Verification of Railway Network Models with EVEREST

João Martins
joao.martins@efacec.com
EFACEC
Porto, Portugal

José M. Fonseca
jose.fonseca@efacec.com
EFACEC
Porto, Portugal

Rafael Costa
INESC TEC
Braga, Portugal
rafael.b.costa@inesctec.pt

José C. Campos
jose.campos@di.uminho.pt
INESC TEC & U.Minho
Braga, Portugal

Alcino Cunha
alcino@di.uminho.pt
INESC TEC & U.Minho
Braga, Portugal

Nuno Macedo
INESC TEC & U.Porto
Porto, Portugal
nmacedo@fe.up.pt

José N. Oliveira
INESC TEC & U.Minho
Braga, Portugal
jno@di.uminho.pt

ABSTRACT

Models – at different levels of abstraction and pertaining to different engineering views – are central in the design of railway networks, in particular signalling systems. The design of such systems must follow numerous strict rules, which may vary from project to project and require information from different views. This renders manual verification of railway networks costly and error-prone.

This paper presents EVEREST, a tool for automating the verification of railway network models that preserves the loosely coupled nature of the design process. To achieve this goal, EVEREST first combines two different views of a railway network model – the topology provided in *signalling diagrams* containing the functional infrastructure, and the precise coordinates of the elements provided in *technical drawings* (CAD) – in a unified model stored in the railML standard format. This railML model is then verified against a set of user-defined infrastructure rules, written in a custom modal logic that simplifies the specification of spatial constraints in the network. The violated rules can be visualized both in the signalling diagrams and technical drawings, where the element(s) responsible for the violation are highlighted.

EVEREST is integrated in a long-term effort of EFACEC to implement industry-strong tools to automate and formally verify the design of railway solutions.

CCS CONCEPTS

• **Applied computing** → **Computer-aided design**; • **Computing methodologies** → **Model verification and validation**; • **Software and its engineering** → **Specification languages**.

KEYWORDS

railway engineering, railway network model verification, formal infrastructure rule specification, railML

ACM Reference Format:
João Martins, José M. Fonseca, Rafael Costa, José C. Campos, Alcino Cunha, Nuno Macedo, and José N. Oliveira. 2022. Verification of Railway Network Models with EVEREST. In *ACM/IEEE 25th International Conference on Model Driven Engineering Languages and Systems (MODELS '22), October 23–28, 2022, Montreal, QC, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3550355.3552439

MODELS '22, October 23–28, 2022, Montreal, QC, Canada

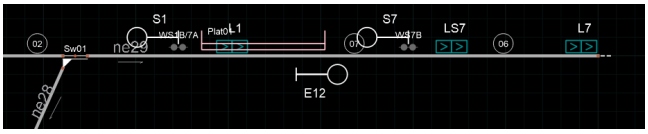
ACM ISBN 978-1-4503-9466-6/22/10…$15.00
https://doi.org/10.1145/3550355.3552439

1 INTRODUCTION

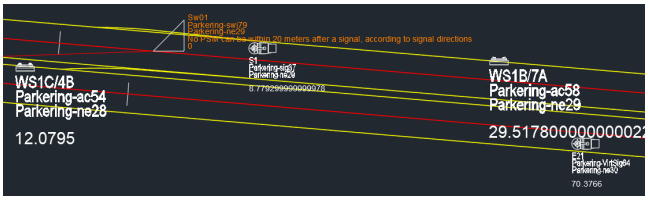
The design of railway signalling systems involves various teams, with different expertise, working on different views of the system design. One of the challenges of this activity is how information is exchanged among the different teams and kept consistent in the different views, typically a manual, and consequently, error-prone process. Despite this challenge, the end goal is to produce a design which meets the rules imposed by regulatory entities and additional end-user specific requirements. This requires the design to be verified, but for infrastructural requirements this is again a task that is typically done manually, since most design tools do not properly support the verification of such requirements.

This paper presents EVEREST (*Efacec Verification of Railway nEtworkS Tool*), a toolset developed by EFACEC, a leading Portuguese company in the area of railway signalling systems, in partnership with academia. EVEREST provides mechanisms for synchronizing and merging two different views of a railway network model: (a) the *topology* provided in *signalling diagrams* describing the network's functional infrastructure (developed in tools such as RaIL-AiD[1]); and (b) the precise coordinates of the elements of the physical system as captured by standard *technical drawings* (drawn in AutoCAD[2]). The unified model resulting from this process, stored in the railML® standard format[3], can then be automatically verified by EVEREST against a set of user-defined infrastructure rules. A

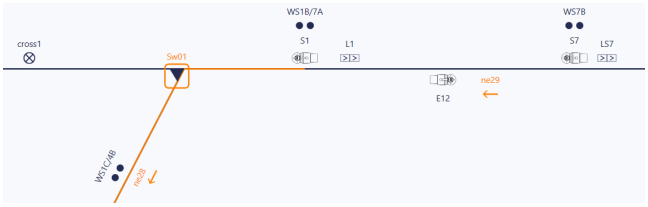
[1]https://www.rail-aid.com/, last visited May 12, 2022.
[2]https://www.autodesk.com/products/autocad/, last visited May 12, 2022.
[3]railML (*Railway Markup Language*) [17] is an open, XML-based format, that has evolved to a *de facto* standard for data exchange of railway network models.



