M)</Id>
<EntranceSignal>ST1662</EntranceSignal>
<ExitSignal>ST1658</ExitSignal>
<InAdjacentInterlocking>false</InAdjacentInte
<RouteIn>false</RouteIn>
<RouteOut>false</RouteOut>
<IsMain>true</IsMain>
<IsWarner>false</IsWarner>
<IsCallOn>false</IsCallOn>
<IsShunt>false</IsShunt>
<IsPoSA>false</IsPoSA>
<HasAutoMode>true</HasAutoMode>
```

Fig. 3. The description of the same route in LDL (left) and XML (right) data sources (abridged). The LDL style is oriented more towards visualisation while the XML part is purely logical.

verification rules we employ the Goal Structuring Notation (GSN) [18] to demonstrate, via a mixture of formal and informal argumentations, in a top-down hiearchical development, that in their entirety low-level verification rules are sufficient to establish high-level safety goals.

IV. Data preparation

In this section we describe the data preparation step. It comprises three main stages. First is the reformulation of a source data as a number of named sets of pairs. From these we extract and type mathematical structures - relations and functions - representing source data. Finally we proceed with the verification of completeness and consistency of the constructed model.

As a departure point we assume that railway data is made available in a structured, computer-readable format, such as XML, JSON, RDF or similar. Such data typically defines a number of entities each characterised by a set of properties (see Fig. 3). In our approach, the extraction of a mathematical model from a data source is mostly automatic so we rely on syntax alone and do not attempt to give any interpretation to tags used in data layout or interpret the contents of data fields.

A typical XML entity takes the following form:

```
<c>
 <a>V</a>
 <b><d>...</d></b>
 <g>Y</g> <g>Z</g>
</c>
```

It can be interpreted as the definition of an object of kind `c` with property `a` taking value *V*, property `g` being a list or a set made of values *Y* and *Z*. Property `b` is a complex definition made of other entities. Structured data, such as an XML file, is generated according to some template; in the ideal scenario such a template (that is, the abstract grammar of a document) would be formally defined as an XML Schema or a document type definition (DTD). In our case, such information was not available and also not essential due to the subsequent type inference stage and a far richer rule-based verification approach. To illustrate the point, the following could be a valid definition of another instance of `c` entity:

```
<c>
 <a>A</a>
 <b><d>...</d><d>...</d></b>
</c>
```

Here, property `g` is absent altogether while `b` appears to be a set or a sequence property. All such definitions would need to be reconciled under a common mathematical model.

In our case this is a set-theoretic representation of relations, functions and predicates. By set theoretic we mean, among other things, using the graph of relation to represent various entities of a model. For some relation $r \in \mathrm{A} \leftrightarrow \mathrm{B}$ mapping elements of set A to elements of set B, its graph is a set of all mappings $a \mapsto b$ such that $a \mapsto b \in r$: $r = \{a_1 \mapsto b_1, \ldots, a_n \mapsto a_n, \ldots\}$.

At times it is convenient to flip the relation and deal with its converse: $r^{-1} = \{b_1 \mapsto a_1, \ldots, b_n \mapsto b_n, \ldots\}$. A function is a relation that possess the functional property of mapping each element of its domain to a unique element in the range. The set of all functions over A and B is $\mathbf{F} = \{r \mid r \in \mathrm{A} \leftrightarrow \mathrm{B} \wedge \forall a, b, c \cdot a \mapsto b \in r \Rightarrow a \mapsto c \notin r\}$.

We write $f \in \mathrm{A} \nrightarrow \mathrm{B}$ to say that f is a functional relation between A and B. In particular, such f is called a partial function as there may be elements in A that are not mapped to anything by f. It rather useful (and not hard for finite structures) to establish various sub-classes of a function that have stronger properties. These are total functions $\mathbf{Ft} = \{f \mid f \in \mathbf{F} \wedge \mathrm{dom}(f) = \mathrm{A}\}$ - functions mapping every element of A, surjective functions $\mathbf{Fs} = \{f \mid f \in \mathbf{F} \wedge \mathrm{ran}(f) = \mathrm{B}\}$ and injective functions $\mathbf{Fi} = \{f \mid f \in \mathbf{F} \wedge f^{-1} \in \mathbf{F}^{-1}\}$. Functions that are total, surjective and injective are called bijections.

