Participatory Verification of Railway Infrastructure by Representing Regulations in RailCNL (author version)[⋆] [⋆⋆]

Bjørnar Luteberget[1], John J. Camilleri[2], Christian Johansen[3], and Gerardo Schneider[2]

[1] RailComplete AS, Sandvika, Norway
[2] Department of Computer Science and Engineering,
Chalmers University of Technology and University of Gothenburg, Sweden
[3] Department of Informatics, University of Oslo, Norway
bjlut@railcomplete.no
{john.j.camilleri|gerardo}@cse.gu.se
cristi@ifi.uio.no

Abstract. Designs of railway infrastructure (tracks, signalling and control systems, etc.) need to comply with comprehensive sets of regulations describing safety requirements, engineering conventions, and design heuristics. We have previously worked on automating the verification of railway designs against such regulations, and integrated a verification tool based on Datalog reasoning into the CAD tools of railway engineers. This was used in a pilot project at Norconsult AS (formerly Anacon AS). In order to allow railway engineers with limited logic programming experience to participate in the verification process, in this work we introduce a controlled natural language, RailCNL, which is designed as a middle ground between informal regulations and Datalog code. Phrases in RailCNL correspond closely to those in the regulation texts, and can be translated automatically into the input language of the verifier. 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, allowing domain experts to examine the correctness of each translation step and better identify sources of errors. We also describe our design methodology, based on CNL best practices and previous experience with creating verification front-end languages.

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 in 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. Firstly, 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. Secondly, we want to allow the users to actively participate in maintaining the verification properties, i.e. to change and adjust them when needed.[4] Thirdly, we want that the domain experts are able to create their own specifications and feed these into the verification engine, e.g. define specific expert knowledge as verification conditions.[5]

Involving the user in the design of a system is well-studied in the field of participatory design [19,8]. We use the term *participatory verification* when talking about methods for including the end user in the verification process. The goal is to make automated verification techniques accessible to engineers with little programming experience.

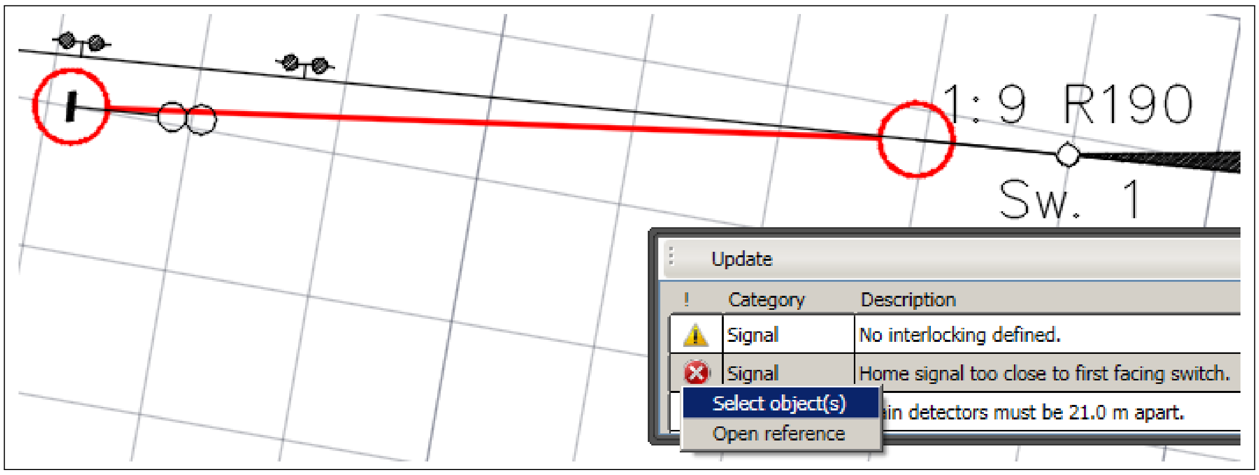
We have previously demonstrated [13,12] an efficient verification and troubleshooting tool integrated into the CAD-based program used by railway planning engineers. This tool performs a lightweight type of verification which we call *static infrastructure verification*, and the results are updated continuously as the engineer is modifying the station (see Fig. 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 evaluated with

[⋆] !This is an author's produced version, and not the final one published by Springer, which can be found at `http://dx.doi.org/10.1007/978-3-319-66197-1_6`

[⋆⋆] Supported by the Norwegian Research Council project *RailCons* – Aut. Methods and Tools for Ensuring Consistency of Railway Designs, and by the Swedish Research Council grant nr. 2012-5746 – Reliable Multilingual Digital Communication: Methods and Applications.

[4] Authorities typically make small adjustments to regulations several times per year, whereas engineering best practices can be revised at any time.