(a) Signalling diagram as depicted by RaIL-AiD



(b) AutoCAD technical drawing



(c) Signalling diagram as depicted by EVEREST

Figure 1: Different views of the example area in the EVEREST ecosystem after positioning and verification

key requirement in the development of EVEREST was to keep the *loosely coupled* approach of the signalling design process, namely allowing the different teams to still interact with the design tools that better suit their concerns.

The main contributions of EVEREST are the following: **a loosely coupled and light-weight railway network model verification workflow**, that blends the activity of signalling engineers and technical designers in an automated and non-intrusive way; **a formal, high-level language for expressing railway infrastructure rules** on top of railML models, that simplifies the specification of spatial constraints in the network; **an implementation of the proposed methodology** in a toolset with components that 1) enable RaIL-AiD and AutoCAD to inter-operate via railML models; 2) the subsequent automatic verification of conformance against a catalogue of user-specified infrastructure rules; 3) reporting about violated rules in the different views of the network model.

Paper structure. Section 2 presents in more detail the proposed EVEREST workflow and toolset. Section 3 presents the language proposed to defined infrastructure rules, while Section 4 details the implementation of the various components of the toolset. Section 5 presents the evaluation of the verification capabilities of EVEREST. Lastly, Section 6 compares the approach with related work and Section 7 draws conclusions and points directions for future work.

2 EVEREST TOOLSET

The design of a railway signalling system usually involves two teams, one of signalling engineers responsible for the signalling system design that interacts with a railway design tool – in the case of EFACEC, RaIL-AiD – and another of technical designers

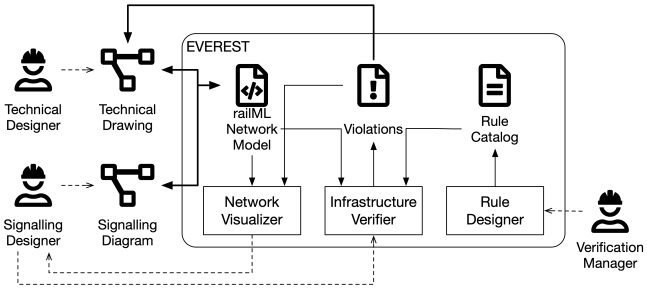


Figure 2: Overview of the EVEREST workflow

responsible for integrating that design in the complete infrastructure that interacts with a technical drawing tool – in the case of EFACEC, AutoCAD. Figure 1 shows these views for (part of) a real (anonymized) area that will be used as a running example, namely as a signalling diagram during railway design in Fig. 1a and as a technical drawing in Fig. 1b (information introduced by EVEREST is also shown, which is explained later). During development, the two teams interact with each other and eventually agree on a consistent view of the network model. At this point, in some cases aided by a *verification manager* with expertise on the verification of infrastructure rules, they manually assess whether the resulting design conformed to the regulations imposed for that project.

Figure 2 provides an abstract overview of how EVEREST can integrate this workflow. The two teams initially interact with the usual tools, preserving the loosely coupled nature of the two engineering tasks. However, EVEREST then merges the information contained in both those models in a single network model represented in railML. Many railway design tools such as RaIL-AiD allow the exporting of a network topology and the respective signalling diagram to railML, but these models are incomplete, lacking the precise location of infrastructure elements (namely signals), or the precise length of track segments. This exportation into railML is the first step towards merging the network information. Then, the elements in this railML are automatically imported into the technical drawing to bootstrap the positioning process. Once the technical designers position the elements, their precise coordinates are propagated back into the railML model. This phase is supported by the EVEREST AutoCAD plugin, that, among others, provides methods to check that all elements are already positioned and have their precise location calculated. The drawing in Fig. 1b already shows (in white) the elements positioned along the track. This approach solves the consistency problem as the two views are not concurrently updated, and technical designers are not expected to change the signalling diagram nor signal engineers element locations.

