Efficient Verification of Railway Infrastructure Designs Against Standard Regulations

Christian Johansen · Bjørnar Luteberget

2017-01-27

Abstract In designing safety-critical infrastructures s.a. railway systems, engineers often have to deal with complex and large-scale designs. Formal methods can play an important role in helping automate or check various tasks. Especially for railway designs formal methods have been used in verifying the safety of so-called interlockings through model checking, which deals with state change and rather complex properties, usually incurring considerable computational burden (e.g., the state-space explosion problem). In contrast, we focus on static infrastructure models and are interested in checking requirements coming from design guidelines and regulations. Our goal is to automate the manual work of the railway engineers through software that is fast enough to do verification on-the-fly, thus being able to be included in the railway design tools, much like a compiler in an IDE.

In consequence, this paper describes the integration of formal methods into the railway design process, by formalizing relevant technical regulations 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, developed under the name RailCOMPLETE®. We apply this tool chain in a Norwegian railway project, the upgrade of the Arna railway station.

Keywords railway designs, automation, logic programming, signalling, railway infrastructure, railML, CAD, Datalog

This work was partially supported by the RailComplete AS company and the project RailCons funded by the Norwegian Research Council.

Christian Johansen
Department of Informatics, University of Oslo, Norway
E-mail: cristi@ifi.uio.no

Bjørnar Luteberget
RailComplete AS, Sandvika, Norway (formerly Anacon AS) and
Department of Informatics, University of Oslo, Norway
E-mail: bjornar.luteberget@railcomplete.no

1 Introduction

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 is 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. [37]).

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 regulations, writing tables, and coordinating disciplines. Many of these manual checks are simple enough to be automated. 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.

With the purpose of increasing the degree of automation, we present results on integrating formal methods into the railway design process by the following means:

- Formalizing rules governing track and signalling layout, and interlocking.
- Using the standardized "railway markup language" railML[1]. as basis and exchange format for the formalization.
- Modeling the concepts describing a railway design in the logic of Datalog; and developing an automated generation of the model from the railML representation.
- Developing a prototype tool and integrating it 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 [48]. We also show the integration with existing railway engineering workflow by using CAD models directly. This enables us to verify compliance with regulations continuously as the design process changes the station layout and interlocking. Based on railML [35] as intermediary language, our results can be easily adopted by anyone who uses this international standard.

The approach presented in this paper could be applied also to other engineering disciplines, such as catenary power lines, track works, and others, which have similar design regulations and often make use of a similar CAD environment. However, this paper uses the signalling and interlocking design process and shows how it can be improved by automation using formal methods.

The work uses as case study the software and the design (presently under development) used in the *Arna-Fløen* upgrade project,[2] 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 that the tool's performance is acceptable for interactive work within the CAD software.

[1] railML.org: `https://www.railml.org/`

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

The paper is organized as follows. Section 2 presents aspects of the railway domain relevant for this work. Section 3 presents our approach to extending CAD programs with domain-specific data, in our case for railway signalling based on the railML format. Section 4 presents our formalization of the rules and concepts of railway design as logical formulas amenable for the Datalog implementation and checking. Section 5 proposes a tool chain that extends CAD with formal representations of signalling layout and interlocking. Section 6 provides information about our tool implementation, including details about counterexample presentation and empirical evaluation using the case study. Section 7 describes our results of performing verification in an incremental manner for better integration with interactive design tools. We conclude in Section 8 with related and future work.

2 The railway signalling design process

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

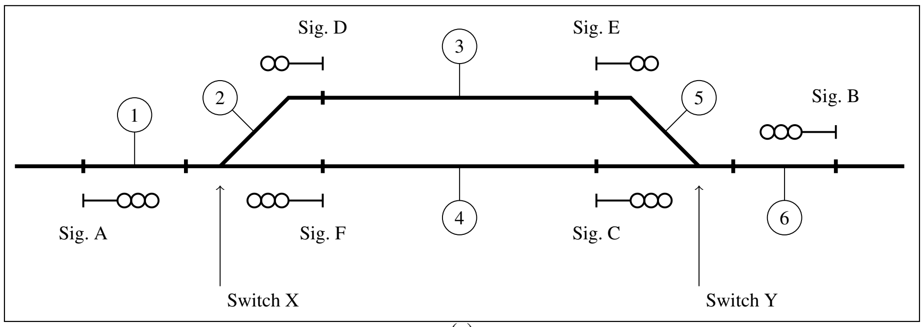
Railway construction projects rely heavily on *computer aided design* (CAD) tools to map out railway station layouts. The various disciplines within a project, such as civil works, track works, signalling, or catenary power lines, work with *coordinated CAD models*. These CAD models contain a major part of the work performed by engineers, and are a collaboration tool for communication between disciplines. The signalling component layout is worked out by the signalling engineers as part of the design process. Signals, train detectors, derailers, etc., are drawn using symbols in a 2D geographical CAD model. An example of a layout drawing made from a CAD model is given in Figure 1.

Track layout details, which are input for 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.

2.2 Interlocking Specification

An interlocking is an interconnection of signals and switches to ensure that train movements are performed in a safe sequence [37]. 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 (or safety zones), must not be in use.



(a)

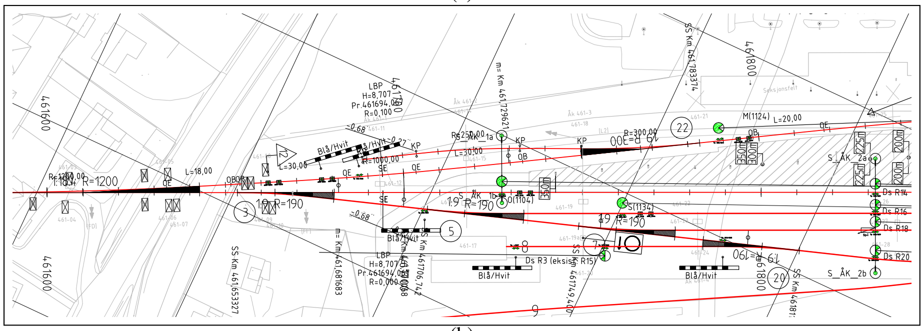

(b)

Fig. 1: (a) Example *schematic* construction drawing. (b) Cut-out from 2D geographical CAD model (construction drawing) of preliminary design of the Arna station signalling.

Route	Start	End	Sw. pos	Detection sections	Conflicts
AC	A	C	X right	1, 2, 4	AE, BF
AE	A	E	X left	1, 2, 3	AC, BD
BF	B	F	Y left	4, 5, 6	AC, BD
BD	B	D	Y right	3, 5, 6	AE, BF

Fig. 2: Example of a *tabular interlocking*, showing available routes and their conditions.

3 Semantic CAD

Civil engineering construction projects, such as railway projects, make heavy use of computer-aided design (CAD) tools to model the *geometric* aspects of the construction project and its product. The origins of CAD tools are in the computerizing of traditional *drafting*, which produces human-readable technical drawings that are used as plans and documentation for the construction. Mainstream CAD tools are mainly concerned with manipulating databases of geometrical objects constituting 2D or 3D representations of spatial properties, and the production of human-readable drawings which depict these geometrical structures.

The DWG file format created for the Autodesk AutoCAD software is a de facto standard in many engineering disciplines, and this format has also been adopted by several other CAD software packages.

3.1 Grouping geometry into blocks

Grouping together several geometrical features into a single unit is in CAD terminology called “making a *block*”. This allows the CAD user to create models more efficiently, by reusing commonly used components. The blocks, which may represent things such as chairs, doors, or railway signals, also create the opportunity to store higher-level information in a CAD model, other than the purely geometrical description. For example, if one uses a railway signal block to model a signal in a railway station, a program can count the number of signals in a model.

This idea can be extended by adding any number of attributes to a block. For a railway signal, we can add attributes that describe e.g. to which track it signals, along with its type, function, and direction. The CAD object database does then not only contain the geometrical objects, such as lines, curves, triangles, cubes, etc., but groups these primitives into higher-level concepts which are closer to the representation that one uses to reason about the actual working of the railway infrastructure.

With a good library of blocks (which we call a *symbol library*), the engineer can more efficiently build the geometric CAD models which lead to human-readable drawings, but they are also building a machine-readable model of high-level railway concepts. We call this *semantic CAD*. While this concept is also a part of building information modeling (BIM), BIM also includes many other concepts such as 3D visualization, time (“4D”), and cost (“5D”).