[5] Such expert knowledge is often seen as proprietary valuable assets of the company.



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

Fig. 1: CAD integrated verification engine, displaying errors and warnings after checking the model extracted from the CAD design against railway regulations on-the-fly.

railway engineers from RailCOMPLETE AS[6] our prototype, they raised yet 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 (CNL), which we call RailCNL, meant to be used as an intermediate representation between natural language texts (i.e. the railway regulations) and Datalog [20] logic programs. RailCNL aims to be human-friendly enough for our domain experts to work with to overcome the above challenges, and thus getting them involved in using and improving the automated verification tool. At the same time, the language is a formal language which can be automatically translated into Datalog.

In our collaboration with Norwegian railway engineers, we have focused on regulations in Norwegian language[7], but our general approach (Section 2) is language-independent. In Section 3 we present RailCNL, a user-friendly verification front-end language for static railway infrastructure analysis. This comes with an automatic translation into Datalog (Section 3.3), and backwards tracing integrated into the CAD program, where marked-up original regulation texts are used together with the CNL text to explain regulatory violations found in the model (Section 3.4). In Section 4 we extract a design methodology from our experience with RailCNL,and conclude in Section 5 by describing the coverage of the defined CNL, and presenting related and future work.

2 Approach to Participatory Verification for Railway Regulations

To promote participatory verification of infrastructure railway designs against regulations, we design a CNL for expressing railway regulations and expert knowledge, integrating it with our previously developed verification engine. Fig. 2 presents the overall workflow of using the railway CNL integrated with the engineer's CAD-based environment and our verification engine. Static infrastructure verification requires:

1. *Models:* railway infrastructure plans, typically created by arranging the station layout using CAD-based programs, e.g. extensions of Autodesk AutoCAD.
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 [12] which allows efficient automatic reasoning. Describing verification properties using logical rules in Datalog is not new (along with other logics like temporal [2] or dynamic logics [5,3]), and we expected that the declarative style of Datalog would make it easy for railway engineers to read and write such properties. However, a pilot project with the RailCOMPLETE engineers showed that they were not proficient enough in logic programming to understand our encodings.

To allow the engineers to participate in the verification process, we develop the controlled natural language RailCNL for representing properties on a higher level of abstraction, make them 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.

[6] `http://railcomplete.no`
[7] The examples presented in this text are English translations of originally Norwegian content.

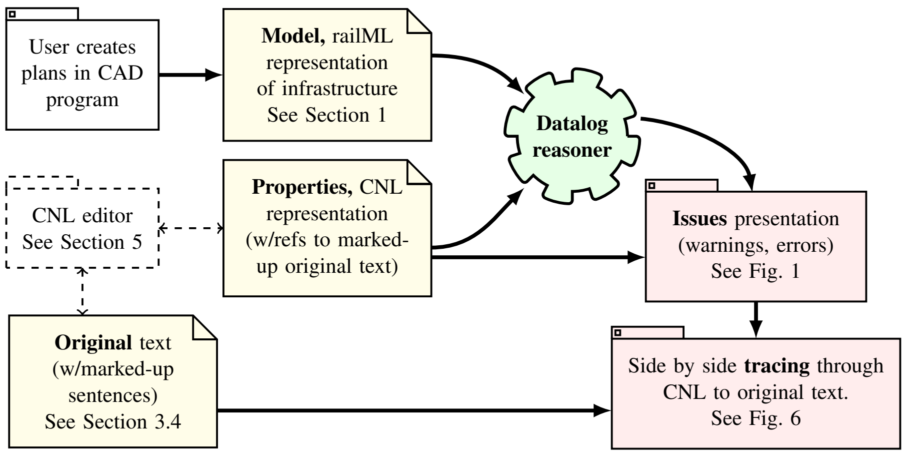

User creates plans in CAD program
Model, railML representation of infrastructure See Section 1
Datalog reasoner
CNL editor See Section 5
Properties, CNL representation (w/refs to marked-up original text)
Issues presentation (warnings, errors) See Fig. 1
Original text (w/marked-up sentences) See Section 3.4
Side by side **tracing** through CNL to original text. See Fig. 6

Fig. 2: Verification process overview. *Models* come directly from the CAD program, which engineers are already familiar with. *Properties* come from paraphrasing the regulations using CNL, which in turn are translated into Datalog. The reasoner outputs *issues* (warnings and errors) which are presented to the user in the CAD program by highlighting the objects involved in the violation. Issues are traced back to the original text (i.e. the regulations) though identifiers on the marked-up sentences.

– 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 clarify semantics, as is done in the original texts.
– 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. If not, they can edit the CNL text.

