Acknowledgments

This research has been partly supported by the Norwegian Research Council project RailCons – Automated Methods and Tools for Ensuring Consistency of Railway Designs, and by the Swedish Research Council under grant nr. 2012-5746 – Reliable Multilingual Digital Communication: Methods and Applications.

Address for Bjørnar Lateberget:
RailComplete AS, Sandvika, Norway
E-mail: bjlut@railcomplete.no

Address for Christian Johansen (né Cristian Prisacaria):
Department of Informatics, University of Oslo, P.O. Box 1080 Blindern, 0316 Oslo, Norway.
E-mail: cristi@ifi.uio.no

Address for John J. Camilleri and Gerardo Schneider:
Department of Computer Science and Engineering, Chalmers University of Technology and University of Gothenburg, Sweden.
E-mail: john.j.camilleri|gerardo@cse.gu.se
Abstract

Previously, we have worked on verification of railway regulations against CAD designs, and integrated a verification engine into the tool chain of railway engineers. This tool was used successfully in a pilot project at RailCOMPLETE AS, (formerly Anacon AS). However, the engineers were reluctant to accept the verification results because they did not have control over the properties being verified. To allow engineers with limited logic programming experience to participate in defining and maintaining the verification properties, we design a controlled natural language (CNL) which contains constructs corresponding closely to the regulation texts. The CNL is translated automatically into the Datalog input language of the verification engine. We demonstrate a prototype system which, upon detecting regulation violations, traces back from errors in the design through the CNL to the marked-up original text, thus allowing domain experts to examine the correctness of each translation step. We describe a methodology based on CNL best practices and previous experience with creating verification front-end languages. By designing the CNL’s structure specifically to support our use case, the language stays natural and readable for non-programmers, allowing railway engineers to better understand verification properties and to participate in improving the system.
Contents

1 Introduction 1

2 Approach to Participatory Verification for Railway Regulations 2

3 Controlled Natural Language 4
   3.1 Grammatical Framework ........................................... 4

4 Design Methodology for a Verification Front-End Language 6
   4.1 Abstract Syntax ...................................................... 8
      4.1.1 Middle Ground: Modelling the Text vs. Modelling the Logic 8
      4.1.2 Vocabulary: Static vs. Dynamic .............................. 9
   4.2 Logic-Driven Design in RailCNL .................................. 9
      4.2.1 Top-Level Statement Types .................................... 9
      4.2.2 Generic Ontology Module .................................... 10
      4.2.3 Layout Module ................................................... 10
      4.2.4 Area Module ..................................................... 11
   4.3 Text-Driven Design ................................................ 11
      4.3.1 Another Example: Train Running Times ..................... 12
   4.4 Concrete Syntax .................................................... 12
      4.4.1 Keeping Ambiguity under Control ......................... 13
   4.5 Translating CNL into the Target Logic Formalism ............ 14
      4.5.1 Predicate Representation .................................... 14
      4.5.2 Explicit Variables ............................................. 14
      4.5.3 Ontology Assertions .......................................... 15
      4.5.4 Ontology Restrictions ........................................ 15
      4.5.5 Disjunctive Normal Form ..................................... 15
   4.6 Simplifications and optimizations ............................... 16
      4.6.1 Layout Properties ............................................. 16
      4.6.2 Inlining .......................................................... 17
   4.7 Bi-directional Translation ....................................... 17
   4.8 Matching Vocabulary .............................................. 17
   4.9 Maintenance of the Language ................................... 17

5 Tool Integration 18
   5.1 CAD Integration ..................................................... 18
   5.2 Requirements Tracing .............................................. 18
   5.3 Editors: Reading vs. Writing in a CNL ......................... 19

6 Conclusions 19
   6.1 Related Work ....................................................... 20
      6.1.1 Railway Verification ........................................... 20
      6.1.2 Domain Specific Languages for Railway Verification .... 21
      6.1.3 Controlled Natural Languages ............................... 21
      6.1.4 Controlled Natural Language Editors ..................... 21
   6.2 Future Work ......................................................... 21

A Appendix: Overview of Norwegian regulation contents 24

B Appendix: Example content from regulations 26
1 Introduction

Automated formal verification techniques have the potential to greatly increase the efficiency of engineering. However, verification engines are not easy to take up in industrial practice. Even if the verification process is fully automated, integrating the tools into the users’ workflow and formalizing properties and models requires careful thinking and domain expertise. The gap between automated verification and domain expert users is often caused by the lack of user involvement. The users are usually not experts in verification techniques, i.e., they do not know how to write properties into the verifier’s language, nor how to build models for the verifier, nor how to interpret the output of the verifier when violated properties are found. In our case, the users are expert engineers from the railway domain, designing railway infrastructure.

We want to allow the end users to participate in the verification process. This is for various purposes. First, the domain experts need to understand the verification properties that the tool actually verifies, as well as the model of the system that the tool works with. This would allow them to trust the verification. Secondly, we want to allow the users to actively participate in maintaining the verification properties, i.e., to change and adjust them when needed. This would bring the flexibility that the users often ask from the verification. Thirdly, we want that the domain experts are able to create their own specifications and feed these into the verification engine, e.g., build specific expert knowledge into verification conditions. This would allow the experts to take full advantage of the verification technique.

The concept of involving the user is well rooted and studied in the field of participatory design [31, 20]. In consequence, we like to use the term participatory verification when talking about methods for including the end user into the verification process, having the goal of making automated verification techniques accessible to and adopted by engineers with little programming experience.

In our case we develop a verification technique for the domain of railway engineering, and especially the signalling discipline, which is characterised by heavy national regulations, long-standing engineering practices, a need for inter-operability, and (national and international) standardization. Due to the high safety requirements, the railway design norms recommend the use of formal methods, and for the higher safety integrity levels (SIL), they “highly recommend” them (cf. e.g. [6]).

During the signalling design, even small changes in station layout and interlocking may require thorough investigation to prove that the designs remain internally consistent and still adhere to the rules and regulations of the national (and international) administrations. Many of these manual checks are simple enough to be automated with little computational effort, so that the verification can be performed inside existing engineering software, on-the-fly while the design is being created.

We have previously demonstrated [24] an efficient verification and troubleshooting tool integrated into the railway CAD programs, which are widely used by railway planning engineers. This tool performs a lightweight type of verification which we call static infrastructure verification, and the results are updated continuously as the engineer is modifying the station (see Figure 1).

However, the Prolog-like formal logical specification language that we used for describing railway rules and regulations is not easy for inexperienced programmers to write. Ideally, railway engineers should be able to read the logical specifications to ensure that they correctly represent the engineering domain. Furthermore, engineers should themselves be able to maintain and extend the rule base with limited support from verification experts. When we presented the railway engineers from RailCOMPLETE AS with our prototype they raised another concern: how could they trace the violation, which the tool displays graphically, back to the source documents?

These observations have led us to develop a controlled natural language, which we call RailCNL, meant to be used as an intermediate representation between natural language texts (i.e., the railway regulations) and Datalog logic programs. RailCNL aims to be human-friendly enough for our domain experts to work with to

1 Authorities typically make small adjustments to regulations several times per year, whereas engineering best practices can be revised at any time.
2 Such expert knowledge is often seen as proprietary valuable assets of the company.
3 http://railcomplete.no
overcome the above challenges, and thus getting them involved in using and improving the automated verification tool (our case is focused on the properties that the tool verifies).

