UiO : Department of Informatics
University of Oslo

Rule-based Consistency Checking of Railway Infrastructure Designs

Bjørnar Luteberget, Christian Johansen, and Martin Steffen

Research report 450, January 2016

ISBN 978-82-7368-415-8
ISSN 0806-3036

Acknowledgments

The first author was partially supported by the project RailCons, — *Automated Methods and Tools for Ensuring Consistency of Railway Designs,* with number 248714 funded by the Norwegian Research Council.

Rule-based Consistency Checking of Railway Infrastructure Designs

Bjørnar Luteberget, Christian Johansen, and Martin Steffen

January 2016

Abstract

Railway systems designs deal with complex and large-scale, safety-critical infrastructures, where formal methods play an important role, especially in verifying the safety of so-called interlockings through model checking. Model checking deals with state change and rather complex properties, usually incurring considerable computational burden. In contrast to this, we focus on static infrastructure properties, based on design guidelines and heuristics. The purpose is to automate much of the manual work of the railway engineers through software that can do verification on-the-fly. In consequence, this paper describes the integration of formal methods into the design process, by formalizing relevant technical rules and expert knowledge. We employ a variant of Datalog and use the standardized "railway markup language" railML as basis and exchange format for the formalization. We describe a prototype tool and its (ongoing) integration in industrial railway CAD software. We apply this tool chain in a Norwegian railway project, the upgrade of the Arna railway station.

1 Introduction

Railway systems are complex and large-scale, safety-critical infrastructures, with increasingly computerized components. The discipline of railway engineering is characterised by heavy national regulatory oversight, high and long-standing safety and engineering standards, a need for inter-operability and (national and international) standardization. Due to the high safety requirements, the railway design norms and regulations recommend the use of formal methods (of various kinds), and for the higher safety integrity levels (SIL), they "highly recommend" them (cf. e.g. [6][4]).

Railways require thoroughly designed control systems to ensure safety and efficient operation. The railway signals are used to direct traffic, and the *signalling component layout* of a train station can be crucial to its traffic capacity. Another central part of a railway infrastructure, e.g., of a single railway station, is the so-called *interlocking*, which refers generally speaking to the ensemble of systems tasked to establish safe, conflict-free routes of trains through stations . A more

narrow interpretation of "interlocking" are the principles, the routes, the signalling and movements of trains have to follow to ensure safe operation (cf. [27]). While formal methods play a crucial role, especially in designing the signalling and interlocking, Railway construction projects are heavy processes that integrate various fields, engineering disciplines, different companies, stakeholders, and regulatory bodies. When working out railway designs a large part of the work is repetitive, involving routine checking of consistency with rules, writing tables, and coordinating disciplines. Many of these manual checks are simple enough to be automated with computational results that can be used inside existing engineering software. The repetition comes from the fact that even small changes in station layout and interlocking may require thorough (re-)investigation to prove that the designs remain internally consistent and still adhere to the rules and regulations of the national (and international) rail administration agencies.

This paper presents results on integrating formal methods into the railway design process, with the purpose of increasing the degree of automation as follows:

- We formalize rules governing track and signalling layout, and interlocking.
- The standardized "railway markup language" railML [31] is used as basis and exchange format for the formalization.
- We model the concepts describing a railway design in the logic of Datalog; and develop an automated generation of the model from the railML representation.
- The prototype tool has been integrated in existing railway CAD software.

We illustrate the logical representation of signalling principles and show how they can be implemented and solved efficiently using the Datalog style of logic programming [36]. We also show the integration with existing railway engineering workflow by using CAD models directly. This enables to verify rules continuously as the design process changes the station layout and interlocking. Based on railML [31]as intermediary language, our results can be easily adopted by anyone that uses this international standard. The work uses as case study the software and the design (presently under development) used in the *Arna-Fløen* upgrade project,[1] a major infrastructure activity of the Norwegian railway system, with planned completion in 2020. The Arna train station is located on Northern Europe's busiest single-track connection (between Arna and Bergen), which is being extended to a double-track connection. Thus, the train station is currently undergoing an extensive overhaul, including significant new tunnel constructions and specifically a replacement of the entire signalling and control system. The case study is part of an ongoing project in Anacon AS (now merged with Norconsult), a Norwegian signalling design consultancy. It is used to illustrate the approach, test the implementation, and to verify

[1] `http://www.jernbaneverket.no/Prosjekter/prosjekter/Arna---Bergen`

that the tool's performance is acceptable for interactive work within the CAD software.

The rest of the paper is organized as follows. Section 2 discusses aspects of the railway domain relevant for this work. Section 3 proposes a tool chain that extends CAD with formal representations of signalling layout and interlocking. Section 4 presents our formalization of the rules and concepts governing general principles of railway design as logical formulas amenable for the Datalog implementation and checking. Section 5 provides more information about the implementation, including details about counterexample presentation and empirical evaluation of the tool using the case study. We conclude in Section 6 with related and future work.

2 Background on Railway Signalling Domain

The signalling design process results in a set of documents which can be categorized into (a) track and signalling component layout, and (b) interlocking specification, and an (c) automatic train control specification. The first two categories are considered in this paper.

2.1 Track and Signalling Component Layout

Track layout details, as input to the signalling design, are often given by a separate division of the railway project. At an early stage and working at a low level of detail, the signalling engineer may challenge the track layout design, and an iterative process may be initiated.

Railway construction projects rely heavily on *computer aided design* (CAD) tools to map out railway station layouts. The various disciplines within a project, such as ground works, track works, signalling, or catenary power lines, work with coordinated *CAD models*. These CAD models contain a major part of the work performed by engineers, and are a collaboration tool for communcation between disciplines. The signalling component layout is worked out by the signalling engineer as part of the design process. Placement of signals, train detectors, derailers, etc. is drawn using symbols in a 2D geographical CAD model.

2.2 Interlocking Specification

An interlocking is an interconnection of signals and switches to ensure that train movements are performed in a safe sequence [27]. Interlocking is performed electronically so that, e.g., a green light (or, more precisely, the *proceed aspect*) communicating the movement authority required for a train to travel through a station can only be lit by the interlocking controller under certain conditions. Conditions and state are built into the interlocking by relay-based circuitry or by computers running interlocking software. Most interlocking specifications use a *route-based tabular* approach, which means that a train station is divided into possible *routes*,

which are paths that a train can take from one signal to another. These signals are called the *route entry signal* and *route exit signal*, respectively. An *elementary route* contains no other signals in-between. The main part of the interlocking specification is to tabulate all possible routes and set conditions for their use. Typical conditions are:

- *Switches* must be positioned to guide the train to a specified route exit signal.
- *Train detectors* must show that the route is free of any other trains.
- *Conflicting routes*, i.e. overlapping routes, must not be in use.

3 Proposed Railway Signalling Design Tool Chain

Next we describe the tool chain that we propose for automating the current manual tasks involved in the design of railway infrastructures. In particular, we are focused on integrating and automating those simple, yet tedious, rules and conditions usually used to maintain some form of consistency of the railway, and have these checks done automatically. Whenever the design is changed by an engineer working with the CAD program, our verification procedure would help, behind the scenes, verifying any small changes in the model and the output documents. Violations would either be automatically corrected, if possible, or highlighted to the engineer. Thus, we are focusing on solutions with small computational overhead when working with CAD tools (running on standard computers).

3.1 Computer-Aided Design (CAD) Layout Model

CAD models, which ultimately correspond to a database of geometrical objects, are used in railway signalling engineering. They may be 2D or 3D, and contain mostly spatial properties and textual annotations, i.e., the CAD models focus on the *shapes* of objects and *where* to place them. The top level of the document, called the *model space block*, contains geometrical primitives, such as lines, circles, arcs, text, and symbols. It also contains *block references*, which consists of a reference to a *block*, and an *insertion point* where the block is located. Block references are typically used to create reusable components of a CAD document.

Geometric elements may represent the physical geometry directly, or symbolically, such as text or symbols. A railway signalling CAD model will contain both physical geometry and symbols, typically track centerlines with horizontal geometry and signalling equipment as symbol blocks with insertion point at their physical location. However, the verification of signalling and interlocking rules requires information about object properties and relations between objects such as which signals and signs are related to which track, and their identification, capabilities, and use. This information is better modelled by the railway-specific hierarchical object model called railML [26].

