Preprint

RailCNL: A Controlled Natural Language for Railway Design Verification Specifications

Bjørnar Luteberget · John J. Camilleri · Christian Johansen · Gerardo Schneider

2019-05-07

Acknowledgements We would like to thank Martin Steffen and Aarne Ranta for numerous fruitful 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. This work was supported by the Norwegian Research Council under the grant *RailCons – Automated Methods and Tools for Ensuring Consistency of Railway Designs*.

Abstract Designs for 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 drawing tools of railway engineers. We would like railway engineers with limited logic programming experience to participate in the verification process, and we propose general principles of participatory verification which are exemplified throughout the paper. For our specific need for participatory verification, we introduce a controlled natural language (CNL), called 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 a design methodology based on CNL best practices and previous experience with creating verification front-end languages useful for designing similar CNLs.

B. Luteberget
RailComplete AS, Sandvika, Norway
E-mail: bjlut@railcomplete.no

J.J. Camilleri
Chalmers University of Technology and University of Gothenburg, Sweden
E-mail: john.j.camilleri@cse.gu.se

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

G. Schneider
Chalmers University of Technology and University of Gothenburg, Sweden
E-mail: gerardo@cse.gu.se

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

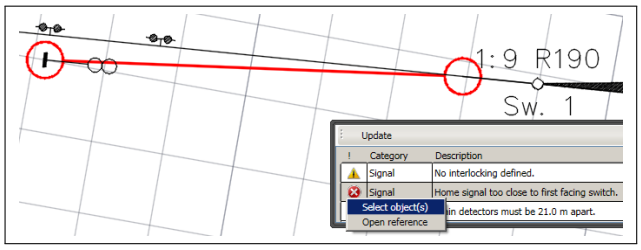


Fig. 1: CAD integrated verification engine, displaying errors and warnings after checking the model extracted from the CAD design against railway regulations on-the-fly.

and adjust them when needed.[1] 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.[2]

Involving the user in the design of a system is well-studied in the field of participatory design [49,22]. 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 or verification experience.

We have previously demonstrated [34,33] 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 our verification prototype with railway engineers from RailCOMPLETE AS[3], they raised yet another concern: how could they trace the violation, which the tool displays graphically, back to the regulations 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 [53] 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.

The contributions of this work are twofold. We propose participatory verification as an agenda for making verification methods more accepted by their intended users. Section 2 describes a general view on participatory verification, and the rest of the paper is concerned with applying this this view in a specific use case, namely the writing of verification properties by engineers, with the proposed solution of using controlled natural languages. The second contribution is our methodology for designing controlled natural languages for verification, described in Section 3, and the RailCNL made specific for railway regulations and specifications, which is described in detail in Section 4 and evaluated in Section 5. To fully evaluate the usefulness of using controlled natural languages for verification properties, we present two tools making use of RailCNL in Section 6. The concluding Section 7 contains some more related works and suggestions for possible continuations of this work.

This article is an extended and revised version of the paper presented at SEFM 2017 [32]. Besides adding considerably more examples and explanations to sections 3 and 4, this paper also includes the following new contributions: Section 2 about participatory verification is new, as well as the CNL editor survey in Section 6.2, and a specialized text editor tool was developed and described in Section 6.3. Moreover, this article has a new structure allowing the reader to follow the content from the most general ideas presented as participatory verification in Section 2 to more specific one in subsequent sections until reaching the tools Section 6.

2 Participatory Verification

We propose to adopt techniques from participatory design [49] into formal verification processes. We try to convey here how the formal verification process can be seen as a participatory design process, pointing out what stages and components from verification can be enhanced by participatory design ideas, so to make verification more user friendly, and to properly include the user in those verification tasks where their participation is needed.

Formal verification techniques are developed by theoretical computer science researchers based on mathematical models, like automata, and logical formalisms, like temporal logics, aiming to verify complex properties of complex systems, where human reasoning or testing techniques cannot encompass the large amount of details. The formal verification technique is implemented into a verification engine which takes as input:

– a *model* (i.e., the system with some details omitted) and

[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`

– a *property* to be verified;

and returns an answer:

– either *correct* when the model satisfies the property, or
– *error* when the model violates the property and
– an *explanation* of why the violation happens.

A plethora of verification engines exists, each for different kinds of systems (avionics, railways, software, microchip) and different kinds of properties of interest (deadlocks, safety, availability, correctness). Many of these have reached enough maturity to be usable in industry on more than proof of concept scenarios. A main impediment in the wider adoption of verification engines and techniques is the high level of specialized expertise that is required in order to:

– build models that the engine will accept and work nicely with,
– write properties,
– understand the output explanations of these engines,
– understand what the verification engine does (how the verification algorithm works) so to increase trust.

Usually those that can perform these tasks are the same experts that also built the respective verification engine, with knowledge of specific types of logics, specific kinds of mathematical objects representing the models, and deep understanding of the verification algorithms employed in order to be able to decipher the outputs of the verification engine and know how to use them to repair the system problem discovered by the verification process.

However, one would want the users of the verification engine to be the respective engineers that design and work with the respective real life complex systems like avionics designers, software engineers, railway engineers. These are persons with high level of skill in their own field, but not in the field of verification. We want persons that work with defining the specifications for the complex system and the various safety regulations and standards, to be the ones writing also the verification properties. We want those that build the complex systems to be able to run the verification algorithm, understanding what it is supposed to do for them, on the models provided by their peer engineers on the properties provided by regulators, and to understand the explanations of the verification tool when properties are not satisfied, i.e., to be able to use these explanations to debug their implementations and designs.

Like in any other product design process, the verification experts should go out of the loop as soon as they have finished designing and developing a specific verification engine (e.g., a verification tool built for some specific avionics system and specific kinds of safety properties as commissioned by some specific company). The verification engine is the product, and the users are the various field engineers. The users should be able to use a product in their daily tasks without any help from experts. However, since the final product is such a complex system, we need additional tools around the verification engine that would help the users participate in the verification process without needing interaction with verification experts.

Therefore, in participatory verification we define two phases and two main classes of software components.

Phase D: The design and development of the verification method and engine for the specific verification task (or area).

Phase V: The verification time when field engineers use the verification tools to debug and check their designs against specific regulations/properties.

Verification software components are seen as organized into:

Expert components include the verification algorithm and engine, various optimization modules, various translation tools between various specific mathematical structures as input to these modules and engine, including models like automata and property description languages like temporal logics.

User components include the UI tools, e.g., for displaying models, for writing properties, various modules for interacting with existing field tools and their UIs, various graphical/diagrammatic languages, editors for domain specific languages.

We observe that usually most of the resources are spent on expert components, whereas the user components are often disregarded in favor of hiring verification experts to use the expert verification components in doing the verification tasks during the verification phase. If some verification tool suite is becoming so popular as to attract companies, then investments in user components appear. The rest of our paper is *concerned with Phase V and User components*.

In participatory verification the concept of *"participation"* comes in two flavors:

(i) Users participate during the Phase D in usability studies [13], helping the verification experts to develop a tool best fit for the specific verification task and for the field in which the engineers are supposed to use the new tool.
(ii) Users actively participate in the verification process, during Phase V, to define the models and properties for the verification engine and to interact and understand the outputs from the engine.

Traditionally, participatory design (or interaction design) is concerned with Phase D of building a product until its delivery on the market, and less concerned with Phase V, when the product is used, unless subsequent versions of improved products are planned.

It is a particular challenge of participatory verification to achieve the second form of participation because of the complexity of the product. It is often the case with verification tools that the difficult learning curve required to use them is too big for the field engineers to overcome. Because of this, too many good verification algorithms and methods are not adopted.

Participatory verification aims to increase adoption of verification techniques, making two fundamental observations:

Sympathy for the verification tool: if the end user is involved in the development (specification, testing, etc) of a complex tool for them (thus prone to seeing the bugs along the way), then the user will be more aware of which features of the tool are difficult to implement, and thus buggy, and which work well.

Empathy for the intended user: if the developer works with the stated intention of making the tool *for the end user*, knowing the capabilities of the end user, how she normally will use the tool, spending enough time on tailoring the tool to the actual expressed user needs, then the user will require little learning and effort for using the new tool with her normal working methods, also making use of all the features of the tool.

In our previous work [33] we have had the role of the developers, building a verification engine for railway designs. We have tried to follow the *Empathy* guidelines, working closer with the engineers and integrating our engine into the engineers' design tools. We observed their working procedures and tools as well as interviewed representatives. However, we did not achieve the *Sympathy* goals, one of the major impediments being the opacity of the verification method, including the encoding of the regulations that our engine was working with. Moreover, it was clear in the end that the engineers would not be able to write or change any of the verification properties by themselves.

The remainder of the section presents our solution for allowing the user of a verification method to participate in the definition of the properties to be checked. The rest of the paper goes into further details, presenting first a methodology that we devised and followed for building such front-ends, then defining an actual constrained natural language that we called RailCNL, and finishing with the way we use RailCNL in practice both for reading and writing properties.

2.1 Approach to Participatory Verification for Railway Regulations

To promote participatory verification of infrastructure railway designs against regulations, we design a property specification language for expressing railway regulations and expert knowledge, integrating it with our previously developed verification engine. Fig. 2 presents the overall workflow of using the property language with special-made tooling, integrated with the engineer's CAD-based environment and our verification engine. Specifically, railway infrastructure static 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 [33] which allows efficient automatic reasoning. The reasoning happens continuously, in the background, while the user is working on the CAD drawing. Violations of regulations and best practices are presented to the user in the CAD program graphically, on the drawing.

We are not concerned with the model because in our case it is automatically generated from CAD drawings, which is already the tool of choice for engineers, thus they are actively involved in making the models while drawing in the CAD-based RailCOMPLETE framework.

Describing verification properties using logical rules in Datalog is not new (along with other logics like temporal [3] or dynamic logics [18,4]), 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, making 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.
- 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 regulations texts where commenting is needed since the expected natural semantics of some regulations needs confirmation in certain cases (e.g., "yes, this rule applies even when (...)").