In this paper we present the addition of the following features to our lightweight verification system: (1) RailCNL, a user-friendly verification front-end language for static railway infrastructure analysis (Sec. 4.1), (2) automatic translation of RailCNL into Datalog (Sec. 4.5), and (3) backwards tracing integrated into the CAD program, where marked-up original regulation texts is used together with the CNL text to explain regulatory violations found in the model (Sec. 5.2).

2 Approach to Participatory Verification for Railway Regulations

This section describes the overall approach that we take to participatory verification of static infrastructure railway plans/drawings against regulations. For this we design and integrate with our previously developed verification engine, a CNL for expressing railway regulations and expert knowledge. Figure 2 presents the overall workflow of using the railway CNL integrated with a CAD environment and our verification. Railway engineers already use a CAD program for most of their design work (typically Autodesk AutoCAD is used in Norwegian railways), and seamless integration with these existing tools can increase adoption and use.

To perform static infrastructure verification, we need the following input:

1. **Model:** railway infrastructure plans, typically created by arranging the station geographical layout using CAD programs. Also, interlocking specifications are defined, typically in a tabular format. See Figure 3 for an example of a CAD drawing showing a railway station.

2. **Properties:** regulations and expert knowledge, extracted from regulatory and best-practices documents.

The formalization of these into Datalog is described in our previous work [24] which allows efficient automatic reasoning. The reasoning happens continuously, in the background, while the user is working on the CAD drawing. Violations are presented to the user in the CAD program which allows quick tracing back to the source of the error.

In [24] we described verification properties using logical rules in Datalog, which is a common practice in verification (maybe using other logics like temporal [25, 32, 2] or dynamic logics [16, 3]). We were expecting that the declarative style of Datalog would be easily accepted by railway engineers for reading and writing such properties themselves. However, when we trialled it with the RailCOMPLETE engineers they were not proficient enough in logic programming to understand our encodings.
Our approach to allowing the railway engineers to actively get involved in the verification process is concerned with the properties and not with the model. This is because in our case the model is automatically generated from CAD drawings, which already is the tool of choice for engineers, thus they are actively involved in making the models while drawing in the CAD-based RailCOMPLETE framework. For reading and writing properties, however, we develop the controlled natural language RailCNL for representing properties on a higher level of abstraction, closer to the original text, while still retaining the possibility for automatic translation into Datalog. This approach has the following advantages:

- RailCNL is domain-specific, i.e., tailored both to the types of logical statements needed by the verification engine, and to the regulations terminology. This allows concise and readable expressions, increasing naturalness and maintainability.

- The language closely resembles natural language, and can be read by engineers with the required domain knowledge without learning a programming language.

- A separate textual explanation (such as comments used in programming) is not needed for presenting violations textually, as the properties are now directly readable as natural text. Comments could still be used, e.g., to clarify edge cases or to describe the CNL semantics. This type of commenting is even needed in the original regulations texts, as even the expected natural semantics of some regulations needs confirmation in certain cases (“yes, this rule applies even when (...)").

- Statements in RailCNL can be linked to statements in the original text, so that reading them side by side reveals to domain experts whether the CNL paraphrasing of the natural text is valid.

The RailCNL has a modular design (see Figure 4) where domain-specific constructs are separated from generic constructs. However, CNL modules are not always trivially composable, and care must be taken to retain naturalness while avoiding ambiguity when increasing the complexity of the language. We give examples of such trade-offs throughout Section 4.
3 Controlled Natural Language

Controlled natural languages (see [21] for a recent survey) are constructed languages resembling natural language (such as English) with additional restrictions on grammar and vocabulary. The restrictions are typically aimed at reducing the ambiguity and complexity of unrestricted natural language. A controlled natural language may or may not also be a formal language. Non-formal controlled natural languages, such as ASD Simplified Technical English and Basic English aim to guide authors to write clearer texts and make them easier to read for non-native English speakers. Wyner et al. [36] give high-level recommendations on how to design controlled natural languages ranging from informal to formal, general to domain-specific, simple to complex.

To be effective as verification front-end language, a CNL needs to be a fully formal language which can be automatically parsed and translated. This requires making careful trade-offs between preserving naturalness and avoiding ambiguity.

Formal CNLs have been used as front-end languages for formal systems in applications such as knowledge representation [14], contract languages [1], or legal texts [9].

Several of these languages focus on knowledge representation (in the style of OWL), covering universal and existential quantifiers over individuals, properties and relations. An ontology language can be a good starting point for designing a CNL, as many domains require reasoning about objects, classes, properties, relations and values. Instead of adopting all constructs from an existing ontology language, or even adopting an existing ontology CNL, selecting the specific constructs needed in the domain helps avoid ambiguities and retain naturalness.

3.1 Grammatical Framework

Grammatical Framework (GF) is a programming language for multilingual grammar applications [27]. A Grammatical Framework program defines a grammar consisting of an abstract syntax and one or more concrete syntaxes.

An abstract syntax consists of categories and constructors (functions), corresponding to a set of algebraic
data types, which define the abstract syntax tree (AST) of the language. The following is an example of abstract syntax used to form sentences about distance restrictions on railway objects:

```plaintext
abstract Railway = {
cat Object; Length; Restriction; Statement;
fun
    Signal, Switch, Detector : Object;
    LengthMeters : Int -> Length;
    GreaterThan, LessThan : Restriction;
    ObjectSpacing : Object -> Object -> Restriction -> Length -> Statement; }
```

To express that signals should not be closer than 20m from a switch, we write:

```
AST:  ObjectSpacing Signal Switch GreaterThan (LengthMeters 20)
```

The concrete syntax creates a mapping from the tree-structured abstract syntax to text. Applying this mapping is called *linearization*. GF concrete syntaxes are invertible so that the concrete syntax also defines a *parser* for the language. This inversion is complete except for situations with ambiguities in the concrete syntax. Therefore, and especially when designing formal language front-ends, it is essential to limit the possible ambiguities in the language to get an exact correspondence between the concrete language (linearization) and the AST. Concrete syntax definitions in GF assign concrete data types to the abstract categories, e.g. strings or record types, and provide implementations of the constructors as functions.

A concrete syntax for the above AST concerning railway object spacing is:

```plaintext
concrete RailwayEng of Railway = {
    lincat Object = Str; Length = Str;
    Restriction = Str; Statement = Str;
    lin
        Signal = "signal";
        Switch = "switch";
        Detector = "detector";
        LengthMeters i = i.s ++ "m";
        GreaterThan = "greater_than";
        LessThan = "less_than";
        ObjectSpacing o1 o2 r l = "a" ++ o1 ++ "must be" ++ r ++ l ++ "from" ++ o2;
}
```

After both an abstract syntax and a corresponding concrete syntax has been defined, we can parse this language:
A switch must be more than 20 m from a signal

We can also linearize the language from the abstract syntax:

AST: ObjectSpacing Detector Signal LessThan (LengthMeters 2)

Text: a detector must be less than 2 m from a signal

Although this example is close to natural language, extending the language in the same style would quickly run into trouble trying to cover all the linguistic variation that arises from composing complex sentences. For example, adding words to the vocabulary which start with vowel sounds would require the article “a” to be replaced with “an” in these cases, breaking the compositionality of the program.

The Grammatical Framework project also features the resource grammar library (RGL), which is a comprehensive linguistic model of natural languages with a unified API for forming sentences, and implementations of this API for 32 languages. The RGL hides the grammatical details of the natural language, and allows an abstract syntax to be mapped into many natural languages with minimal effort. Since the RGL encapsulates the linguistic complexity of the natural languages, the effort needed to map an AST into another natural language is minimized, often reducing to simply providing the domain-specific vocabulary.