A predicate is a function of form $p \in X \to \mathrm{BOOL}$ where BOOL is the boolean type. Predicates are treated just like functions and relations.

Let us see how an XML element `c` may be mapped into a relational model. To illustrate the translation we label the fields with indexed identifiers and, since we deal with finite number of objects, assume that each entity declaration is serially numbered:

```
<c>
 <a>ca(k)</a>
 <b><d>bd(l)</b></b>
 <g>cg(k, m)</g><g>cg(k, m + 1)</g>
</c>
```

In the above, k is the serial index of the occurrence of `c`, l is the index for `d` and so on. When processing an XML entry above we extend the relational model in the following manner:

$$\begin{aligned}
&\mathsf{ca}' = \mathsf{ca}' \cup \{k \mapsto ca(k)\} \\
&\mathsf{cbd}' = \mathsf{cbd} \cup \{k \mapsto l, \ldots\} \\
&\mathsf{cg}' = \mathsf{cg}' \cup \{k \mapsto \langle cg(k, m), cg(k, m+1)\rangle\} \\
&\mathsf{bd}' = \mathsf{bd}' \cup \{l \mapsto bd(l)\}
\end{aligned}$$

Here ca is a relation between `c` index set and all values x found in `<a>`x`</a>`. Relation cbd maps between index sets of `c` and `b` to model the association of complex entity of type

b with this particular instance of c. The d value of element b is defined in a separate relation. Finally, multiple values g are represented as an ordered sequence.

Dealing with such relations is inordinately uncomfortable: index sets are artificial constructs that introduce a necessary but a cumbersome level of indirectness. Fortunately, in almost all situations we have seen so far they can be removed. For this we use a heuristic to identify which property of an object is likely to act in the capacity of object identifier. The heuristic looks for textual properties that are present in all element instances; if there are several such properties, the one that would render a relation function is preferred.

The next step is to infer relation types. Take some r. The elements of its domain $\mathrm{dom}(r)$ are postulated to be drawn from a fresh given set. The name of this set is lexically bound, when possible, to the source element type. The elements of range $\mathrm{ran}(r)$ could be one of the following: integer, real or boolean literals, typed identifiers (the type being some given set associated with a record type), untyped identifiers or their power sets, sequences and Cartesian products. The types of untyped elements are solved via a variant of HindleyMilner type system.

At this stage there are three possibilities: data cannot be consistently typed due to irreconcilable typing restrictions; there are some unbound type variables; everything is typed. The first situation indicates serious inconsistencies in the source data. The second may be resolved automatically by generated fresh given sets for each unbound variable. It is however, still highly undesirable situation as well as these sets would remain abstract (not enumerated) which may affect the verification process.

From our experience with the real data sets, it is necessary to manually introduce several typing constraints to guide the type inference procedure. In the experimental data set, even after introducing all meaningful typing hints, there remained a small subset that cannot be typed due to unification errors and has to be ignored (until data generation procedure is changed).

Once typing is done, for the typed subset of data, we compute whether each relation is functional, surjective, total and etc. This is done by simply going through the elements of a relation and checking for certain conditions. Such a check generates a long list of verification statements about relation functionality, surjectivity and so on. These are passed to provers and of everything is fine, the data model is annotated to reflect additional properties. It is especially beneficial to detect functions as makes functional application operator $f(x)$ available and also bijections that have a range of useful properties.

The final stage of model construction establishes that basic data properties such as range and domain types and totality, uniqueness of mappings, injectivity and so on are in agreement with informal description of the data structures. The descriptions of data structures is provided by domain experts or made available from documentation.