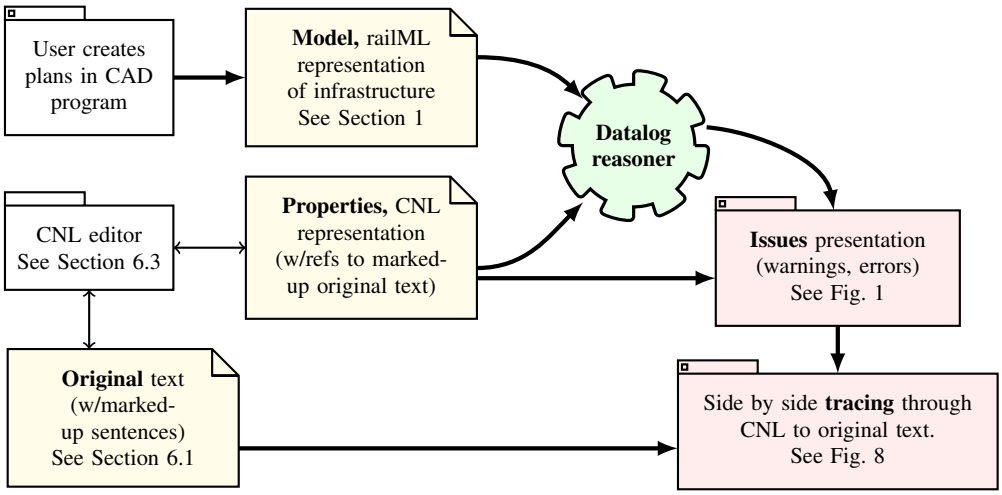


Fig. 2: Verification process overview. *Models* come directly from the CAD program, which engineers are already familiar with. *Properties* come from paraphrasing the regulations using CNL, which in turn are translated into Datalog. The reasoner outputs *issues* (warnings and errors) which are presented 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.

– 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 Design Methodology for a Verification Front-End Language

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. [59] 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 survey of CNLs, see Kuhn [27], whereas in Section 6.2 we survey CNL editors and their properties.

3.1 Using the Grammatical Framework to build CNLs

Grammatical Framework (GF) is a programming language for multilingual grammar applications [41]. 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, minimizing the effort needed to map an abstract syntax into another natural language, often reducing to simply providing the domain-specific vocabulary. This makes GF a valuable tool for building CNLs.

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:

```
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:

```
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␣a" ++ o2;
}
```

After both an abstract syntax and a corresponding concrete syntax has been defined, we can parse this language:

Text: *a switch must be more than 20 m from a signal*

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

We can also linearize the language from the 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 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 = mkA2 (mkA "more") (mkPrep "
      than");
  LessThan = mkA2 (mkA "less") (mkPrep "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.

3.2 Design methodology overview

Our methodology is based on CNL and GF best practices; in particular, Ranta et al. [46] 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. [45] 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 as well as on the works from [20,2,9] that created domain-specific CNLs as verification front-ends.

We present here the methodology that we apply in Section 4 to design RailCNL, a verification front-end language for describing rules for static railway infrastructure verification. This methodology combines concrete advises from the above works with our own experience from creating a railway infrastructure verification platform [34,33].

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 spe-

cific 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 the 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 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, or the need for structure on larger scales than a single sentence. Section 3.4 addresses each of these concerns.

Developing a specialized translation algorithm (see Section 4.2) instead of going through the GF typing system 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.

3.3 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. 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. Our railway verification method is used for *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 language.
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.

Our approach to handling this difficulty is to split the process of designing the abstract syntax into two parts.

(a) 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 (as noted in [45, 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:
```
main_signal(X) :- signal(X),
                  type(X,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.

(b) 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. This approach follows the notion of *language oriented programming* [57], 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. Sim-

ilarly, we use the CNL as a middle-ground between the original texts and the verification system.

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 6.1). The logic-driven design phase is exemplified for our RailCNL in Section 4.1.

3.4 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. [46, 45]). 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 keep ambiguity under control 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*. A straight-forward way to add such modifying adjectives as a prefix to class names would be to add a constructor, for example:

```
Adjective : String -> Class -> Class
```

This constructor 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.

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

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

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. 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 could 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.
Alternatively, we can increase the verbosity of the syntax, to reduce the likelihood of causing ambiguity when embedded in a longer statement. 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.

3.5 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 is 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 [20] 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 6).

3.6 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 [45]). 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 4.2 describes the main techniques that we used for RailCNL.

4 RailCNL: a Front-End Language for Railway Verification

With RailCNL we aim to cover the following content (see Section 5 on page 15 for a detailed account of the coverage that we achieve):

⟨*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 (GF notation shown in Appendix 8). 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.

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[4]) which give obligations or recommendations on the design of the railway infrastructure.

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

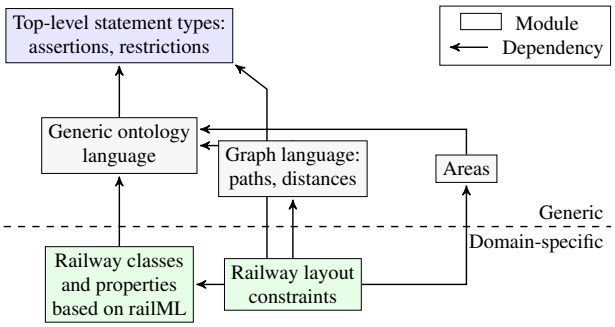


Fig. 4: Modules of the RailCNL (boxes) and their dependencies (arrows). The *generic* modules could be reused when building CNLs for verification in other domains. The *specific* modules are, however, tailored to railway regulations.

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 Appendix 8 at page 25 for an excerpt of the RailCNL grammar in GF notation), 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).

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 paper.

4.1 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 (as presented in Section 3). 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.

4.1.1 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.

Example 1 (Parse tree for a constraint statement)
CNL: *A signal which has type main is a main signal.*
AST:

```
OntologyAssertion
  (SubjectPropertyRestriction
    (StringClass "signal")
    (MkPropertyRestriction
      (StringProperty "type")
      (Eq (MkValue
            (StringTerm "main")))))
  (ConditionClass
    (StringClass "main_signal"))
```

– **Obligation**: design requirements on the railway infrastructure. The CAD model is checked for compliance, and violations are presented as errors to the user.

Example 2 (Parse tree for an obligation statement)
CNL: *A vertical segment must have length greater than 20.*
AST:

```
OntologyRestriction Obligation
  (SubjectClass (StringClassGen1
                  "vertical segment"))
  (ConditionPropertyRestriction
    (MkPropertyRestriction
      (StringProperty "length")
      (Gt (MkValue (StringTerm "20")))))
```

– **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 3 (Parse tree for a recommendation stmt.)
CNL: *A switch should be placed on a straight segment.*
AST:

```
PlacementRestriction Recommendation
  (SubjectClass (StringClass "switch"))
  (SingleArea
    (NoRestrictionArea
      (NonSpecificArea
        (MkNamedArea "straight segment"))))
```

4.1.2 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. As an example, we define "railway object":

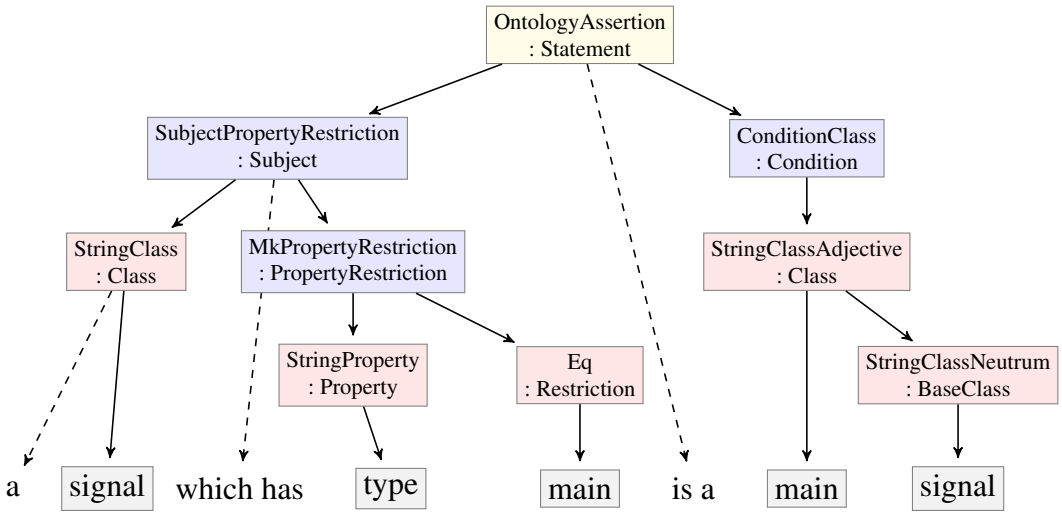


Fig. 5: The parse tree of the ontology assertion statement from Example 1.

Example 4 (Parse tree for using class names)
CNL: *A signal is a railway object.*
AST:

```
OntologyAssertion
  (SubjectClass
    (StringClassNoAdjective
      (StringClassNeutrum "signal")))
  (ConditionClassRestriction
    (MkClassRestriction
      (StringClassAdjective
        "railway"
        (StringClassNeutrum "object"))))
```

– **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 5 (Parse tree using properties and values)
CNL: *A project which is a new construction should have quality normal or high.*
AST:

```
OntologyRestriction Recommendation
  (SubjectCondition
    (StringClassNoAdjective
      (StringClassNeutrum "project"))
    (ConditionClassRestriction
      (MkClassRestriction
        (StringClassAdjective
          "new" (StringClassNeutrum
                  "construction")))))
  (ConditionPropertyRestriction
    (MkPropertyRestriction
      (StringProperty "quality")
      (OrRestr
       (Eq (MkValue (StringTerm "normal")))
       (Eq (MkValue (StringTerm "high"))))))
```

– **Restrictions**: Equality (shown as `Eq` in the AST example above) is a common case of restriction for which we simply choose the wording "to be". Other restriction types such as greater than (`Gt`), less than (`Lt`), etc. are worded more verbosely.