The resource grammar library defines a comprehensive set of linguistic categories such as noun phrases (NP), verb phrases (VP), clauses (Cl) and sentences (S) which can be used to compose texts. The type-safety enforced by the GF compiler on the constructors which use these linguistic categories ensures that the compositions are grammatical. Each language resource in the RGL implements these categories with the required attributes for that particular language. For example, the English grammar contains a determiner phrase $a_Det$, which can be linearized as “a” or “an”, depending on the composition of the noun phrase in which it is used.

The example from above can be re-written to use the English resource grammar as follows:

concrete RailwayEngRGL of Railway = open SyntaxEng, ParadigmsEng, SymbolicEng, (Res=ResEng) in {
  lincat Object = N; Length = NP;
  Restriction = A2; Statement = S;
  lin
    Signal = mkN "signal";
    Switch = mkN "switch";
    Detector = mkN "detector";
    LengthMeters i = symb (i.s ++ "m");
    GreaterThan = mA2 (mA "more") (mPrep "than");
    LessThan = mA2 (mA "less") (mPrep "than");
    ObjectSpacing obj1 obj2 restriction length =
      mkS (mkCl (mkNP a_Det obj1)
        (mkVP (mkVP must_VV (mkVP (mkAP restriction length)))))
        (SyntaxEng.mkAdv from_Prep (mkNP a_Det obj2)));
}

Using the resource grammar library allows us to separate the concerns of composing sentences from the concern of inflections and word ordering.

4 Design Methodology for a Verification Front-End Language

Our methodology is based on CNL and Grammatical Framework best practices. Ranta et al. [30] describe the construction of a controlled natural language with a focus on translating from one natural language to another, showing how to construct a semantic model and map it into natural language using the resource grammar library, and how to avoid or handle ambiguity in parsing and translating. In a later report, Ranta et al. [29] give
explicit best practices gathered from Grammatical Framework projects, such as (1) using a modular structure separating generic and domain-specific parts of the grammar, (2) letting the AST model the semantics of the text, as opposed to the logic of the underlying formalism, and (3) trade-offs in modelling language restrictions purely in context-free grammar versus using dependent types.

We also build on the following works on creating domain-specific CNLs as verification front-ends. Johannisson [19] describes a CNL targeting the Object Constraint Language (OCL) for use in reasoning about Java program correctness in the KeY system. The language features dynamic vocabulary based on input UML diagrams. Vocabulary changes are achieved by re-compiling the grammar using the GF compiler when needed. A conflict detection framework was created by Angelov et al. [1] where Grammatical Framework is used to map the contract language CL onto a CNL. Statement modalities, such as obligation, permission and prohibition, are applied to actions and conjunctions of actions with operations such as and, or, concurrency, sequence, and choice. The structure of the CNL is modelled after the CL language. After translating the CNL into CL syntax, the statements are used in a conflict analysis tool. A CNL approach for manipulating C-O diagrams by Caimilleri et al. [9] uses several interfaces to a common XML interchange format. A visual diagram editor, a CNL with text editor support, and a spreadsheet representation can be each used for different needs. The common format is in turn translated into timed automata for reasoning about system properties. Queries for tax fraud detection by Calafato et al. [8] were modelled with a GF CNL which was translated into database queries and stream filters for letting tax fraud professionals iterate quickly on designing filters for tax investigation.

Each of the above works covers some aspects of constructing a domain-specific CNL, some also geared towards verification applications. We present here the methodology that we apply to design RailCNL, a verification front-end language for describing rules for static railway infrastructure verification. This methodology is detailed in the next sections, and combines concrete advices from the above experiences with our own experience from creating a railway infrastructure verification platform.

With RailCNL, we aim to reasonably cover the following content:

1. Definitions of railway-domain terms from a set of basic terms given by the object types present in the CAD program and the railML exchange format.

2. Regulations (from infrastructure manager Bane NOR’s technical regulations) which give obligations or recommendations on the design of the railway infrastructure.

3. Expert knowledge given in textual form apart from official regulations, used to gather and formalize engineering practice.

RailCNL has been developed to support Norwegian language regulations. All the examples presented below have been translated into English for the purpose of presenting them in this text.

The main activities of this methodology are:

1. Define an abstract syntax which is able to represent statements of relevant texts. We suggest two sub-activities to help manage the difficulty and complexity of modelling domain-specific, yet diverse and informally structured, texts:

   (a) Logic-driven design where basic (often non-domain-specific) constructs which are known from the verification logic are added in a “bottom-up” fashion.

   (b) Text-driven design where highly domain-specific constructs are added to the language to model specific examples in original texts in a “top-down” fashion.

2. Write a concrete syntax, mapping the abstract syntax into one or more natural languages, using Grammatical Framework and its resource grammar library.

https://trv.jbv.no/
3. Create a **translation** from the abstract syntax to the target logic formalism, i.e. the verification properties expressed in the input language of the solver.

In theory, these steps could be performed one after the other, each depending only on the previous steps in the list. In practice, however, the activities have subtle cross-dependencies, for example the need for reducing ambiguity by encoding more restrictions in the types, the usage of restricted keywords, and the need for structure on larger scales than a single sentence. Section 4.4 addresses each of these concerns.

Developing a specialized translation algorithm (see Section 4.5) instead of going through the GF framework is encouraged when the end result is a complex logical language, as in our case. In the translation algorithm we can also incorporate various optimization aspects (see Section 4.6).

### 4.1 Abstract Syntax

Given a body of informal specifications it might not be feasible nor desirable to aim for complete coverage in any sense, for the following reasons:

1. The text might have more or less non-normative content intended only to give readers a better understanding of the subject matter.

2. Parts of the normative content might not be suitable for modelling in the target verification tool. For example, overly broad statements such as “the system shall ensure safety in all possible conditions” are often part of regulations even if they do not lend themselves to any direct action. In our railway verification, we have limited to static infrastructure properties, whereas any properties requiring dynamic analysis are left to other stages (and tools) of the system design. A CNL can still be designed to model more properties than those which are translatable into the verification logic.

3. The available body of text might be large and complex, and covering all parts of it could require diverse domain knowledge from various disciplines. In our railway case, we focus on the disciplines of track and signalling design, as these are the sub-disciplines of railway engineering for which we have had access to domain experts during the design of our verification system.

Furthermore, starting from arbitrary sentences in the natural text and trying to cover it with the CNL will often prove to be a daunting task, given the variety of sentence structures, variety of contexts and levels of abstraction, and variety of domain knowledge needed to make sense of the statements.

#### 4.1.1 Middle Ground: Modelling the Text vs. Modelling the Logic

Our approach to handling this difficulty is to use a combination of computer scientists’ understanding and domain experts’ understanding to suggest possible building blocks for the language. We define some language constructs that the regulations might be expressed through, and only later use the actual text and adjust the CNL to better match it, preserving the basic concepts underneath. We call this first part logic-driven design, as opposed to the second part which we call text-driven design. These two phases are not completely separate or independent by any means, but provide a useful mental model for approaching the design of the CNL’s AST. This approach follows the notion of language oriented programming [34], where identifying a high-level language to be used as a middle ground between bottom-up and top-down programming breaks the system design into two parts which can be handled separately. Similarly, we use the CNL as a middle-ground between the original texts and the verification system.

Even when deciding on the basic logic of the language, it might still be wise to abstract away from the details of the underlying verification logic (as noted in [29 Sec. 5.2]). In our railway verification case, even if many regulations can be concisely expressed as Datalog programs, the abstract syntax of these programs might not resemble the structure of the original text they were expressed in. As an example, Datalog does not nest
predicates, so explicitly naming variables is required to express that an object has both a class and a property, while in natural language, a named variable would not be needed for such a statement.