The verification of signalling and interlocking rules requires information about 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 *railML* [35]. In the CAD industry-standard DWG file format, 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,

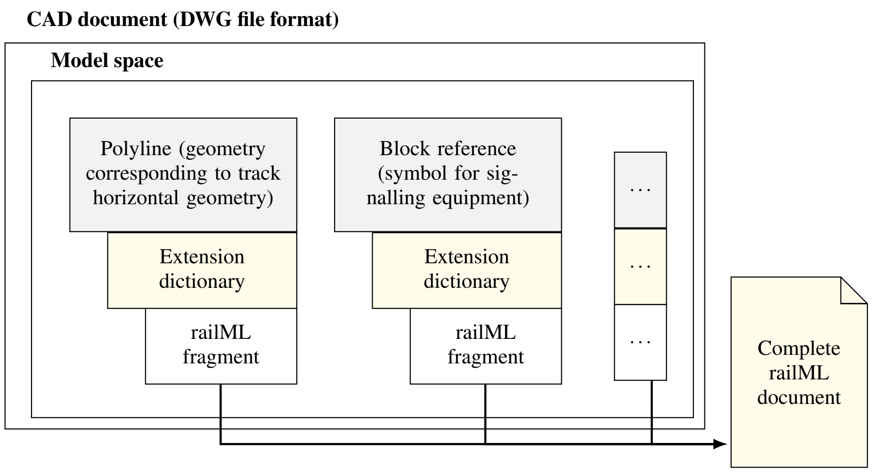

CAD document (DWG file format)
Model space
Polyline (geometry corresponding to track horizontal geometry)
Extension dictionary
railML fragment
Block reference (symbol for signalling equipment)
Extension dictionary
railML fragment
...
...
...
Complete railML document

Fig. 3: railML integrated into a CAD database

see Figure 3 . Thus, we can compile the complete railML representation of the station from the CAD model.

3.2 Object type descriptions

It is necessary to decide which objects in the CAD model should be associated with which data types, i.e. what attributes should be stored in the symbols. This is comparable to specifying an object's *class* in an object-oriented programming language. To do this, we create an *object type description* which augments the symbol library with class information. Whenever the user adds a symbol, its data editor is determined by the assigned class, and vice versa: when e.g. a railML object is imported into CAD, its corresponding symbol is inserted in the graphical model.

3.3 Interlocking and train protection systems

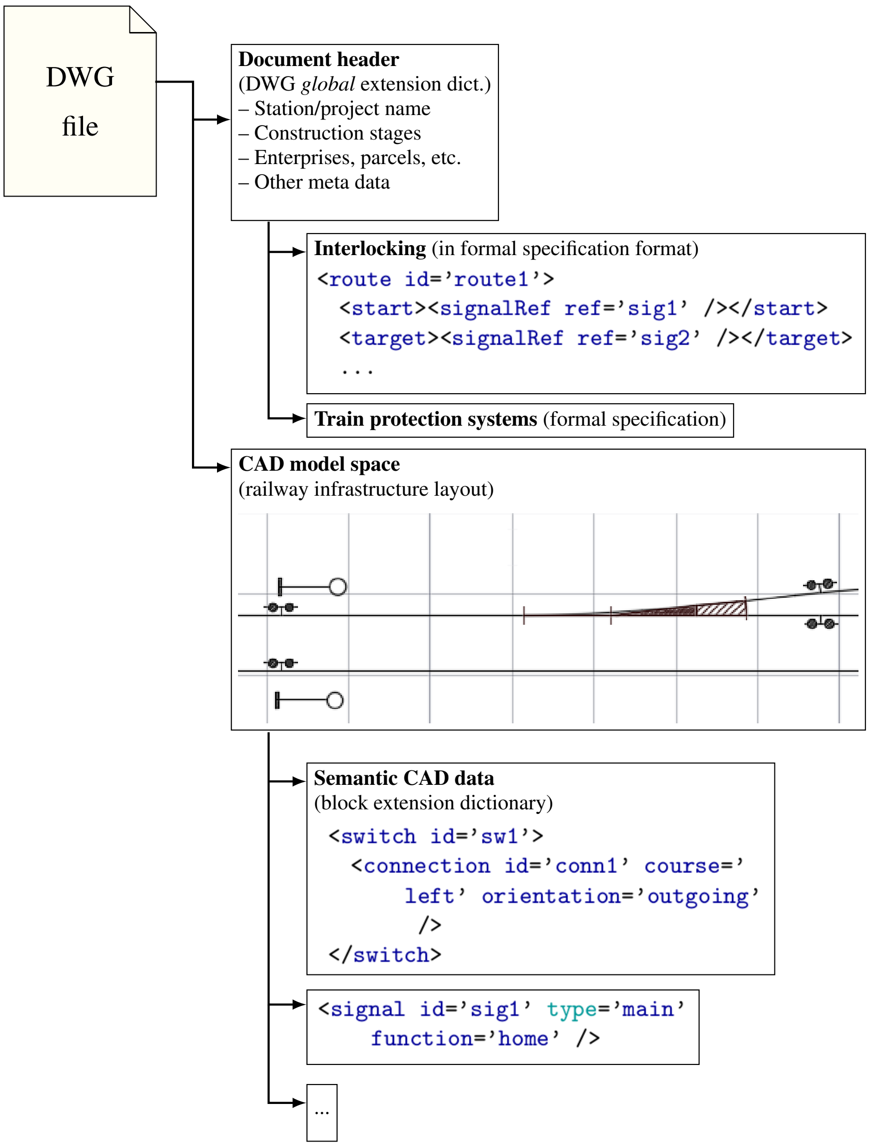
Besides the CAD model layout, the design of a railway station's signalling consists also of specifications for the interlocking and train protection (speed control) systems. These specifications are used to build the interlocking controllers and speed controllers, and they model the behavior of the signalling equipment and its interaction with trains. These systems are tightly linked to the station layout.

A formal representation of the interlocking and train protection 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, geographical layout of the train station also contains a machine-readable model which fully describes both the component layout and the functional specification of the interlocking and train protection systems. This allows analysis of the operational aspects of the train station directly in a familiar editable CAD model. See Figure 4 for an overview of this architecture.

4 Logic programming and knowledge-base systems

We have shown how to associate semantic information with CAD symbols, but in order to automatically verify rules and regulations on this railway infrastructure model, we need a computer program which can check each property for violations with the given model as input.

A straight-forward approach to making such a program could be to create some search function on the graph implicit in the track network. This procedure should allow, for example, to find the nearest object of a given type, or to find all paths between two points. Then we would describe a checking procedure for each rule. Consider for example, the *home signal* regulation from Property 1, which says "A *home main signal* shall be placed at least 200 m in front of the first controlled, facing switch in the entry train path.". Checking such a property can be done by iterating over tracks, locating station boundaries, starting a search function to locate the relevant facing switches, starting another search backwards to check that there is a home signal, and so on. The amount of code required to do this in a mainstream programming language can become large, and this code is often very specific to a given railway administration.


DWG
file
Document header
(DWG *global* extension dict.)
– Station/project name
– Construction stages
– Enterprises, parcels, etc.
– Other meta data
Interlocking (in formal specification format)
Train protection systems (formal specification)
CAD model space
(railway infrastructure layout)
Semantic CAD data
(block extension dictionary)
...

```
<route id='route1'>
  <start><signalRef ref='sig1' /></start>
  <target><signalRef ref='sig2' /></target>
  ...
```

```
<switch id='sw1'>
  <connection id='conn1' course='
      left' orientation='outgoing'
       />
</switch>
```

```
<signal id='sig1' type='main'
    function='home' />
```

Fig. 4: Semantic CAD document organization including interlocking specification.

Better suited to manage the large amounts of code required for a large number of rules, is *logic programming*, which allows rule descriptions that are much closer to the original specifications than in a mainstream programming language.

4.1 Logic programming

Logic programming [36] is a family of programming languages based on formal logic. Logic programs are *declarative*, i.e. they describe properties of the solution of a problem rather than a calculation procedure for finding the solution. This separates the concerns of expressing rules about railway systems from the algorithms required to do automatic analysis. This separation allows one to systematically maintain a large set of rules, and decouple the tool implementation from the set of concepts, rules and expert knowledge that is specific to a railway administration.

We have successfully used the Datalog language [48], a subset of the more well-known Prolog language, for verifying many properties given as technical rules and expert knowledge. It allows concise formulations of railway concepts, and queries can be efficiently calculated.