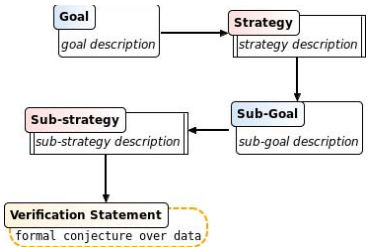
With the data model ready, one can start writing verification conditions over it. To express such conditions we use a mathematical language loosely based on the Event-B notation for expressions and predicates [3]. The notation offers first-order logic operators such as $\forall, \wedge, \neg$, set-theoretic operators such as set union/difference/complement, relational image, function application and set comprehension. There is a basic support for integer, real and machine floating number arithmetic.

V. Goal structuring notation with formal leaves

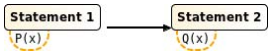
In our approach a safety case is a tree that has safety goal as its root and formal verification conditions as its leaves. We employ a visual derived from the popular Goal Structuring Notation (GSN) [18] although we have so far found the use only for the bare essentials of GSN: elaboration of goals into sub-goal and strategies. We omit contexts and replace solutions with verification statements. Some extra notation is added in aid of verification structuring.

A safety case in SafeCap is defined with the following four basic node types: goal, strategy, verification statement and definition. Verification statement is a node containing a statement formulate in the first-order logic and ZFC set theory. A definition helps to structure verification statements and reuse auxiliary elements and lemmata.

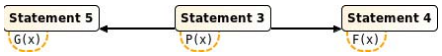
The principal elaboration strategy is to proceed from a goal node one or more strategy nodes and then sub-goals that are addressed by further strategies or sub-goals. At the end, a strategy must be satisfactorily met by a verification statement.



There is a number of special connectors for verification statement nodes. The most common is the formal consequence relation where one node is said to be implied by the other. In the following diagram, it is stated that $\frac{H \vdash Q(x)}{H \vdash P(x)}$ (note that inference is in the opposing direction to the edge):



A verification statement may be decomposed into two or more statements. A decomposed statement is not instantiated with specific data but is rather proven as a general theorem. The decomposing statements must be decomposed further on instantiated and proven for each data source. In the following we have that $\frac{H \vdash F(x) \wedge G(x)}{H \vdash P(x)}$:



A definition, lemma or a named block of lemmata may be brought into the context of a given verification statement. This allows one to combine several results to prove a more general consequence often without data instantiation. The following diagram depicts $H, S(x) \vdash P(x)$:

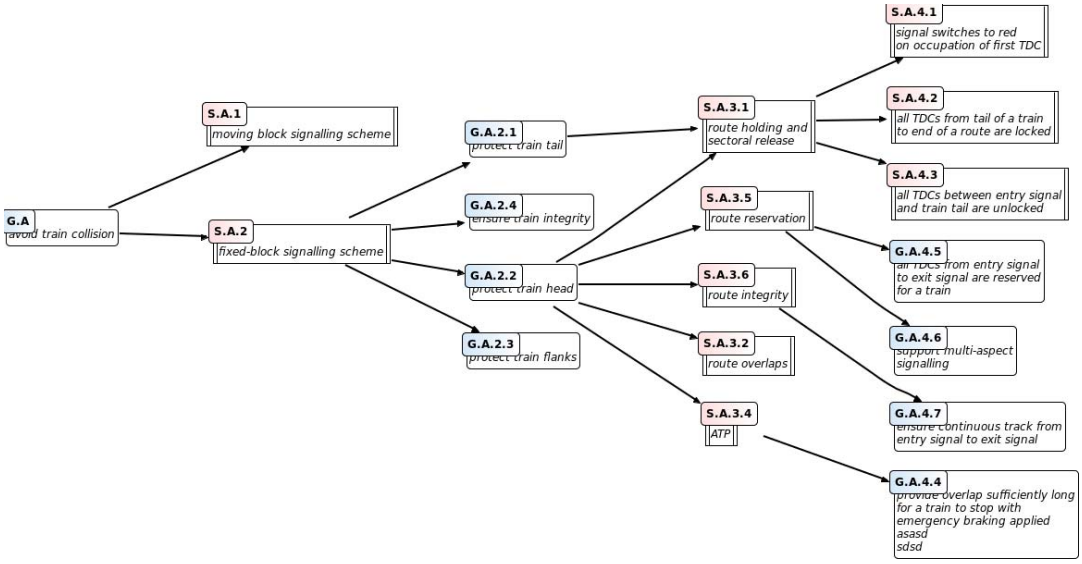


Fig. 4. The top-level part of a safety case addressing the absence of train collisions (an excerpt).