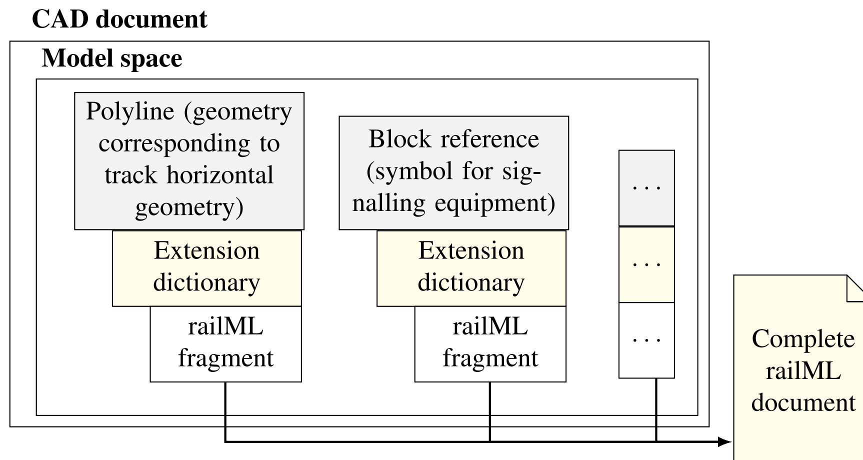


Figure 1: RailML integrated into a CAD database

3.2 Integrating Layout Model with railML

CAD programs were originally designed to produce paper drawings, and common practice in the use of CAD programs is to focus on *human-readable* documents. The database structure, however, may also be used to store machine-readable information. In the industry-standard DWG format [10], each geometrical object in the database has an associated *extension dictionary*, where add-on programs may store any data related to the object. Our tool uses this method to store the railML fragments associated with each geometrical object or symbol, see Figure 1. Thus, we can compile the complete railML representation of the station from the CAD model.

3.3 Interlocking and Automatic Train Control (ATC) Specifications

Besides the CAD model layout, the design of a railway station consists also of specifications for the interlocking and ATC. These specifications model the behavior of the signalling, and are tightly linked to the station layout. A formal representation of the interlocking and ATC specifications is embedded in the CAD document in a similar way as for the railML infrastructure data, using the document's global extension dictionary. Thus, the single CAD document showing the human-readable layout of the train station also contains a machine-readable model which fully describes both the component layout and the functional specification of interlocking and ATC. This allows a full analysis of the operational aspects of the train station directly in a familiar editable CAD model.

3.4 Overall Tool Chain

Figure 2 shows the overall tool chain. The software allows checking of rules and regulations of static infrastructure (described in this paper) inside the CAD en-

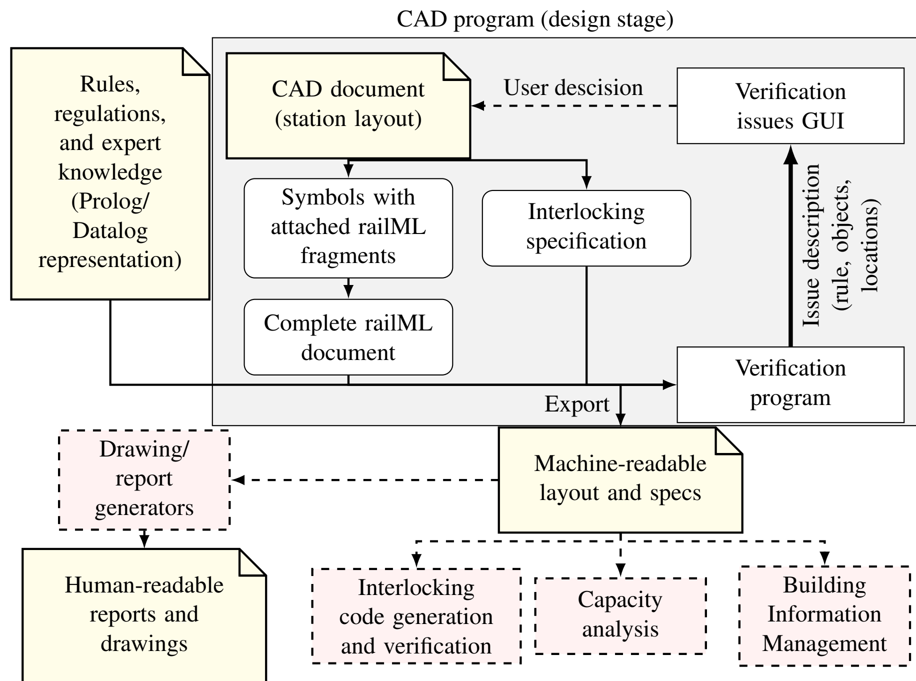


Figure 2: Railway design tool chain. The CAD program box shows features which are directly accessible at design time inside the CAD program, while the export creates machine-readable (or human-readable) documents which may be further analysed and verified by external software (shown in dashed boxes).

vironment, while more comprehensive verification and quality assurance can be performed by special-purpose software for other design and analysis activities.

Generally, analysis and verification tools for railway signalling designs can have complex inputs, must account for a large variety of situations, and usually require long running times. Therefore, we limit the verification inside the design environment to static rules and expert knowledge, as these rules require less dynamic information (timetables, rolling stock, etc.) and computational effort, while still offering valuable insights. This situation may be compared to the tool chain for writing computer programs. Static analysis can be used at the detailed design stage (writing the code), but can only verify a limited set of considerations. It cannot fully replace testing, simulation and other types of analysis, and must as such be seen as a part of a larger tool chain.

Other tools, that are external to the CAD environment, may be used for these more calculation heavy or less automated types of analysis, such as:

- Code generation and verification for interlockings, possible through the formal verfication framework of Prover Technologies.

- Capacity analysis and timetabling, performed using OpenTrack, LUKS, or Treno.
- Building information management (BIM), including such activities as life-cycle information management and 3D viewing, are already well integrated with CAD, and can be seen as an extension of CAD.

The transfer of data from the CAD design model to other tools is possible by using standardized formats such as railML, which in the future will also include an interlocking specification schema [3].

4 Formalization of Rule Checking

To achieve our goal of automating checking of the consistency of railway designs we need formal representations of both the designs and the consistency rules.

The rules can be seen as the static part, e.g., kept in a a domain specific language or GUI, thus hiding the logical notation. Obtaining the formal, logical representation of the designs will be done on-the-fly through a CAD module, whereas the logical representation of the rules will be done manually by the engineers through a domain specific language or GUI.

The logical consistency checking that we deal with turns out to require only simple, computationally tractable, forms of logics. In particular, we do not go into linear-time temporal logics (LTL) [29] and the automata-based model checking [21], as needed for checking safety of the actual interlocking programs [5]. Nevertheless, the verification methodology is the same: The logical representation of the designs (called the model) and of the rules (called properties) are fed into the verification engine (SAT/SMT or Datalog) which is doing satisfiability checking, thus looking for an interpretation of the logical variables that would satisfy the formulas. More precisely, the rules are first negated, then conjoined with the formulas representing the model. Therefore, looking for a satisfying interpretation is the same as looking for a way to violate the rules. When found, the interpretation contains the information about the exact reasons for the violation. The reasons, or counter-example, always involves some of the negated rules as well as some parts of the model. In other words, a good railway design is one for which the satisfiability engine returns a negative answer, because it cannot find a satisfying variable interpretation.

We formalize the correctness properties (i.e., technical rules and expert knowledge) as predicates over finite and real domains. Using a logic programming framework, we will include the following in the logical model:

1. Predicate representation of input document *facts*, i.e. track layout and interlocking.
2. Predicate representation of derived concept *rules*, such as object properties, topological properties, and calculation of distances.

3. Predicate representation of technical rules.

Each of these categories are described in more detail below, after we present the logical framework we employ.

4.1 Datalog

Declarative logic programming is a programming language paradigm which allows clean separation of logic (meaning) and computation (algorithm). This section gives a short overview of Datalog concepts. See [36] for more details. In its most basic form it is a database query, like in the SQL language, over a finite set of atoms which can be combined using conjunctive queries, i.e. expressions in the fragment of first-order logic which includes only conjunctions and existential quantification.