Ideally, we would like the railway engineers themselves, without much programming education, to be able to create and maintain the set of rules which is used for the verification. This separation of logic and algorithm is a step in this direction, because non-IT experts can work on the rules without considering how the calculations are implemented. However, the strict formalism and subtle semantics of logic programming are still a challenge for an inexperienced programmer.

Still we think that it is feasible for inexperienced logic programmers to do some of the maintenance of a rule base for the following reasons:

1. The most basic concepts, such as connectedness, distances, directions, etc., rarely need to be redefined, and may be specified by an expert programmer, and then reused.
2. Naming or documenting the basic concepts in a way that is understandable by railway engineers allows them to use these concepts without considering the actual definitions.
3. Rule formulations are often so succinct that they can be understood even without knowledge of the logic programming syntax.
4. Modification (updating) of rules, for example following a change in regulation from the railway administration, often preserves the structure of the specification, adding only a similar clause or the change of a numeric constant.
5. Templates for common rule structures can be given, so that implementing some types of rules becomes a matter of specifying e.g. only object types, directions, and distances.

We envision that a common rule base would be exchanged between all engineers working with a railway administration, and that the rule base would be worked out partly by software experts, partly by railway experts. Also, the rule base should be fairly constant, like the regulations, requiring an update frequency of perhaps once per year.

We do, however, concede that Datalog programming in general is outside what would be expected competency for a railway engineer. A higher-level domain-specific language including relevant constructs for a railway signalling design knowledge base could improve the likelihood of railway engineers being successful in creating and maintaining the regulations. Even more, this language could allow each company and each engineer to experiment with encoding different design heuristics and expert knowledge to see the effects on the verification and the design. Our planned future work (see Section 8.2) includes defining such a language and also on using a controlled natural language syntax to improve ease of comprehension.

4.2 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 [48,1,36] for more details. In its most basic form Datalog is a database query, as 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 [24, Chap. 3].

Adding fixed-point operators to conjunctive queries is a common way to mitigate the above problem while preserving decidability and polynomial time complexity.

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, along with terms corresponding to the predicate arguments, forming an expression like $p(\overrightarrow{a})$, where $\overrightarrow{a} = (a_1, a_2, \ldots, a_n)$. *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

$$r_0(\overrightarrow{x}) \;\text{:--}\; \exists \overrightarrow{y} : r_1(\overrightarrow{x_1}, \overrightarrow{y_1}), r_2(\overrightarrow{x_2}, \overrightarrow{y_2}), \ldots, r_k(\overrightarrow{x_k}, \overrightarrow{y_k}),$$

with $\bigcup_{1 \le i \le k} \overrightarrow{x_i} = \overrightarrow{x}$ and $\bigcup_{1 \le i \le k} \overrightarrow{y_i} = \overrightarrow{y}$. Datalog uses the Prolog convention of interpreting identifiers starting with a capital letter as variables, and other identifiers as constants, e.g., the clause

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

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 a graph edge relation between vertices a and b. Graph searches can now be encoded by making a transitive closure over the edge relation:

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

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

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

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 [48, 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.

4.3 Knowledge-base system

With Datalog as specification language, we build a *knowledge-base system* to perform the verification. A knowledge-base system consists of a set of facts and rules, along with an inference engine which answers queries by applying logical inference rules. For an introduction to knowledge-base systems in general, see [48, Chap. 3] or [41, Chap. 8 and 12]. We give here an overview of how we encode railway signalling properties as Datalog predicates, which in turn may be automatically checked for consistency. In our verification tool, we organize our knowledge base in the following manner:

1. **Input documents:** Predicate representation of input document, i.e. track layout and interlocking, are represented as *facts* which are converted from the railML representation stored and maintained in the CAD database by a CAD plug-in program.
2. **Derived concepts:** Predicate representation of derived concept *rules*, such as object properties, topological properties, and calculation of distances. A library of general railway concepts and administration-specific concepts and definitions are kept in a rule base which is re-used between projects.
3. **Technical rules and expert knowledge:** Predicate representation of technical rules or expert knowledge as logic programming *rules*, which encode the administration-specific rules and expert knowledge that is checked and errors reported to the user by the verification tool.
4. **Inference engine:** A Datalog evaluation engine is used as inference engine; in our case the XSB Prolog tabled logic programming system [47].

Each of these aspects are described in more detail below.

4.3.1 Input documents

Each of the XML elements and attributes is translated into a corresponding predicate. An example of translating a railML *switch* element into predicate representation is given below.

```
<switch id='sw1'>                                        switch(sw1).
  <connection id='conn1' course='left'                   connection(conn1).
       orientation='outgoing' />                   →     belongsTo(sw1,conn1).
</switch>                                                course(conn1,left).
                                                         orientation(conn1,outgoing).
```

4.3.2 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):

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

- Element name:

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

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

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

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

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

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

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

  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`):

$$\begin{aligned}&\text{signalType}(a,t) \leftarrow (\text{element}_a.\texttt{type}=t), t \in \{\text{main, distant, shunting, combined}\}\,,\\ &\text{signalFunction}(a,f) \leftarrow (\text{element}_a.\texttt{function} = f),\\ &\qquad\qquad a \in \text{Atoms}, f \in \{\text{home, intermediate, exit, blocking}\}\,.\end{aligned}$$

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

$$\text{signalType}(a,t) \Rightarrow signal(a),$$

$$\text{signalFunction}(a,f) \Rightarrow signal(a).$$

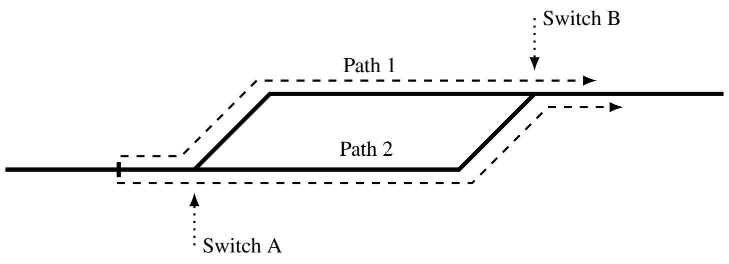


Fig. 5: Switches give rise to branching paths

These are only a few examples of predicates that are extracted from the railML document. The translator from railML to predicate form needs only to consider XML elements, attributes and sub-elements, not the specifics of railML and its type hierarchy. The complete structure 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 5. 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:

– Connection element and reference:

$$\begin{aligned}\text{connection}(a) &\leftarrow \text{element}_a \text{ is of type } \texttt{connection},\\ \text{connection}(a,b) &\leftarrow (\text{element}_a.\texttt{ref} = b).\end{aligned}$$

– Connection course and orientation:

$$\begin{aligned}\text{connectionCourse}(a,c) &\leftarrow (\text{element}_a.\texttt{course} = c), c \in \{\text{left, straight, right}\}\\ \text{connectionOrientation}(a,o) &\leftarrow (\text{element}_a.\texttt{orientation} = o),\\ &\quad a \in \text{Atoms}, o \in \{\text{outgoing, incoming}\}.\end{aligned}$$

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):

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

4.3.3 Interlocking.

An XML schema for tabular interlocking specifications is described in [5], and this format is used here, anticipating that it will become part of the railML standard schema in the future. We give some examples of how this schema is translated into predicate form:

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

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

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

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

4.4 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.