Example 6 (Parse tree using restrictions)
CNL: *A main signal should have height which is greater than 1.5m and less than 5.0m.*
AST:

```
OntologyRestriction Recommendation
  (SubjectClass
    (StringClassAdjective
      "main" (StringClassNeutrum "signal")))
  (ConditionPropertyRestriction
   (MkPropertyRestriction
     (StringProperty "height")
     (AndRestr
      (Gt (MkValue (StringTerm "1.5m")))
      (Lt (MkValue (StringTerm "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 7 (Parse tree using relations)
CNL: *A distant signal should have one or more associated signals.*
AST:

```
OntologyRestriction Obligation
  (SubjectClass (StringClassAdjective
      "distant"
      (StringClassNeutrum "signal")))
  (ConditionRelationRestriction
    ManyRelation (StringClassAdjective
      "associated"
      (StringClassMasculine "signals")))
```

4.1.3 Layout Module

For writing statements about the topology of the railway track, e.g. about paths as illustrated in Fig. 6c, 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. (See Fig. 6a)

Example 8 (Parse tree using path restriction)
CNL: *All paths from a station border to the first facing switch must pass an entry signal.*
AST:

```
AllPathsObligation
  (SubjectClass
    (StringClassAdjective "station"
      (StringClassMasculine "border")))
  (FirstFound FacingSwitch)
  (PathContains (AnyDirectionObject
    (AnySearchSubject (SubjectClass
      (StringClassAdjective "entry"
        (StringClassNeutrum "signal"))))))
```

- **Distance restrictions**: See also Fig. 6b.

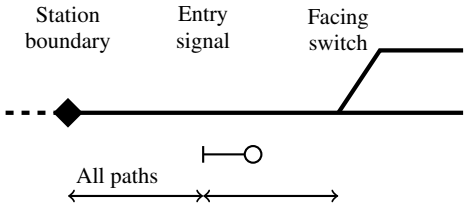
Example 9 (Parse tree using distance restriction)
CNL: *The distance from an entry signal to the first facing switch must be greater than 200m.*
AST:

```
DistanceObligation
  (SubjectClass
    (StringClassAdjective
      "entry" (StringClassNeutrum "signal")))
  (FirstFound FacingSwitch)
  (Gt (MkValue (StringTerm "200m")))
```

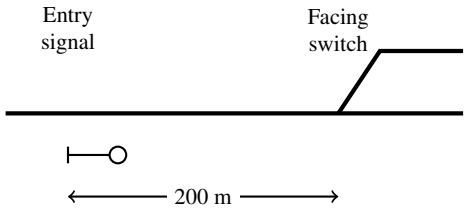
4.1.4 Area Module

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. 6d).

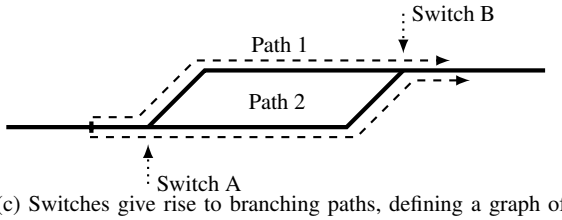
- **Subject constructor**: the `Subject` is extended to add a prepositional phrase containing area information, such as being inside of a tunnel or on a bridge.
- **Placement restriction**: extends the constructors for the type `OntologyRestriction` to allow restrictions on object being inside areas.



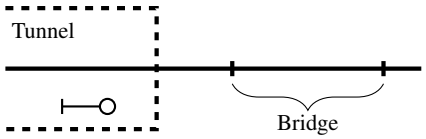
(a) Path restrictions are constructed from a subject, a goal, a quantifier and a condition.



(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. 6: Conditions on railway geographical layout as supported by RailCNL.

Example 10 (Parse tree using areas)
CNL: *A main signal should not be placed in tunnel or bridge.*
AST:

```
PlacementRestriction NegativeRecommendation
  (StringClassAdjective
    "main" (StringClassNeutrum "signal"))
  (MkArea
    (OrArea
      (SingleAreaConj (NoRestrictionArea
        (NonSpecificArea TunnelArea)))
      (SingleAreaConj (NoRestrictionArea
        (NonSpecificArea BridgeArea)))))
```

4.1.5 Signalling Layout Regulations

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

Example 11 (Heuristic for axle counter placement.)

CNL: *Running time at nominal speed from an axle counter to an adjacent axle counter must be less than 25s.*
AST:

```
RunningTimeObligation NominalSpeed
  (SubjectClass
    (StringClassAdjective
      "axle"
      (StringClassMasculine "counter")))
  (AnyFound (AnyDirectionObject
              SubjectOtherImplied))
  (Lt (MkValue (StringTerm "25s")))
```

4.2 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. 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.

Example 12 (Datalog translation of a subject constructor)

CNL: *A signal which has height 4.5m*
Datalog:

```
signal(X), height(X, 4.5).
```

The subject is the starting point for the translation, as other parts of the phrase refer back to the subject. In the following example, we first 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 to the subject ("X").

Example 13 (Datalog translation using implicit variable reference)

CNL: *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).
```

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.

Example 14 (Ontology assertion which would result in disjuntive head and is thus not expressible in Datalog)

CNL: *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 ontology restrictions.

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 12 below) and then declaring a rule which uses the negation of these restrictions (`!r1_found`) in order to yield a counter-example.

Example 15 (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 12), 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.

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.

Simplifications and optimizations. 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 $next(e_1, e_2, d)$. It could be possible to define general *connected* and *distance* predicates, as we have used in our previous work [34]. 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 design 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 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.

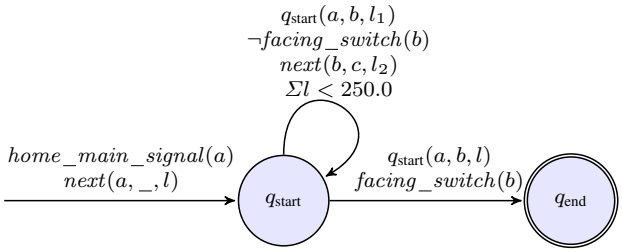


Fig. 7: Datalog rules used to execute the distance search from "home main signal" to "first facing switch".

To achieve a local search, we avoid the global *distance* and *connected* predicates, and use instead the underlying *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 *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.

Example 16 (Datalog distance search)

CNL: *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).
```

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.

Example 17 (Inlining)

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

Datalog:

```
sign_found(Subj0) :- sign(Subj0),
    tangent(Subj0, Val2), Val2 > 94.
sign_recommendation(Subj0) :-
    sign(Subj0), !sign2_found(Subj0).
```

After performing inlining simplification we get instead:

Datalog:

```
sign_recommendation(Subj0) :- sign(Subj0),
    tangent(Subj0, Val2), Val2 >= 94.
```

5 Evaluation of RailCNL Coverage of Norwegian regulations

Table 1 is based on an analysis of phrases from a selection of technical regulations from Norwegian infrastructure manager Bane Nor (`https://trv.jbv.no/`). Content from regulations concerning the engineering disciplines of railway tracks and signalling were selected for the evaluation because they were the focus of the RailCOMPLETE development, and as such was the domains for which the company had available expertise. (See Appendix 9 at page 26 for representative examples and Appendix 10 for a comprehensive overview of the Norwegian technical regulations that we worked with.)

Each sentence or table cell of the original text was classified according to the following:

1. **Sentence type**:classifying the sentence into *formalia* (headings, captions, etc), *meta* (describing the text), *applicability* (declaring scope, referring to other sections, etc.), *normative* (considered relevant for translation to RailCNL), or *other*.
2. **Discipline**: identifying whether phrases were belonging to another discipline than what the chapter heading had declared.
3. **Stage**: identifying whether the phrase was relevant for the planning stage of a railway construction project, excluding e.g. generic construction or operation statements.
4. **Static checkability**: for normative statements relevant for planning, there is also the possibility that they do not fit into the layout and specification part of the planning, thus not being suitable for static infrastructure verification. This is most notably the case for railway interlocking (control system) regulations, where statements about the dynamic behavior of the control system (concerning e.g. latency, timeouts, and state) are typically not part of the station-specific specification, and are not relevant for static infrastructure verification.

Table cells from the regulations were considered separate phrases, e.g. a number in a table cell was re-phrased as a self-contained CNL statement using information from the row and column headers. Phrases that were reasonably naturally expressible in RailCNL (either straight-forwardly in the basic logic, or after adding appropriate domain-specific constructs), were counted as *covered*. The results are detailed in Table 1.

6 Tool Integration

There are three ways of making use of a CNL for participatory verification.

Reading is the most simple to implement, but offers least benefits. For participatory verification purposes, only allowing an engineer to read in natural language the verification properties that the verifier works with is already valuable as it establishes trust in the opaque verification mechanisms. Moreover, using the CNL would shield the engineer from various logical formalisms that are used by complex processes like formal verification or certification. In Section 6.1 we show how using RailCNL only in a reading mode allows us to provide real help to the engineer in understanding the errors reported by our verification engine in a way that the engineer can make sense out of.

Template Editing, as a limited tool-support for editing phrases, would offer simple forms of editing, in addition to all the reading benefits. Changing a numerical value or changing words by selecting from a list of choices is easy to do and easy to understand for users, without requiring full understanding of the formal grammar behind the structure of the phrase. In the railway domain, the national regulations change seldom, and when they do, it is often enough to do only simple changes, e.g., when a new speed limit is imposed we only need to change a number. Besides being useful to regulators, template editing can also be used by engineers in companies to adjust regulations or properties that their specific designs

Engineering 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.

need (maybe only temporarily, like for debugging purposes).

Writing and editing phrases allows the CNL to be used at its full potential, but developing an editor which allows a user to edit phrases without having a thorough understanding of the formal CNL grammar is a challenge. Section 6.2 presents the many difficulties and features of CNL editors. Model-checking is not only useful for verifying compliance with regulations, but more importantly engineers would like to add 'rule-of-thumb' properties which they normally abide by when making new designs. These are usually considered valuable and proprietary for a company. The RailCNL editor that we present in Section 6.3 is meant to allow railway engineers to write properties in their natural domain language which can be verified by our engine, so that their know-how is kept in-house, without the need for an external expert (in verification).

6.1 Traceability Support in RailCNL

Verification tools 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. 8(top) 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. Fig. 8(middle) shows how the error message can be traced back through the Datalog code, the AST, and the CNL code, to the original, highlighted, regulations text Fig. 8(bottom).

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. 9). When the verification program finds a violation among the regulations, it outputs the identifier of the rule that has been violated, enabling the tracing.

When producing new regulatory texts or writing down expert knowledge for which a CNL exists, the approach of *embedded controlled language* [43] can be used to create natural texts where some sentences are directly parsable into verification properties. The other parts of such a text is then considered to be comments or explanations, similar to the programming approach known as *literate programming*.

6.2 An Overview of CNL Editors and their Features and Properties

While a CNL is usually designed to be easy to read without any prior training, the process of *writing* in a restricted language is less straightforward. In order to write in CNL, users need to be able to construct phrases which are correct with respect to the particular syntactic restrictions of that language, and which moreover have the intended meaning.

A CNL that presents itself as user-friendly may not expect that its users are willing to study its rules thoroughly before attempting to compose something in that language. This poses a challenge: the user wants to compose a phrase which is structurally correct, yet without having to know the rules governing that structure, or even seeing the underlying structure at all. This challenge is often aided by some software interface or tool designed specifically to help the user construct phrases which are valid in the particular CNL. We refer to such tools generally as *CNL editors*.

In this section, we give an overview of the state of the art in this area and compare the features of various CNL editors found in the literature. This section is useful to understand the choices that we made when developing the RailCNL editor presented in Section 6.3.

When we talk about CNL editors, we are implicitly assuming that the goal of the CNL in question is formalization, and thus that there exists some formal definition of the CNL itself, along with a parser. Therefore, we will not consider CNLs that have not been formalized or for which no parser exists.

There are two predominant paradigms to CNL editing:

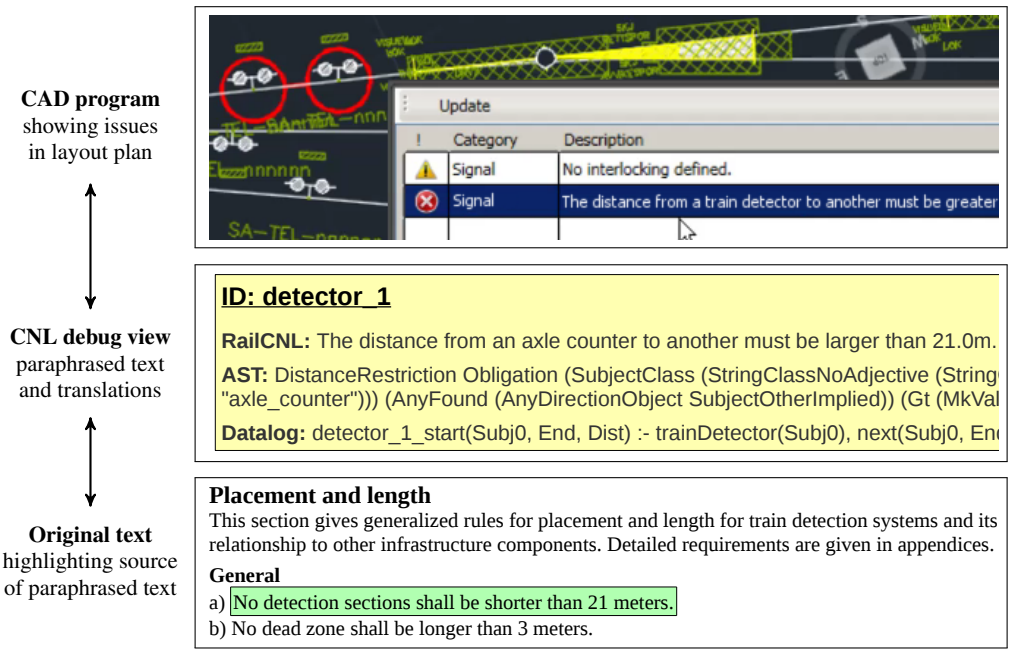


Fig. 8: Tracing of requirements backwards from CAD program through CNL to marked-up original texts. From a regulation violation presented as a warning or error, the user can browse to the corresponding regulatory text, shown side by side with the CNL text.

```
==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

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

CNL properties with references to labels

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

- **Structural editing**: where the user is building a formal representation (generally a tree) in a structural way, prevented from going outside the bounds of the CNL.
- **Surface editing**: where the user is inputting text, with varying degrees of guidance from the editor, which eventually needs to be checked for conformity.

These can be seen as the opposite ends of a spectrum where most CNL editors out there can be placed neatly at either end, while only a few fall in between.

This distinction goes back to [58], who outline the design issues that arise in the construction of language-based editors. Even though [58] treats primarily programming languages and not CNLs, many of the ideas discussed there are quite relevant to CNL editing:

> *"The assumption that all users are willing and able to think exclusively in tree terms is clearly false. In practice, many users have a pluralistic view of the programs they manipulate, seeing them sometimes as tree structures, sometimes as symbol sequences, and sometimes as character texts."*

The authors put forward the idea of *pluralistic editors*, which should support editing on these different levels of abstraction. These ideas are not surprising, and seem to serve as the basis for most of the CNL editors we have seen.

6.2.1 Structural Editing

The idea of structural editing for programming languages goes back at least to Mentor [12] and the Cornell Program Synthesizer [50]. These early editors were based on the philosophy that since programs are not text, it does not make sense to treat them as such. *"They are hierarchical compositions of computational structures and should should be edited, executed, and debugged in an environment that consistently acknowledges and reinforces this viewpoint."*[50]. These programming tools put the underlying structure of the

language very much in focus, minimizing the role of their text-based concrete syntax. This demands full understanding by the user of the language's grammar. While this idea did not really catch on for mainstream programming languages (which are still very much written and debugged using text-based formats) it still comes up in a number of CNL editors, in various guises. Scott and Power [39] introduce the idea of WYSIWYM editing (What You See Is What You Meant) for multilingual authoring, where users edit on the level of semantic representation, in their case a knowledge base, which is linearized into "feedback texts" in multiple languages for the user during the editing phase.

This idea is also mirrored in the syntax tree editing tools for CNLs defined using the Grammatical Framework [42] of which there are a few different implementations [23, 38, 7]. In these editors the user is directly building a tree in a top-down fashion by choosing the functions to be used at each node. The partial tree can be immediately linearized in multiple languages while the user works, including holes for yet unspecified nodes. Sub-trees can be inserted by parsing free text, but no guidance is provided to the user at this stage; the parse either succeeds or fails, and ambiguities must be fixed manually.

Ljunglöf [30] proposes the use of interactive tree building for dialogue management. The general idea is that a tree is built by asking questions to the user and then the tree is refined based on the responses obtained, filling in the missing holes until a complete tree is obtained. There is however no concrete tool associated with this work.

6.2.2 Structured Surface Editing

This class of editors comes somewhere in between the two major extremes described above. They are *structural* in the sense that you are directly building a tree and cannot stray outside the CNL, but they are also *surface-based* because the user is interacting with the editor on the surface string level, not necessarily seeing the underlying structure at all.

Menu-based input. Within this class, a number of editors tend to use similar user interface components, in particular the ideas of templates with holes which are filled by selecting completions from some kind of menu. An early example of this idea can be found in [51], which continued into various other works including NLMenu [52], ROSY [5], and MenuGen [17], which all follow more or less the same ideas. The authors of [55] discuss a general architecture of a menu-based input system for a multilingual translation systems, underlining the benefits of menu-based input for avoiding user input error.

The authors of [16] coined the term *Conceptual Authoring* for their input system for building database queries. Continuing this menu-based idea, their interface replaces the traditional query-writing as plain text with a set of templates with holes which are filled using menus. All editing operations are defined directly on an underlying logical representation, governed by a predefined ontology. The method avoids the problems associated with parsing, and is particularly well suited for query interfaces to closed-domain systems.

Similarly, the Phrasomatic editor[5] is a web-based interface for multilingual CNL phrases, powered by GF grammars. It has a distinctly menu-driven approach where the kinds of phrases one can write are pre-determined, and the user's task is to fill slots from the existing lexicon. The interface is hard-coded to be grammar-specific, in the sense that it is not generated by the grammar alone. This approach makes sense for small languages where the number of possible choices at each point is small, and there is no benefit to be had by allowing free-text input.

A slightly different approach within this class of editors is the MUSTE editor[6] [31], which is an experiment in keyboard-free structural editing on the surface level, where the tree is not revealed to the user. There is no text input at all, instead, the user edits an existing phrase by clicking words to show possible replacements. Clicking multiple times changes the scope currently under focus, extending from words to sub-phrases, and allowing potentially any sub-tree to be edited in accordance with the underlying grammar. It is an experiment in editing techniques and does not scale to larger CNLs.

Blocks and frames. Some editors are based on the concept of blocks which fit together and reveal the underlying structure of the text. The Blockly project[7] provides a UI library for building such kinds of interfaces. It is mainly promoted as an educational tool for teaching people to understand the structure behind programming languages. Color-coding and visual connectors are used to give some type information, for example distinguishing between statements and expressions. Alice[8] is a programming environment for creating animations and program simple games in 3D, which uses an editor based on this block-based interface.

In a similar fashion, the ATTAC-L editor [54, 11] uses "bricks" as the basic building blocks of its language, which can be pieced together and which reveal a lot of the structure behind the CNL. The representation which the user works with ends up as some kind of tree-like structure combined with snippets of text embedded in it.