**Datalog:**

\[
\text{main_signal}(X) :\text{-} \text{signal}(X), \text{type}(X, \text{main}).
\]

By designing a language to have a level of abstraction closer to how the original texts are written, the details of the underlying formal language, its logic, or the verification system, might be changed without devaluing the knowledge base built by encoding domain knowledge into the front-end language. For example, the ontological statements in RailCNL could also be translated into an ontology language such as OWL.

On the other hand, focusing exclusively on linguistic structures will not be sufficient to bridge the gap between informal and formal specifications. To make use of the CNL input in the verification, the language needs to be designed with the target translation in mind. A consequence of this compromise is that the language will seldom be able to cover the exact wordings used in the original texts. We accept this consequence and aim instead to provide a user-friendly comparison of original text and CNL text for traceability (see Section 5.2).

### 4.1.2 Vocabulary: Static vs. Dynamic

If the full vocabulary of the language is known in advance, we can define constant constructors which represent each atomic concept. In this case, the resource grammar library provides functions for setting up the required morphological variations of lexical categories, for example by giving the stem and gender of a noun.

However, we would like our CNL to be able to also define new terms and subsequently construct statements using these new terms. This would imply that the vocabulary is not static, and cannot be compiled in advance.

The most important lexical category for dynamic vocabulary would be nouns, possibly composite nouns. Most of the morphological variation for these (in Norwegian) would be given by their gender. A work-around for dynamic vocabulary could be to encode the gender in the abstract syntax. This would allow natural and consistent use of gender for noun added dynamically to the vocabulary, but this technique ties the AST to the concrete language and could thus make it harder to handle several concrete syntaxes.

To avoid excessive ambiguity caused by allowing arbitrary words in the grammar, we can declare a set of keywords which should never be parsed in the arbitrary names category. This can be implemented in the Grammatical Framework’s runtime library as a callback function which disqualifies certain parses by examining the arbitrary word.

Another work-around used in [19] is to write new vocabulary items back into the GF source code for the language and recompile the GF grammar. We rule this approach out for this project to avoid having to distribute the GF compiler and Haskell runtime with our CAD tool (see Section 5.2).

### 4.2 Logic-Driven Design in RailCNL

In the logic-driven design activity, we used our own knowledge of the verification logic and also interviewed railway engineers about their expectations for logical structures which should be present in the language. The engineers we have interacted with are representatives of the signalling and track design disciplines for Norwegian railways, and the choice of language constructs is mostly motivated by covering the needs of these disciplines in the setting of static infrastructure verification.

#### 4.2.1 Top-Level Statement Types

Most normative sentences in railway regulations are classified into one of the following types, or their negation (see details in Appendix ??):

- **Constraint**: logical constraints on the railway infrastructure model. These sentences can be used by the Datalog reasoner to infer new statements. **Example:** A signal of type main is a main signal.
4.2.2 Generic Ontology Module

Statements about classes of objects and their properties form a natural basis for knowledge representation. We allow arbitrary string tokens to represent class names, property names and values, and compose these in various ways.

- **Class names**: are arbitrary words, optionally prefixed with another arbitrary word. The reason for allowing this is to give the CNL the power to define new words. *Example: the class “railway object” is defined with the statement “A signal is a railway object”. (All given examples are formalized in Appendix ??.)*

- **Properties and values**: can be arbitrary string tokens. These can be joined by “and” or “or” both on the level of values and of properties. *Example: A project which is a new construction should have quality normal or high.*

- **Restrictions**: Equality is a common case of restriction for which we simply choose the wording “to be”. Other restriction types such as greater than, less than, etc., are worded more verbosely. *Example: A main signal should have height which is more than 1.5m and less than 5.0m.*

- **Relations**: the basic ontology module contains multiplicity restrictions on relations. In the layout module presented below, we will see how relations are used when writing statements which are concerned with more than one object simultaneously.

  *Example: A distant signal should have one or more associated signals.*

4.2.3 Layout Module

For writing statements about the topology of the railway track graph, e.g., about paths as in Figure 5, we use the following language constructs:

- **Goal object**: modifies the Subject type defined in the ontology module to add conditions which make sense in a railway graph search, such as the object’s orientation (same direction or opposite direction as the source object), the search’s direction (forwards or backwards from the source object) or the termination properties of the search (search only for the nearest, or search for all).

- **Path condition**: argument to the search constructors which specifies what restrictions are placed on the paths from source to goal object. Typically this concerns the presence of yet another object in the path.

  *Example: A distant signal should have one or more associated signals.*

\[ \text{Figure 5: Switches give rise to branching paths, defining a graph of railway tracks.} \]
• **Path restrictions:** the combination of the source object, goal object and path conditions. *Example:* All paths from a station border to the first facing switch must pass an entry signal.

![Diagram](image.png)

• **Distance restrictions:** *Example:* The distance from an entry signal to the first facing switch must be greater than 200m.

![Diagram](image.png)

4.2.4 Area Module

The areas module covers both planar areas and linear sections. For example, an object being inside a tunnel means that it is contained in the planar area occupied by the tunnel. An example of a linear section is to describe that an object is placed in a curved section on the track. This is a property which occupies a linear section of the track.

• **Subject constructor:** the Subject is extended to add a prepositional phrase containing area information, such as being inside of a tunnel or on a bridge.

• **Placement restriction:** extends OntologyRestriction to allow restrictions on object being inside areas. *Example:* A signal should not be placed in tunnel or bridge.

![Diagram](image.png)

4.3 Text-Driven Design

In this activity we identify from the original regulations text any adjustments that need to be added to the constructs above to cover these original statements. An example from our railway verification language is the following statement about the rotation of signs.

**Text:** To avoid undesirable reflections, signs should not be oriented orthogonally to the track, but rather at an angle of 4º.

To represent this sentence in RailCNL, we first throw away the non-normative explanatory part (“to avoid undesirable reflections”), and then paraphrase into a new sentence using constructs from the logic-driven design phase:

**Text:** A sign should have orientation relative to the track tangent which is greater than 94deg.
4.3.1 Another Example: Train Running Times

A variation on the distance restriction is to use running time (travel time) from one place to another. These are typically used as heuristics for the control system’s maximum performance. This running time can be e.g. nominal speed (the allowable speed on the distance), or maximum dynamic time (maximum speeds taking acceleration and braking into account).

Keeping the running time between axle counters below a certain limit helps ensure capacity on heavily used railways. Example: Running time at nominal speed from an axle counter to another must be less than 25s.

4.4 Concrete Syntax

The abstract syntax described above is mapped into a natural language by using the GF resource grammar library. Each category of the abstract syntax is mapped into a linearization type, often a record data structure from the resource grammar library. For example, the Subject category of RailCNL is assigned the complex noun (CN) record type, and the Statement category is assigned the utterance (Utt) type. Each function in the abstract syntax is mapped into natural language by using the resource grammar library API, which combines e.g. noun phrases and verb phrases into utterances in a language-independent way. How to use this API for creating linearizations is well covered in the Grammatical Framework documentation and literature (e.g. [30, 29]).

See Figure 6 for an example parse tree from RailCNL.
4.4.1 Keeping Ambiguity under Control

The motivation for formally controlled natural languages is that they can be unambiguously parsed as long as the language is restricted enough. Languages written using GF are often restricted to a pre-compiled vocabulary, to be able to identify structure and handle morphological variation. For our verification application, however, we need users to be able to define new terms dynamically, e.g. class names, and afterwards write statements using both built-in and user-defined classes. Allowing arbitrary string tokens in too many places makes the parser generate many parse trees for a single statement. The problem can be mitigated by several means.