4.4.1 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 ($a \in$ Atoms, $d \in$ Direction):

$$\begin{aligned}\text{switchFacing}(a,d) \leftarrow \exists c,o : \text{switch}(a) \wedge \text{switchConnection}(a,c) \wedge\\ switchOrientation(c,o) \wedge orientationDirection(o,d).\end{aligned}$$

$$\text{switchTrailing}(a,d) \leftarrow \neg\text{switchFacing}(a,d)$$

4.4.2 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 ($a,b \in$ Atoms):

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

$$\begin{aligned}\text{connected}(a,b) \leftarrow \text{directlyConnected}(a,b) \vee (\exists c_1,c_2 : \text{connection}(c_1,c_2) \wedge\\ \text{directlyConnected}(a,c_1) \wedge \text{connected}(c_2,b)).\end{aligned}$$

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

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

$$\begin{aligned}\text{following}(a,b,d) \leftarrow\ & \text{directlyFollowing}(a,b,d) \vee\\ & \exists c_1,c_2 : \text{connection}(c_1,c_2) \wedge \text{directlyFollowing}(a,c_1,d)\\ & \wedge \text{following}(c_2,b,d)\end{aligned}$$

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

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

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

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

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

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

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

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

4.4.3 Interlocking properties.

Predicates such as existsPathWithoutSignal(a,b) which defines the method for finding elementary routes, and existsPathWithDetector(a,b) for finding adjacent train detectors, will be used as building blocks for the interlocking rules. We show here a recursive rule used for finding elementary routes:

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

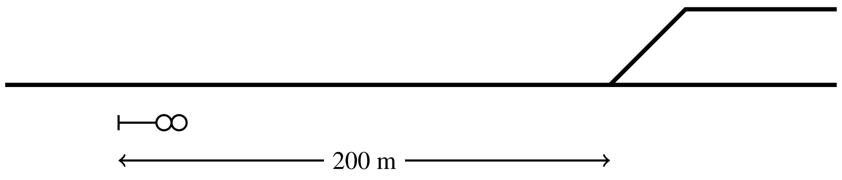
$$\begin{aligned}\text{existsPathWithoutSignal}(a,b,d) \leftarrow\ & \text{following}(a,b,d) \wedge \\ & (\neg(\exists x : \text{signal}(x) \wedge \text{between}(a,x,b)) \vee \\ & (\exists x : \text{between}(a,x,b) \wedge \text{existsPathWithoutSignal}(a,x,d) \wedge \\ & \text{existsPathWithoutSignal}(x,b,d)).\end{aligned}$$

4.5 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. All technical rules presented herein are based on the Norwegian infrastructure manager's regulations[3]. 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.

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

Property 1 (Layout: Home signal) *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 may be represented in the following way:

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

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

Checking for rule violations can be expressed as:

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

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

Property 2 (Layout: Minimum detection section length) *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:

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

Property 3 (Layout: Exit main signal) *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:

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

– Station boundaries should be preceded by an exit signal:

$$\begin{aligned}\text{exitSignalBefore}(x,d) \leftarrow & \exists s : \text{signalType}(s,\text{exit}) \wedge \text{following}(s,x,d) \\ \text{ruleViolation}_3(b) \leftarrow & \exists d : \text{stationBoundary}(b) \wedge \neg\text{exitSignalBefore}(b,d).\end{aligned}$$

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.*

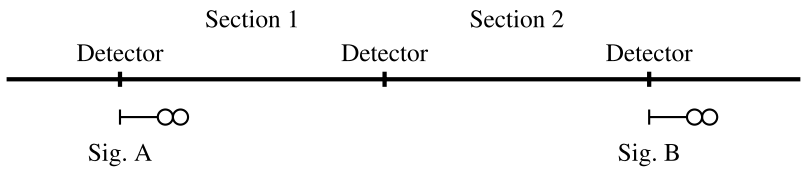
This can be represented as follows:

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

$$\begin{aligned}&\text{ruleViolation}_4(a,b,d) \leftarrow \text{defaultRoute}(a,b,d) \wedge \\ &\quad \neg(\exists r : \text{trainRoute}(r) \wedge \text{trainRouteStart}(r,a) \wedge \text{trainRouteEnd}(r,b)).\end{aligned}$$

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.*



Tabular interlocking:

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

Property 5 can be represented as follows:

$$\begin{aligned}\text{existsPathWithDetector}(a,b) \leftarrow & \exists d : \text{following}(a,b,d) \wedge \text{trainDetector}(x) \wedge \\ & \text{between}(a,x,b).\end{aligned}$$

$$\begin{aligned}\text{adjacentDetectors}(a,b) \leftarrow & \text{trainDetector}(a) \wedge \text{trainDetector}(b) \wedge \\ & \neg\text{existsPathWithDetector}(a,b),\end{aligned}$$

$$\begin{aligned}\text{detectionSectionOverlapsRoute}(r,d_a,d_b) \leftarrow\ & \text{trainRoute}(r) \wedge \\ & \text{start}(r,s_a) \wedge \text{end}(r,s_b) \wedge \\ & \text{adjacentDetectors}(d_a,d_b) \wedge \text{overlap}(s_a,s_b,d_a,d_b),\end{aligned}$$

$$\begin{aligned}\text{detectionSectionCondition}(r,d_a,d_b) \leftarrow\ & \text{detectionSectionCondition}(c) \wedge \\ & \text{belongsTo}(c,r) \wedge \text{belongsTo}(d_a,c) \wedge \text{belongsTo}(d_b,c).\end{aligned}$$

$$\begin{aligned}\text{ruleViolation}_5(r,d_a,d_b) \leftarrow\ & \\ & \text{detectionSectionOverlapsRoute}(r,d_a,d_b) \wedge \\ & \neg\text{detectionSectionCondition}(r,d_a,d_b).\end{aligned}$$

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

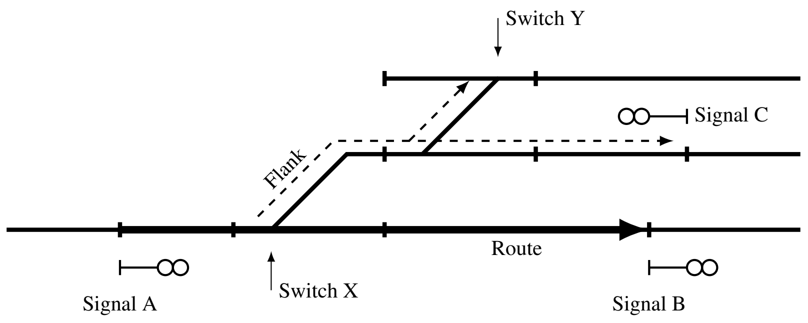


Fig. 6: The dashed path starting in switch X must be terminated in all branches by a valid flank protection object, in this case switch Y and signal C. (Property 6)

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):

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

$$\begin{aligned}\text{flankProtectionRequired}(r,x,d) \leftarrow\ & \text{trainRoute}(r) \wedge \text{start}(r,s_a) \wedge \\ & \text{end}(r,s_b) \wedge \text{switchOrientation}(x,o) \wedge \text{between}(s_a,x,s_b) \wedge \\ & \text{orientationDirection}(o,o_d) \wedge \text{oppositeDirection}(o_d,d).\end{aligned}$$

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

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

– 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:

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

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

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

5 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 (more details can be found in [26]). 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).

Figure 7 shows the overall tool chain. The software allows checking of rules and regulations of static infrastructure (described in this paper) inside the CAD environment, 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, they must account for a large variety of situations, and they 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 less 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 properties. 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 types of analysis, which are less automated or require heavier computation, such as:

– **Code generation and verification for interlockings** is possible e.g. through the formal verification framework of Prover Technology[4].

[4] Prover Technology AB: `http://www.prover.com/`

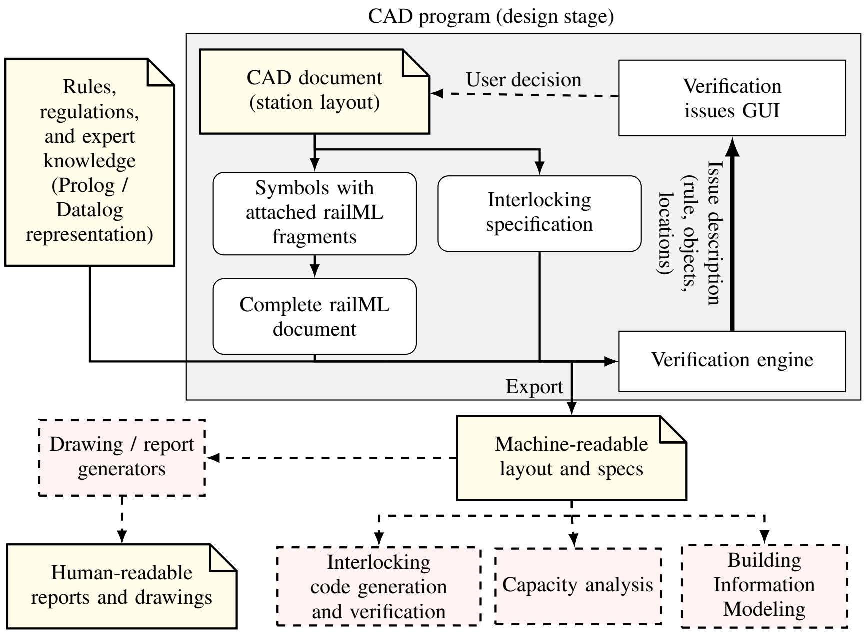

CAD program (design stage)
Rules, regulations, and expert knowledge (Prolog / Datalog representation)
CAD document (station layout)
User decision
Verification issues GUI
Symbols with attached railML fragments
Interlocking specification
Issue description (rule, objects, locations)
Complete railML document
Verification engine
Export
Drawing / report generators
Machine-readable layout and specs
Human-readable reports and drawings
Interlocking code generation and verification
Capacity analysis
Building Information Modeling

Fig. 7: 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 analyzed and verified by external software (shown in dashed boxes).