Conjunctive queries alone, however, cannot express the properties needed to verify railway signalling. For example, given the layout of the station with tracks represented as edges between signalling equipment nodes, graph reachability queries are required to verify some of the rules. This corresponds to computing the transitive closure of the graph adjacency relation, which is not expressible in first-order logic [20, Chap. 3].

Adding fixed-point operators to conjunctive queries is a common way to mitigate the inexpressibility of this type of graph queries while preserving decidability and polynomial time complexity. Fixed-point operators on finite structures amount to some form of iteration producing sets that are monotonically growing. Thus, when using a finite set of atoms, termination is guaranteed.

The Datalog language is a first-order logic extended with least fixed points. We define the Datalog language as follows: *Terms* are either constants (atoms) or variables. *Literals* consist of a *predicate* P with a certain arity n, alongs with terms corresponding to the predicate arguments, forming an expression like $P(\vec{a})$, where $\vec{a} = (a_1, a_2, \dots, a_n)$. *Clauses* consist of a *head* literal and one or more *body* literals, such that all variables in the head also appear in the body. Clauses are written as

$$R_0(\vec{x}) \;\text{:-}\; \exists \vec{y} : R_1(\vec{x}, \vec{y}), R_2(\vec{x}, \vec{y}), \dots, R_k(\vec{x}, \vec{y}).$$

Datalog uses the Prolog convention of intepreting identifiers starting with a capital letter as variables, and other identifiers as constants. E.g., the clause

$$a(X, Y) \;\text{:-}\; b(X, Z), c(Z, Y)$$

has the meaning of

$$\forall x, y : ((\exists z : (b(x, z) \wedge c(z, y))) \rightarrow a(x, y)) \,.$$

Clauses without body, which cannot then contain any variables, are called *facts*, those with one or more literals in the body are called *rules*. No nesting of literals is allowed. However, recursive definitions of predicates are possible. For example,

let $edge(a, b)$ be an graph edge relation between vertices a and b. Graph searches can now be encoded by making a transitive closure over the edge relation:

$$\begin{aligned} path(a,b) &:- \ edge(a,b). \\ path(a,b) &:- \ edge(a,x), path(x,b). \end{aligned}$$

In the railway domain, this can be used to define the *connected* predicate, which defines whether two objects are connected by railway tracks:

$$\begin{aligned} directlyConnected(a,b) &:- \ track(t), belongsTo(a,t), belongsTo(b,t). \\ connected(a,b) &:- \ directlyConnected(a,b). \\ connected(a,b) &:- \ directlyConnected(a,x), connection(x,c), \\ &\quad\ connected(c,b). \end{aligned}$$

Here, the *connection* predicate contains switches and other connection types. Further details of relevant predicates are given in the sections below.

Another common feature of Datalog implementations is to allow negation, with *negation as failure* semantics. This means that negation of predicates in rules is allowed with the interpretation that when the satisfiability procedure cannot find a model, the statement is false. To ensure termination and unique solutions, the negation of predicates must have a *stratification*, i.e. the dependency graph of negated predicates must have a topological ordering (see [36, Chap. 3] for details).

Datalog is sufficiently expressive to describe static rules of signalling layout topology and interlocking. For geometrical properties, it is necessary to take sums and differences of lengths, which requires extending Datalog with arithmetic operations. A more expressive language is required to cover all aspects of railway design, e.g. capacity analysis and software verification, but for the properties in the scope of this paper, a concise, restricted language which ensures termination and short running times has the advantage of allowing tight integration with the existing engineering workflow.

4.2 Input Documents Representation

4.2.1 Track and signalling objects layout in the railML format.

Given a complete railML infrastructure document, we consider the set of XML elements in it that correspond to identifiable objects (this is the set of elements which inherit properties from the type `tElementWithIDAndName`). The set of all IDs which are assigned to XML elements form the finite domain of constants on which we base our predicates (IDs are assumed unique in railML).

$$\text{Atoms} := \{a \mid \text{element}.\texttt{ID} = a\}\,.$$

We denote a railML element with *ID* $= a$ as element_a. All other data associated with an element is expressed as predicates with its identifying atom as one of the arguments, most notably the following:

- Element type (also called class in railML/XML):

$$\begin{aligned} track(a) &\leftarrow \text{element}_a \text{ is of type } \texttt{track}, \\ signal(a) &\leftarrow \text{element}_a \text{ is of type } \texttt{signal}, \\ balise(a) &\leftarrow \text{element}_a \text{ is of type } \texttt{balise}, \\ switch(a) &\leftarrow \text{element}_a \text{ is of type } \texttt{switch}. \end{aligned}$$

- Element name:

$$name(a, n) \leftarrow (\text{element}_a.\texttt{name} = n).$$

- Position and absolute position (elements inheriting from `tPlacedElement`):

$$pos(a, p) \leftarrow (\text{element}_a.\texttt{pos} = p), \quad a \in \text{Atoms}, p \in \mathbb{R},$$

$$absPos(a, p) \leftarrow (\text{element}_a.\texttt{absPos} = p), \quad a \in \text{Atoms}, p \in \mathbb{R}.$$

- Geographical coordinates (for elements inheriting from `tPlacedElement`):

$$geoCoords(a, q) \leftarrow (\text{element}_a.\texttt{geoCoords} = q), \quad a \in \text{Atoms}, q \in \mathbb{R}^3.$$

- Direction (for elements inheriting from `tOrientedElement`):

$$dir(a, d) \leftarrow (\text{element}_a.\texttt{dir} = d), \quad a \in \text{Atoms}, d \in \text{Direction},$$

  where $\text{Direction} = \{up, down, both, unknown\}$, indicating whether the object is visible or functional in only one of the two possible travel directions, or both.
- Signal properties (for elements of type `tSignal`):

$$\begin{aligned} signalType(a, t) \leftarrow &(\text{element}_a.\texttt{type} = t), \\ &a \in \text{Atoms}, t \in \{\text{main, distant, shunting, combined}\}, \end{aligned}$$

$$\begin{aligned} signalFunction(a, f) \leftarrow &(\text{element}_a.\texttt{function} = f), \\ &a \in \text{Atoms}, f \in \{\text{home, intermediate, exit, blocking}\}. \end{aligned}$$

  Consistency axioms would impose that *signalType* and *signalFunction* be applied only to *signal* elements:

$$signalType(a, t) \Rightarrow signal(a),$$

$$signalFunction(a, f) \Rightarrow signal(a).$$

The above list give only a few examples of predicates that are extracted from the railML document. The translator from railML (XML documents) to predicate form needs only to consider XML elements, attributes and sub-elements, not the specifics of railML and its type hierarchy. The whole expressivity of railML as such is carried over directly to the logic programming environment. The *switch* element is the object which connects tracks with each other and creates the branching of paths, see Figure 3. A switch belongs to a single track, but contains *connection* sub-elements which point to other connection elements, which are in turn contained in switches, crossings or track ends. For connections, we have the following predicates:

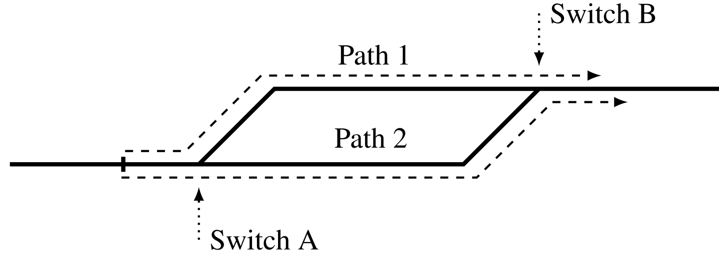


Figure 3: Switches give rise to branching paths

- Connection element and reference:

$$connection(a) \leftarrow \text{element}_a \text{ is of type } \texttt{connection},$$

$$connection(a,b) \leftarrow (\text{element}_a.\texttt{ref} = b).$$

  The connection relation should always be symmetric, i.e. $\forall a, b : connection(a,b) \rightarrow connection(b,a)$, and this will be checked by a consistency predicate.

- Connection course and orientation:

$$connectionCourse(a,c) \leftarrow (\text{element}_a.\texttt{course} = c),\quad a \in \text{Atoms}, c \in \{\text{left,straight,right}\},$$

$$connectionOrientation(a,o) \leftarrow (\text{element}_a.\texttt{orientation} = o),\quad a \in \text{Atoms}, o \in \{\text{outgoing,incoming}\}.$$

To encode the hierarchical structure of the railML document, a separate predicate encoding the parent/child relationship is added: This is required because the predicate representation does not implicitly contain the hierarchy of the XML representation, where elements are declared inside other elements.