The idea of *frame-based editing* introduced in [29] also falls within this class, and aims to combine the advantages

[5] `http://www.phrasomatic.net/` by Michal Boleslav Měchura in 2011.

[6] *"MUSTE: Multimodal semantic text editing"* by Ljunglöf, Peter. `https://heatherleaf.github.io/muste/`

[7] `https://developers.google.com/blockly/`

[8] "Alice — Tell stories. Build games. Learn to program." `http://www.alice.org/`

of block-based and text-based editing systems for programming languages. In essence, they propose a text-editing interface with additional visual markups to aid understanding, and template-and-menu editing of code to reduce syntax and type errors. The focus of this work is programming languages, and to our knowledge their approach has not been applied to CNLs.

6.2.3 Surface Editing

By surface editing, we mean that the user is composing input phrases rather than constructing them from menus. This allows more freedom of input and the user may construct incorrect phrases which will later get rejected by the CNL parser.

The most basic kind of surface editor can be seen as a typical parser which allows arbitrary input but returns an error when it is not syntactically correct. This is familiar to us from the world of programming languages, and any CNL which has a parser written for it can provide this functionality. The quality and helpfulness of the supplied error messages can of course vary greatly, which directly impacts the user experience.

To improve on this, most CNL editors provide some kinds of cues to guide the user in the direction of writing something correct. The standard paradigm here is that of sequential left-to-right textual input with suggested completions for every word or phrase. The completions themselves may often be categorized and/or sorted in some way.

One such example is the WebALT project's WExEd tool for designing multilingual mathematics exercises, which uses TextMathEditor for input of individual phrases [10]. The input language here is a multilingual CNL for mathematics, which converts phrases into objects in the OpenMath formalism.

Attempto Controlled English (ACE), one of the best known modern CNLs, also has a few different editors for it which all follow this paradigm [28,24,21]. They provide predictive text editing where completions are split into 'function word', 'proper name', 'verb', 'variables', 'nouns', etc. along with some pop-up warnings when the user enters something incorrect. The editor has undergone a usability study [25], and the project itself has even produced a grammar notation for CNLs focusing on predictive editors and anaphora [26].

The GF runtime also provides incremental parsing, which has been used to create various predictive editors. The standard example of this interface is the Minibar[9], which has been used in various other applications (for example [8,6, 9]).

[9] Minibar by Thomas Hallgren: `http://cloud.grammaticalframework.org/minibar/minibar.html`

Other CNL editors which also follow this paradigm include the ECOLE editor [47] for the PENG language, and its web-based successor for PENG^{ASP} [15]. These editors include the ability to add out-of-vocabulary (OOV) words, support for anaphora across different phrases, and completions sorted by category in drop-down menus. The Ask Data Anything editor [48] uses a similar approach for allowing users to write complex database queries in a custom CNL. The editor also includes some automatic correction of erroneous input using fuzzy matching, rather than strictly preventing incorrect user input.

The KANTOO controlled language checker [35] is an editor for the KANT Controlled English (KCE) which takes a *prescriptive* approach: rather than forcing user input to be correct with respect to the CNL, the user is allowed free text input and is then given diagnostic information suggesting what can be improved, such as missing constituents, punctuation, or incorrect coordination between phrases.

This approach is also taken in the MuTUAL editor [37], designed for assisting non-professional writers in creating Japanese texts that conform to a set of writing rules for enabling translation to English. The tool allows free text input, detecting problems in the source text in real-time and providing diagnostic messages for interactive rewriting. The editor uses highlighting to indicate rule violations and prescribed terms, provides suggestions for alternate expressions and shows the CNL rules to aid users in making their text conformant. The tool also comes with an extensive user evaluation [36].

6.2.4 Search-based Editing

In search-based editors the user inputs free text, but rather than being fed to a parser, this input is used to *search* for closely matching phrases within the CNL.

The work of [1] generates phrases from a CNL grammar, and combines this with a full text search engine. The advantage is that text-based search is well studied and one can use off-the-shelf search engines like Apache solar. However having to exhaustively generate phrases from a CNL can still be a bottleneck when the language is non-trivial in size, or even infinite as in our case.

In [44] it is presented a more advanced approach where instead of relying on exhaustive generation and a text search engine, phrases from both the input and the CNL are represented as vectors with infinite dimensionality. By considering only the non-zero elements, which are finite in number, closeness between input and valid CNL phrases can then be computed using cosine similarity. While this has been shown to work well on small CNL grammars, scalability is a problem because of the large number of comparisons which need to be made when searching for matches.

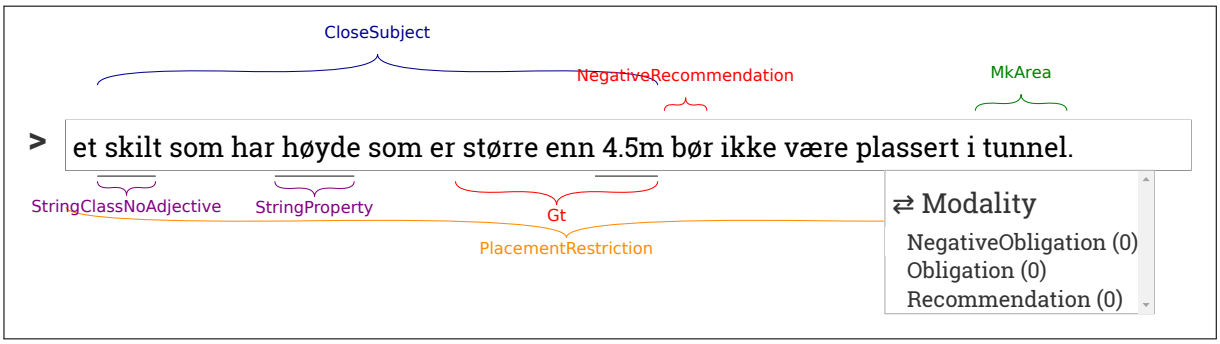


Fig. 10: Example from the RailCNL phrase editor demonstrating the menu for substituting constructors.

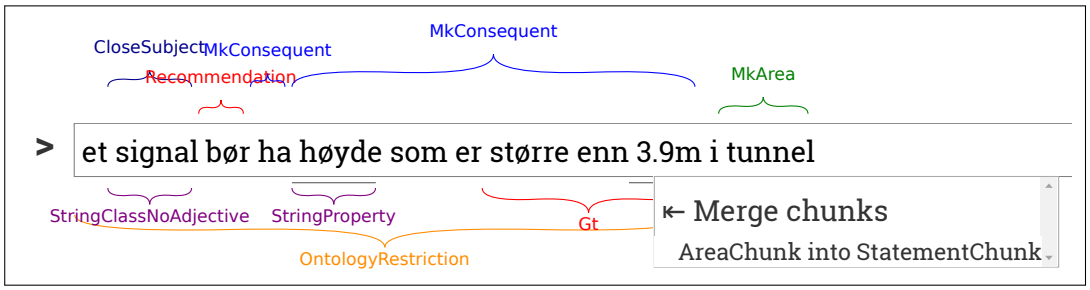


Fig. 11: Example from the RailCNL phrase editor demonstrating the menu for merging chunks.

6.3 An Editor for RailCNL

Taking clues from several of the approaches for building CNL editors described in Section 6.2, we developed an editor for Grammatical Framework languages with the specific use case of having railway regulations written by railway engineers using RailCNL. [10]. The figures in this section are actual screen-shots from the editor, and the input texts are therefore not translated into English.

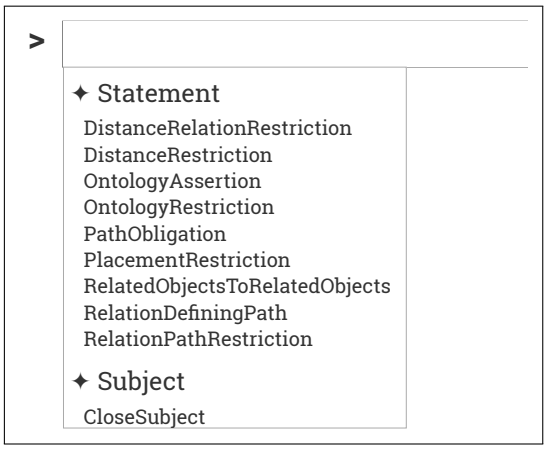
The RailCNL editor consists of a text input field containing the phrase which is being edited. There are no restrictions on the input to the text field, so the phrase can be empty, unparsable, partially parsable, or fully parsable. Whenever any change is made to the text, the parser will re-evaluate the text and update a drop-down menu and a partial parse tree visualization. The editor is thus mainly a surface editor (as described in Section 6.2.3), specifically an unrestricted text editor like mainstream programming language editors, but with two menu-based features: (1) a drop-down list giving menu choices relevant for the current text phrase and cursor position, and (2) a partial visualization of the parse tree where selected abstract syntax node types are drawn with a given a color above or below the text input field. The drop-down menu may be compared to the auto-complete feature of mainstream programming editors, while the partial parse tree visualization can be seen as a hybrid between a full parse tree visualization and the kind of syntax highlighting used in mainstream programming editors.

The interaction between the text input, drop-down menu, and tree visualization works as follows:

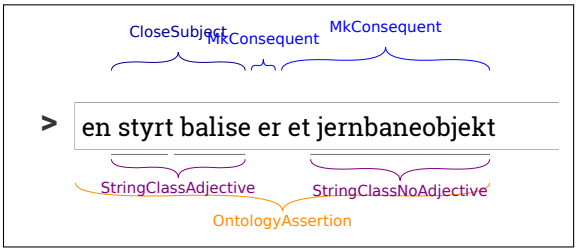
- Menu choices exist for any **concrete words** which are grammatically correct to insert at the current position. This works similarly to the auto-complete feature of mainstream programming editors, and the choices are supplied by the *predictive parsing* capability of GF.

- If the parsing has failed, and there is no top-level constructor of the parse tree, the menu will contain suggestions to **insert constructor**. When the text is empty, this part of the menu will suggest all top-level constructors, giving an overview of possible sentence structures. If the user chooses to insert constructors for which not all arguments are available in the current set of chunks, the smallest tree that has the required type is then constructed.