Railway infrastructure topology, signalling objects, and interlocking specifications should be automatically transferred to a code generation and verification tool to help automate interlocking implementation. The transfer of data from the CAD design model to interlocking code generation tools is possible by using standardized formats such as railML, which in the future will also include an interlocking specification schema [5].

- **Capacity analysis and timetabling** can be performed using e.g. OpenTrack[5], LUKS[6], or Treno[7].
  OpenTrack is a simulation tool which allows stochastic capacity analysis, running time analysis, and other types of analyses. By transferring data directly from a CAD model, such analyses can be performed at an early stage in the design process, greatly increasing the possibility for design decisions to be affected by capacity analysis. This allows a more agile and dynamic design process, so that the end goals of the railway administration can be met, and costs of re-designing and re-building can be minimized.
- **Building information modeling (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.

[5] OpenTrack: simulation of railway networks, `http://www.opentrack.ch/`

[6] LUKS: analysis of lines and junctions, `http://www.via-con.de/en/development/luks`

[7] treno: timetable reliability & network operations analyser, University of Trieste.

```
%| 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.
```

Fig. 8: Structured comments on rule violation expression

The object type definitions described in Section 3 above may be used to associate 3D models to symbols in the 2D geographical layout. Semantic information can then be preserved when transferring information between 2D and 3D representations. 3D tools for design and presentation are now becoming widely used on new railway projects.[8]

6 Tool Implementation

In this section we describe the main aspects of our tool, which implements the verification and the integration into the CAD program, as described in Figure 7.

The XSB Prolog interpreter [47] was used as a back-end for the implementation as it offers tabled predicates which have the same characteristics as Datalog programs, while still allowing general Prolog expressions such as arithmetic operations.

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.

6.1 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 8 for an example of how we do this). 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. We implemented a prototype CAD add-on program for Autodesk AutoCAD (see Figure 9 for a screen-shot).

[8] http://www.jernbaneverket.no/Prosjekter/Inter-City-/3d/

	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.

6.2 Case Study Results

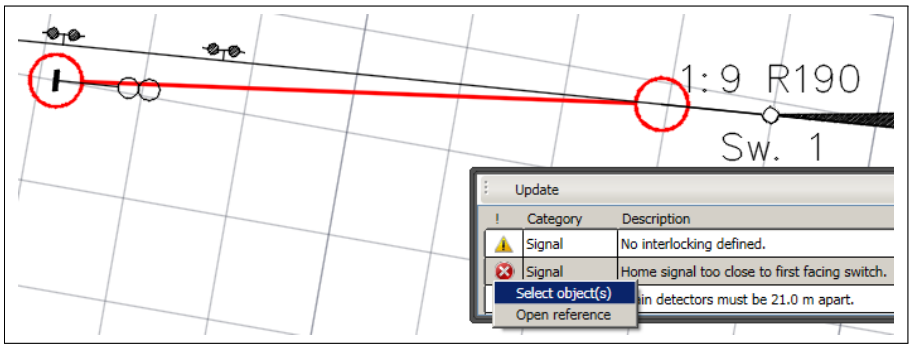
The rules concerning signalling layout and interlocking from *Jernbaneverket*[9] described above have been checked against the model (i.e., railML representation) of the Arna-Fløen project, which is an ongoing design project in Anacon AS (now merged with Norconsult AS). Each object was associated with one or more construction phases, which we call phase *A* and phase *B*, which also corresponds to two operational phases. The 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.

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 5, has been in use for more than 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 automated 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.

The tight integration into the CAD program and, as such, into the engineer's design process, creates the demand for fast re-evaluation of all conclusions upon small changes to the railway designs.

Usually, engineers start with an empty or draft design and add/change one object at a time. The performance figures presented in Table 1 show that the current implementation is

[9] The Norwegian Railway Authorities (`http://www.jbv.no`).



!	Category	Description
⚠	Signal	No interlocking defined.
⊗	Signal	Home signal too close to first facing switch.
		in detectors must be 21.0 m apart.

Fig. 9: Counterexample presentation within an interactive CAD environment.

well acceptable for “one-shot” validation even for realistic designs with running times in the range of seconds. However, it is not fast enough to *smoothly* and transparently be integrated such that it can automatically rerun the complete verification for each small change.

An alternative approach that promises to be more efficient is *incremental verification*: instead of solving logic programs from scratch for each verification run, it tries to materialize all consequences of the base facts and then maintains this view under fact updates. Incremental verification is further discussed in Section 7 below.

7 Incremental Verification

While the static infrastructure verification process as developed so far in this text certainly can improve on the current practice of railway signalling design as it is, the full potential of a “light-weight” verification is still unused because of the perceived separation of design activity and verification activity. A verification tool which runs invisibly alongside the design, giving feedback on the current state of the design at any time could have a higher impact on the design process.

The common use case for running the railway design CAD tool in general is that one performs a series of small changes. Indeed, we have found in the collaborations with railway engineers that large portions of the design phase have the goal of *efficiently handling changes* in track layouts, component capabilities, performance requirements, etc. The verification could, instead of being called whenever final version printouts are being made, instantly report potential problems in the design as soon as this information is available.

This requires lowering the running time of the verification, hopefully to less than one second, while keeping in mind that our prototype verification tool should eventually be able to scale up to much larger stations, projects spanning several stations, and significantly larger knowledge bases. Exploiting the fact that the design work is incremental, also evaluating the Datalog programs incrementally seems to be a promising solution to this challenge.

In this section we give an overview of approaches and algorithms for incremental Datalog and the tools that are available. We study these from the viewpoint of our application domain and evaluate initial performance on our case study.

7.1 Incremental evaluation of Datalog

Datalog systems use rules to derive a set of consequences (*intensional* facts), from a given set of base facts (*extensional* facts). Typically, Datalog systems use a *bottom-up* (or *forward-chaining*) evaluation strategy, where all possible consequences are materialized [48, Chap. 3] [1, Chap. 13] . This simplifies query answering to simply looking up values in the materialization tables. Any change to the base facts, however, will invalidate the materialization. Several approaches have been suggested to reduce the work required to find a new materialization after changing the base facts.

First, if considering only addition of facts to positive Datalog programs, i.e. without negation, then the standard *semi-naive* algorithm [48, Chap. 3] [1, Chap. 13] is already an efficient approach, as it correctly handles additions to the materialization in an incremental manner. The real challenge is the non-monotonic changes, i.e., when removing facts appearing positively in rules or adding facts appearing negatively in rules. Non-monotonicity is essential in our railway infrastructure verification rules. Graph reachability is prominent in many of the regulations for railway signalling, so efficiently maintaining rules involving transitivity is also essential.

Some algorithms, such as truth maintenance systems [8], work by storing more information (in addition to the logical consequences) about the *supporting facts* for derived facts, so that removal of supporting facts may or may not remove a derived fact, depending on whether the support is still sufficient. This allows efficient removal of facts, at the cost of requiring more time and memory for normal derivations. Inspired by the truth maintenance systems of Doyle [8], the XSB Prolog system implements *incremental tabling* [46] by keeping such sets of supporting facts in memory. Figure 10. shows deduced facts for a graph reachability query. In this case, whenever there are several paths connecting a pair vertices of the graph, the `reach` fact for the two vertices is deduced in several ways. In the approach taken in XSB Prolog, different sets of facts that independently prove a derived fact are stored in tables. Whenever changes are made to base facts, the sets of supporting facts can be removed, and as long as the set is not emptied, the derived fact still holds.

```
edge(0,1)
edge(0,2)
edge(1,1)
edge(1,2)
```
(a)

Answer	Supports
reach(0,1)	⟨1,{edge(0,1)}⟩, ⟨2,{reach(0,1),edge(1,1)}⟩
reach(0,2)	⟨1,{edge(0,2)}⟩, ⟨2,{reach(0,1),edge(1,2)}⟩

(b)

Fig. 10: Edge relation and corresponding support sets for a reachability predicate (example from [42]).