- Object belongs to (e.g. a is a signal belonging to track b):

$$belongsTo(a,b) \leftarrow b \text{ is the closest XML ancestor of } a \text{ whose element type inherits from } \texttt{tElementWithIDAndName}.$$

4.2.2 Interlocking.

An XML schema for tabular interlocking specifications is described in [3], and this format is used here with the expectation that it will become part of the railML standard schema in the future. We give some examples of how XML files with this schema are translated into predicate form:

- Train route with given direction d, start point a, and end point b ($a, b \in$ Atoms, $d \in$ Direction):

$$trainRoute(t) \leftarrow \text{element}_t \text{ is of type } \texttt{route}$$
$$start(t,a) \leftarrow (\text{element}_t.\texttt{start} = a)$$
$$end(t,b) \leftarrow (\text{element}_t.\texttt{end} = b)$$

- Conditions on detection section free (a) and switch position (s, p):

$$detectionSectionCondition(t, a) \leftarrow (a \in \text{element}_t.\texttt{sectionConditions}),$$
$$switchPositionCondition(t, s, p) \leftarrow ((s, p) \in \text{element}_t.\texttt{switchConditions}).$$

4.3 Derived Concepts Representation

Derived concepts are properties of the railway model which can be defined independently of the specific station. A library of these predicates is needed to allow concise expression of the rules to be checked.

4.3.1 Object properties.

Properties related to specific object types which are not explicitly represented in the layout description, such as whether a switch is *facing* in a given direction, i.e. if the path will branch when you pass it:

- Switch facing or trailing ($a \in \text{Atoms}$, $d \in \text{Direction}$):

$$switchFacing(a, d) \leftarrow \exists c, o : switch(a) \wedge switchConnection(a, c) \wedge switchOrientation(c, o) \wedge orientationDirection(o, d).$$

$$switchTrailing(a, d) \leftarrow \neg switchFacing(a, d)$$

4.3.2 Topological and geometric layout properties.

Predicates describing the topological configuration of signalling objects and the train travel distance between them are described by predicates for **track connection** (predicate $connected(a, b)$), **directed connection** (predicate $following(a, b, d)$), **distance** (predicate $distance(a, b, d, l)$), etc. The track connection predicate is defined as:

- There is a **track connection** between object a and b ($a, b \in \text{Atoms}$):

$$directlyConnected(a, b) \leftarrow \exists t : track(t) \wedge belongsTo(a, t) \wedge belongsTo(b, t),$$

$$connected(a, b) \leftarrow directlyConnected(a, b) \vee (\exists c_1, c_2 : connection(c_1, c_2) \wedge directlyConnected(a, c_1) \wedge connected(c_2, b)).$$

- There is a **directed connection** between object a and b ($a, b \in \text{Atoms}$, $d \in \text{Direction}$, $p_a, p_b \in \mathbb{R}$):

$$directlyFollowing(a, b, d) \leftarrow directlyConnected(a, b) \wedge position(a, p_a) \wedge position(b, p_b) \wedge ((d = \text{up} \wedge p_a < p_b) \vee (d = \text{down} \wedge p_a > p_b))$$

$$following(a, b, d) \leftarrow directlyFollowing(a, b, d) \vee \exists c_1, c_2 : connection(c_1, c_2) \wedge directlyFollowing(a, c_1, d) \wedge following(c_2, b, d)$$

- The **distance** (along track) in a given direction between object a and b ($a, b \in$ Atoms, $d \in$ Direction, $p_a, p_b, l \in \mathbb{R}$):

$$\begin{aligned} \mathit{directDistance}(a,b,d,l) \leftarrow\ & \mathit{directlyFollowing}(a,b,d) \wedge \\ & \mathit{position}(a,p_a) \wedge \mathit{position}(b,p_b) \\ & \wedge\, l = |p_b - p_a| \end{aligned}$$

$$\begin{aligned} \mathit{distance}(a,b,d,l) \leftarrow\ & \mathit{directDistance}(a,b,d,l) \vee \\ & \exists c_1, c_2, l_1, l_2 : \mathit{connection}(c_1,c_2) \\ & \wedge\, \mathit{directDistance}(a,c_1,d,l_1) \\ & \wedge\, \mathit{distance}(c_2,b,d,l_2) \wedge l = l_1 + l_2 \end{aligned}$$

- Object is located **between** a and b ($a, x, b \in$ Atoms, $d \in$ Direction):

$$\mathit{between}(a,x,b,d) \leftarrow \mathit{following}(a,x,d) \wedge \mathit{following}(x,b,d)$$

$$\mathit{between}(a,x,b) \leftarrow \exists d : \mathit{between}(a,x,b,d)$$

- A path between a and b **overlaps** with a path between c and d ($a, b, c, d \in$ Atoms):

$$\mathit{overlap}(a,b,c,d) \leftarrow \exists e : \mathit{between}(a,e,b) \wedge \mathit{between}(c,e,d)$$

4.3.3 Interlocking properties.

Properties such as $\mathit{existsPathWithoutSignal}(a,b)$ for finding elementary routes, and $\mathit{existsPathWithDetector}(a,b)$ for finding adjacent train detectors will be used as building blocks for the interlocking rules.

- Signals a and b have a path between them without any other signals in between:

$$\begin{aligned} \mathit{existsPathWithoutSignal}(a,b,d) \leftarrow\ & \mathit{following}(a,b,d) \wedge \\ & (\neg(\exists x : \mathit{signal}(x) \wedge \mathit{between}(a,x,b)) \vee \\ & (\exists x : \mathit{between}(a,x,b) \wedge \mathit{existsPathWithoutSignal}(a,x,d) \wedge \\ & \mathit{existsPathWithoutSignal}(x,b,d)). \end{aligned}$$

4.4 Rule Violations Representation

With the input documents represented as facts, and a library of derived concepts, it remains to define the technical rules to be checked. Technical rules are based on [17]. The goal of the consistency checking is to confirm that no inconsistencies exist, in which case no further information is required, or to find inconsistencies and present them in a way that allows the user to understand the error and to adjust their design accordingly. Rules are therefore expressed negatively, as rule *violations*, so that a query corresponding to the rule is empty whenever the rule is consistent with the design, or the query contains counterexamples to the rule when they exist. Some examples of technical rules representing conditions of the railway station layout are given below.

Property 1 (Layout: Home signal [17]) *A* home main signal *shall be placed at least 200 m in front of the first controlled, facing switch in the entry train path.*

See also Figure 4 for an example. Property 1 may be represented in the following way:

$$\begin{aligned} \mathit{isFirstFacingSwitch}(b,s) \leftarrow\ & \mathit{stationBoundary}(b) \land \mathit{facingSwitch}(s) \land \\ & \neg(\exists x : \mathit{facingSwitch}(x) \land \mathit{between}(b,x,s)), \end{aligned}$$

$$\begin{aligned} \mathit{ruleViolation}_1(b,s) \leftarrow\ & \mathit{isFirstFacingSwitch}(b,s) \land \\ & (\neg(\exists x : \mathit{signalFunction}(x, \text{home}) \land \mathit{between}(b,x,s)) \lor \\ & (\exists x,d,l : \mathit{signalFunction}(x, \text{home}) \land \\ & \land \mathit{distance}(x,s,d,l) \land l < 200). \end{aligned}$$

Checking for rule violations can be expressed as:

$$\exists b, s : \mathit{ruleViolation}_1(b,s),$$

which in Prolog/Datalog query format becomes `ruleViolation1(B,S)?`.