After all precise locations and measurements are incorporated in the railML model, the EVEREST toolset can be used for verifying its conformance with required infrastructure rules. These rules are written in a DSL supported by the EVEREST *Rule Designer* component by a team aided by the verification manager, which builds a catalog of rules transversal to the various projects of the company. The language was designed to be simple enough so that only basic knowledge of logic and formal specification is required by members of this team. For instance, consider the rule stating

```
rule      := scope :: fol
scope     := route | track
fol       := multOp expr | expr exprComp expr
           | expr numComp expr | spatial | expr
           | ( fol ) | qtOp (var : expr),+ | fol
           | fol binFormOp fol | not fol
spatial   := everywhere [ range ] fol
           | nowhere [ range ] fol
           | somewhere [ range ] fol
           | fol until [ range ] fol
qtOp      := all | some
binFormOp := and | or | implies | iff
multOp    := one | lone | some | no
expComp   := in | =
numComp   := < | > | ≤ | ≥
expr      := var | railML | const | expr binNumOp expr
           | unExpOp expr | ( expr ) | expr binExpOp expr
binExpOp  := . | | | & | → | \
binNumOp  := + | − | * | /
unExpOp   := ~ | ^ | #
range     := ( [ | ] ) [ expr ] .. [ expr ] ( ] | [ )
railML    := id
var       := id
const     := number | string | true | false
```

Figure 3: Concrete syntax of the EVEREST rule language.

that no railway switches appear in the 20 meters following a signal. In EVEREST, this would be written as:

```
route :: everywhere
  (some signalIS implies everywhere [0..20] no switchIS)
```

The rule language draws inspiration both from metric interval linear temporal logic [3], with temporal modalities and time intervals adapted to the spatial context, and the (first-order) relational logic of Alloy [10], to simplify the navigation along the attributes and children of the numerous elements in the railML schema. In the rule above, the first identifier determines the *scope* of the rule. Here, the rule will be evaluated for all the routes defined in the signalling diagram, with attributes of routes implicitly projected for each route being evaluated. A *route* is a unidirectional path for train movements in the railway network, between an entry and an exit signal, traversing switches that should be locked in a predefined position by the interlocking system responsible for preventing conflicting train movements. Another possible scope is a *track* segment between topological elements (for example, switches): if movement is allowed in both directions two independent unidirectional track segments are considered. The fact that all scope elements are unidirectional is what allows us to use a linear model of space. For example, the modal operator **everywhere** in the above rule quantifies on all positions along each route, possible restricted by a range interval. The identifiers that occur in the rules are those of the railML model. The rule refers to `signalIS` and `switchIS`, that in railML contain the physical characteristics of signals and switches (the infrastructure level). If information about their function and usage was required (the interlocking level), `signalIL` and `switchIL` should be used instead. Elements such as these, which now have a precise location extracted from the technical drawing, are implicitly projected for each location. Thus, the rule simply states that in every location of a route where there is some signal, in every other location in the succeeding 20 meters there are no switches.

The EVEREST Rule Designer provides some support to write and maintain this catalog of rules. A flexible type-checker is implemented to detect badly written rules. Some basic versioning functionalities are provided. Moreover, EVEREST supports the definition of parameterized rules that can be instantiated for different projects, and expression macros to simplify the writing of frequently occurring expressions and tame the verbosity of railML, which frequently requires a long navigation chain along attributes and children of different elements to fetch the required information.

The interaction of the signalling designer with the EVEREST *Infrastructure Verifier* is essentially selecting the rules relevant for the current project from the catalog. Once the rules are selected, the Verifier automatically evaluates them for the provided network model, reporting any element of the scope that has violated the rule. These are presented back to the different parties in different perspectives: not only are they presented in the signalling diagram of the EVEREST *Visualizer*, but are also integrated back into the technical drawing. For our running example, violations were found for the rule defined above: in one of the routes, switch `Sw01` is too close to signal `S1`. Figure 1c shows this violation reported in the Visualizer (highlighting the violating route and `Sw01`), and Fig. 1b its visualization in the technical drawing (`Sw01` was labelled with the description of the violated rule and painted in orange).

3 RULE LANGUAGE

As mentioned above, the EVEREST rule language draws inspiration both from metric interval linear temporal logic [3] and Alloy [10]. To promote flexibility, it shares the Alloy motto that "everything is a relation". In particular, the railML network models over which rules are evaluated will be represented by sets of relations, capturing the different XML tags and attributes.

3.1 Syntax and Semantics

The full syntax of the EVEREST rule language is provided in Fig. 3. The semantics of a formula will be defined over an abstract network model M, a map that assigns to each railML tag or attribute i a *tuple set* whose atoms are either constants or railML element ids. For instance, $M($`signalIL`$)$ could be $\{\langle$`SL1`$\rangle, \langle$`SL2`$\rangle\}$, while $M($`isVirtual`$)$ (`isVirtual` being one of the possible attributes of element `signalIL`) could be $\{\langle$`SL1`, **true**$\rangle, \langle$`SL2`, **false**$\rangle\}$.