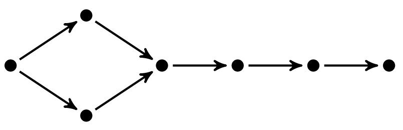
Another class of algorithms, working *without* additional “bookkeeping”, can be more efficient if the re-evaluation of sets of facts is relatively easy compared to re-materializing all facts. The Propagation-Filtering algorithm [17] works on each removed fact separately, propagating it through to all rules which depend on it, while also after each step of the propagation performing a query for alternative support which would end the propagation. In contrast, the Delete-Rederive (DRed) algorithm [16] is rule-oriented and works on sets of facts, first over-approximating all possible deletions that may result from a change in base facts, then re-deriving any still-supported facts from the over-deleted state before finally continuing semi-naive materialization on newly added facts.

An example where the DRed algorithm is less efficient is graph reachability, which can be encoded on the following form:

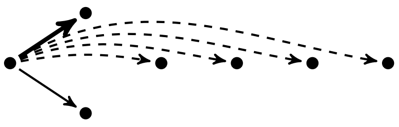
$$\begin{aligned}\text{path}(x,y) &\leftarrow \text{edge}(x,y),\\ \text{path}(x,y) &\leftarrow \text{edge}(x,z) \wedge \text{path}(z,y).\end{aligned}$$

Figure 11 shows key differences in update approaches for the example of a graph reachability from a given node.

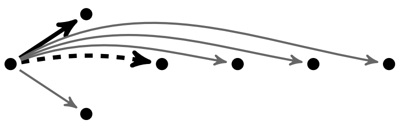
Recently, the Forward/Backward/Forward (FBF) algorithm [32] used in RDFox improved the DRed algorithm in most cases by searching for alternative support (and caching the results) for each potentially deleted fact before proceeding to the next fact. Notably, this method performs better on rules involving *transitivity,* as deletions do not propagate further than necessary.



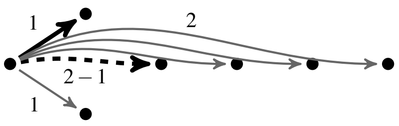
(a) Edge relation visualized as arrows between objects (each element is an arrow $e(a,b)$).



(b) DRed algorithm: removing one edge (thick line) triggers re-evaluation of many dependent edges (dashed lines)



(c) FBF algorithm: removing one edge (thick line) causes re-evaluation of dependent edge (thick dashed line), but confirmation that this edge is still valid stops further propagation.



(d) Counting approach: removing one edge (thick line) causes re-evaluation of dependent edge (thick dashed line), but because this edge has multiple derivations, it is still valid, and propagation can stop. Note that a pure counting approach is not sufficient in this case because of the recursive reachability rule.

Fig. 11: Different approaches to incremental evaluation demonstrated on a reachability program using an edge relation. Using the edge relation in (a), the reachability from the first vertex is calculated, and update strategies for (b) DRed, (c) FBF, and (d) a counting approach are exemplified.

This method is used in the Semantic Web tool *RDFox*[10], which has a high performance on multicore processors with in-memory databases. We are considering RDFox as an alternative candidates for the back end of our incremental railway infrastructure verification procedure.

7.2 Tools and performance

This section summarizes a survey of tools first presented in [27], and describes tools that feature incremental evaluation and Datalog, and which have the maturity required for a future in industrial applications. The logic programs for our verification make use of recursive predicates, stratified negation, and arithmetic. Therefore, we pay particular attention to tools that at least satisfy these needs. In addition, we are looking for high performance on relatively small (in-memory) data sets, so light-weight library-style logic engines are preferred. High-performance distributed "big data" type of tools have less value in this context.

XSB Prolog, continuously developed since 1990, has constantly been pushing the state of the art in high-performance Prolog. XSB is especially known for its tabling support [47], which allows fast Datalog-like evaluation of logic programs without restricting ISO Prolog in any way. The tabling support was extended to allow incremental evaluation [42], and these features have been under continued development and seem to have reached a mature state [46]. For some applications, however, the additional memory usage for incremental tabling can lead to a significant increase in the total memory needed.

[10] RDFox: scalable in-memory RDF triple store with share memory parallel Datalog reasoning, `http://www.cs.ox.ac.uk/isg/tools/RDFox/`

RDFox is a multicore-scalable in-memory RDF triple store with Datalog reasoning. It reads semantic web formats (RDF/OWL) and stores RDF triples, but also includes a Datalog-like input language which can describe SWRL rules. This rule language has been extended to include stratified negation and arithmetic. The RDFox system also implements the new FBF algorithm for incremental evaluation [32].
RDFox stores internally only triples as in RDF (subject, predicate, and object), which, in Datalog, corresponds to only using unary and binary predicates. A method of reifying the rules for higher-arity Datalog predicates into binary predicates allows RDFox to calculate any-arity Datalog programs. However, this requires separate rules for each component (argument) of the predicate, and when doing incremental evaluation, the FBF algorithm's backward chaining step then examines all combinations of components (arguments) potentially involved in such a higher-arity predicate. Because of this problem, using RDFox incrementally did not improve running times in our case study, suggesting a need for native support for n-ary predicates in RDFox.

LogicBlox is a programming platform [2] for combining transactions with analytics in enterprise application areas including web-based retail planning and insurance. It uses a typed, Datalog-based custom language LogiQL and has a comprehensive development framework. It claims support for incremental verification, but we could not evaluate it on our railway example due to absence of freely downloadable distributions.

Dyna is a promising new Datalog-like language for modern statistical AI systems [10]. It has currently not matured sufficiently for our application, but its techniques are promising, and we hope to see it more fully developed in the future.

Many other Datalog tools are available (around 30), few of them supporting incremental evaluation. An overview and our brief evaluation of them can be found in the technical report [29], and a more general overview of Datalog tools can be found in the Wikipedia page. [11]

7.3 Performance

Table 2 compares the running time and memory usage for the verification case study of Arna station presented in Section 6, extended to use the incremental capabilities of XSB Prolog. The extra bookkeeping required in XSB to prepare for incremental evaluation requires more time and memory than non-incremental evaluation, so we include both non-incremental and from-scratch incremental evaluation in the table for comparison. We show how updates can be calculated faster than from-scratch evaluation by moving a single object (an axle counter) in and out of a disallowed area near another object (regulations require at least 21.0 m separation between train detectors). Without using abstraction methods, the case study verification uses over 2 GB of memory. So, for any hope of handling larger stations on a standard laptop or workstation, this must be reduced. We were not able to reduce memory usage in this case study using the abstraction methods in XSB (version 3.6.0).

While currently none of the tools seem to satisfy all conditions we hoped for in our integration, notably efficiency, but also maturity and stability, it should also be noted that the need for incremental evaluation has been identified by the community not only as theoretically interesting, but also as of practical importance. The RDFox developers aim to support incremental updates of higher-arity predicates in a later version. The XSB project has made efforts to improve its abstraction mechanisms, so future versions might become feasible for our use. If reducing the memory usage would require adapting a Datalog algorithm (such as

[11] `https://en.wikipedia.org/wiki/Datalog#Systems_implementing_Datalog`

			Testing station	Arna phase *A*	Arna phase *B*
Relevant components			15	152	231
Interlocking routes			2	23	42
Datalog input facts			85	8283	9159
XSB:					
Non-incremental verif.:	Running time:	(s)	0.015	2.31	4.59
	Memory	(MB)	20	104	190
Incremental verif. baseline:	Running time	(s)	0.016	5.87	12.25
	Memory	(MB)	21	1110	2195
Incr. single object update:	Running time	(s)	0.014	0.54	**0.61**
	Memory	(MB)	22	1165	2267

Table 2: Case study size and running times on a standard laptop.

DRed), then XSB's unrestricted Prolog might be a challenge. A different approach would be to extend another efficient Datalog tool, such as Soufflé[12] to do incremental evaluation, which could require a significant effort.

8 Conclusions, Related and Further Work

We have demonstrated a logical formalism in which railway layout and interlocking constraints can be modelled and technical regulations can be expressed, and which can be decided by logic programming methods (Datalog in particular) with polynomial time complexity. This allows verification of railway signalling designs against infrastructure manager 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. We have demonstrated this approach on an ongoing railway design project from the Anacon AS company and using the standard regulations from the Norwegian railway authorities. We have implemented a prototype and integrated it in the engineer's CAD design tool suite. Even though preliminary tests show good performance, we saw the need for faster verification methods, and thus looked into incremental verification tools for Datalog. In this respect we presented our summary of findings and our test results on our railway use case.