Our extension is mostly conservative (the only novelty is the definition node which may be seen as a form a comment in an informal context) yet the coupling with verification process is very tight. Instead of mapping between safety cases and formal reasoning done in parallel [20], [4]. We use a safety case as a structured documentation that helps in collecting and discussing verification conditions. In a longer term it might be beneficial to have a methodological connection between the verification process and the construction of a safety case [7].

VI. Case Study

The testing ground for the technique is a data set provided by Siemens Rail Automation UK. The data is made of roughly 12MB of XML and structured text files describing topology and signalling of Reading station with signalling split into three overlapping interlocking areas.

One immediate issue what that a part of the data is not XML but a proprietary text-based format called LDL (originating at Invensys Rail) with a number of undocumented custom extensions. The SafeCap Platform has an import facility for LDL files but this silently ignores unknown data fields. We thus developed a new, more basic import tool that treats XML-based and LDL-based data on the same footing of an abstract catalogue-based data representation.

There is a considerable overlap between various parts of this data set. Many of them are not trivial to spot and for historic reason same elements are sometimes known under differing names. In addition, no provision for distinguishing between sets of elements and a sequence of elements. A strict interpretation would require regarding any multiplicity as a sequence or a tuple rather than a set. This is inefficient from the verification viewpoint. To counter this, we allow a user to manually demote sequence and tuples types into set types. For the case of tuples, a unified type (which might not exist for incompatible types) is used.

At the very basic level, we can posit conjectures over the data properties and then check them by combing a conjecture with the data model to see if a contradiction arises. For instance, to check that each section of a given interlocking area is at least 30 meters long one could write property P_1 of the following form:

$$P_1 \coloneqq \min(\text{"TRACK.LENGTH"}[\\ \quad \mathrm{dom}(\mathrm{dom}(\\ \quad\quad \text{"INTERLOCKING.M\_SECTION\_LIST"}(RgStRelf))) \\ \quad]) > 30$$

Here "INTERLOCKING.M_SECTION_LIST" is a function name defined in a data source. It is taken into quotes to escape characters clashing with operator syntax. Expression "INTERLOCKING.M_SECTION_LIST"$(RgStRelf)$ defines a set of triplets and $\mathrm{dom}(\mathrm{dom}(..))$ projects out the set made of first elements from all the triplets. This gives a set of track names. Relational image "TRACK.LENGTH"$[..]$ converts the set names in a set of track lengths; finally $\min(..)$ gives the minimal value of the set. Next, the property is conjoined with the mathematical model of data $\mathbf{H}$ to form conjecture $\mathbf{H} \vdash P_1$. The conjecture undergoes a conservative filtering to remove parts of data model $\mathbf{H}$ irrelevant to P_1 and form a less constrained model $\mathbf{H}_f, \mathbf{H} \subseteq \mathbf{H}$. The typing information is removed and all the literal values are encoded as integers. The un-typing and coding process has fairly modest impact on proof success per se but without it some tools cannot ingest and parse otherwise typically a very large input file.