As the location of the elements along the track is essential to the evaluation of rules, we extract auxiliary information from railML to support the definition of the semantics. Localized elements are those containing a `spotLocation` children element in the railML schema, such as `switchIS` or `signalIS`. A spot location is basically a position along a track segment (denoted *net element* in railML) with a direction (normal, reverse, or both, in relation to the origin of the net element). This is the railML element whose information is enriched with a precise location after the positioning is done in the technical drawing. A localized element may have multiple spot locations assigned, such as switches which are at the intersection of 3 net elements. We assume the existence of the following functions, that can be derived *a priori* from a railML file: spots(e), the locations of element e; entry(s), the locations where a scope s

$$
\begin{array}{lcl}
{}_M[\![a \mid b]\!]^l_s & \triangleq & {}_M[\![a]\!]^l_s \cup {}_M[\![b]\!]^l_s \\
{}_M[\![a \,\&\, b]\!]^l_s & \triangleq & {}_M[\![a]\!]^l_s \cap {}_M[\![b]\!]^l_s \\
{}_M[\![a \setminus b]\!]^l_s & \triangleq & {}_M[\![a]\!]^l_s \setminus {}_M[\![b]\!]^l_s \\
{}_M[\![a \,.\, b]\!]^l_s & \triangleq & {}_M[\![a]\!]^l_s \bullet {}_M[\![b]\!]^l_s \\
{}_M[\![a \rightarrow b]\!]^l_s & \triangleq & {}_M[\![a]\!]^l_s \times {}_M[\![b]\!]^l_s \\
{}_M[\![\sim b]\!]^l_s & \triangleq & {}_M[\![a]\!]^{l\circ}_s \\
{}_M[\![\hat{} \, b]\!]^l_s & \triangleq & {}_M[\![a]\!]^{l+}_s \\
{}_M[\![\# \, b]\!]^l_s & \triangleq & \{\langle |{}_M[\![a]\!]^l_s| \rangle\}
\end{array}
$$

$$
\begin{array}{lcl}
{}_M[\![a + \; b]\!]^l_s & \triangleq & \{\langle n+m \rangle\} \text{ s.t. } {}_M[\![a]\!]^l_s = \{\langle n \rangle\} \wedge {}_M[\![b]\!]^l_s = \{\langle m \rangle\} \\
{}_M[\![a * \; b]\!]^l_s & \triangleq & \{\langle n \times m \rangle\} \text{ s.t. } {}_M[\![a]\!]^l_s = \{\langle n \rangle\} \wedge {}_M[\![b]\!]^l_s = \{\langle m \rangle\} \\
{}_M[\![a \,/\, b]\!]^l_s & \triangleq & \{\langle n \div m \rangle\} \text{ s.t. } {}_M[\![a]\!]^l_s = \{\langle n \rangle\} \wedge {}_M[\![b]\!]^l_s = \{\langle m \rangle\}
\end{array}
$$

$$
\begin{array}{lcl}
{}_M[\![i]\!]^l_s & \triangleq & \{t \mid t \in M(i)\!\downarrow\! s \wedge \\
 & & \quad (t = \langle x \rangle \wedge x \in dom(\text{spots})) \Rightarrow l \in \text{spots}(x)\} \\
{}_M[\![v]\!]^l_s & \triangleq & M(v) \\
{}_M[\![n]\!]^l_s & \triangleq & \{\langle n \rangle\} \\
{}_M[\![u]\!]^l_s & \triangleq & \{\langle u \rangle\} \\
{}_M[\![\texttt{true}]\!]^l_s & \triangleq & \{\langle \texttt{true} \rangle\}
\end{array}
$$

Figure 4: Semantics of the rule language expressions.

starts; $\text{dist}(s, l_1, l_2)$, the distance between locations l_1 and l_2 along scope s; and $\text{between}(s, l_1, l_2, l)$, determining whether location l is between locations l_1 and l_2 along scope s. Although scope elements s (routes and tracks) are not localized, we abuse the notation and have $\text{spots}(s)$ return all spot locations traversed by s.

An expression a is always evaluated in the context of a network model M, and a location l of a scope element s. We denote this value by ${}_M[\![a]\!]^l_s$, a tuple set calculated as defined in Fig. 4. Expressions are essentially built by combining railML identifiers (along with numeric, string, and Boolean constants) with relational and arithmetic operators. Whenever an identifier is a railML property of a scope element, it is implicitly projected on that element. Operation $M(i)\!\downarrow\! s$ projects identifier i if its domain is the scope type, leaving it unchanged otherwise. For instance, `routeEntry` relates routes with their entry signal (at interlocking level). A possible value for this identifier would be $M(\texttt{routeEntry}) = \{\langle \text{R1}, \text{SL1} \rangle, \langle \text{R2}, \text{SL2} \rangle\}$. In a rule with scope **`route`**, a call to `routeEntry` is projected on each scope element. For example, when evaluating a rule for route `R1` we would have ${}_M[\![\texttt{routeEntry}]\!]^l_{\text{R1}} = \{\langle \text{SL1} \rangle\}$. In a rule with scope **`track`** the full binary relation would be returned. Localized elements are also filtered according to the current location l. If an identifier i refers to a localized element, it is only considered if the current l is among is locations. Constants are also interpreted as tuple sets to allow combination with other expressions, so, for instance, constant 1 is interpreted as the tuple set $\{\langle 1 \rangle\}$.