**Type-level Restrictions** The railway term “main signal” is the common way to refer to a signal which, in an object-oriented programming model, has the property that its type is main. A straightforward way to add such modifying adjectives as a prefix to class names would be to add a constructor `Adjective : String -> Class -> Class`, which can be used to add adjectives to any class, and with the result of this operation also being a class. However, this approach would mean that any amount of arbitrary words ending with a class name (which could also be an arbitrary string) would be a valid parse. An undesired example is that the subject “a main signal which has height 10m” could be parsed as the class “10m” with six modifying prefix adjectives.

Usually, only one or two adjectives are prefixed to a class name. We can encode this restriction in the type system by separating the adjective-prefixed class name from the non-prefixed one. We also add two adjective constructors, one for adding an adjective and one for transforming the type `BaseClass` into `Class` without prefixing an adjective.

```
Adjective : String -> BaseClass -> Class
NoAdjective : BaseClass -> Class
```

In this way, we greatly reduce the ambiguity introduced by adding arbitrary prefixed strings to class names. We have used this approach a number times, for example to add optional names to areas as described above.

**Reserved Keywords** Using unrestricted strings as building blocks of our language resembles the use of identifiers to name variables in programming languages. Classical programming language grammars have reserved keywords which cannot be used as variable names. Similarly, we can use the GF parser call-backs system to remove parses which contain certain words (such as “which”, “has”, “is”, “must”, “be”, etc.) as arbitrary names.

**Weighted Constructors** The GF parser has support for probabilistic grammars, which work by assigning weights (probabilities) to the constructors of the abstract syntax. By assigning a low weight to any constructor which uses the `String` category, we can ensure that built-in syntax is always prioritized over arbitrary tokens.

**Syntactic Guides** Mainstream programming languages rely on assigning precise meanings to punctuation and symbols to guide their parsers to recognizing the structures of the language. If we are willing to compromise on the natural text-like look of a CNL, it is possible to use this technique, for example to delimit phrases which are to be parsed as a certain category. For example, in the following sentence we could use the curly brackets to indicate a class name (now allowing any number of tokens), and the square brackets to indicate placement.

Text:  

```
A {home main signal} should not be placed [in a tunnel].
```

For a human reader, if the meaning of the statement is preserved when ignoring the brackets, the CNL can still be said to be readable as natural text.
An alternative to using punctuation and symbols is to increase the verbosity of the syntax. Compare the following examples:

**Text:** A signal of height 5.0m.
**Text:** A signal which has height which is equal to 5.0m.

The second one is less likely to cause ambiguity when embedded in a longer statement. Adjusting the verbosity of the syntax is a method for making a trade-off between naturalness/conciseness and potential ambiguity.

### 4.5 Translating CNL into the Target Logic Formalism

Verification using formal methods relies on expressing properties in a logic formalism. Properties are translated, usually from some informal textual representation, into the formal input language of a verification tool. Instead of computer scientists writing the properties directly into the formal language, the CNL verification front-end methodology relies on being able to produce an automated translation from the CNL’s AST to the verification logic. If the target logic has been kept in mind when designing the AST, it should be possible to write a general translation using techniques from programming language compilers. In this section we describe the techniques that we used to create a fully automatic translation of RailCNL into Datalog. Each top-level constructor in the CNL definition has a translation function into the Datalog AST.

#### 4.5.1 Predicate Representation

We employ the following predicate conventions:

- Class membership as `classname(object)`.
- Object properties as `propertyname(object, value)`.
- Relation between objects as `relationname(object, otherobject)`.

#### 4.5.2 Explicit Variables

The subject of the sentences of the Ontology module defines an arbitrary individual whose definition does not depend on other information. To translate it into Datalog, it is sufficient to invent a single variable, denoting the arbitrary individual.

**Text:** A signal which has height 4.5m.
**Datalog:** `signal(X), height(X, 4.5).`

The subject makes the starting for the translation into Datalog, as other parts of the sentence refer back to the subject. In the following example we first need to process the SubjectCondition part of the sentence (“A signal which has height 4.5m”), find a fresh variable name for it, and then process the consequent (“is a tall signal”), which implicitly refers back to the subject (“X”).

**Text:** A signal which has height 4.5m is a tall signal.
**AST:** 
```
OntologyAssertion (SubjectCondition (StringClassNoAdjective (StringClassNeutrum "signal")) (ConditionPropertyRestriction (MkPropertyRestriction (StringProperty "height") (Eq (MkValue (StringTerm "4.5m"))))) (ConditionClassRestriction (MkClassRestriction (StringClassAdjective "tall" (StringClassNeutrum "signal")))))
```
**Datalog:** `tall_signal(X) :- signal(X), height(X, 4.5).`
4.5.3 Ontology Assertions

As seen in the previous example, translations of ontology assertions take the subject, construct a rule body from it, then take the consequent condition, and create a rule head containing a rule head from it. As Datalog allows only single predicates as rule heads, this means that we cannot write assertions which imply disjunctions. For example, the following text can be parsed by our CNL parser, but not translated to Datalog.

Text: A signal has height 4.0m or 4.5m.

AST: OntologyAssertion (SubjectClass (StringClassNoAdjective (StringClassNeutrum "signal"))) (ConditionPropertyRestriction (MkPropertyRestriction (StringProperty "height") (OrRestr (Eq (MkValue (StringTerm "4.0m"))) (Eq (MkValue (StringTerm "4.5m"))))))

This limitation corresponds to the theoretical restrictions on Datalog. Allowing such sentences is the defining characteristic of a Datalog extension called Datalog with disjunctive heads, which has higher computational complexity than plain Datalog. For example, three-coloring of a graph would be expressible in Datalog with disjunctive heads. Note that merely checking that all signals have height either 4.0m or 4.5m is certainly expressible in Datalog, and is covered by the ontology restrictions described below.

4.5.4 Ontology Restrictions

For ontology restriction, such as obligations (“must”) and recommendations (“should”), the Datalog rule head contains a predicate which stores any violations of the text.

Text: A signal must have height 4.0m or 4.5m.

AST: OntologyRestriction Obligation (SubjectClass (StringClassNoAdjective (StringClassNeutrum "signal"))) (ConditionPropertyRestriction (MkPropertyRestriction (StringProperty "height") (OrRestr (Eq (MkValue (StringTerm "4.0m"))) (Eq (MkValue (StringTerm "4.5m"))))))

Datalog: rule1_found(Subj0) :- signal(Subj0), height(Subj0, 4.0).
rule1_found(Subj0) :- signal(Subj0), height(Subj0, 4.5).
rule1_obl(Subj0) :- signal(Subj0), !rule1_found(Subj0).

4.5.5 Disjunctive Normal Form

As Datalog does not (necessarily) have an or operator, nor negation over complex terms, these must be factored out into separate rules and auxiliary predicates. This transformation can be performed by considering the result of the transformation of a sentence to be a set of rules, and the result of the partial translation (such as adding a class or property constraint to a rule) to be a set of conjunctions which are prefixes of the final rules.

The following example shows an or-statement translated into several rules.

Text: A main signal should not be placed in tunnel or bridge.