Every condition is checked twice: once in the positive (i.e., as given) form and once in the negative form. Both cases must have a definite answer (that is, *unknown* result for either case renders the whole condition *false*) so that a conjecture is assumed to hold only when the positive is proven to be truth and the negative is proven false. The double check addresses potential well-definedness problems such as applying "TRACK.LENGTH" to an element x typed to be track but

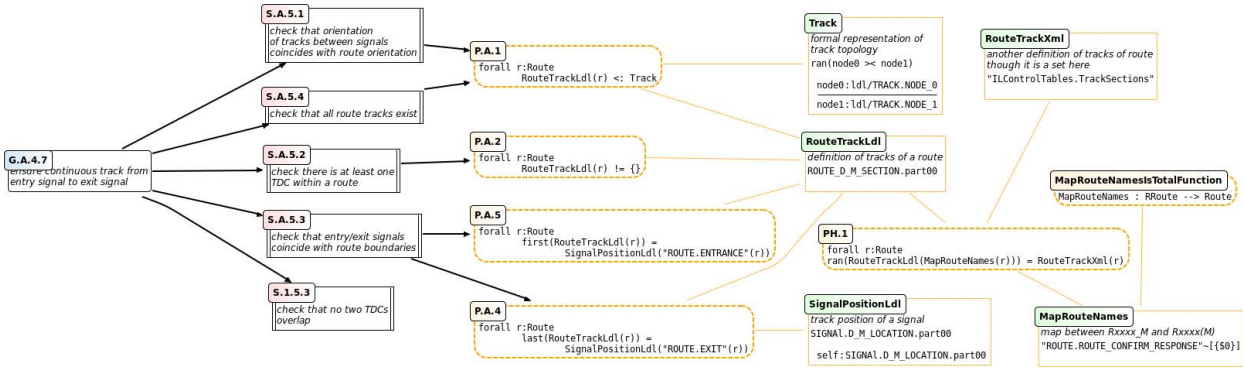


Fig. 5. Zooming into one of the goals and its formal leaves.

not defined in the domain of relation "TRACK.LENGTH" as well as an internally inconsistent data model.

When a conjecture of the form $\forall x.P(x)$ is found to be false, the tool automatically constructs two siblings: a predicate $\neg P(x)$ (with x defined in a context) and expression $\{x \mid \neg P(x)\}$. If models for these may be found (not all tools can return models), they provide a single or complete counter-example to the original statement. For certain types of expressions (sequences and sets and single elements of signals, routes, tracks, points, etc.) a counter-example may be visualised on a track diagram. This helps to spot patterns in data that are not easily discernible in a textual form. As an illustration, should we find that P_1 does not hold, we can restate as

$$P_1' \coloneqq \forall x \cdot x \in \mathrm{dom}(\mathrm{dom}(\text{"INTERLOCKING.M\_SECTION\_LIST"}(\mathit{RgStRelf}))) \Rightarrow \text{"TRACK.LENGTH"}(x) > 30$$

to emphasize that we are looking for track elements that do not satisfy property "TRACK.LENGTH"$(x) > 30$. The tool will highlight all such tracks on a schema diagram.

Real-life verification conditions appear as an elaboration of a GSN diagram. Let us consider, as an illustration, the safety case for the avoidance of train collision. There are two major strategies to address this goal: fixed-block and moving-block signalling. Reading station uses the former so we focus on the sub-goals elaborating the fixed-block strategy. An excerpt of the corresponding safety case diagram is shown in Fig. 4 (the diagram only depict a brief node label; there is a more expansive description behind it). In the diagram we use automatic hierarchical labelling of the form $\{\mathrm{G} \mid \mathrm{S}\}.\langle \textit{global goal id} \rangle.\langle \textit{layer} \rangle.\langle \textit{index} \rangle$. The diagram is built together with domain experts and gradually breaks down high-level concepts into low-level ones.