3 RailCNL: a Front-End Language for Railway Verification

A controlled natural languages (CNL) is a constructed language resembling a natural language (such as English) but with added restrictions on its grammar and vocabulary. The restrictions are typically aimed at reducing the ambiguity and complexity of unrestricted natural language. A CNL may or may not also be a formal language, depending on its intended use. Wyner et al. [22] give high-level recommendations on how to design controlled natural languages ranging from informal to formal, general to domain-specific, simple to complex. For a recent survery of CNLs, see Kuhn [9].

Grammatical Framework (GF) is a programming language for multilingual grammar applications [16]. A GF program defines a grammar consisting of an *abstract syntax* and one or more *concrete syntaxes*. The 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 encapsulates the linguistic complexity of the underlying natural languages, making the effort needed to map an abstract syntax into another natural language minimal, often reducing to simply providing the domain-specific vocabulary. This makes GF a valuable tool for building CNLs (see [11] for details).

3.1 RailCNL Grammar

With RailCNL, we aim to cover the following content (also see Table 1 on page 10):

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 technical regulations[8]) 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.

An English version of RailCNL's core grammar is presented in Fig. 3. The full grammar is defined in GF (see [11]), which has some advantages over classical BNF parsers: (i) separation of abstract syntax and concrete syntax; (ii) resource grammar library for natural languages, allowing us to compose sentences in natural language while abstracting away from morphological details; (iii) modularity and extensibility, which we need for evolving a domain-specific language alongside its application; and (iv) tool support for managing text (editors, predictive parsing, visualization).

[8] Norwegian infrastructure manager Bane NOR's regulations: `https://trv.jbv.no/`

3.2 RailCNL Modules and Examples

RailCNL has a modular design (see Fig. 4) where domain-specific constructs are separated from generic ones. 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 a summary of such trade-offs in Section 4. We describe below the main modules and constructs of RailCNL, with examples of CNL text and the corresponding abstract syntax tree (AST) obtained from the GF parser (see [11] for more examples).

Top-Level Statement Types Most normative sentences in railway regulations are classified into one of the following types, or their negation:

- **Constraint**: logical constraints on the railway infrastructure model. These sentences can be used by the Datalog reasoner to infer new statements.
- **Obligation**: design requirements on the railway infrastructure. The CAD model is checked for compliance, and violations are presented as errors to the user.
- **Recommendation**: design heuristics for railway infrastructure. The CAD model is checked for compliance, but violations are presented as warnings or for information only, which can be dismissed from the view.

Example 1 (Parse tree for an obligation statement.)

CNL: *A vertical segment must have length greater than 20.0m.*

AST:
```
OntologyRestriction Obligation
   (SubjectClass (StringClassAdjective "vertical"
       (StringClass "segment")))
   (ConditionPropertyRestriction (MkPropertyRestriction
       (StringProperty "length")
       (Gt (MkValue (StringTerm "20.0m")))))
```

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.
- **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.
- **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 greater 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.*

Layout Module For writing statements about the topology of the railway track, e.g. about paths as illustrated in Fig. 5c, 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) the search’s direction (forwards or backwards) or the termination properties of the search.
- **Path condition**: argument to the search constructors which specifies what restrictions are placed on the paths from source to goal object.
- **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.* (See Fig. 5a)
- **Distance restrictions**: See Fig. 5b and Example 2.

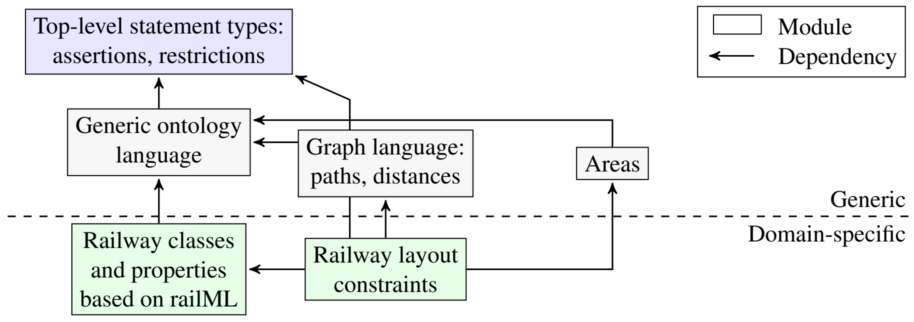
Example 2 (Parse tree for a railway layout statement.)