[10] RailCNL editor prototype demo: `https://luteberget.github.io/ControlText/`



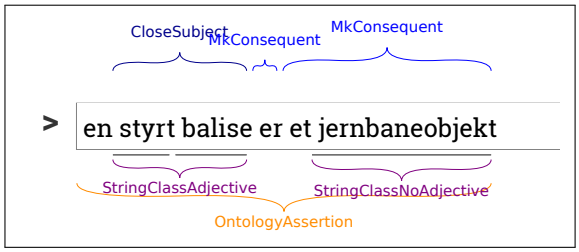
– When the parser has provided a full or partial parse tree, then a **partial parse tree visualization** is performed by consulting a list of selected types in the parse tree which are drawn as curly braces over the relevant part of the text using the *bracketed linearization* capability of GF. This allows the users to see while they are typing, what the parser is able to recognize.



– When the cursor is positioned inside a part of the text that has a valid parse tree, the menu shows suggestions for **substituting constructors**. All constructors which match the type of the tree nodes that cover the current cursor position have a corresponding list of alternative constructors, ranked by the number of terms that do not match in number and type. In the example below, the cursor is positioned on the word *"bør"* ("should" in English translation), which is tagged with the type *Modality*. Other modalities are suggested, giving an overview of possible expression types and thereby letting the user learn about the language as they are editing. See Fig. 10.

– If more than one chunk is recognized, clicking the top-level constructor in one chunk will search for nodes in the trees of the other chunks where a constructor substitution would allow the current chunk to be inserted, thereby **merging chunks**. In the example below, the user has a valid phrase of type OntologyRestriction, and has tried to add a condition to the end saying that the restriction applies only in tunnels. However, the grammar specifies that the area modifier applies to the subject part of the sentence, and must therefore be moved further ahead in the sentence to be valid. When the cursor is placed on the area chunk, the editor will add a menu choice that will merge the area chunk into the statement chunk, producing a fully parsable phrase. See Fig. 11.

– Dynamic vocabulary is used in the RailCNL system, for example, the user might define and name a new class, property or relation which is not part of the standard vocabulary. The grammar accepts this by allowing arbitrary strings in certain positions. Such unparsed words are highlighted in the editor by underlining. In the example below, "styrt balise" and "jernbaneobjekt" are part of the dynamic vocabulary.



The RailCNL editor works similarly to a programming source code editor. The design is based on the assumption that the user is a professional willing to learn about how the formal language and the verification system works. Even so, the editor's parse tree visualization and contextual menu-based operation encourages experimentation, exploration, and learning from examples.

The prototype RailCNL editor was implemented in JavaScript and HTML, which is suitable for demonstrations and entry into on-line databases, but a reimplementation using desktop application technologies should be considered to allow working with files on each engineer's local file system. The editor implementation uses the the grammar file produced by the Grammatical Framework for parsing and related functions through the Grammatical Framework runtime API. It also uses an editor-specific configuration file containing types and colors for the partial tree visualization, and for on-line documentation of constructors, which customizes the editor to the specific grammar.

7 Conclusions

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). 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.

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 Section 5). 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 implemented using the Grammatical Framework and its resource grammar library. 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. The CNL literature, and Grammatical Framework specifically, contains a lot of explicit and implicit knowledge about constructing languages with natural syntax, and this paper makes explicit some of this knowledge gathered from the CNL and GF community by describing details of the language design methodology in a general way.

A formal CNL with well-chosen linearizations can be very natural, and often perfectly readable for a non-programmer with the required domain knowledge; and we used this in Section 6.1 in our application to traceability of verification error messages. 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, which we reviewed in Section 6.2. We have been guided by these existing experiences when creating one such editor for RailCNL, presented in Section 6.3, which we plan to integrate in the RailCOMPLETE CAD environment, and carry out a usability study on its efficacy.

Focusing on *empathy* towards the intended user in the participatory verification process has shown us that integrated systems with familiar graphical user interfaces is an important requirement to engage non-programmer engineer users to adopt new tools into their daily routines. We hope that prioritizing according to the principles of participatory verification will in the long run help verification tools and techniques based on formal methods take hold in industrial practice, also outside the fields of software and electrical engineering.

7.1 Related Work

Johannisson [20] describes a CNL targeting the Object Constraint Language (OCL) for use in reasoning about Java program correctness in the KeY system [4]. 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. [2] present a conflict detection framework where GF is used to map the contract language $\mathcal{CL}$ [40] 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. [9] 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. [56], while also considering the verification to be an activity which is separate from design and implementation. James et al. [19] 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.

7.2 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.

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.

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 [14] 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 engineers, maybe a proprietary version of RailCNL would be used internally by a company, including specific proprietary knowledge of the domain and practice.

References

1. Agfjord, M., Angelov, K., Fredelius, P., Marinov, S.: Grammar-based suggestion engine with keyword search. In: Proceedings of The Fifth Swedish Language Technology Conference (SLTC 2014), Uppsala, Sweden (2014)
2. Angelov, K., Camilleri, J.J., Schneider, G.: A Framework for Conflict Analysis of Normative Texts Written in Controlled Natural Language. The Journal of Logic and Algebraic Programming **82**(5-7), 216–240 (2013). DOI 10.1016/j.jlap.2013.03.002
3. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press (2008)
4. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software: The KeY Approach. Springer (2007)
5. Blum, E.J.: ROSY – Menü-basiertes Parsing natürlicher Sprache unter besonderer Berücksichtigung des Deutschen. Diploma thesis, FB. Informatik, University of Saarland, Saarbrücken (1987)
6. Calafato, A., Colombo, C., Pace, G.J.: A controlled natural language for tax fraud detection. In: B. Davis, G.J. Pace, A.Z. Wyner (eds.) International Workshop on Controlled Natural Language (CNL 2016), *Lecture Notes in Computer Science*, vol. 9767, pp. 1–12. Springer (2016). DOI 10.1007/978-3-319-41498-0_1
7. Camilleri, J.J.: GF Syntax Editor. Online: `http://cloud.grammaticalframework.org/syntax-editor/editor.html` (2012). Accessed 2017-11-20
8. Camilleri, J.J., Pace, G.J., Rosner, M.: Controlled natural language in a game for legal assistance. In: M. Rosner, N.E. Fuchs (eds.) International Workshop on Controlled Natural Language (CNL 2010). Revised Papers, *Lecture Notes in Computer Science*, vol. 7175, pp. 137–153. Springer (2010). DOI 10.1007/978-3-642-31175-8_8
9. Camilleri, J.J., Paganelli, G., Schneider, G.: A CNL for contract-oriented diagrams. In: B. Davis, K. Kaljurand, T. Kuhn (eds.) International Workshop on Controlled Natural Language (CNL 2014), *Lecture Notes in Computer Science*, vol. 8625, pp. 135–146. Springer (2014). DOI 10.1007/978-3-319-10223-8_13
10. Cohen, A., Cuypers, H., Poels, K., Spanbroek, M., Verrijzer, R.: WExEd — WebALT exercise editor for multilingual mathematics exercises. In: M. Seppälä, S. Xambo, O. Caprotti (eds.) WebALT Conference and Exhibition (WebALT 2006), pp. 141–145 (2006). URL `http://www.win.tue.nl/~amc/pub/wexed.pdf`
11. De Troyer, O., Van Broeckhoven, F., Vlieghe, J.: Creating story-based serious games using a controlled natural language domain specific modeling language. In: M. Ma, A. Oikonomou (eds.) Serious Games and Edutainment Applications: Volume II, pp. 567–603. Springer International Publishing, Cham (2017). DOI 10.1007/978-3-319-51645-5_25
12. Donzeau-Gouge, V., Kahn, G., Lang, B., Mélèse, B.: Document structure and modularity in mentor. In: W.E. Riddle, P.B. Henderson (eds.) Software Engineering Symposium on Practical Software Development Environments, pp. 141–148. ACM (1984). DOI 10.1145/800020.808259
13. Dumas, J.S., Dumas, J.S., Redish, J.: A practical guide to usability testing. Intellect books (1999)
14. Fowler, M.: Domain-Specific Languages. Addison-Wesley Professional (2010)
15. Guy, S., Schwitter, R.: Architecture of a web-based predictive editor for controlled natural language processing. In: B. Davis, K. Kaljurand, T. Kuhn (eds.) International Workshop on Controlled Natural Language (CNL 2014), *Lecture Notes in Computer Science*, vol. 8625, pp. 167–178. Springer (2014). DOI 10.1007/978-3-319-10223-8_16
16. Hallett, C.: Generic querying of relational databases using natural language generation techniques. In: N. Colineau, C. Paris, S. Wan, R. Dale, A. Belz (eds.) INLG 2006 - Proceedings of the Fourth International Natural Language Generation Conference, pp. 95–102. The Association for Computer Linguistics (2006). URL `http://www.aclweb.org/anthology/W06-1414`
17. Hammerich, S.: Menübasierte generierung natürlicher sprache. Project thesis (studienarbeit), University of Hamburg (1999)
18. Harel, D., Tiuryn, J., Kozen, D.: Dynamic Logic. MIT Press (2000)
19. James, P., Roggenbach, M.: Encapsulating formal methods within domain specific languages: A solution for verifying railway scheme plans. Mathematics in Computer Science **8**(1), 11–38 (2014). DOI 10.1007/s11786-014-0174-0
20. Johannisson, K.: Natural language specifications. In: Beckert et al. [4], pp. 317–333. DOI 10.1007/978-3-540-69061-0_7
21. Kaljurand, K., Kuhn, T.: A multilingual semantic wiki based on attempto controlled english and grammatical framework. In: P. Cimiano, Ó. Corcho, V. Presutti, L. Hollink, S. Rudolph (eds.) International Conference on The Semantic Web: Semantics and Big Data (ESWC 2013), *Lecture Notes in Computer Science*, vol. 7882, pp. 427–441. Springer (2013). DOI 10.1007/978-3-642-38288-8_29
22. Kensing, F., Blomberg, J.: Participatory design: Issues and concerns. Computer Supported Cooperative Work (CSCW) **7**(3), 167–185 (1998). DOI 10.1023/A:1008689307411
23. Khegai, J., Nordström, B., Ranta, A.: Multilingual syntax editing in GF. In: A.F. Gelbukh (ed.) International Conference on Computational Linguistics and Intelligent Text Processing (CICLing 2003), *Lecture Notes in Computer Science*, vol. 2588, pp. 453–464. Springer (2003). DOI 10.1007/3-540-36456-0_48
24. Kuhn, T.: AceWiki: Collaborative ontology management in controlled natural language. In: C. Lange, S. Schaffert, H. Skaf-Molli, M. Völkel (eds.) Semantic Wiki Workshop (SemWiki 2008), *CEUR Workshop Proceedings*, vol. 360. CEUR-WS.org (2008). URL `http://ceur-ws.org/Vol-360/paper-5.pdf`
25. Kuhn, T.: How controlled english can improve semantic wikis. In: C. Lange, S. Schaffert, H. Skaf-Molli, M. Völkel (eds.) Semantic Wiki Workshop (SemWiki 2009), *CEUR Workshop Proceedings*, vol. 464. CEUR-WS.org (2009). URL `http://ceur-ws.org/Vol-464/paper-03.pdf`
26. Kuhn, T.: Codeco: A practical notation for controlled english grammars in predictive editors. In: M. Rosner, N.E. Fuchs (eds.) International Workshop on Controlled Natural Language (CNL 2010), Revised Papers, *Lecture Notes in Computer Science*, vol. 7175, pp. 95–114. Springer (2010). DOI 10.1007/978-3-642-31175-8_6
27. Kuhn, T.: A survey and classification of controlled natural languages. Computational Linguistics **40**(1), 121–170 (2014). DOI 10.1162/COLI_a_00168
28. Kuhn, T., Schwitter, R.: Writing support for controlled natural languages. In: Australasian Language Technology Association Workshop (ALTA 2008) (2008). URL `http://attempto.ifi.uzh.ch/site/pubs/papers/alta2008_kuhnschwitter.pdf`