Recall that the overall objective is to arrive at strategies concrete enough that one is able to transcribe them into formal verification conditions. For the illustration purposes we focus on goal G.A.4.7 that speaks about signal to signal track continuity. It is a fairly trivial condition but is still has to be broken down to meet the conceptual level of input data. It is approximately at this stage of a safety case that we are to match concepts of strategies and goals to the concepts making up the data model. The diagram in Fig. 5 illustrates such an expansion of a low-level informal goal elaborates first into a collection of fine-grained informal strategies and then into formal verification conditions.

At this stage there is one-to-one mapping between an informal strategy which should be convincing enough to a domain expert. Consider, for instance, property S.A.5.1. A route is always directed: it is delineated by entry and exist signals. A piece of track may be uni- or two-directional. It would be illegal to form a route made of track that cannot be traversed in the direction of the route. The verification condition simply states that the path of some route r, RouteTrackLdl(r) is also a path (i.e., sub-graph) within the track topology defined by relation Track. The same verification statement addresses the track existence concern.

Once proven, a property like $P.A.1$ may be added to the data model $\mathbf{H}$ as a new axiom to help discharge further properties.

To assist in the legibility of verification, a diagram may contain definition nodes such as Track and RouteTrackLdl. In the source data, the track topology is defined by a pair of functions mapping from a track to its first or second node. Such a representation proves awkward so we project out from it the adjacency relation by taking the range of a direct product of the two functions. The result is known as new entity Track. Notice that Track is not instantiated or computed: we only provide its axiomatic definition as shown on the diagram.

The same kind of data is often available in several forms. To get additional confidence in the definitions such as Track and RouteTrackLdl we state helper conditions expressing how they correspond to each other and various model elements. The diagram in Fig. 5 shows one simple example where the same track topology is extracted from different parts of the data set and the equivalence of definitions is stated in a helper property $PH.1$.

In the current version there 72 verification goals many of which split into smaller sub-goals to amount to over 200 hundred different theorems. It takes just under 7 minutes to go through all these conditions with all the tools running in parallel on 8 core i7 CPU with 32GB RAM. Among these we have identified four condition not satisfied by the current and confirmed to be genuine problems. Due to the confidentially issues we cannot discuss these in any detail.

VII. Discussion

Instead of building a model that directly operates with the concepts of train movement and interpretation of signalling control, we use low-level static verification conditions formulated semantically as close to the level of source data as it is feasible. The resultant conditions may be tricky to interpret but this is compensated for by constructing a safety case and linking it with the formal conditions.

Perhaps the most prevalent validation technique in the railway industry is simulation. Simulation engines range from coarse-grained time stepping of a national railway network to a detailed model of various aspects of mechanical performance of specific rolling stock over certain track. Validation concerns span from the analysis of digital communication protocols connecting trains and regional control to stressing of tunnels and bridges by passing trains. Simulation is widely applied for time table optimisation and interactive 3D simulation is sometimes used for driver training. RailSys [14] and OpenTrack [13] are two of the more well-known simulation suites applied in time table optimisation and general analysis of signalling performance.

The main attraction of simulation is that it does not require deep understanding of railway functioning. Simulation tools present many aspects of railway performance in an intuitive, visual manner helping to quickly obtain the big picture of overall layout and signalling performance. There is, however, no guarantee of safety as simulation can only ever consider a tiny proportion of all scenarios.

The safety challenge of railways and the fact that collision and derailment properties may be dealt with within the setting of discrete, inertia-less train movement makes railway safety verification especially appealing for formal method practitioners. The principal idea of railway model checking is quite simple: a model of train movement laws is combined with the definitions of track topology and signalling rules. A model checking tool attempts to go through all or many execution scenarios to confirm that unsafe scenarios are ruled out. The list of modelling notations used in this setting is practically endless. Notable examples include Coloured Petri nets [5], process algebra CSP [8], a continuation work based on the model-based notation ASM [9], an algebraic language Maude [2] and the B Method together with ProB model checking tool [12].