CNL: *Distance from an entry signal to first facing switch must be greater than 200.0 m.*

AST:
```
DistanceRestriction Obligation
   (SubjectClass (StringClassAdjective "entry"
     (StringClass "signal")))
   (FirstFound FacingSwitch)
   (Gt (MkValue (StringTerm "200.0m")))
```

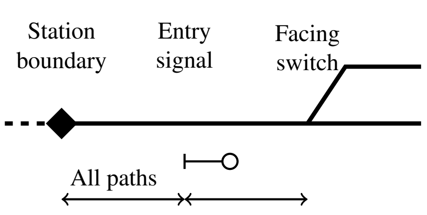
```
⟨Statement⟩ ::= ⟨OntologyAssertion⟩
  | ⟨OntologyRestriction⟩
  | ⟨DistanceRestriction⟩
  | ⟨PathRestriction⟩
  | ⟨PlacementRestriction⟩
  | (...) // Partial grammar
⟨OntologyAssertion⟩ ::= ⟨Subject⟩ ⟨Condition⟩
⟨OntologyRestriction⟩ ::= ⟨Subject⟩ ⟨Modality⟩ ⟨Condition⟩
⟨DistanceRestriction⟩ ::= 'the distance from' ⟨Subject⟩
    'to' ⟨GoalObject⟩ ⟨Modality⟩ ⟨Restriction⟩
⟨PathRestriction⟩ ::= ⟨PathQuantifier⟩ 'from' ⟨Subject⟩ 'to'
    ⟨GoalObject⟩ ⟨Modality⟩ ⟨PathCondition⟩
⟨PlacementRestriction⟩ ::= ⟨Subject⟩ ⟨Modality⟩
    'be placed in' ⟨Area⟩
⟨Modality⟩ ::= 'must' | 'shall not'
  | 'should' | 'should not'
⟨PathQuantifier⟩ ::= 'all paths'
  | 'no paths' | (...)
⟨PathCondition⟩ ::= 'pass' ⟨DirectionalObject⟩
⟨GoalObject⟩ ::= ⟨DirectionalObject⟩
  | 'the first' ⟨DirectionalObject⟩
⟨DirectionalObject⟩ ::= ⟨SearchSubject⟩
  | 'a facing switch'
  | 'a trailing switch'
  | ⟨SearchSubject⟩ ⟨RelativeDirection⟩
⟨RelativeDirection⟩ ::= 'same dir.'
  | 'opposite dir.'
⟨SearchSubject⟩ ::= 'a' ⟨Subject⟩
  | 'another'
⟨Area⟩ ::= ⟨BaseArea⟩
  | ⟨BaseArea⟩ 'which has' ⟨PropertyRestriction⟩
  | ⟨Area⟩ 'or' ⟨Area⟩
  | ⟨Area⟩ 'and' ⟨Area⟩
⟨BaseArea⟩ ::= 'tunnel' | 'bridge'
  | 'local release area' | ⟨Identifier⟩
⟨Subject⟩ ::= 'a' ⟨Class⟩
  | 'a' ⟨Class⟩ 'which' ⟨Condition⟩
⟨Condition⟩ ::= 'is a' ⟨ClassRestriction⟩
  | 'has' ⟨PropertyRestriction⟩
  | 'is a' ⟨ClassRestriction⟩ 'which has'
    ⟨PropertyRestriction⟩
⟨PropertyRestriction⟩ ::= ⟨Property⟩ ⟨ValueRestriction⟩
  | (...) // and/or
⟨ClassRestriction⟩ ::= ⟨Class⟩
  | (...) // and/or
⟨ValueRestriction⟩ ::= ⟨Value⟩
  | 'not equal to' ⟨Value⟩
  | 'less than' ⟨Value⟩
  | (...) // ≤, >, ≥
  | (...) // and/or
⟨Value⟩ ::= ⟨Identifier⟩ | ⟨Number⟩ ⟨Unit⟩
⟨Property⟩ ::= ⟨Identifier⟩
⟨Class⟩ ::= ⟨Identifier⟩
```

Fig. 3: English version of RailCNL's core grammar in BNF. Some linguistic complexity such as subject-verb agreement is ignored here; the actual grammar is fully specified as GF code, which is ideally suited for handling such cases.

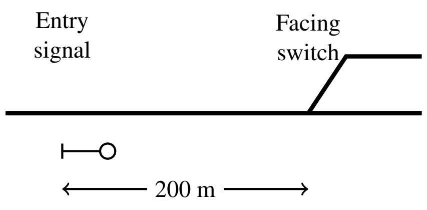


Top-level statement types: assertions, restrictions
Module
Dependency
Generic ontology language
Graph language: paths, distances
Areas
Generic
Domain-specific
Railway classes and properties based on railML
Railway layout constraints

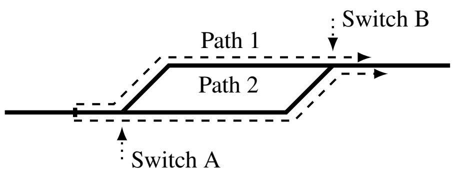
Fig. 4: Modules of the RailCNL (boxes) and their dependencies (arrows). The *generic* modules could be reused when building CNLs for verification in other domains. The *specific* modules are, however, tailored to railway regulations.