This paper is an extension and combination of three previous conference papers, i.e., we extended our initial results from [28] with more explanations and background material; we combined and explained the logical work in more context, some of which was presented in [26] to the practitioners from the railway domain; we explained the need for incremental verification and provided our findings and conclusions, part of which were presented in [27]. This paper, thus, presents our results in a more uniform and integrated manner, giving a better picture of the overall tool chain and putting the problem well in context. Our future work is detailed in the following.

8.1 Related work.

Railway control systems and signalling designs are a fertile ground for formal methods. See [3,11] for an overview of 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

[12] Soufflé: a Datalog compiler, `http://souffle-lang.org/`

survey, see [19]. In particular, safety of interlockings has been intensively formalized and studied, using for instance VDM [14] and the B-method, resp. Event-B [23]. 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. [6,51,9] [38,30,15,9]. Critically evaluating practicality, [12] investigated applicability of model checking for interlocking tables using NuSMV resp. Spin, two prominent representatives of BDD-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, [18] uses *bounded* model checking [7] for interlockings. Instead of attempting an exhaustive 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 to use SAT solving techniques in the analysis. The paper uses a variant of linear temporal logic (LTL) for safety property specification and employs so-call k-induction. The work of [50] investigates how 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 [4]. Prover is an automated theorem prover, using Stålmarck's method [45] 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 [20] uses a Prolog database to represent the topology and the layout, where for the the verification, the work uses a separate SAT solver. Similarly, the work of [33,34] uses 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 have seen renewed research interest in Datalog, see for instance the collection [31]. Datalog has in particular been used for formalizing and efficiently implementing program analyses [43,49], whereas [44] presents 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.

Lodemann et al. [25] use semantic technologies to automate railway infrastructure verification. Their scope is still wider than this paper in the computational sense, with the full expressive power of OWL ontologies, running times on the order of hours, and the use of separate interactive graphical user interfaces rather than integration with design tools.

8.2 Future work.

In the future work with RailComplete AS, we will focus on extending the rule base to contain more relevant signalling and interlocking regulations, and also on evaluating the performance of our verification on a larger scale. Design information and rules about other railway control systems, such as geographical interlockings and train protection systems could also be included. The current work is assuming Norwegian regulations, but the European Rail Traffic Management System is expected to dominate in the future.

Involving railway engineers in knowledge base development is somewhat hindered by the fact that Datalog and logic programming, though declarative and concise, are still programming languages, and a good intuition about language semantics is required for efficient and correct development. We have started developing a higher-level domain-specific language which compiles to Datalog, for the purpose of including railway engineers more closely in the knowledge base development. Using the techniques of controlled natural languages (CNL) [22], the language is formally defined and parsable, yet readable as natural language. Writing text in a controlled natural language is a harder problem, as the formal grammar limits what can be expressed, and also how it is written. This problem can be mitigated by means of specialized text editors (e.g., as provided by the Grammatical Framework [39],[40, Chap. 7] or Attempto Controlled English [13,21]) which guide the user towards writing acceptable sentences. Adding specific language constructs for static infrastructure analysis of railway designs ensures that the level of abstraction is close to the original regulation text. Our future work will include testing this language with railway engineers and integrating knowledge base development into the railway design tool chain.

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.

Acknowledgements We are grateful to Claus Feyling from RailComplete AS and Martin Steffen from University of Oslo for valuable inputs.

References

1. Abiteboul, S., Hull, R., Vianu, V. (eds.): Foundations of Databases, 1st edn. Addison-Wesley Longman Publishing Co., Inc. (1995)
2. Aref, M., ten Cate, B., Green, T.J., Kimelfeld, B., Olteanu, D., Pasalic, E., Veldhuizen, T.L., Washburn, G.: Design and implementation of the LogicBlox system. In: T.K. Sellis, S.B. Davidson, Z.G. Ives (eds.) The 2015 ACM SIGMOD International Conference on Management of Data, pp. 1371–1382. ACM (2015). DOI 10.1145/2723372.2742796
3. Bjørner, D.: New results and trends in formal techniques for the development of software in transportation systems. In: G. Tarnai, E. Schnieder (eds.) Proceedings of the Symposium on Formal Methods for Railway Operation and Control Systems (FORMS'03), pp. 1–20. L'Harmattan Hongrie (2003)
4. Borälv, A., Stålmarck, G.: Prover technology in railways. In: M.G. Hinchey, J.P. Bowen (eds.) Industrial-Strength Formal Methods, International Series in Formal Methods, pp. 329–305. Springer-Verlag (1999)
5. Bosschaart, M., Quaglietta, E., Janssen, B., Goverde, R.M.P.: Efficient formalization of railway interlocking data in RailML. Information Systems **49**, 126–141 (2015). DOI http://dx.doi.org/10.1016/j.is.2014.11.007
6. Busard, S., Cappart, Q., Limbrée, C., Pecheur, C., Schaus, P.: Verification of railway interlocking systems. In: J. Pang, Y. Liu, S. Mauw (eds.) Proceedings 4th International Workshop on Engineering Safety and Security Systems, ESSS, *Electronic Proceedings in Theoretical Computer Science*, vol. 184, pp. 19–31. Open Publishing Association (2015). DOI 10.4204/EPTCS.184.2
7. Clarke, E.M., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Formal Methods in System Design **19**, 7–34 (2001)
8. Doyle, J.: A truth maintenance system. Artificial Intelligence **12**(3), 231–272 (1979). DOI 10.1016/0004-3702(79)90008-0
9. Eisner, C.: Using symbolic model checking to verify the railway stations of Hoorn-Kersenboogerd and Heerhuowaard. In: L. Pierre, T. Kropf (eds.) Correct Hardware Design and Verification Methods, 10th IFIP WG 10.5 Advanced Research Working Conference, CHARME '99, *Lecture Notes in Computer Science*, vol. 1703, pp. 97–109. Springer-Verlag (1999)
10. Eisner, J., Filardo, N.W.: Dyna: Extending Datalog for modern AI. In: de Moor et al. [31], pp. 181–220