Almost all model checking approaches allow automatic instantiation of template models making application of model checking relatively straightforward for engineers. Many tools are able to report a sequence of steps leading to a safety violation. While model checkers are able to analyse many more scenarios than a simulator this comes at a price of reduced expressiveness (i.e., inability to reason about track geometry) and proof certificate is generally not ultimate: there could be a false negative (i.e., the absence of an error report in case an error is present but not discovered) when a model is too large to analyse exhaustively.

Model checking imposes limitations on the model size and performs best with a relatively limited logical language. Theorem proving overcomes these limitations and offers potentially unlimited opportunities for verifying safety with the utmost level of rigour. Theorem proving is not necessarily an all-manual process: there is a large and successful community developing automated theorem provers [19]. At the moment, automated prove support is best in the domain of first order logic and set theory; an attempt at reasoning about continuous train dynamics is likely to require an intervention by a highly skilled verification expert - the kind of people mostly found in academia.

In our future work we will evaluate our approach using larger stations and junctions, such as St. Pancras and King's Cross, and work on its deployment in industry by improving its usability and automation, and by integrating it into the development process. Our plan is to develop interfaces to other types of railway data (e.g. RailML) and evaluate the feasibility of the approach with Network Rail (UK).

References

[1] A. Iliasov, I. Lopatkin, A. Romanovsky. Practical Formal Methods in Railways - The SafeCap Approach. 2014.

[2] I. Chieh Yu P. Enger A. M. Hagalisletto, J. Bjørk. Constructing and Refining Large-Scale Railway Models Represented by Petri Nets. *IEEE Transactions on Systems, Man, and Cybernetics, Part C*, pages 444–460, 2007.

[3] J.-R. Abrial. *Modelling in Event-B*. Cambridge University Press, 2010.

[4] F. Bitsch. Safety patterns the key to formal specification of safety requirements. In Udo Voges, editor, *Computer Safety, Reliability and Security*, volume 2187 of *LNCS*. Springer, 2001.

[5] C. W. Janczura. *Modelling and Analysis of Railway Network Control Logic using Coloured Petri Nets*. PhD thesis, School of Mathematics and Institute for Telecommunications Research, University of South Australia, 1998.

[6] D. Davies. Automatic train protection for the rail network in britain: A study. Technical report, Royal Academy of Engineering., 2000.

[7] E. Denney and G. Pai. Evidence arguments for using formal methods in software certification. 2013.

[8] K. Winter. Model Checking Railway Interlocking Systems. In Proceeding of the 25th Australian Computer Science Conference (ACSC 2002). 2002.

[9] K. Winter and N. Robinson. Modelling Large Railway Interlockings and Model Checking Small Ones. In Proceeding of the Australian Cumputer Science Conference (ACSC 2003). 2003.

[10] N. Bjørner L. De Moura. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08. Springer, 2008.

[11] A. Voronkov L. Kovács. First-order theorem proving and vampire. In *Proceedings of the 25th International Conference on Computer Aided Verification (CAV)*, 2013.

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

[13] OpenTrack simulator. website. Available at http://www.opentrack.ch/.

[14] RailSys simulation platform. website. Available at http://http://www.rmcon.de.

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

[16] SafeCap Project. Project website. 2012. Available at http://safecap.cs.ncl.ac.uk.

[17] S. Schulz. System Description: E 1.8. In Ken McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *Proc. of the 19th LPAR, Stellenbosch*, volume 8312 of *LNCS*, 2013.

[18] R. Weaver T. Kelly. The goal structuring notation a safety argument notation. In *Proc. of Dependable Systems and Networks 2004 Workshop on Assurance Cases*, 2004.

[19] TPTP. Thousands of Problems for Theorem Provers. Available at www.tptp.org/.

[20] E. Troubitsyna Y. Prokhorova, L. Laibinis. Facilitating Construction of Safety Cases from Formal Models in Event-B. *Information and Software Technology*, 60:5176, 2015.