29. Kölling, M., Brown, N., Altadmri, A.: Frame-based editing. Journal of Visual Languages and Sentient Systems **3** (2017). DOI 10.18293/VLSS2017-012. URL https://kclpure.kcl.ac.uk/portal/files/71018111/Frame_based_editing.pdf
30. Ljunglöf, P.: Dialogue management as interactive tree building. In: Workshop on the Semantics and Pragmatics of Dialogue (DiaHolmia 2009). Stockholm, Sweden (2009)
31. Ljunglöf, P.: Editing syntax trees on the surface. In: Nordic Conference of Computational Linguistics (NoDaLiDa 2011), pp. 138–145 (2011). URL http://aclweb.org/anthology/W/W11/W11-4619.pdf
32. Luteberget, B., Camilleri, J.J., Johansen, C., Schneider, G.: Participatory Verification of Railway Infrastructure by Representing Regulations in RailCNL. In: A. Cimatti, M. Sirjani (eds.) Software Engineering and Formal Methods, *Lecture Notes in Computer Science*, vol. 10469, pp. 87–103. Springer (2017)
33. Luteberget, B., Johansen, C.: Efficient verification of railway infrastructure designs against standard regulations. Formal Methods in System Design **52**(1), 1–32 (2018). DOI 10.1007/s10703-017-0281-z
34. Luteberget, B., Johansen, C., Steffen, M.: Rule-based consistency checking of railway infrastructure designs. In: E. Ábrahám, M. Huisman (eds.) 12th International Conference on Integrated Formal Methods, *Lecture Notes in Computer Science*, vol. 9681, pp. 491–507. Springer (2016). DOI 10.1007/978-3-319-33693-0_31
35. Mitamura, T., Baker, K., Nyberg, E., Svoboda, D.: Diagnostics for interactive controlled language checking. In: Controlled Language Application Workshop (CLAW 2003), pp. 237–244 (2003)
36. Miyata, R., Hartley, A., Kageura, K., Paris, C.: Evaluating the usability of a controlled language authoring assistant. The Prague Bulletin of Mathematical Linguistics **108**(1), 147–158 (2017). DOI 10.1515/pralin-2017-0016
37. Miyata, R., Hartley, A., Paris, C., Kageura, K.: Evaluating and implementing a controlled language checker. In: K.S. Choi, S. Nam (eds.) Controlled Language Applications Workshop (CLAW 2016), pp. 30–35. KAIST (2016)
38. Moreno, M.S.M., Bringert, B.: Interactive multilingual web applications with grammatical framework. In: B. Nordström, A. Ranta (eds.) International Conference in Advances in Natural Language Processing (GoTAL 2008), *Lecture Notes in Computer Science*, vol. 5221, pp. 336–347. Springer (2008). DOI 10.1007/978-3-540-85287-2_32
39. Power, R., Scott, D., Evans, R.: What you see is what you meant: direct knowledge editing with natural language feedback. In: European Conference on Artificial Intelligence (ECAI 1998), pp. 677–681 (1998)
40. Prisacariu, C., Schneider, G.: A Dynamic Deontic Logic for Complex Contracts. The Journal of Logic and Algebraic Programming (JLAP) **81**(4), 458–490 (2012). DOI 10.1016/j.jlap.2012.03.003
41. Ranta, A.: Grammatical Framework. Journal of Functional Programming **14**(2), 145–189 (2004). DOI 10.1017/S0956796803004738
42. Ranta, A.: Grammatical Framework: Programming with Multilingual Grammars. CSLI Publications, Stanford (2011)
43. Ranta, A.: Embedded controlled languages. In: B. Davis, K. Kaljurand, T. Kuhn (eds.) International Workshop on Controlled Natural Language (CNL 2014), *Lecture Notes in Computer Science*, vol. 8625, pp. 1–7. Springer-Verlag (2014). DOI 10.1007/978-3-319-10223-8_1
44. Ranta, A., Angelov, K., Höglind, R., Axelsson, C., Sandsjö, L.: A mobile language interpreter app for prehospital/emergency care. In: Medicinteknik dagarna. Västerås (2017). URL http://www.trippus.se/eventus/userfiles/86430.pdf
45. Ranta, A., Camilleri, J., Détrez, G., Enache, R., Hallgren, T.: Grammar tool manual and best practices. Tech. rep., MOLTO Deliverable D2.3, MOLTO Consortium, Göteborg (2012). http://www.molto-project.eu/biblio/deliverable/grammar-tools-and-best-practices
46. Ranta, A., Enache, R., Détrez, G.: Controlled language for everyday use: The MOLTO phrasebook. In: CNL 2012, *Lecture Notes in Computer Science*, vol. 7175, pp. 115–136. Springer-Verlag (2012). DOI 10.1007/978-3-642-31175-8_7
47. Schwitter, R., Ljungberg, A., Hood, D.: ECOLE — a look-ahead editor for a controlled language. In: Controlled Language Applications Workshop (CLAW 2003), pp. 141–150 (2003). URL http://web.science.mq.edu.au/~rolfs/papers/CLAW03-ECOLE.pdf
48. Seganti, A., Kaplanski, P., Campo, J.D.N., Cieslinski, K., Koziolkiewicz, J., Zarzycki, P.: Asking data in a controlled way with ask data anything NQL. In: B. Davis, G.J. Pace, A.Z. Wyner (eds.) International Workshop on Controlled Natural Language, (CNL 2016), *Lecture Notes in Computer Science*, vol. 9767, pp. 58–68. Springer (2016). DOI 10.1007/978-3-319-41498-0_6
49. Sharp, H., Rogers, Y., Preece, J.: Interaction design: beyond human-computer interaction. John Wiley (2007)
50. Teitelbaum, T., Reps, T.W.: The cornell program synthesizer: A syntax-directed programming environment. Communications of the ACM **24**(9), 563–573 (1981). DOI 10.1145/358746.358755
51. Tennant, H.R., Ross, K.M., Saenz, R.M., Thompson, C.W., Miller, J.R.: Menu-based natural language understanding. In: M.P. Marcus (ed.) Annual Meeting of the Association for Computational Linguistics (ACL 1983), pp. 151–158. ACL (1983). URL http://aclweb.org/anthology/P/P83/P83-1023.pdf
52. Thompson, C.W.: Using menu-based natural language understanding to avoid problems associated with traditional interfaces to databases. Ph.D. Dissertation, Deptartment of Computer Science, University of Texas, Austin (1989)
53. Ullman, J.D.: Principles of Database and Knowledge-Base Systems. W. H. Freeman & Co., New York (1983)
54. Van Broeckhoven, F., Vlieghe, J., De Troyer, O.: Using a controlled natural language for specifying the narratives of serious games. In: H. Schoenau-Fog, L.E. Bruni, S. Louchart, S. Baceviciute (eds.) International Conference on Interactive Digital Storytelling (ICIDS 2015), *Lecture Notes in Computer Science*, vol. 9445, pp. 142–153. Springer (2015). DOI 10.1007/978-3-319-27036-4_13
55. Vertan, C., von Hahn, W.: Menu choice translation — a flexible menu-based controlled natural language system. In: Controlled Language Application Workshop (CLAW 2003) (2003)
56. Vu, L.H., Haxthausen, A.E., Peleska, J.: A domain-specific language for railway interlocking systems. In: Proceedings of the 10th Symposium on Formal Methods for Automation and Safety in Railway and Automotive Systems, FORMS/FORMAT 2014, pp. 200–209. Technische Universität Braunschweig (2014)
57. Ward, M.P.: Language-oriented programming. Software - Concepts and Tools **15**(4), 147–161 (1994)
58. Welsh, J., Broom, B., Kiong, D.: A design rationale for a language-based editor. Software: Practice and Experience **21**(9), 923–948 (1991). DOI 10.1002/spe.4380210904
59. Wyner, A.Z., Angelov, K., Barzdins, G., Damljanovic, D., Davis, B., Fuchs, N.E., Höfler, S., Jones, K., Kaljurand, K., Kuhn, T.: On controlled natural languages: Properties and prospects. In: N.E. Fuchs (ed.) Controlled Natural Language, Workshop on Controlled Natural Language (CNL 2009), *Lecture Notes in Computer Science*, vol. 5972, pp. 281–289. Springer (2009). DOI 10.1007/978-3-642-14418-9_17