11. Fantechi, A., Fokkink, W., Morzenti, A.: Some trends in formal methods applications to railway signalling. In: S. Gnesi, T. Margaria (eds.) Formal Methods for Industrial Critical Systems, pp. 61–84. John Wiley & Sons Inc. (2012)
12. Ferrari, A., Magnani, G., Grasso, D., Fantechi, A.: Model checking interlocking control tables. In: E. Schnieder, G. Tarnai (eds.) 8th Symposium on Formal Methods for Automation and Safety in Railway and Automotive Systems (FORMS/FORMAT 2010), pp. 107–115. Springer-Verlag (2011). DOI 10.1007/978-3-642-14261-1
13. Fuchs, N.E., Kaljurand, K., Kuhn, T.: Attempto controlled english for knowledge representation. In: C. Baroglio, P. Bonatti, J. Maluszynski, M. Marchiori, A. Polleres, S. Schaffert (eds.) Reasoning Web, *Lecture Notes in Computer Science*, vol. 5224, pp. 104–124. Springer (2008). DOI 10.1007/978-3-540-85658-0_3
14. Fukuda, M., Hirao, Y., Ogino, T.: VDM specification of an interlocking system and a simulator for its validation. In: U. Becker, E.L. Schneider (eds.) 9th IFAC Symposium Control in Transportation Systems, pp. 218–223. IFAC (2000)
15. Gnesi, S., Lenzini, G., Latella, D., Abbaneo, C., Amendola, A., Marmo, P.: Automatic Spin validation of a safety critical railway control system. In: IEEE Conference on Dependable Systems and Networks, pp. 119–124. IEEE Computer Society Press (2000)
16. Gupta, A., Mumick, I.S., Subrahmanian, V.S.: Maintaining views incrementally. In: P. Buneman, S. Jajodia (eds.) SIGMOD International Conference on Management of Data, SIGMOD '93, pp. 157–166. ACM (1993). DOI 10.1145/170035.170066
17. Harrison, J.V., Dietrich, S.W.: Maintenance of materialized views in a deductive database: An update propagation approach. In: K. Ramamohanarao, J. Harland, G. Dong (eds.) Proceedings of the Workshop on Deductive Databases held in conjunction with the Joint International Conference and Symposium on Logic Programming, *Technical Report*, vol. CITRI/TR-92-65, pp. 56–65. Department of Computer Science, University of Melbourne (1992)
18. Haxthausen, A.E., Peleska, J., Pinger, R.: Applied bounded model checking for interlocking system designs. In: S. Counsell, M. Núñez (eds.) Software Engineering and Formal Methods Collocated Workshops, *Lecture Notes in Computer Science*, vol. 8368, pp. 205–220. Springer-Verlag (2014)
19. Hinchey, M.G., Bowen, J.P. (eds.): Industrial-Strength Formal Methods. International Series in Formal Methods. Springer-Verlag (1999)
20. Kanso, K., Moller, F., Setzer, A.: Automated verification of signalling principles in railway interlocking systems. In: A. Miller, M. Calder (eds.) Proceedings of the Eighth International Workshop on Automated Verification of Critical Systems (AVoCS 2008), *Electronic Notes in Theoretical Computer Science*, vol. 250, pp. 19–31. Elsevier (2009). DOI http://dx.doi.org/10.1016/j.entcs.2009.08.015
21. Kuhn, T.: A principled approach to grammars for controlled natural languages and predictive editors. Journal of Logic, Language and Information **22**(1), 33–70 (2013). DOI 10.1007/s10849-012-9167-z
22. Kuhn, T.: A survey and classification of controlled natural languages. Computational Linguistics **40**(1), 121–170 (2014). DOI 10.1162/COLI_a_00168
23. Lecomte, T., Burdy, L., Leuschel, M.: Formally checking large data sets in the railways. In: F. Ishikawa, A.. Romanovsky (eds.) Advances in Developing Dependable Systems in Event-B. In conjunction with ICFEM 2012, Technical Report. Newcastle University (2012)
24. Libkin, L.: Elements of Finite Model Theory. Texts in Theoretical Computer Science. An EATCS Series. Springer-Verlag (2004). DOI 10.1007/978-3-662-07003-1. URL `http://dx.doi.org/10.1007/978-3-662-07003-1`
25. Lodemann, M., Luttenberger, N., Schulz, E.: Semantic computing for railway infrastructure verification. In: 7th International Conference on Semantic Computing (ICSC), pp. 371–376. IEEE (2013). DOI 10.1109/ICSC.2013.69
26. Luteberget, B., Feyling, C.: Automated verification of rules and regulations compliance in CAD models of railway signalling and interlocking. In: C. Brebbia, J. Mera, N. Tomii, P. Tzieropoulos (eds.) Computers in Railways XV, pp. 153–165. WIT Press (2016)
27. Luteberget, B., Johansen, C., Feyling, C., Steffen, M.: Rule-based incremental verification tools applied to railway designs and regulations. In: J. Fitzgerald, C. Heitmeyer, S. Gnesi, A. Philippou (eds.) 21st International Symposium on Formal Methods (FM), *Lecture Notes in Computer Science*, vol. 9995, pp. 772–778. Springer-Verlag (2016). DOI 10.1007/978-3-319-48989-6_49
28. Luteberget, B., Johansen, C., Steffen, M.: Rule-based consistency checking of railway infrastructure designs. In: E. Ábrahám, M. Huisman (eds.) 12th International Conference on integrated Formal Methods (iFM 2016), *Lecture Notes in Computer Science*, vol. 9681, pp. 491–507. Springer (2016). DOI 10.1007/978-3-319-33693-0_31
29. Luteberget, B., Johansen, C., Steffen, M.: Rule-based consistency checking of railway infrastructure designs (long version). Technical report 450, University of Oslo, Dept. of Informatics (2016). URL `http://www.ifi.uio.no/~msteffen/download/16/rulebasedconsistency-rep.pdf`

30. Mirabadi, A., Yazdi, M.B.: Automatic generation and verification of railway interlocking tables using FSM and NuSMV. Transport Problems: An International Scientific Journal **4**, 103–110 (2009)
31. de Moor, O., Gottlob, G., Furche, T., Sellers, A. (eds.): Datalog Reloaded. First International Workshop 2010, *Lecture Notes in Computer Science*, vol. 6702. Springer-Verlag (2011)
32. Motik, B., Nenov, Y., Piro, R.E.F., Horrocks, I.: Incremental update of datalog materialisation: the backward/forward algorithm. In: B. Bonet, S. Koenig (eds.) Twenty-Ninth AAAI Conference on Artificial Intelligence, pp. 1560–1568. AAAI Press (2015). URL `http://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/9660`
33. Nakamatsu, K., Kiuchi, Y., Chen, W., Chung, S.: Intelligent railway interlocking safety verification based on annotated logic program and its simulator. In: IEEE International Conference on Networking, Sensing and Control, vol. 1, pp. 694–699. IEEE (2004). DOI 10.1109/ICNSC.2004.1297524
34. Nakamatsu, K., Kiuchi, Y., Suzuki, A.: EVALPSN based railway interlocking simulator. In: M.G. Negoita, R.J. Howlett, L.C. Jain (eds.) Knowledge-Based Intelligent Information and Engineering Systems, *Lecture Notes in Artificial Intelligence*, vol. 3214, pp. 961–967. Springer-Verlag (2004)
35. Nash, A., Huerlimann, D., Schütte, J., Krauss, V.P.: RailML — a standard data interface for railroad applications. In: J. Allan, C. Brebbia, R. Hill, G. Sciutto, S. Sone (eds.) Computers in Railways IX, pp. 233–240. WIT Press (2004)
36. Nilsson, U., Maluszynski, J.: Logic, Programming, and Prolog, 2nd edn. John Wiley & Sons, Inc. (1995)
37. Pachl, J.: Railway Operation and Control. VTD Rail Publishing (2015)
38. Pavlovic, O., Ehrich, H.: Model checking PLC software written in function block diagram. In: Third International Conference on Software Testing, Verification and Validation, ICST, pp. 439–448. IEEE (2010)
39. Ranta, A.: Grammatical framework. Journal of Functional Programming **14**(2), 145–189 (2004). DOI 10.1017/S0956796803004738
40. Ranta, A.: Grammatical Framework: Programming with Multilingual Grammars. CSLI Publications (2011)
41. Russell, S.J., Norvig, P.: Artificial Intelligence: A Modern Approach, 2 edn. Pearson Education (2003)
42. Saha, D., Ramakrishnan, C.R.: Incremental evaluation of tabled logic programs. In: C. Palamidessi (ed.) Logic Programming, 19th International Conference, ICLP, *Lecture Notes in Computer Science*, vol. 2916, pp. 392–406. Springer (2003). DOI 10.1007/978-3-540-24599-5_27
43. Smaragdakis, Y., Bravenboer, M.: Using Datalog for fast and easy program analysis. In: de Moor et al. [31], pp. 245–251
44. Smaragdakis, Y., Bravenboer, M., Lhoták, O.: Pick your contexts well: Understanding context-sensitivity (the making of a precise and scalable pointer analysis). In: T. Ball, M. Sagiv (eds.) 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, (POPL), pp. 17–30. ACM (2011)
45. Stalmårck, G.: 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)
46. Swift, T.: Incremental tabling in support of knowledge representation and reasoning. Theory and Practice of Logic Programming **14**(4-5), 553–567 (2014). DOI 10.1017/S1471068414000209
47. Swift, T., Warren, D.S.: XSB: Extending Prolog with tabled logic programming. Theory and Practice of Logic Programming **12**(1-2), 157–187 (2012). DOI 10.1017/S1471068411000500
48. Ullman, J.D.: Principles of Database and Knowledge-Base Systems (Volume I & II). Computer Society Press (1988)
49. Whaley, J., Avots, D., Carbin, M., Lam, M.S.: Using Datalog with binary decision diagrams for program analysis. In: K. Yi (ed.) Third Asian Symposium on Programming Languages and Systems (APLAS), *Lecture Notes in Computer Science*, vol. 3780, pp. 97–108. Springer-Verlag (2005)
50. Winter, K.: Optimising ordering strategies for symbolic model checking interlocking control tables. In: T. Margaria, B. Steffen (eds.) 5th International Symposium on Leveraging Applications of Formal Methods, Verification, and Validation (ISOLA'12), Part II, *Lecture Notes in Computer Science*, vol. 7610, pp. 246–260. Springer-Verlag (2012)
51. Winter, K., Johnston, W., Robinson, P., Strooper, P., van den Berg, L.: Tool support for checking railway interlocking designs. In: T. Cant (ed.) Proceedings of the 10th Australian Workshop on Safety Critical Systems and Software, pp. 101–107. ACM (2006)