Property 2 (Layout: Minimum detection section length [17]) *No train detection section shall be shorter than 21 m. I.e., no train detectors should be separated with less than 21 m driving distance.*

This property is represented as follows:

$$\begin{aligned} \mathit{ruleViolation}_2(a,b) \leftarrow & \exists d,l : \mathit{trainDetector}(a) \land \mathit{trainDetector}(b) \land \\ & \mathit{distance}(a,b,d,l) \land l < 21.0. \end{aligned}$$

Property 3 (Layout: Exit main signal [17]) *An exit main signal shall be used to signal movement exiting a station.*

This property can be elaborated into the following rules:

- No path should have more than one exit signal:

$$\begin{aligned} \mathit{ruleViolation}_3(s) \leftarrow & \exists d : \mathit{signalType}(s, \text{exit}) \land \mathit{following}(s, s_o, d) \land \\ & \neg \mathit{signalType}(s_0, \text{exit}). \end{aligned}$$

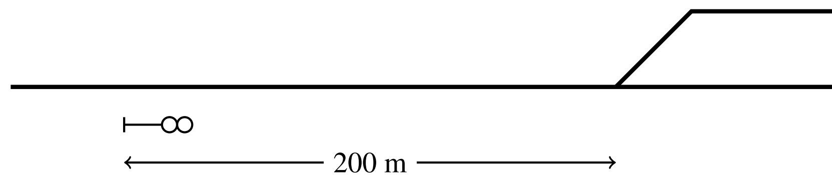
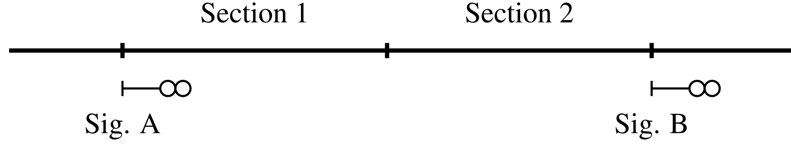


Figure 4: A home main signal shall be placed at least 200 m in front of the first controlled, facing switch in the entry train path. (Property 1)



Tabular interlocking:

Route	Start	End	Sections must be clear
AB	A	B	1, 2

Figure 5: Track sections which overlap a route must have a corresponding condition in the interlocking. (Property 5)

- Station boundaries should be preceded by an exit signal:

$$exitSignalBefore(x, d) \leftarrow \exists s : signalType(s, \text{exit}) \wedge following(s, x, d)$$
$$ruleViolation_3(b) \leftarrow \exists d : stationBoundary(b) \wedge \neg exitSignalBefore(b, d).$$

A basic property of tabular interlockings is that each consecutive pair of main signals normally has an elementary train route associated with it, i.e.:

Property 4 (Interlocking: Elementary routes) *A pair of consecutive main signals should be present as a route in the interlocking.*

This can be represented as follows:

$$defaultRoute(a, b, d) \leftarrow signalType(a, \text{main}) \wedge signalType(b, \text{main}) \wedge direction(a, d) \wedge direction(b, d) \wedge following(a, b, d) \wedge existsPathWithoutSignal(a, b, d),$$

$$ruleViolation_4(a, b, d) \leftarrow defaultRoute(a, b, d) \wedge \neg(\exists r : trainRoute(r) \wedge trainRouteStart(r, a) \wedge trainRouteEnd(r, b)).$$

This type of rule is not absolutely required for a railway signalling design to be valid and safe. Some rules are hard constraints, where violations may be considered to be errors in the design, while other rules are soft constraints, where violations may suggest that further investigation is recommended. This is relevant for the counterexample presentation section below.

Property 5 (Interlocking: Track clear on route) *Each pair of adjacent train detectors defines a track detection section. For any track detection sections overlapping the route path, there shall exist a corresponding condition on the activation of the route.*

See Figure 5 for an example. Property 5 can be represented as follows:

$$existsPathWithDetector(a, b) \leftarrow \exists d : following(a, b, d) \wedge trainDetector(x) \wedge between(a, x, b).$$

$$adjacentDetectors(a, b) \leftarrow trainDetector(a) \wedge trainDetector(b) \wedge \neg existsPathWithDetector(a, b),$$

$$detectionSectionOverlapsRoute(r, d_a, d_b) \leftarrow trainRoute(r) \wedge start(r, s_a) \wedge end(r, s_b) \wedge adjacentDetectors(d_a, d_b) \wedge overlap(s_a, s_b, d_a, d_b),$$

$$detectionSectionCondition(r, d_a, d_b) \leftarrow detectionSectionCondition(c) \wedge belongsTo(c, r) \wedge belongsTo(d_a, c) \wedge belongsTo(d_b, c).$$

$$ruleViolation_5(r, d_a, d_b) \leftarrow detectionSectionOverlapsRoute(r, d_a, d_b) \wedge \neg detectionSectionCondition(r, d_a, d_b).$$

Property 6 (Interlocking: Flank protection [17]) *A train route shall have flank protection.*

For each switch in the route path and its associated position, the paths starting in the opposite switch position defines the *flank*. Each flank path is terminated by the first flank protection object encountered along the path. The following objects can give flank protection:

1. *Main signals*, by showing the *stop* aspect.
2. *Shunting signals*, by showing the *stop* aspect.
3. *Switches*, by being controlled and locked in the position which does not lead into the path to be protected.
4. *Derailers*, by being controlled and locked in the derailing state.

An example situation is shown in Figure 6. While the indicated route is active (A to B), switch X needs flank protection for its left track. Flank protection is given by setting switch Y in right position and setting signal C to *stop*. Property 6 can be elaborated into the following rules:

- All flank protection objects should be eligible flank protection objects, i.e. they should be in the list of possible flank protection objects, and have the correct orientation (the *flankElement* predicate contains the interlocking facts):

$$flankProtectionObject(a, b, d) \leftarrow ((signalType(a, \text{main}) \wedge dir(a, d)) \vee (signalType(a, \text{shunting}) \wedge dir(a, d)) \vee switchFacing(a, d) \vee derailer(a)) \wedge following(a, b, d).$$

$$flankProtectionRequired(r, x, d) \leftarrow trainRoute(r) \wedge start(r, s_a) \wedge end(r, s_b) \wedge switchOrientation(x, o) \wedge between(s_a, x, s_b) \wedge orientationDirection(o, o_d) \wedge oppositeDirection(o_d, d).$$

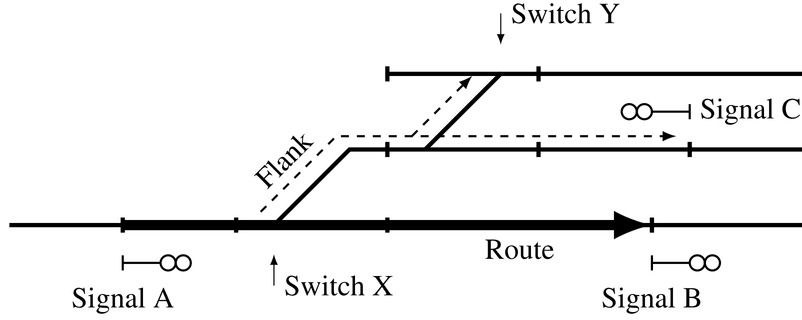


Figure 6: The dashed path starting in switch X must be terminated in all branches by a valid flank protection object, in this case switch Y and signal C. (Property 6)

$$\begin{aligned} \mathit{flankProtection}(r,e) \leftarrow & \mathit{flankProtectionRequired}(r,x,d) \wedge \\ & \mathit{flankProtectionObject}(e,x,d). \end{aligned}$$

$$\begin{aligned} \mathit{ruleViolation}_6(r,e) \leftarrow & \mathit{flankElement}(r,e) \wedge \\ & \neg \mathit{flankProtection}(r,e). \end{aligned}$$

- There should be no path from a model/station boundary to the given switch, in the given direction, that does not pass a flank protection object for the route:

$$\begin{aligned} & \mathit{existsPathWithFlankProtection}(r,b,x,d) \leftarrow \\ & \quad \mathit{flankElement}(r,e) \wedge \mathit{flankProtectionElement}(e,x,d) \wedge \\ & \quad \mathit{between}(b,e,x). \end{aligned}$$

$$\begin{aligned} & \mathit{existsPathWithoutFlankProtection}(r,b,x,d) \leftarrow \\ & \quad \neg \mathit{existsPathWithFlankProtection}(r,b,x,d) \vee \\ & \quad (\mathit{between}(b,y,x) \wedge \neg \mathit{flankProtectionElement}(e,y,d) \wedge \\ & \quad \mathit{existsPathWithoutFlankProtection}(r,b,y,d) \wedge \\ & \quad \mathit{existsPathWithoutFlankProtection}(r,y,x,d)). \end{aligned}$$

$$\begin{aligned} \mathit{ruleViolation}_6(r,b,x) \leftarrow & \ \mathit{stationBoundary}(b) \wedge \\ & \mathit{flankProtectionRequired}(r,x,d) \wedge \mathit{following}(b,x,d) \wedge \\ & \mathit{existsPathWithoutFlankProtection}(r,b,x,d). \end{aligned}$$

5 Tool Implementation

In this section we describe the main aspects of our tool implementation. The XSB Prolog interpreter was used as a back-end for the implementation of a verification procedure, as it offers tabled predicates which have the same characteristics as Datalog programs [35], while still allowing general Prolog expressions such as arithmetic operations.

```
%| rule: Home signal too close to first facing switch.
%| type: technical
%| severity: error
homeSignalBeforeFacingSwitchError(S,SW) :-
    firstFacingSwitch(B,SW,DIR),
    homeSignalBetween(S,B,SW),
    distance(S,SW,DIR,L), L < 200.
```

Figure 7: Structured comments on rule violation expression

The translation from railML to Datalog facts assumes that the document is valid railML, which may be checked with general XML schema validators, or a specialized railML validator.