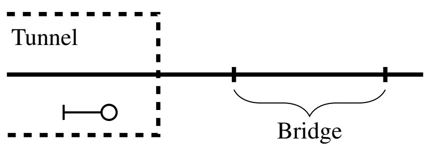
(a) Path restrictions are constructed from a subject, a goal, a quantifier and a condition.



(b) Distance restrictions are constructed from a subject, a goal, and a value restriction.



(c) Switches give rise to branching paths, defining a graph of railway tracks.



(d) Area containment can refer to either a planar region or an interval on a track.

Fig. 5: Conditions on railway geographical layout as supported by RailCNL.

Area Module The area module modifies subjects to express whether they are inside a planar area, such as station areas, tunnels or bridges, or belongs to a linear segment of a track, such as being located in a curve or on an incline (see Fig. 5d).

3.3 Translating RailCNL into Datalog

To make use of RailCNL in the verification tool, ASTs obtained by parsing CNL phrases with the GF runtime are transformed into Datalog rules (a description of how this is implemented can be found in Section 4.3). Each top-level constructor in the CNL definition has a translation function into the Datalog AST.

Predicate Conventions. We employ the following predicate conventions:

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

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, we create a new variable denoting the arbitrary individual. The subject makes the starting point for the translation, as other parts of the sentence refer back to the subject.

Ontology Restrictions. For ontology restrictions, such as obligations ("must") and recommendations ("should"), the Datalog rule head contains a predicate which captures any violations of the text. This is achieved by first defining the restrictions themselves (`r1_found` in Example 3 below) and then declaring a rule which uses the negation of these restrictions (`!r1_found`) in order to yield a counter-example.

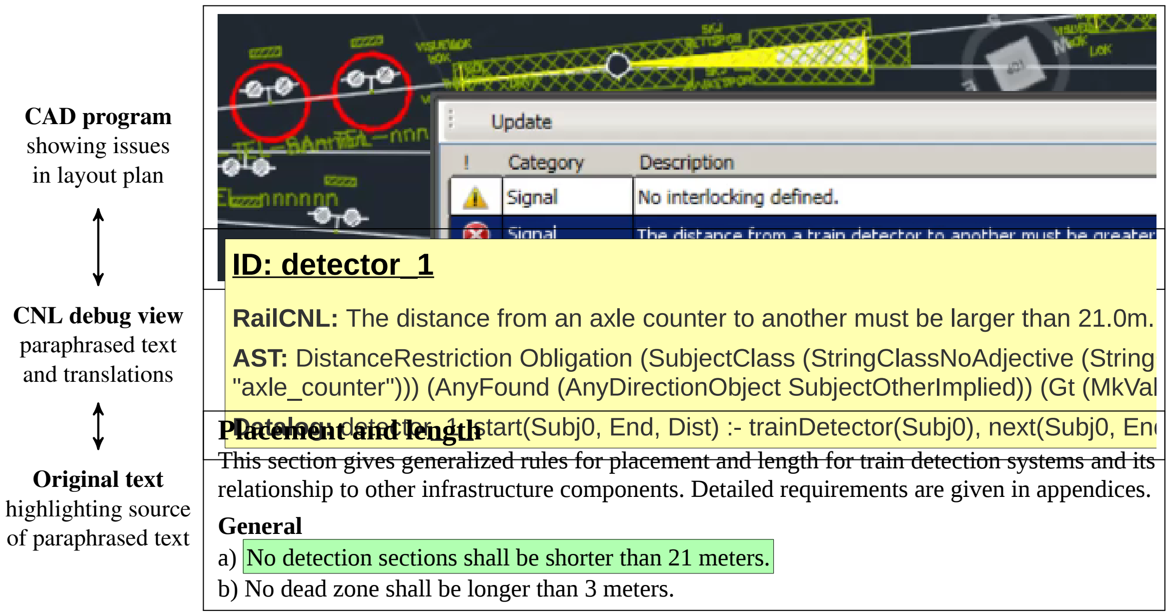
Example 3 (Datalog translation of an ontology restriction.)

CNL: *A signal must have height 4.0m or 4.5m.*

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

Datalog:
```
r1_found(Subj0) :- signal(Subj0), height(Subj0, 4.0).
r1_found(Subj0) :- signal(Subj0), height(Subj0, 4.5).
r1_obl(Subj0) :- signal(Subj0), !r1_found(Subj0).
```

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 translation of a sentence to be a *set of rules* (such as the two definitions of `r1_found` in Example 3), 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.