8 Appendix: Excerpts from RailCNL Grammar as written in GF

The complete set of files can be downloaded from `https://github.com/luteberget/RailCNL.git`.

```
-- Overall grammar, combining modules.
abstract RailCNL = Statement,
 Ontology, Graph, Area ** {}
```

```
-- Grammar for expressions about
-- railway infrastructure layout.
abstract Layout = Ontology ** {
 cat DirectionalObject; GoalObject;
     PathCondition; SearchSubject;
 fun
  -- Convert subjects from Ontology
  -- module into search goals.
  AnySearchSubject : Subject
   -> SearchSubject;
  SameDirObject, OppositeDirObject,
  AnyDirObject : SearchSubject
   -> DirectionalObject;
  -- Specify restrictions on distance
  -- between sets of objects.
  DistanceRestriction : Modality
   -> Subject -> GoalObject
   -> Restriction -> Statement;  }
```

```
abstract Area = Graph ** {
  cat BaseArea; NamedArea; SingleArea;
    AreaConj; Area;
 fun
  -- Arbitrary area type from string.
  MkNamedArea : String -> Area;
(...)
  -- Use area as a Subject modifier.
  SubjectArea : OpenSubject -> Area
   -> Subject;
  -- Statement about area containment.
  PlacementRestriction : Modality
   -> Subject -> AreaConj
   -> Statement;  }
```

```
-- Partial concrete grammar in Norwegian
-- for the Ontology module.
concrete OntologyNor of Ontology = open
 SyntaxNor, ParadigmsNor,
 (RailLex = RailCNLLexiconNor) in {
 lincat (...)
   Class = CN; Property = CN;
   Subject = CN; Statement = Utt;
   Modality = {vv:VV; typ:ModalityType};
(...)
 lin
(...)
  -- Modalities
  Obligation
   = {vv = RailLex.must_VV; typ = MPos};
  NegativeObligation
   = {vv = RailLex.shall_VV; typ = MNeg};
  -- Apply restriction to ontology.
  OntologyRestriction mod subj cond =
   mkUtt (mkS
    (case mod.typ of {
      MNeg => negativePol;
      MPos => positivePol })
    (mkCl (forall_CN subj)
          (mkVP mod.vv cond)));  }
```

```
abstract Ontology = Statement ** {
 -- Partial grammar in the Railway
 -- CNL for expressing classes and
 -- properties of classes.
 cat BaseClass; Class; Property;
  Value; ConsequentCondition;
  OpenSubject; Subject; Condition;
  Restriction; ClassRestriction;
  Modality; PropertyRestriction;
 fun
  -- Class name from string.
  StringClass
   : String -> BaseClass;
  -- Class prefix string.
  StringClassAdjective
   : String -> BaseClass -> Class;
  -- Class name without prefix.
  StringClassNoAdjective
   : BaseClass -> Class;
  -- Property name from string.
  StringProperty
   : String -> Property;
  Gt, Gte, Lt, Lte, Eq, Neq
   : Value -> Restriction;
  -- Combine restrictions by 'and'/'or'
  AndRestr, OrRestr
   : Restriction -> Restriction
     -> Restriction;
  -- Combine property restrictions
  AndPropRestr, OrPropRestr
   : PropertyRestriction
     -> PropertyRestriction
     -> PropertyRestriction;
  -- Subject from Class and Condition
  SubjectCondition
   : Class -> Condition -> OpenSubject;
  -- Use class/property as condition
  ConditionClassAndPropertyRestriction
   : Class -> PropertyRestriction
     -> Condition;
  ConditionRelationRestriction
   : RelationMultiplicity -> Class
     -> Condition;

  -- Modalities: must/should and neg'd
  Obligation, NegativeObligation,
  Recommendation, NegativeRecommendation
   : Modality;

  -- Assertion statement about ontology
  OntologyAssertion
   : Subject -> ConsequentCondition
     -> Statement;

  -- Restriction statement
  OntologyRestriction
   : Modality -> Subject
     -> ConsequentCondition -> Statement;
}
```

Fig. 12: Grammar excerpts from RailCNL implementation in Grammatical Framework.

9 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.

Original text	English translation	Comments
Source: **Overbygning**: 530 Prosjektering, Kap. 8 Helsveist spor, 2.1.		
De store krefter som kan forekomme i et helsveist spor stiller strenge krav til sporets konstruksjon.	The large forces that may occur in a welded track makes stringent demands on the track construction.	This sentence is not normative, and is unlikely to have any use in automated verification.
Source: **Overbygning**: 530 Prosjektering, Kap. 8 Helsveist spor, 2.1.3 a)		
Ballasten skal på linjen og i hovedspor på stasjoner være fullverdig grovpukk (av størrelse 31.5 – 63 mm)	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)	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.
Source: **Overbygning**: 530 Prosjektering, Kap. 8 Helsveist spor, 2.1.2 a)		
Minste kurveradius for helsveist med betongsviller skal være 250 m.	The lowest allowable radius of curvature for whole welded track on concrete sleepers is 250 m.	This is a typical example of static infrastructure verification, expressible in Datalog as: error(Segment) :- trackSegment(Segment), trackSegmentRadius(Segment, Radius), Radius ¡ 250.
Source: **Signal**: 550 Prosjektering, Kap. 6 Lyssignal, 2.1.2 j)		
Et innkjørhovedsignal skal plasseres $\geq$ 200 meter foran innkjørtogveiens første sentralstilte, motrettede sporveksel, se Figur 5.	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).	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 ¡ 200.
Source: **Signal**: 550 Signal, Kap. 5 Forriglingsutrustning, 2.8.1 Dekningsgivende objekt		
Følgende objekt kan være dekningsgivende: Hovedsignal, Dvergsignal, Sporveksel, Sporsperre, Avsporingstunge, Signal E35 Stoppskilt. Et hovedsignal skal vise signal "Stopp" for å være dekningsgivende.	The following objects can provide flank protection: main signal, shunting signal, switch, derailer, derailing tongue, signal E35 stop sign. A main signal must display "stop" to provide flank protection.	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**).

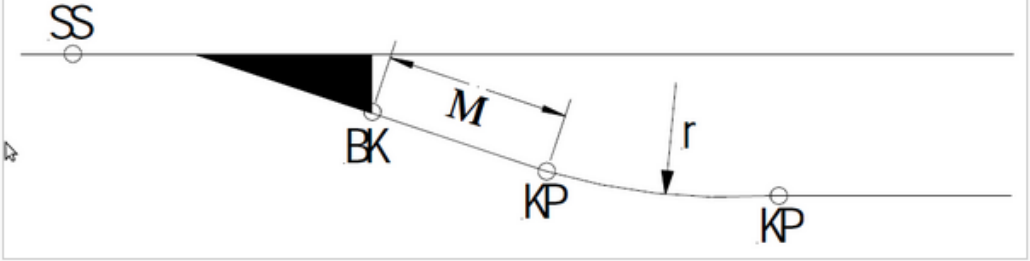
Original text	English translation	Comments
Source: **Signal**: 550 Prosjektering, Kap. 6 Lyssignal, 2.1.2 i)		
Et hovedsignal bør ikke plasseres i tunneler, på bruer, eller andre steder hvor en eventuell togstans og dermed muligheten for avstigning, vil medføre fare.	A main signal should not be placed in tunnels, on bridges, or other places where halting trains and thus the possibility of disembarking, can impose dangers.	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.
Source: **Signal**: 550 Prosjektering, Kap. 5 Forriglingsutrustning, 4.1.1.1 i)		
For at en togvei skal kunne fastlegges, skal et objekt som gir dekning til togveien være dekningsgivende.	For a train route to be deactivated, any object giving flank protection must be in a protecting state.	This regulation concerns only the state of the control system, and as such relates to the implementation of the control system and not the static infrastructure specification.
Source: **Overbygning**: 530 Prosjektering, Kap. 5 Sporets trasé, 3.1 Dimensjonerende parametre		
See table below.	(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

	Symbol	Definisjon	Normale krav	Minste krav
a)	$R_{\min}$	minste radius	250 m	190 m
b)	h_{maks}	maksimal verdi for overhøyden [1)]	150 mm	
c)	h_{avsp}	grense for overhøyde pga. avsporingsfaren ved lave hastigheter	$\frac{R-100}{2}$ mm	
d)	$\left(\frac{\Delta h}{L}\right)_{maks}$	grenseverdi for rampestigning	1:400	
e)	I_{maks}	grenseverdi for manglende overhøyde [2)]	R ≤ 600: 100 mm R > 600: 130 mm	

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

Original text	English translation	Comments
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.



Figur 4: Krav til rettlinje M ved linjedelende sporveksel

Source: **Overbygning**: 530 Prosjektering, Kap. 5 Sporets trasé, 5 Største hastighet – sporets geometri

Hastigheten i en kurve skal ikke være større enn: $V = 0,291 \cdot \sqrt{R\,(h + I_{\text{maks}})}$ (5) Hvis ligning 5 i tilfeller med falsk overhøyde gir lavere verdi enn 20 km/h gjelder $V = 20$ km/h.	The speed in a curve shall not exceed: $V = 0,291 \cdot \sqrt{R\,(h + I_{\text{maks}})}$ (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.

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

Dersom lyssignal er vridd eller på annen måte kommet ut av stilling skal dette utbedres snarest.	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.

10 Appendix: Overview of Norwegian Regulation Contents

The technical regulations (*"Teknisk regelverk"*) can be found at `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.