5.1 Counterexample Presentation

When rule violations are found, the railway engineer will benefit from information about the following:

- Which rule was violated (textual message containing a reference to the source of the rule or a justification in the case of expert knowledge rules).
- Where the rule was violated (identity of objects involved).

Also, classification of rules based on e.g. *discipline* and *severity* may be useful in many cases. In the rule databases, this may be accomplished through the use of *structured comments*, similar to the common practice of including structured documentation in computer programs, such as JavaDoc (see Figure 7 for an example). A program parses the structured comments and forwards corresponding queries to the logic programming solver. Any violations returned are associated with the information in the comments, so that the combination can be used to present a helpful message to the user. A prototype CAD add-on program for Autodesk AutoCAD was implemented, see Figure 8.

5.2 Case Study Results

The rules concerning signalling layout and interlocking from *Jernbaneverket* [17] described above were checked in the railML representation of the Arna-Fløen project which is an ongoing design project in Anacon AS (now merged with Norconsult). Each object was associated with one or more construction phases, which we call phase A and phase B, which also corresponds with two operational phases. The station CAD model that was used for the work with the Arna station (phase A and B combined) included 25 switches, 55 connections, 74 train detectors, and 74 signals. The interlocking consisted of 23 and 42 elementary routes in operational phase A and B respectively.

	Testing station	Arna phase A	Arna phase B
Relevant components	15	152	231
Interlocking routes	2	23	42
Datalog facts	85	8283	9159
Running time (s)	0.1	4.4	9.4

Table 1: Case study size and running times on a standard laptop.

The Arna station design project and the corresponding CAD model has been in progress since 2013, and the method of integrating railML fragments into the CAD database, as described in Section 3, has been in use for about one year. Engineers working on this model are now routinely adding the required railML properties to the signalling components as part of their CAD modelling process. This allowed a fully automatic transfer of the railML station description to the verification tool. Several simplified models were made also for testing the correct functioning of the concept predicates and rule violation predicates. The rule collection consisted of 37 derived concepts, 5 consistency predicates, and 8 technical predicates. Running times for the verification procedure can be found in Table 1.

6 Conclusions, Related and Further Work

We have demonstrated a logical formalism in which railway layout and interlocking constraints and technical rules may be expressed, and which can be decided by logic programming proof methods with polynomial time complexity. This allows verification of railway signalling designs against infrastructure manager rules and regulations. It also allows to build and maintain a formally expressed body of expert knowledge, which may be exchanged between engineers and automatically checked against designs.

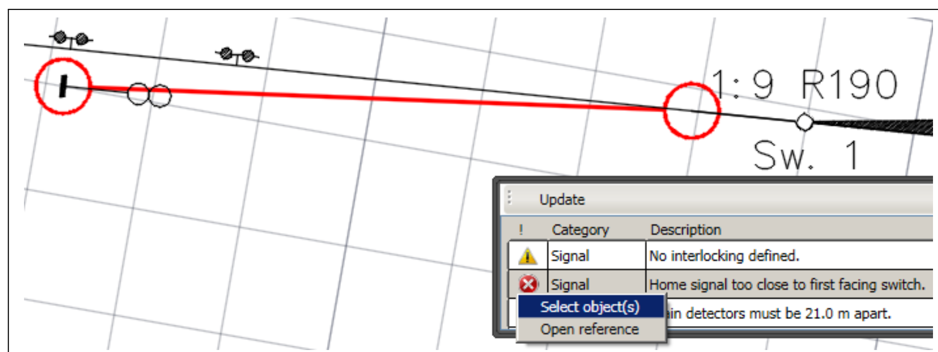


Figure 8: Counterexample presentation within an interactive CAD environment.

Related work.

Railway control systems and signalling designs are a fertile ground for formal methods. See [1, 11] for an overview over various approaches and pointers to the literature, applying formal methods in various phases of railway design. For a slightly more dated state-of-the-art survey, see [16]. In particular, safety of interlockings has been intensively formalized and studied, using for instance VDM [13] and the B-method, resp. Event-B [19]. Model checking has proved particularly attractive for tackling the safety of interlocking, and various model checkers and temporal logics have been used, cf. e.g. [5, 38, 9] [28, 23, 14, 9]. Critically evaluating practicability, [12] investigate applicability of model checking for interlocking tables using NuSMV resp. Spin, two prominent representatives of BBD-based symbolic model checking, resp. explicit state model checking. The research shows that interlocking systems of realistic size are currently out of reach for both flavors of general purpose model checkers. To mitigate the state-space explosion problem, [15] uses *bounded* model checking [7] for railway designs and interlocking systems.Instead of attempting an exaustive coverage of the state-space, symbolically or explicitly, bounded model checking analysis (the behavior of) a given system only up to a given bound (which is raised incrementally in case analyzing a problem instance is inconclusive).This restriction allows use use SAT solving techniques in the analysis. The paper uses a variant of linear temporal logic (LTL) for property specification (concentrating on safety properties and including existential quantification for) and employs so-call k-induction. [39] investigates to exploit domain-specific knowledge about interlocking verification to obtain good variable orderings when encoding the systems to be verified in a BDD-based symbolic model checker. An influential technology is the tool-based support for verified code generation for railway interlockings from Prover AB Sweden [30][2]. Prover is an automated theorem prover, using Stålmarck's method [34] of tautology checking.

Also logic (programming) languages, like Prolog or Datalog, have been used for representing and checking various aspects of railway designs. For the verification of signalling of an interlocking design [18] uses a Prolog data base to represent the topology and the layout, where for the the verification, the work uses a separate SAT solver. As this work, [24][25] use logic programming for verification of interlocking systems. In particular, the work uses a specific version of so-called annotated logic, namely annotated logic programs with strong negation, ALPSN). In general and beyond the railway system domain, recent times hav seen renewed research interest in Datalog, see for instance the collection [8]. Datalog has in particular been used for formalizing and efficiently implementing program analyses [33, 37]. [32] present Doop, a context-sensitive points-to analysis framework for Java.

The mentioned works generally include *dynamic* aspects of the railway in their checking, like train positions and the interlocking state. This is in contrast to our work, which focuses on checking against a formalization of the general design

rules issued by the regulatory bodies, thus concentrating on static aspects such as the signalling layout. This makes the notorious state-space explosion problem less urgent and makes an integration into the standard design workflow within the existing CAD tool practical. A description of using semantic web technologies for checking static railway layout properties can be found in [22].

Future work.

In the future work with RailComplete AS, we will focus on extending the rule base to contain all relevant signalling and interlocking rules from [17] , evaluating the performance of our verification on a larger scale. Design information and rules about other railway control systems, such as geographical interlockings and Automatic Train Control (ATC) systems could also be included. The current work is assuming Norwegian regulations, but the European Rail Traffic Management System (ERTMS) is expected to be used in the future, and the impact on verification should be investigated.

Finally, we plan to extend from consistency checking to optimization of designs. Optimization requires significantly larger computational effort, and the relation between Datalog and more expressive logical programming frameworks could become relevant.

Acknowledgments.

We thank Anacon AS and RailComplete AS, especially senior engineer Claus Feyling, for guidance and support on railway and signalling design methodology and philosophy.

References

[1] D. Bjørner. New results and trends in formal techniques for the development of software in transporation systems. In L'Harmattan Hongrie, editor, *Proceedings of the Symposium on Formal Methods for Railway Operation and Control Systems (FORMS'03)*. Springer-Verlag, 2003.