AST: PlacementRestriction NegativeRecommendation (SubjectClass (StringClassAdjective "main" (StringClassNeutrum "signal"))) (OrArea (SingleArea (NoRestrictionArea (NonSpecificArea TunnelArea))))
(SingleArea (NoRestrictionArea (NonSpecificArea BridgeArea))))

Datalog: r1_rec(S0) :- main_signal(S0), in(S0, A1), tunnel(A1).
r1_rec(S0) :- main_signal(S0), in(S0, A1), bridge(A1).
4.6 Simplifications and optimizations

4.6.1 Layout Properties

Creating Datalog rules for layout properties requires reasoning about paths and distances of a directed graph. We start from a relation describing edges of the graph, from $e_1$ to $e_2$ with distance $d$ is $\text{next}(e_1, e_2, d)$. It could be possible to define general connected and distance predicates, as we have used in our previous work [24]. However, this can become inefficient, especially if using a bottom-up materializing Datalog solver, which would then compute the transitive closure of the whole graph and distances of all paths in the graph. For a single project concerning a small to medium-sized train station, this might be acceptable as verification procedure, but to achieve our goal of on-the-fly verification and/or large-scale verification of railway lines spanning many stations, or even a whole national network, we must ensure that rules can be localized. For example, specifying minimum distances between objects (such as the minimum separation of 21.0m for train detectors) should not lead to calculation of distances between all pairs of train detectors.

To achieve a local search, we use the $\text{next}$ relation directly when translating the CNL to Datalog. We think of the search as a state machine with one Datalog rule corresponding to each state, see Figure 7. One or more searching states are recursively defined to create a transitive closure of the $\text{next}$ relation, guarded by distance conditions, path conditions, etc. Finding the search goal, under the given conditions, leads to an accepting state, which is a relation containing the violation of the specification given in the text. By introducing more states, more complex path conditions can be represented – however, the naturalness of the input language puts a natural limit on the number of states which can arise from a single sentence. More complex situations should be described by defining (naming) intermediate concepts.

```
figure 7: Datalog rules used to execute the distance search from “home main signal” to “first facing switch”.
```

Text: The distance from a home main signal to the first facing switch must be greater than 250.0m.

AST: DistanceRestriction Obligation (SubjectClass (StringClassNoAdjective (StringClassNeutrum "home_main_signal"))) (FirstFound FacingSwitch) (Gt (MkValue (StringTerm "250.0m")))

Datalog:

```
r1_goal(Goal0) :- switch(Goal0).

r1_start(S0,E,D) :- signal(S0), next(S0, E, D).

r1_start(S0,E,D) :- r1_start(S0, M, D0), next(M, E, D1), D=D0+D1, D < 250.0, !rule1_goal(M).

r1_end(S,E,D) :- r1_start(S,E,D), rule1_goals(E).
```

A general advice for writing compilers is that the basic code generation algorithm should focus on correctness, which is best obtained by keeping it simple. Optimizations in programming language compilers are often kept as a separate optional step which operates on a low-level intermediate language, if not on the machine code itself. For the Datalog output from our compiler, we have kept this in mind, and have tried to keep the Datalog code generation as general as possible. Separating optimizations from the basic code generation also means
that modifying or extending the CNL, which requires a corresponding modification of the Datalog translator, becomes easier.

Using this general code generation, some code is more verbose than a hand-written Datalog program would be. Where significant, we simplify the Datalog code as an optional rewriting step after code generation. This step increases the readability and efficiency of the code.

4.6.2 Inlining

Simple instances of OntologyRestriction statements can often be written as a single rule. However, the general translation procedure splits this up into finding correct instances (predicate name ending in “found”), and a separate rule identifying the same objects with a negation of the found rule (predicate name ending in “obligation” or “recommendation”). Whenever the found predicate has only a single atom which is different from the obligation/recommendation rule, then it can be inlined into the same rule.

Text: A sign should have angle to the track tangent which is greater than 94°.

Datalog: sign_found(Subj0) :- skilt(Subj0), tangent(Subj0, Val2), Val2 > 94.

sign_recommendation(Subj0) :- skilt(Subj0), !skilt2_found(Subj0).

After performing inlining simplification we get instead.

Datalog: sign_recommendation(Subj0) :- skilt(Subj0), tangent(Subj0, Val2), Val2 >= 94.

4.7 Bi-directional Translation

Converting from Datalog to RailCNL might be complicated. Straight-forwardly writing an explicit transfer function from Datalog to RailCNL might not be feasible. Also, if we consider the RailCNL grammar an (extensible) extension of the Datalog grammar, then there might be several stages of refinement going from the Datalog language to the RailCNL language. These factors suggest that a term rewriting system might be suitable. In this case, we should be able to show that the Datalog language is a normal form. See [?] for an approach of using rewriting rules in a first order logic natural language generation system, going from the low level logic representation to a higher level AST which generates more natural sentences.

It might not be a top priority to convert from Datalog to RailCNL, but it could also be used to illustrate how extending the RailCNL with better constructs improves the naturalness of the language.

4.8 Matching Vocabulary

The Norwegian regulations are written in Norwegian and use other terms for class names, properties and relations than the railML representation does. After identifying the class names from the CNL, they will be looked up in a Norwegian/railML dictionary. For example, Norwegian “akselteller” is mapped into the railML class “trainDetector” with the “axlecounting” property.

4.9 Maintenance of the Language

Some of the constructs in the CNL are highly specific to the text we are modelling, which is expected since the text freely uses a wide range of background railway knowledge, general engineering and mathematical knowledge. The main challenge in designing such a CNL is to find the underlying concepts, and to strike a balance between matching the level of abstraction on which the original text is based and introducing many special-purpose language constructs. More generally, any domain-specific language (DSL) must to some extent evolve alongside the needs of the applications it supports. See [13] for a general treatment of DSLs.

We envisage that RailCNL will evolve over time to include both new terminology appearing in new regulations as well as new knowledge and engineering practices. Therefore, the language would be maintained by
5 Tool Integration

5.1 CAD Integration

In [24], we presented and demonstrated a verification tool for static infrastructure properties based on evaluation of Datalog rules. The tool is integrated into the RailCOMPLETE® software, a professional railway CAD program for producing and editing railML representations of railway infrastructure. The railML format [?] is an international standard for describing railway infrastructure, time tables, and rolling stock information. The railML description is transformed into a Datalog for verification.

Our prototype implementation uses XSB Prolog which does conventional top-down Prolog search, combined with tabling of recursive predicates, ensuring the Datalog properties of termination and polynomial running time. Fig. 8 shows the graphical representation indicating to the engineer which regulation is violated.

5.2 Requirements Tracing

Verification methods usually output a counter-example when the requirements are violated by the model. It is often difficult to understand from the counter-example which (possibly several) of the requirements have been violated, and why. We use the notion of tracing to trace such errors from the verification output all the way to the original text regulations (explained in Figure 8).

When using a CNL as a verification front-end language, previously written specifications in unrestricted natural language can seldom be parsed directly. We mark-up sentences of the original text with an identifier, and created a separate document containing the formalized representation using RailCNL, containing the identifiers
==General==

a) \(<label id="detector_1">No detection section shall be shorter than 21 meters. \</label>\)

b) No dead zones shall be longer than 3 meters.

\[\text{<rule class="static-infrastructure-datalog" textref="detector_1">}
\]

\[\text{<RailCNL>}
\]

The distance from an axle counter to another must be greater than 21.0m.

\[\text{</RailCNL>}
\]

\[\text{</rule>}
\]

Figure 9: Excerpt of original text marked-up with sentence identifiers, and properties represented in CNL with references to original text.

as references back into the original text as explained in Figure 9.

When the verification program finds a violation among the regulations or expert knowledge, it outputs the identifier of the rule which has been violated, allowing tracing back through the Datalog code, the AST, and the CNL code, to the original text, allowing railway engineers to see the how each step is linked together. Figure 8 shows our prototype tool presenting a problem in the CAD view, and how it is traced back to the relevant regulations text.