!	Category	Description
⚠	Signal	No interlocking defined.
✖	Signal	The distance from a train detector to another must be greater

ID: detector_1

RailCNL: The distance from an axle counter to another must be larger than 21.0m.

AST: DistanceRestriction Obligation (SubjectClass (StringClassNoAdjective (String "axle_counter"))) (AnyFound (AnyDirectionObject SubjectOtherImplied)) (Gt (MkVal

Datalog: detector_1 start(Subj0, End, Dist) :- trainDetector(Subj0), next(Subj0, En

Placement and length

This section gives generalized rules for placement and length for train detection systems and its relationship to other infrastructure components. Detailed requirements are given in appendices.

General

a) No detection sections shall be shorter than 21 meters.

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

Fig. 6: Tracing of requirements backwards from CAD program through CNL to marked-up original texts. From a regulation violation presented as a warning or error, the user can browse to the corresponding regulatory text, shown side by side with the CNL text.

```
==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.
```

Original text
marked up with
labels

↕

CNL properties
with references
to labels

```
<rule class="static-infrastructure-datalog"
      textref="detector_1">
  <RailCNL>
    The distance from an axle counter to another
    must be greater than 21.0m.
  </RailCNL>
</rule>
```

Fig. 7: Excerpt of original text marked-up with sentence identifiers, and properties represented in CNL with references to original text.

Vocabulary Matching. 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.

3.4 Tool Integration

Verification tool usually output a counter-example when the requirements are violated by the model. It is often difficult to understand from the counter-example which of the (possibly several) 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. Fig. 6 shows our prototype tool (running as a plug-in for the AutoCAD program used by Norwegian railway engineers) presenting a problem in the CAD view, and how it is traced back through the Datalog code, the AST, and the CNL code, to the original regulations text. We mark-up sentences of the original text with an identifier, and create a separate document containing the formalized representation using RailCNL, using the identifiers as references back into the original text (Fig. 7). When the verification program finds a violation among the regulations, it outputs the identifier of the rule which has been violated, enabling the tracing.

4 Design Methodology for a Verification Front-End Language

Our methodology is based on CNL and GF best practices; in particular, Ranta et al. [18] describe the construction of a CNL by creating an abstract syntax corresponding to a semantic model, mapping it into natural language, and also how to avoid or handle ambiguity in parsing and translating. In a later report, Ranta et al. [17] give explicit best practices, such as: (i) using a modular structure separating generic and domain-specific parts of the grammar, (ii) letting the AST model the semantics of the text, as opposed to the logic of the underlying formalism, and (iii) trade-offs in modelling language restrictions purely in context-free grammar versus using dependent types. We expand on these best practices in the context of creating intermediate languages for writing diverse natural text in a form which is translatable into formal verification properties.

The main activities for defining a verification front-end language using GF 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.
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 practice, the above activities may 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 4.2 addresses each of these concerns.

4.1 Abstract Syntax

Attempting to formally model a body of informal specifications in its entirety may be neither feasible nor desirable, for a variety of reasons:

1. The text might have some amount of 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.
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.

Therefore, starting from arbitrary sentences in the natural text and trying to cover them with the CNL will often prove to be a daunting task. Our approach to handling this difficulty is to split the process of designing the abstract syntax into two parts.

We start with a *logic-driven design*, where we define basic concepts in a bottom-up fashion, such as classifying the statement types (*constraints*, *restrictions*, etc.) and describing sets of objects based on their class and their properties. 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.

Next follows a *text-driven design* phase, where we look for text samples that can be captured in the CNL, and make adjustments and additions to the grammar to cover them. This phase might eventually lead to finding new basic building blocks, such as adding the *graph module* to RailCNL for describing railway layout, or adding *relations* to the ontology module. However, it is easy to get carried away and construct a highly nested language which has too much freedom and therefore becomes difficult to parse. Until the need for more generality is proven, each newly added construct is kept specific.

Alternating between the logic-driven and the text-driven phases can be useful for handling complexity and discovering the middle ground between informal specifications and verification logic. 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 3.4).

4.2 Concrete Syntax

The abstract syntax is mapped into a natural language using the GF resource grammar library (RGL), which is well-covered in the GF documentation and literature (e.g. [18,17]). Each category of the abstract syntax is mapped into a linearization type, often a record data structure. For example, the `Subject` category of RailCNL is assigned the complex noun (`CN`) record type, and `Statement` is assigned to utterance (`Utt`).