Like Alloy, the fundamental relational operation is composition (`.`), which simplifies the navigation along children elements and attributes. For example, `routeEntry.isVirtual` in a rule with scope **`route`** determines whether the respective entry signal is

$$
\begin{array}{lcl}
{}_M[\![\textbf{everywhere } r\ \phi]\!]^l_s & \triangleq & \\
\quad \forall l' \in \text{spots}(s) \mid \text{dist}(s, l, l') \in {}_M[\![r]\!]^l_s \Rightarrow {}_M[\![\phi]\!]^{l'}_s & & \\
{}_M[\![\textbf{nowhere } r\ \phi]\!]^l_s & \triangleq & \\
\quad \forall l' \in \text{spots}(s) \mid \text{dist}(s, l, l') \in {}_M[\![r]\!]^l_s \Rightarrow \neg {}_M[\![\phi]\!]^{l'}_s & & \\
{}_M[\![\textbf{somewhere } r\ \phi]\!]^l_s & \triangleq & \\
\quad \exists l' \in \text{spots}(s) \mid \text{dist}(s, l, l') \in {}_M[\![r]\!]^l_s \wedge {}_M[\![\phi]\!]^{l'}_s & & \\
{}_M[\![\phi \textbf{ until } r\ \psi]\!]^l_s & \triangleq & \\
\quad \exists l' \in \text{spots}(s) \mid \text{dist}(s, l, l') \in {}_M[\![r]\!]^l_s \wedge {}_M[\![\phi]\!]^{l'}_s \wedge & & \\
\qquad \forall l'' \in \text{spots}(s) \mid \text{between}(s, l, l', l'') \Rightarrow {}_M[\![\psi]\!]^{l''}_s & &
\end{array}
$$

$$
\begin{array}{lcl}
{}_M[\![\texttt{[} a \texttt{ .. } b \texttt{]}]\!]^l_s & \triangleq & [n, m] \text{ s.t. } {}_M[\![a]\!]^l_s = \{\langle n \rangle\} \wedge {}_M[\![b]\!]^l_s = \{\langle m \rangle\} \\
{}_M[\![\texttt{[} a \texttt{ .. } b \texttt{ [}]\!]^l_s & \triangleq & [n, m[\text{ s.t. } {}_M[\![a]\!]^l_s = \{\langle n \rangle\} \wedge {}_M[\![b]\!]^l_s = \{\langle m \rangle\} \\
{}_M[\![\texttt{[} a \texttt{ .. [}]\!]^l_s & \triangleq & [n, \infty[\text{ s.t. } {}_M[\![a]\!]^l_s = \{\langle n \rangle\} \\
{}_M[\![\texttt{] } a \texttt{ .. } b \texttt{]}]\!]^l_s & \triangleq &]n, m] \text{ s.t. } {}_M[\![a]\!]^l_s = \{\langle n \rangle\} \wedge {}_M[\![b]\!]^l_s = \{\langle m \rangle\} \\
{}_M[\![\texttt{] .. } b \texttt{]}]\!]^l_s & \triangleq &]-\infty, m] \text{ s.t. } {}_M[\![b]\!]^l_s = \{\langle m \rangle\} \\
{}_M[\![\texttt{] } a \texttt{ .. } b \texttt{ [}]\!]^l_s & \triangleq &]n, m[\text{ s.t. } {}_M[\![a]\!]^l_s = \{\langle n \rangle\} \wedge {}_M[\![b]\!]^l_s = \{\langle m \rangle\} \\
{}_M[\![\texttt{] .. [}]\!]^l_s & \triangleq &]-\infty, \infty[
\end{array}
$$

$$
\begin{array}{lcl}
{}_M[\![\textbf{not } \phi]\!]^l_s & \triangleq & \neg {}_M[\![\phi]\!]^l_s \\
{}_M[\![\phi \textbf{ and } \psi]\!]^l_s & \triangleq & {}_M[\![\phi]\!]^l_s \wedge {}_M[\![\psi]\!]^l_s \\
{}_M[\![\textbf{all } v : a \mid \phi]\!]^l_s & \triangleq & \forall t \in {}_M[\![a]\!]^l_s : {}_{M \oplus v \mapsto \{t\}}[\![a]\!]^l_s
\end{array}
$$

$$
\begin{array}{lcl}
{}_M[\![a \textbf{ in } b]\!]^l_s & \triangleq & {}_M[\![a]\!]^l_s \subseteq {}_M[\![b]\!]^l_s \\
{}_M[\![\textbf{some } a]\!]^l_s & \triangleq & |{}_M[\![a]\!]^l_s| > 0 \\
{}_M[\![\textbf{lone } a]\!]^l_s & \triangleq & |{}_M[\![a]\!]^l_s| \leq 1 \\
{}_M[\![a]\!]^l_s & \triangleq & x = \textbf{true} \text{ s.t. } {}_M[\![a]\!]^l_s = \{\langle x \rangle\} \\
{}_M[\![a < b]\!]^l_s & \triangleq & n < m \text{ s.t. } {}_M[\![a]\!]^l_s = \{\langle n \rangle\} \wedge {}_M[\![b]\!]^l_s = \{\langle m \rangle\}
\end{array}
$$