When producing new regulatory texts or writing down expert knowledge for which a CNL exists, the approach of *embedded controlled language* [28] can be used to create natural texts where some sentences are directly parsable into verification properties. The rest of the text is considered to be comments or explanations, similar to the programming approach known as *literate programming*.

5.3 Editors: Reading vs. Writing in a CNL

A formal CNL with a linearization defined using the Grammatical Framework tools can be very natural, and is often perfectly readable for a non-programmer with the required domain knowledge. However, writing statements in the formal CNL is a completely different challenge. In the writing setting, the seeming naturalness of the language can be a trap that makes the user forget about the formal grammar behind, if they even knew about the grammar in the first place. Writing in a formal CNL can be of similar difficulty as writing in a programming language.

However, a standard mitigation to this problem is the use of special-purpose editors which guide the user towards structuring their text according to the formal grammar. Different approaches to constructing CNL editors have been explored (see e.g. [9,23,26]), however we could not find an editor that could be readily used with our RailCNL and easily integrated into our application. Figure 10 shows the graphical user interface used in the editor for the Contract CNL of [9]. As future work, we hope to be able to integrate a CNL editor in our tool chain to give railway engineers better control over maintaining and creating the verification properties.

6 Conclusions

The aim of RailCNL is to formalize in a human-readable manner relevant parts of the technical regulations used in an on-the-fly verification engine integrated within railway construction design software. This type
of verification is limited to static infrastructure analysis, leaving the more heavy-weight analysis, e.g., the implementation of control systems or interlocking specifications, to specialized analysis software such as the products of Prover AB (Sweden) or Systerel (France).

We have collaborated with railway engineers associated with RailCOMPLETE during the design of the language, and have written verification properties together with them. They have also suggested further sets of properties which are not currently covered by the grammar, and suggested where the coverage would be overly verbose and complex. In these cases, we continue working on adding constructs to the language. In particular, the text-driven design is an ongoing process which cannot be complete as long as the rule base is expanding.

We have used the Grammatical Framework which has resource grammar library providing support for some 32 languages. While we have so far used Norwegian language for representing regulations, the system can easily be translated for use with other authorities’ regulations written in other languages. As long as most of the abstract syntax are re-used, the translation into Datalog should also be readily adaptable.

Designing RailCNL is our approach to participatory verification, where the end user, the railway engineers in our case, gets access to the verification properties (i.e., can read and understand) and also can change and add such properties. This allows the engineers to actively participate in the verification by maintaining the rule base and managing their own (often based on experience and best practice) properties. In our case the verification models are already generated automatically from the CAD designs of the engineers. However, in other instances of participatory verification one may need to look at this as well, i.e., to allow the user to define the models in a language (maybe visual, as with the C-O diagrams [9]) fairly usable.

6.1 Related Work

6.1.1 Railway Verification

Railway control systems and signalling designs are a fertile ground for formal methods. See [4] [11] for an overview of various approaches and pointers to the literature on applying formal methods in various phases of railway design. In particular, safety of interlockings has been intensively formalized and studied, using for instance VDM [13] and the B-method, resp. Event-B [22]. 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. [7, 35, 10]. Critically evaluating practicality, [12] investigated applicability of model checking for interlocking tables using NuSMV resp. Spin. To mitigate the state-space explosion problem, [17] uses bounded model checking for interlockings. An influential technology is the verified code generation for interlockings from Prover AB Sweden [5], using an automated theorem prover based on Stålmarck’s method.

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 static aspects of the infrastructure.

6.1.2 Domain Specific Languages for Railway Verification

Other efforts to define domain specific languages for railway verification have typically focused on the implementation of control systems, such as Vu et al. [33], while also considering the verification as a separate activity from design and implementation. James et al. [18] show how to integrate UML modelling of the railway domain with graphical modelling and specification and verification languages, but also keep the focus on verifying the control system of a fixed design. The logical specifications used in formal verification of control system implementations can be more intricate and subtle than the specifications and heuristics used in layout design, and so the emphasis these works have is on formal languages rather than naturalness of expression.

6.1.3 Controlled Natural Languages

Grammatical Framework and its resource grammars has been used for building controlled natural languages in several domains, and there are tutorials and best practice documents available, see e.g. [30].

6.1.4 Controlled Natural Language Editors

6.2 Future Work

The constructs of RailCNL could be expanded from covering mainly the rule input to the static verification procedure, into covering some or all of the following:

1. Control system implementation verification, typically more computationally expensive, and based on other formalisms, such as timed automata, first order logic, etc.

2. Verification of human activities and processes, such as regulations for maintenance scheduling, or regulations for design considerations. (For example, a non-standard design choice for infrastructure could be acceptable only when reasoning has been given for abandoning the standard choice.)

3. Analysis of the regulations in themselves, for example exposing inconsistencies.

4. Tranforming the regulations into a simpler, unambiguous text for human readers.

The creation of CNL editors have already to a certain extent improved the usability and value of CNLs in general, but in our opinion there is much to gain from further development. We would like to explore techniques for creating CNL editors in general, how to tailor them to specific domains and languages, and how to integrate them in a verification tool chain.

Acknowledgements:

We thank Martin Steffen and Aarne Ranta for numerous useful interactions, and Claus Feyling (CEO of RailCOMPLETE AS) for allowing us to use the time of his engineers for testing our results and other railway specific interactions.
References


A Appendix: Overview of Norwegian regulation contents

The technical regulations ("Teknisk regelverk") can be found at [https://trv.jbv.no/](https://trv.jbv.no/) and consists of the following books:

- **Common regulations**: 501 Common regulations
- **Common electrical**: 510 Design and construction
- **Signs**: 515 Placement of signs along the track
- **Superstructure (tracks)**: 530 Design, 531 Construction, 532 Maintenance
- **Substructure**: 520 Design and construction, 522 Maintenance
- **Tunnels**: 521 Design and construction, 523 Maintenance
- **Bridges**: 525 Design and construction, 527 Maintenance
- **Overhead line**: 540 Design, 541 Construction, 542 Maintenance
- **Low voltage and 22 kV**: 543 Design, 544 Construction, 545 Maintenance
- **Power supply**: 546 Design, 547 Construction, 548 Maintenance
- **Signalling**: 550 Design, 551 Construction, 552 Maintenance, 553 Assessment
- **Telecommunications**: 560 Design and construction, 562 Maintenance

Structure of each book:

- Each book repeats the **common regulations** as the first three chapters.
- Following this will typically be a **general** section containing:
  - declaration of the scope of the book,
  - references to relevant standards,
  - definitions of relevant technical terms,
  - qualitative classifications, such as quality classes, risk classes, etc.

- The main part of a book consists typically of 5 to 10 chapters, each detailing a specific technical topic within the discipline. The text consists of:
  - Scope declarations
  - Definitions
  - Non-normative statements
  - Comments
  - Regulations (including tables and figures), with exceptions
  - Examples

The technical regulations contain a lot of generalities which are not necessarily normative, nor directly useful in a design setting. Based on the prioritized use case list, the following parts of the regulations should be considered first in designing and testing the formalization procedure:
1. Superstructure design (track design / Overbygning: 530 Prosjektering), especially regulations and formulas regarding
   - track geometry: curvature, gradients, etc.
   - switches: types, maximum speeds, naming (numbering), etc.