A major motivation for formal CNLs 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 terms. But allowing arbitrary string tokens can introduce ambiguity, i.e. the parser returning many parse trees for a single statement. We mitigate this problem through several means:

Type-level Restrictions The railway term "main signal" is the common way to refer to a signal which is of type *main*. Instead of using a recursively defined constructor for this term (e.g. `Adjective : String -> Class -> Class`), we can restrict the number of adjectives to one or two. This restriction is encoded in the type system by separating the adjective-prefixed class name from the non-prefixed one:

```
StringClassAdjective : String -> BaseClass -> Class
StringClassNoAdjective : BaseClass -> Class
```

Reserved Keywords Using arbitrary names as building blocks of our language resembles the use of identifiers as variables in programming languages. Programming languages have *restricted keywords* which cannot be used as variable names. Similarly, we use the GF parser callbacks system to remove parses which contain function words (such as "*which*", "*has*", "*is*", "*must*", "*be*", etc.) as arbitrary names. These are very unlikely to be needed as class or property 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 ensure that built-in syntax is always prioritized over arbitrary tokens.

Syntactic Guides As in programming languages, special symbols and punctuation can be used as guides for the parser if we are willing to compromise on naturalness. Alternatively, we can increase the verbosity of the syntax, to reduce the likelihood of causing ambiguity when embedded in a longer statement.

4.3 Translation into the Target Logic Formalism

If the abstract syntax is made to faithfully model the logic of the verification system, then the translation into the logic formalism can be made by implementing another GF concrete syntax for the target language. However, target logics are often too low-level to represent regulations directly. GF incorporates dependent type features which could allow for a more concise representation of this translation, but this practice has not yet matured to a state in which it can be said to be a recommended practice (see [17]). For RailCNL we have instead written a separate program (in C#, as it is a part of the verification CAD plugin) which translates from the abstract syntax of the CNL into Datalog. Section 3.3 describes the main techniques used.

5 Evaluation and Conclusions

RailCNL formalizes, in a human-readable manner, relevant parts of the technical regulations and expert knowledge 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).

RailCNL is our approach to *participatory verification*, where the end users (railway engineers, in our case) get full access to the verification properties. This allows them to actively participate in the verification by maintaining the rule base and managing their own properties (often based on experience and best practice).

We have collaborated with railway engineers associated with RailCOMPLETE during the design of the language and the writing of the verification properties. Their feedback on limitations in the coverage of the language and suggestions for simplification will continue to drive the design forwards.

We surveyed the Norwegian railway regulations and counted how much of the relevant regulations our basic RailCNL covers (see results in Table 1, and [11] for methodology and examples). The survey is limited to parts of the regulations covering railway track and signalling, as these are the disciplines that the RailCOMPLETE software development is currently focusing on.

RailCNL is impemented using the Grammatical Framework and its resource grammar library (RGL). While we have used Norwegian for representing regulations, RailCNL could be easily extended with other languages supported by the RGL. This would allow the system to be used for other authorities' regulations written in other languages. As long as most of the abstract syntax is re-used, the translation into Datalog should also be readily adaptable.

Eng. discipline	**Chapter title**	Phrases	Normative	Relevant	Covered	**Coverage**
Track	Planning: general technical	140	74	74	70	95%
Track	Planning: geometry	278	157	152	119	78%
Signalling	Planning: detectors	144	106	35	21	60%
Signalling	Planning: interlocking	376	265	130	81	62%
Total		938	602	391	291	**74%**

Table 1: Coverage evaluation for a subset of Norwegian regulations. *Phrases* of the original text which could be classified as *normative* (i.e. applying some restriction on design) were evaluated for *relevance* to static infrastructure verification. The *coverage* is the percentage of relevant phrases expressible in RailCNL.

Related Work Johannisson [7] describes a CNL targeting the Object Constraint Language (OCL) for use in reasoning about Java program correctness in the KeY system [3]. The language features dynamic vocabulary based on input UML diagrams where vocabulary updates are achieved by re-compiling the grammar using the GF compiler when needed. Angelov et al. [1] present a conflict detection framework where GF is used to map the contract language $\mathcal{CL}$ [15] into a CNL. Statement modalities, such as obligation, permission and prohibition, are applied to complex actions. The structure of the CNL is modelled after the $\mathcal{CL}$ language. Camilleri et al. [4] take a CNL approach to manipulating contract-oriented diagrams using a visual diagram editor, a CNL with text editor support, and a spreadsheet representation as interfaces to a common model, which can be translated into timed automata for reasoning about system properties.

Other efforts to define domain specific languages for railway verification have typically focused on the implementation of control systems, such as Vu et al. [21], while also considering the verification to be an activity which is separate from design and implementation. James et al. [6] show how to integrate UML modelling of the railway domain with graphical modelling and specification and verification languages, also keeping the focus on verifying the control system implementation of a fixed design.