Figure 5: Semantics of the rule language formulas.

virtual. Composition is defined as follows:

$$
\begin{aligned}
R \bullet S = \{&\langle r_1, \ldots, r_{n-1}, s_2, \ldots, s_m \rangle \mid \\
&\langle r_1, \ldots, r_n \rangle \in R \wedge \langle s_1, \ldots, s_m \rangle \in S \wedge r_n = s_1\}
\end{aligned}
$$

for relations R and S with arity n and m, respectively. Other relational operators include union (`|`), intersection (`&`), difference (`\`), Cartesian product (`->`), converse (`~`) and transitive closure (`^`). The cardinality of an expression can also be retrieved (`#`). Arithmetic operations are only well-defined when the value of the operands is a singleton set (i.e., exactly one tuple of arity one). Otherwise, a runtime error will be thrown. Note that such errors cannot be detected statically by the type-checker (to be presented shortly), since it only infers the type of the expressions, and not their cardinality. For instance, an arithmetic expression involving the *optional* integer attribute `approachSpeed` of signals, like in rule

```
track :: everywhere
  (all s : signalIS | s.approachSpeed < 100)
```

will type check and be correctly evaluated as long as an `approachSpeed` is assigned to every signal in the network under analysis.

Formulas combine these expressions using operators from (first-order) metric interval temporal logic, interpreted in the spacial

setting. The semantics of a formula ϕ is defined as ${}_M[\![\phi]\!]^l_s$, determining whether ϕ holds under model M for scope element s at location l, and is defined in Fig. 5 for a kernel of the language, from which the remaining operators can be easily derived.

Spatial properties (the first block in Fig. 5) are built with the unary **everywhere** (property holds in all positions), **nowhere** (property holds in no position) and **somewhere** (property holds in some position), and binary **until** (a property holds in some position and another hold must hold until there). We adopt a point-wise semantics, meaning a formula is only evaluated at spot locations along s where there is some localized element in M. These quantifiers can be restricted by ranges (the second block in Fig. 5), including negative values. For instance, a quantification **everywhere** `[-10,10]` ϕ evaluates ϕ in all positions between 10 meters before and 10 meters beyond the current position. By omission, the default range is `[0..[`, i.e., all positions succeeding the current position.

Spatial properties are combined with the typical Boolean connectives (**and**, **or**, **implies** and **iff**) and first-order quantification (**all** and **some**) (third block in Fig. 5). Abusing notation, the valuation M is also used to store the value of quantified variables. Atomic formulas (last block in Fig. 5) either determine if an expression is contained in another (**in**), the cardinality of an expression (**one**, **lone**, **some** and **no**), or, in the case of numerical expressions, integer inequalities. Likewise arithmetic expressions, some of these have side conditions. Such is the case of Boolean expressions (must evaluate to the singleton tuple $\{\langle \textbf{true} \rangle\}$) and the numeric inequality operations (the operands must evaluate to a singleton number).

Complete rules are created by assigning a scope to a formula. We denote the semantics of a rule S `::` ϕ under model M as ${}_M[\![S :: \phi]\!]$. Evaluating such a rule requires evaluating ϕ for all elements of the scope from the respective starting position, i.e.:

$$\forall \langle s \rangle \in M(S) \mid {}_M[\![S :: \phi]\!]_s^{\mathsf{entry}(s)}$$

3.2 Type system

EVEREST rules must obey certain type rules. Inspired by light-weight type-systems for relational logic [7], the type of an EVEREST expressions is itself also a tuple set whose atoms are atomic types (i.e., `number`, `string`, `bool` or elements of the railML schema). This allows the type of an expression to be calculated by applying the same relational operators at the type level. To calculate the type of an expression, we need a typing context Γ for the railML identifiers extracted from the railML schema. This typing context is defined in a configuration file, described in the next section. We denote the fact that an expression a has type T with arity n under the typing context Γ and scope type S as $\Gamma \vdash_S a \subseteq T^n$.