[2] A. Borälv and G. Stålmarck. Prover technology in railways. In Hinchey and Bowen [16], pages 329–305.

[3] Mark Bosschaart, Egidio Quaglietta, Bob Janssen, and Rob M. P. Goverde. Efficient formalization of railway interlocking data in RailML. *Information Systems*, 49:126–141, 2015.

[4] Jean-Louis Boulanger. *CENELEC 50128 and IEC 62279 Standards*. Wiley-ISTE, March 2015.

[5] S. Busard, Q. Cappart, C. Limbrée, C. Pecheur, and P. Schaus. Verification of Railway Interlocking Systems. *Electronic Proceedings in Theoretical Computer Science, Special Issue for the Proceedings of the 4th International Workshop on Engineering Safety and Security Systems (Workshop at FM'15, Oslo)*, 184, June 2015.

[6] CENELEC (2011). EN50128 – Railway Applications — Communication, Signalling and Processing Systems — Software for Railway Control and Protection Systems, 2011.

[7] E. M. Clarke, A. Biere, R. Raimi, and Y Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19:7–34, 2001.

[8] Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Sellers, editors. *Datalog Reloaded. First International Workshop 2010*, volume 6702 of *Lecture Notes in Computer Science*. Springer-Verlag, 2011.

[9] Cindy Eisner. Using symbolic model checking to verify the railway stations of Hoorn-Kersenboogerd and Heerhuowaard. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods, 10th IFIP WG 10.5 Advanced Research Working Conference, CHARME '99*, number 1703 in Lecture Notes in Computer Science, pages 97–109. Springer-Verlag, 1999.

[10] Harrison Eiteljorg II, Kate Fernie, Jeremy Huggett, and Damian Robinson. *Archaeology Data Service / Digital Antiquity Guides to Good Practice*, chapter CAD: A Guide to Good Practice. Archaeology Data Service, University of York, UK, 2011.

[11] A. Fantechi, W. Fokkink, and A. Morzenti. Some trends in formal methods applications to railway signalling. In *Formal Methods for Industrial Critical Systems*, pages 61–84. John Wiley & Sons Inc., 2012.

[12] A. Ferrari, G. Magnani, D. Grasso, and A. Fantechi. Model checking interlocking control tables. In Eckehard Schnieder and Geza Tarnai, editors, *FORMS/FORMAT 2010*, pages 107–115. Springer-Verlag, 2011.

[13] Mitsuyoshi Fukuda, Yuji Hirao, and Takahiko Ogino. VDM specification of an interlocking system and a simulator for its validation. In *9th IFAC Symposium Control in Transportation Systems 2000 Proceedings Vol.1*, pages 218–223, Braunschweig, 2000. IFAC.

[14] S. Gnesi, G. Lenzini, D. Latella, C. Abbaneo, A. Amendola, and P. Marmo. Automatic Spin validation of a safety critical railway control system. In *Proc. of the IEEE Conference onf Dependable Systems and Networks*, pages 119–124. IEEE Computer Society Press, 2000.

[15] Anne E. Haxthausen, Jan Peleska, and Ralf Pinger. Applied bounded model checking for interlocking system designs. In *Revised Selected Papers of the SEFM 2013 Collocated Workshops on Software Engineering and Formal Methods*, volume 8368 of *Lecture Notes in Computer Science*, pages 205–220. Springer-Verlag, 2014.

[16] Michael G. Hinchey and Jonathan P. Bowen, editors. *Industrial-Strength Formal Methods*. International Series in Formal Methods. Springer-Verlag, 1999.

[17] Jernbaneverket. Teknisk regelverk. `http://trv.jbv.no/`, 2015.

[18] Karim Kanso, Faron Moller, and Anton Setzer. Automated verification of signalling principles in railway interlocking systems. *Electronic Notes in Theoretical Computer Science*, 250(2):19–31, 2009. Proceedings of the Eighth International Workshop on Automated Verification of Critical Systems (AVoCS 2008).

[19] T. Lecomte, L. Burdy, and M. Leuschel. Formally checking large data sets in the railways. In *Proceedings of DS-Event-B 2012: Advances in Developing Dependable Systems in Event-B. In conjunction with ICFEM 2012*, November 2012.

[20] Leonid Libkin. *Elements of Finite Model Theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer-Verlag, 2004.

[21] Orna Lichtenstein and Amir Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Twelfth Annual Symposium on Principles of Programming Languages (POPL) (New Orleans, LA)*, pages 97–107. ACM, January 1985.

[22] M. Lodemann, N. Luttenberger, and E. Schulz. Semantic computing for railway infrastructure verification. In *Semantic Computing (ICSC), 2013 IEEE Seventh International Conference on*, pages 371–376, Sept 2013.

[23] A. Mirabadi and M. B. Yazdi. Automatic generation and verification of railway interlocking tables using FSM and NuSMV. *Transport Problems: An International Scientific Journal*, 2009.

[24] K. Nakamatsu, Y. Kiuchi, W.Y. Chen, and S.L. Chung. Intelligent railway interlocking safety verification based on annotated logic program and its simulator. In *Networking, Sensing and Control, 2004 IEEE International Conference on*, volume 1, pages 694–699, March 2004.

[25] Kazumi Nakamatsu, Yosuke Kiuchi, and Atsuyuki Suzuki. EVALPSN based railway interlocking simulator. In Mircea Gh. Negoita, Robert J. Howlett, and Lakhmi C. Jain, editors, *Knowledge-Based Intelligent Information and Engineering Systems*, volume 3214 of *Lecture Notes in Artificial Intelligence*, pages 961–967. Springer-Verlag, 2004.

[26] Andrew Nash, Daniel Huerlimann, Jörg Schütte, and Vasco Paul Krauss. RailML — a standard data interface for railroad applications. *WIT Press*, 2004.

[27] J. Pachl. *Railway Operation and Control*. VTD Rail Publishing, 2015.

[28] O. Pavlovic and H. Ehrich. Model checking PLC software written in function block diagram. In *ICST'10*, pages 439–448, 2010.

[29] Amir Pnueli. The temporal logic of programs. In *Proceeding of the 18th Annual Symposium on Foundations of Computer Science*, pages 45–57, 1977.

[30] Prover AB homepage. `http://www.prover.com/`, 2015.

[31] railML. The XML interface for railway applications. `http://www.railml.org`, 2016.

[32] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: Understanding context-sensitivity (the making of a precise and scalable pointer analysis). In *Proceedings of POPL '11*. ACM, January 2011.

[33] Yannis Smaragdakis and Martin Bravenboer. Using Datalog for fast and easy program analysis. In de Moor et al. [8].

[34] Gunnar Stalmårck. A system for determining logic theorems by applying values and rules to triplets that are generated from a formula. Swedish Patent No. 467 076 (approved 1992), U.S. Patent No. 5 276 897 (approved 1994), European Patent No. 0403 454 (approved 1995), 1992.

[35] Terrance Swift and David S. Warren. XSB: Extending prolog with tabled logic programming. *Theory Pract. Log. Program.*, 12(1-2):157–187, January 2012.

[36] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems (Volume I & II)*. Computer Society Press, 1988.

[37] J. Whaley, D. Avots, M. Carbin, and M. S. Lam. Using Datalog with binary decision diagrams for program analysis. In K. Yi, editor, *Proceedings of APLAS'05*, volume 3780 of *Lecture Notes in Computer Science*, pages 97–108, 2005.

[38] K. Winter, W. Johnston, P. Robinson, P. Strooper, and L. van den Berg. Tool support for checking railway interlocking designs. In *Proceedings of the 10th Australian Workshop on Safety Critical Systems and Software*, pages 101–107, 2006.

[39] Kirsten Winter. Optimising ordering strategies for symbolic model checking interlocking control tables. In *5th International Symposium on Leveraging Applications of Formal Methods, Verification, and Validation (ISOLA'12), Part II*, volume 7610 of *Lecture Notes in Computer Science*. Springer-Verlag, 2012.

7 Appendix: Example of Program Inputs

The appendix is for reviewing only and should not be regarded as part of the paper. The contents of the appendix, as well as more details and examples will appear in a technical report towards the end of January.