2. Signalling design (*Signal: 550 Prosjektering*), especially
   - signal placement, functions, sighting distance
   - train detector placement, classification
   - switch motors requirements and control system components
   - automatic train protection system (ATC) placement and functions
   - interlocking (control system) routes, conflicts, detection sections, safety classes, flank protection, overlaps, etc.
## B Appendix: Example content from regulations

The following table lists some example excerpts from the regulations along with a translation into English, and a comment about use cases and relevance.

<table>
<thead>
<tr>
<th>Use cases</th>
<th>Original text</th>
<th>English translation</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Overbygning: 530 Prosjektering, Kap. 8 Helsveist spor, 2.1.</strong></td>
<td>De store krefter som kan forekomme i et helsveist spor stiller strenge krav til sporets konstruksjon.</td>
<td>The large forces that may occur in a welded track makes stringent demands on the track construction.</td>
<td>This sentence is not normative, and is unlikely to have any use in automated verification.</td>
</tr>
<tr>
<td>#4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Overbygning: 530 Prosjektering, Kap. 8 Helsveist spor, 2.1.3 a)</strong></td>
<td>Ballasten skal på linjen og i hovedspor på statsjoner være fullverdig grovpukk (av størrelse 31.5 – 63 mm)</td>
<td>The ballast on the line and in the main track at stations must be purely coarse crushed stone (size from 31.5 to 63 mm)</td>
<td>This is a specification which is absolute, and rules out the need for specifying this as a part of the design, because it is not part of a specific station. It can still be valuable to support this sentence in a CNL, and in a formal representation.</td>
</tr>
<tr>
<td>#4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Overbygning: 530 Prosjektering, Kap. 8 Helsveist spor, 2.1.2 a)</strong></td>
<td>Minste kurveradius for helsveist med betongsviller skal være 250 m.</td>
<td>The lowest allowable radius of curvature for whole welded track on concrete sleepers is 250 m.</td>
<td>This is a typical example of static infrastructure verification, expressible in Datalog as: error(Segment) :- trackSegment(Segment), trackSegmentRadius(Segment, Radius), Radius &lt; 250.</td>
</tr>
<tr>
<td>#1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Signal: 550 Prosjektering, Kap. 6 Lyssignal, 2.1.2 j)</strong></td>
<td>Et innkjørhovedsignal skal plasseres ≥ 200 meter foran innkjørtegveiens første sentralstilte, motrettede sporveksel, se Figur 5.</td>
<td>A home main signal shall be placed at least 200 m in front of the first controlled, facing switch in the entry train path (see Figure 5).</td>
<td>This is the example that we have been using most frequently for the RailCons verification tool. Datalog: error(Sig,Sw) :- firstFacing(Bdry, Sw, Dir), homeSignalBetween(Sig, Bdry, Sw), distance(Sig, Sw, Dir, L), L &lt; 200.</td>
</tr>
<tr>
<td>#1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Signal: 550 Signal, Kap. 5 Forriglingsutrustning, 2.8.1 Dekningsgivende objekt</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Use cases</td>
<td>Original text</td>
<td>English translation</td>
<td>Comments</td>
</tr>
<tr>
<td>-----------</td>
<td>---------------</td>
<td>---------------------</td>
<td>----------</td>
</tr>
<tr>
<td>#1, #2</td>
<td>Følgende objekt kan være dekningsgivende: Hovedsignal, Dvergsignal, Sporveksel, Sporsperre, Avsporingstunge, Signal E35 Stoppskilt. Et hovedsignal skal vise signal &quot;Stopp&quot; for å være dekningsgivende.</td>
<td>The following objects can provide flank protection: main signal, shunting signal, switch, derailer, derailing tongue, signal E35 stop sign. A main signal must display &quot;stop&quot; to provide flank protection.</td>
<td>This regulation is relevant both for specifying the control system, and for verifying the implementation. The specification chooses which objects to use for flank protection (static) and what state they can be used in, while the implementation must correctly enforce the conditions saying which message the signal displays (dynamic).</td>
</tr>
<tr>
<td>#1, #3</td>
<td>Et hovedsignal bør ikke plasseres i tunneler, på bruer, eller andre steder hvor en eventuell togstans og dermed muligheten for avstigning, vil medføre fare.</td>
<td>A main signal should not be placed in tunnels, on bridges, or other places where halting trains and thus the possibility of disembarking, can impose dangers.</td>
<td>Here we have an example of a “should” modality, where the static infrastructure verification could issue a warning, but not an error. Also, it could be required to document the alternatives that were considered when deciding on the design.</td>
</tr>
</tbody>
</table>

**Signal:** 550 Prosjektering, Kap. 6 Lyssignal, 2.1.2 i)  

**Signal:** 550 Prosjektering, Kap. 5 Forriglingsutrustning, 4.1.1.1 i)  

**Overbygning:** 530 Prosjektering, Kap. 5 Sporets trasé, 3.1 Dimensjonerende parametre
(a) minimum radius, (b) maximal superelevation, (c) limit on superelevation cause by derailment risk at low speeds, (d) limit for superelevation rate of change, (e) limit for superelevation deficit. Limiting values are organized in a table for use in formulas in other sections.

### Tabell 2: Dimensjonerende parametre for nye baner og linjeomlegginger

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definisjon</th>
<th>Normale krav</th>
<th>Minste krav</th>
</tr>
</thead>
<tbody>
<tr>
<td>a) $R_{\text{min}}$</td>
<td>minste radius</td>
<td>250 m</td>
<td>190 m</td>
</tr>
<tr>
<td>b) $h_{\text{maks}}$</td>
<td>maksimal verdi for overhøyden</td>
<td>150 mm</td>
<td></td>
</tr>
<tr>
<td>c) $h_{\text{tr.xp}}$</td>
<td>grense for overhøyde pga. avsporingsfarer ved lave hastigheter</td>
<td>$\frac{R-100}{2}$ mm</td>
<td></td>
</tr>
<tr>
<td>d) $(\frac{\Delta h}{L})_{\text{maks}}$</td>
<td>grenseverdi for rampestigning</td>
<td>1:400</td>
<td></td>
</tr>
<tr>
<td>e) $I_{\text{maks}}$</td>
<td>grenseverdi for mangelende overhøyde</td>
<td>R ≤ 600: 100 mm</td>
<td>R &gt; 600: 130 mm</td>
</tr>
</tbody>
</table>

**Overbygning: 530 Prosjektering, Kap. 5 Sporets trasé, 3.7 Sporveksler og sporforbindelser**

Avstanden mellom sporveksel og overgangskurve, sirkelkurve, bru eller annen motstående sporveksel skal ikke være mindre enn avstanden M gitt i *Kurver uten overgangskurver*, krav b). M skal imidlertid ikke være kortere enn 6 m. The distance between the switch point and the transition curve, circle curve, bridge or other opposite switch point should not be less than the distance M given in section “curves without transition curves”, requirements b). M shall not be shorter than 6 m. The parameter $M$ is explained by the figure below. Reference is given to another section of the regulations.
**Overbygning:** 530 Prosjektering, Kap. 5 Sporets trasé, 5 Største hastighet – sporets geometri

The speed in a curve shall not exceed:

\[ V = 0.291 \cdot \sqrt{R (h + I_{\text{max}})} \]  

(5)

If Eq. 5 gives a lower value than 20 km/h in situations with false superelevation, then \( V = 20 \) km/h shall be used.

Use of equations with designed and given parameters.

**Signal:** 552 Vedlikehold, Kap. 6 Lyssignal, 3 Lyssignaler

If a signal is twisted or in other ways are out of position, this shall be fixed as soon as possible.

Typical maintenance regulation. Here, it might be sufficient to identify this as a checklist item, for maintenance scheduling and reporting purposes.