Most relational operators essentially test the arity of the expressions. For instance, a union is well-typed if both operands have the same arity, a rule defined as:

$$\Gamma \vdash_S a \mid b \subseteq (T \cup U)^n \quad \equiv \quad \Gamma \vdash_S a \subseteq T^n \wedge \Gamma \vdash_S b \subseteq U^m \wedge n = m$$

Quantifier ranges and arithmetic operators require expressions to be numbers with arity 1 (a set). For instance, the rule for addition is defined as:

$$\Gamma \vdash_S a + b \subseteq \{\langle number \rangle\}^1 \quad \equiv \quad \Gamma \vdash_S a \subseteq \{\langle number \rangle\}^1 \wedge \Gamma \vdash_S b \subseteq \{\langle number \rangle\}^1$$

Figure 6: Infrastructure Verifier of the EVEREST toolset

The scope type S is essentially used to get the type of identifiers that should be projected when the domain is S, namely:

$$\Gamma \vdash_S i \subseteq T^n \quad \equiv \quad \Gamma(i)\downarrow S \subseteq T^n \wedge n > 0$$

$T\downarrow S$ has the same behaviour as before. For instance, since $\Gamma($`routeEntry`$) = \{\langle$`route`, `routeEntry`$\rangle\}$, $\Gamma($`routeEntry`$)\downarrow$**route** $= \{\langle$`routeEntry`$\rangle\}$.

As an example of type inference for expressions, we have

$$\Gamma \vdash_{\textbf{route}} \texttt{routeEntry.refersTo.ref} \subseteq \{\langle \texttt{signalIL} \rangle\}^1$$

because:

$$\begin{aligned} \Gamma \vdash_{\textbf{route}} \texttt{routeEntry} &\subseteq \{\langle \texttt{routeEntry} \rangle\}^1 \\ \Gamma \vdash_{\textbf{route}} \texttt{refersTo} &\subseteq \{\langle \texttt{routeEntry}, \texttt{routeEntryRef} \rangle, \ldots\}^2 \\ \Gamma \vdash_{\textbf{route}} \texttt{ref} &\subseteq \{\langle \texttt{routeEntryRef}, \texttt{signalIL} \rangle, \ldots\}^2 \end{aligned}$$

Once the type of expressions is calculated, formulas ϕ can be type-checked. We say that a formula ϕ is well-typed under typing context Γ for type scope S when $\Gamma \vdash_S \phi$. Spatial and first-order logic operators essentially propagate type-checking to their operands. If an expression appears as a formula, then it must have a Boolean type, and arithmetic inequities must compare numeric expressions. Inclusion formulas additionally test the relevance of the sub-expressions, reporting an irrelevance error if they have nothing in common:

$$\Gamma \vdash_S a \textbf{ in } b \quad \equiv \quad \Gamma \vdash_S a \subseteq T^n \wedge \Gamma \vdash_S b \subseteq U^m \wedge n = m \wedge T \cap U \neq \emptyset$$

For instance, rule

```
route :: routeEntry.refersTo.ref in speedSection
```

would not type check because:

$$\begin{aligned} \Gamma \vdash_{\textbf{route}} \texttt{routeEntry.refersTo.ref} &\subseteq \{\langle \texttt{signalIL} \rangle\}^1 \\ \Gamma \vdash_{\textbf{route}} \texttt{speedSection} &\subseteq \{\langle \texttt{speedSection} \rangle\}^1 \end{aligned}$$

Lastly, we say that an EVEREST rule S `::` ϕ is well-typed under typing context Γ, denoted by $\Gamma \vdash S$ `::` ϕ, when $\Gamma \vdash_S \phi$.

4 IMPLEMENTATION

The EVEREST toolset incorporates two main tools, the EVEREST Design Verification tool and the EVEREST AutoCAD plugin.

4.1 EVEREST Design Verification tool

Main components. The EVEREST Design Verification tool is a standalone application developed in C# with the Windows Presentation Foundation (WPF), with three main components. The **Rule Designer** enables users to create EVEREST rules and manage a rule catalog. For each project, users can select which rules in the catalog should be evaluated. To simplify this process, functionalities such as rule editing, cloning, deleting and grouping are provided. Before adding a rule to the catalog, its syntax and type correctness

```
<railML>
 <infrastructure><topology>...</topology>
  <functionalInfrastructure>
   <signalsIS>
    <signalIS id="S1"><spotLocation netElementRef="ne1"
          applicationDirection="reverse" pos="149"/></signalIS>
    <signalIS id="S2"><spotLocation netElementRef="ne2"
          applicationDirection="reverse" pos="8.8"/></signalIS>
    <signalIS id="S3"><spotLocation netElementRef="ne3"
          applicationDirection="normal" pos="14.9"/></signalIS>
   </signalsIS>...
  </functionalInfrastructure>
 </infrastructure>
 <interlocking>
  <assetsForIL>
   <signalsIL>
    <signalIL id="SL1"><refersTo ref="S1"/></signalIL>
    <signalIL id="SL2"><refersTo ref="S2"/></signalIL>
    <signalIL id="SL3" isVirtual="true"><refersTo
          ref="S3"/></signalIL>
   </signalsIL>
   <routes>
    <route id="R1">
     <routeEntry id="E1"><refersTo ref="SL1"/></routeEntry>
     <routeExit id="X1"><refersTo ref="SL2"/></routeExit></route>
    <route id="R2">
     <routeEntry id="E2"><refersTo ref="SL2"/></routeEntry>
     <routeExit id="X2"><refersTo ref="SL3"/></routeExit></route>
   </routes>...
  </assetsForIL>
 </interlocking>...
</railML>
```