Future Work In working with railway engineers, we discovered language features which could be added to increase the coverage of RailCNL:

1. A notion of scopes and exceptions, so that more complex conditional restrictions can be expressed more naturally.
2. Mathematical formulas as a sub-language.
3. Vague or soft requirements represented not for direct use in verification, but for requiring manual checks at some points.

A formal CNL with well-chosen linearizations can be very natural, and often perfectly readable for a non-programmer with the required domain knowledge. However, writing in a formal CNL can potentially be as difficult as writing in a programming language. A solution to this problem is the use of special-purpose editors which guide the user towards structuring their text according to the underlying formal grammar. Different approaches to CNL editors have been explored (see e.g. [4,10,14]). We plan to investigate these further and integrate one such editor for RailCNL in the RailCOMPLETE CAD environment, and carry out a usability study on its efficacy.

We are continuing our collaboration with Norwegian railway engineers to evaluate the usability of our prototype tools, increase the text coverage and extend the language to handle other railway engineering disciplines such as catenary lines and ground works.

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

1. K. Angelov, J. J. Camilleri, and G. Schneider. A framework for conflict analysis of normative texts written in controlled natural language. *JLAP*, 82(5):216–240, 2013.
2. C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.
3. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. Springer, 2007.
4. J. J. Camilleri, G. Paganelli, and G. Schneider. A CNL for contract-oriented diagrams. In *CNL 2014*, volume 8625 of *LNCS*, pages 135–146. Springer, 2014.
5. D. Harel, J. Tiuryn, and D. Kozen. *Dynamic Logic*. MIT Press, 2000.
6. P. James and M. Roggenbach. Encapsulating formal methods within domain specific languages: A solution for verifying railway scheme plans. *Mathematics in Computer Science*, 8(1):11–38, 2014.
7. K. Johannisson. Natural language specifications. In Beckert et al. [3], pages 317–333.

8. F. Kensing and J. Blomberg. Participatory design: Issues and concerns. *Computer Supported Cooperative Work (CSCW)*, 7(3):167–185, 1998.
9. T. Kuhn. A survey and classification of controlled natural languages. *Computational Linguistics*, 40(1):121–170, March 2014.
10. P. Ljunglöf. Editing syntax trees on the surface. In *NoDaLiDa 2011*, pages 138–145, 2011.
11. B. Luteberget, J. J. Camilleri, C. Johansen, and G. Schneider. Participatory Verification of Railway Infrastructure Regulations using RailCNL (long version). Technical report 465, University of Oslo, 2017.
12. B. Luteberget and C. Johansen. Efficient verification of railway infrastructure designs against standard regulations. *Formal Methods in System Design*, 2017.
13. B. Luteberget, C. Johansen, and M. Steffen. Rule-based consistency checking of railway infrastructure designs. In *IFM 2016*, volume 9681 of *LNCS*, pages 491–507. Springer, 2016.
14. M. Moreno and B. Bringert. Interactive multilingual web applications with grammatical framework. In *Advances in NLP*, volume 5221 of *LNCS*, pages 336–347. Springer, 2008.
15. C. Prisacariu and G. Schneider. A Dynamic Deontic Logic for Complex Contracts. *The Journal of Logic and Algebraic Programming (JLAP)*, 81(4):458–490, 2012.
16. A. Ranta. Grammatical Framework. *J. Functional Programming*, 14(2):145–189, 2004.
17. A. Ranta, J. Camilleri, G. Détrez, R. Enache, and T. Hallgren. Grammar tool manual and best practices. Technical report, MOLTO Deliverable D2.3, MOLTO Consortium, Göteborg, 2012. `http://www.molto-project.eu/biblio/deliverable/grammar-tools-and-best-practices`.
18. A. Ranta, R. Enache, and G. Détrez. Controlled language for everyday use: The MOLTO phrasebook. In *CNL 2012*, volume 7175 of *LNCS*, pages 115–136. Springer-Verlag, 2012.
19. H. Sharp, Y. Rogers, and J. Preece. *Interaction design: beyond human-computer interaction*. John Wiley, 2007.
20. J. D. Ullman. *Principles of Database and Knowledge-Base Systems*. CSPP, New York, 1988.
21. L. H. Vu, A. E. Haxthausen, and J. Peleska. A domain-specific language for railway interlocking systems. In *FORMS/FORMAT 2014*, pages 200–209. TU Braunschweig, 2014.
22. A. Z. Wyner, K. Angelov, G. Barzdins, D. Damljanovic, B. Davis, N. E. Fuchs, S. Höfler, K. Jones, K. Kaljurand, and T. Kuhn. On controlled natural languages: Properties and prospects. In *CNL 2009*, volume 5972 of *LNCS*, pages 281–289. Springer, 2009.