This section contains example input and output of the verification procedure described above in the paper. First, we give an example of a railML document describing a station.

```
<?xml version="1.0" encoding="utf-8"?>
<infrastructure xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns
    :xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http
    ://www.railml.org/schemas/2013">
  <tracks>
    <track id="t1">
      <trackTopology>
        <trackBegin id="tb1" pos="0"> <macroscopicNode /> </
            trackBegin>
        <trackEnd id="te1" pos="500"> <macroscopicNode /> </trackEnd
            >
        <connections>
          <switch id="sw1" pos="100"> <connection id="sw1c" ref="t2bc
              " course="left" orientation="outgoing" /> </switch>
          <switch id="sw2" pos="400"> <connection id="sw2c" ref="t2ec
              " course="right" orientation="incoming" /> </switch>
        </connections>
      </trackTopology>
      <ocsElements>
        <signals>
          <signal id="sig1" pos="50" type="main" function="home" dir=
              "up" />
          <signal id="sig2" pos="350" type="main" function="exit" dir
              ="up" />
        </signals>
        <trainDetectionElements>
          <trainDetector id="ac1" name="Tp(x/1)" pos="48.9"
              axleCounting="true" />
          <trainDetector id="ac2" name="Tp(1/y)" pos="350.1"
              axleCounting="true" />
        </trainDetectionElements>
      </ocsElements>
    </track>
    <track id="t2">
      <trackTopology>
        <trackBegin id="t2b" pos="0"> <connection id="t2bc" ref="
            sw1c" /> </trackBegin>
        <trackEnd id="t2e" pos="300"> <connection id="t2ec" ref="
            sw2c" /> </trackEnd>
      </trackTopology>
      <ocsElements>
        <signals>
          <signal id="sig3" pos="225" type="main" function="exit" dir
              ="up"/>
```

```
        </signals>
        <trainDetectionElements>
          <trainDetector id="ac3" name="Tp(1/z)" pos="225.1"
              axleCounting="true" />
          <trainDetector id="ac4" name="Tp(1/z)" pos="220.6"
              axleCounting="true" />
        </trainDetectionElements>
      </ocsElements>
    </track>
  </tracks>
</infrastructure>
```

A simplified rulebase for static railway infrastructure verification is included below.

```
%|| railcons_ruleset:
%|| title: Example ruleset for static railway infrastructure
    verification

%| rule: X belongs to Y, typically a track.
%| type: definition
belongsTo(X,Y) :- childElement(X,Y).
belongsTo(X,Y) :- belongsTo(X,Z), belongsTo(Z,Y).

%| rule: Element which is connected to a track.
%| type: definition
trackElement(X) :- track(T), childElement(X,T).

%| rule: Station boundary
%| type: definition
stationBoundary(B) :- macroscopicNode(B).

%| rule: Connection exists between objects (switch, track
    continuation, crossing, etc.).
%| type: definition
connection(A,B) :- trackElement(A), trackElement(B), connection(C1
    ), belongsTo(C1,A),
               belongsTo(C2,B), ref(C1,C2).
connection(A,B) :- connection(B,A).

%| rule: Connection with direction (switch, track continuation,
    crossing, etc.)
%| type: definition
connection(A,B,D) :- connection(A,B), trackBegin(B), D='up'.
connection(A,B,D) :- connection(A,B), trackEnd(B), D='down'.
connection(A,B,D) :- connection(B,A,O), oppositeDirection(D,O).

%| rule: Switch position opposite (left/right)
%| type: definition
oppositePosition(left,right).
oppositePosition(right,left).

%| rule: Direction opposite (up/down)
%| type: definition
```

```
oppositeDirection(up,down).
oppositeDirection(down,up).

%| rule: Inconsistent connection information.
%| type: consistency
twoWayConnectionMissingError(X) :- connection(X), ref(X,Y), not(
    ref(Y,X)).

%| rule: Objects belong to same track.
%| type: definition
directlyConnected(A,B) :- track(T), belongsTo(A,T), belongsTo(B,T)
    .

%| rule: Objects are connected by tracks.
%| type: definition
connected(A,B) :- directlyConnected(A,B).
connected(A,B) :- connection(A,B).
connected(A,B) :- connected(A,X), connected(X,B).

%| rule: Objects are following (in given direction) on the same
    track.
%| type: definition
directlyFollowing(A,B,'up') :- directlyConnected(A,B), A \= B, pos
    (A,PA), pos(B,PB), PA < PB.
directlyFollowing(A,B,'down') :- directlyConnected(A,B), A \= B,
    pos(A,PA), pos(B,PB), PA > PB.

%| rule: Objects are following (in direction D).
%| type: definition
following(A,B,D) :- directlyFollowing(A,B,D).
following(A,B,D) :- connection(A,B,D).
following(A,B,D) :- following(A,X,D), following(X,B,D).

%| rule: Objects have a distance of L, on the same track.
%| type: definition
directDistance(A,B,D,L) :- directlyFollowing(A,B,D), pos(A,PA),
    pos(B,PB), PB > PA, L is PB-PA.
directDistance(A,B,D,L) :- directlyFollowing(A,B,D), pos(A,PA),
    pos(B,PB), PB < PA, L is PA-PB.

%| rule: Connection to same track.
%| type: consistency
connectionToSameTrack(A,B) :- connection(A,B), directlyConnected(A
    ,B).

%| rule: Objects have a distance of L, along track.
%| type: definition
distance(A,B,D,L) :- directDistance(A,B,D,L).
distance(A,B,D,L) :- connection(A,B,D), L is 0.
distance(A,B,D,L) :- not(directlyConnected(A,B)), directDistance(A
    ,X,D,L1),
                distance(X,B,D,L2), L is L1+L2.

%| rule: Object between two other objects (along tracks).
```

```
%| type: definition
between(A,X,B) :- following(A,X,D), following(X,B,D).

%| rule: Missing switch orientation.
%| type: consistency
missingSwitchOrientation(X) :- switch(X), not(switchOrientation(X,
    _)).

%| rule: Switch orientation derived from the connection relation.
%| type: definition
switchOrientation(Sw,O) :- switch(Sw), connection(Sw,X,D),
    orientationDirection(O,D).

%| rule: Orientation/direction correspondence (up is outgoing, i.e
    . increasing mileage)
%| type: definition
orientationDirection('outgoing','up').
orientationDirection('incoming','down').

%| rule: Facing switch definition
%| type: definition
switchFacing(SW,DIR) :- switchOrientation(SW,O),
    orientationDirection(O,DIR).

%| rule: First facing switch in station, coming from a macroscopic
     node.
%| type: definition
firstFacingSwitch(B,SW,DIR) :- stationBoundary(B),
   switchFacing(SW,DIR), following(B,SW,DIR).

%| rule: Missing signal type.
%| type: consistency
missingSignalType(X) :- signal(X), not(type(X,_)).

%| rule: Main signal with specified directionality.
%| type: definition
mainSignalDirection(X,D) :- signal(X), dir(X,D), (type(X, 'main')
    ; type(X,'combined')).

%| rule: Home signal exists between two elements.
%| type: definition
homeSignalBetween(S,B,SW) :-
   signal(S), function(S,'home'), between(B,S,SW).

%| rule: Missing home signal in station entry path.
%| type: technical
%| severity: error
missingHomeSignalBeforeFacingSwitch(B,SW) :-
   firstFacingSwitch(B,SW,_),
   (not(homeSignalBetween(_,B,SW))).

%| rule: Home signal too close to first facing switch.
%| type: technical
%| severity: error
```

```
homeSignalBeforeFacingSwitchError(S,SW) :-
   firstFacingSwitch(B,SW,DIR),
   homeSignalBetween(S,B,SW),
   distance(S,SW,DIR,L), L < 200.

%| rule: Train detectors must be 21.0 m apart.
%| type: technical
trainDetectorsTooClose(A,B) :-
   trainDetector(A), trainDetector(B),
   distance(A,B,'up',L), L < 21.0.
```

Finally, the output (YAML format) of the verification tool.


```
issues:
  - rule:
     type: technical
     severity: error
     rule: Missing home signal in station entry path.
    ids:
      - te1
      - sw2
  - rule:
     severity: error
     rule: Home signal too close to first facing switch.
     type: technical
    ids:
      - sig1
      - sw1
  - rule:
     rule: Train detectors must be 21.0 m apart.
     type: technical
    ids:
      - ac4
      - ac3
```