Figure 7: railML excerpt

is checked, according to the rules defined in Section 3.2. The **Infrastructure Verifier** evaluates the selected rules against the railML models of the current project, generating violations for each failed rule (see Fig. 6). Rule evaluation strictly follows the semantics defined in Section 3.1. Note that this process does not require solving because all free identifiers in the rules are defined in the railML model. Rule parsing was implemented with ANTLR [19], with evaluation, syntax and type-checking implemented via callbacks in the abstract syntax tree. The **Network Visualizer** allows the visualization, in a WPF Canvas, of the current signalling diagram. It also provides route and track highlighting and, in case of rule violations, highlights the failing scope elements.

railML processing. The implementation of EVEREST relies on a railML library, also developed in C#, whose main goal is to process railML files and extract relevant information. In particular, it extracts all the topology and signalling information needed for the Visualizer. It also computes the information needed for the Verifier, namely the abstract network model M of the railML file, and the auxiliary functions spots, entry, dist, and between.

The network model, as defined in Section 3.1, is a mapping from railML tags and attributes to a tuple set. To build this map only the <infrastrutcure> and <interlocking> railML sub-schemas are considered, and inside these we only consider the content of the <functionalInfrastrutcture> and <assertsForIL> tags, respectively, which group all relevant elements. For each of the main railML elements a set is created in the model. The railML identifier of the element (in the id attribute) is used to denote the respective atoms. For example, from the railML excerpt in Fig. 7, the following sets would be added to the model:

$$\texttt{signalIS} \mapsto \{\langle \texttt{S1}\rangle, \langle \texttt{S2}\rangle, \langle \texttt{S3}\rangle\}$$
$$\texttt{signalIL} \mapsto \{\langle \texttt{SL1}\rangle, \langle \texttt{SL2}\rangle, \langle \texttt{SL3}\rangle\}$$
$$\texttt{route} \mapsto \{\langle \texttt{R1}\rangle, \langle \texttt{R2}\rangle\}$$

Then, for each children tag, the respective binary relation associating it with its parent is added to the model. If the tag has an id attribute, that identifier is used. Otherwise a new unique identifier is created (represented in italics below). In the case of the railML excerpt, the following relations would be added to the model:

$$\texttt{routeEntry} \mapsto \{\langle \texttt{R1}, \texttt{E1}\rangle, \langle \texttt{R2}, \texttt{E2}\rangle\}$$
$$\texttt{routeExit} \mapsto \{\langle \texttt{R1}, \texttt{X1}\rangle, \langle \texttt{R2}, \texttt{X2}\rangle\}$$
$$\texttt{refersTo} \mapsto \{\langle \texttt{SL1}, \textit{1}\rangle, \langle \texttt{SL2}, \textit{2}\rangle, \langle \texttt{SL3}, \textit{4}\rangle, \langle \texttt{E1}, \textit{4}\rangle, \langle \texttt{E2}, \textit{5}\rangle, \langle \texttt{X1}, \textit{6}\rangle, \langle \texttt{X2}, \textit{7}\rangle\}$$

Finally, the binary relations associating each element with the respective attributes are also added to the model:

$$\texttt{isVirtual} \mapsto \{\langle \texttt{SL3}, \textbf{true}\rangle\}$$
$$\texttt{ref} \mapsto \{\langle \textit{1}, \texttt{S1}\rangle, \langle \textit{2}, \texttt{S2}\rangle, \langle \textit{3}, \texttt{S3}\rangle \langle \textit{4}, \texttt{SL1}\rangle, \langle \textit{5}, \texttt{SL2}\rangle, \langle \textit{6}, \texttt{SL2}\rangle, \langle \textit{7}, \texttt{SL3}\rangle\}$$

The <spotLocation> is ignored in this process, but is used to compute the spots of localized elements. Abstractly, a location is a triple that identifies the net element, the respective direction, and a position in the net element. We normalize these locations, storing in the spots function all the equivalent locations of an element. For example, for elements located in the extremity of a net element we also store the locations in the extremity of the connected net elements, if any (that is the case of switches, for example).

To compute the spots of tracks and routes, first the sequence of track segments of each route and track must be computed. For tracks that is given directly in the railML schema. In the case of a route it can be computed by following it from the track segment where the entry is located until the respective exit, taking into account the branching position of the facing switches. In the case of our railML example, there are no switches, and route R1 traverses segments ne1 and ne2, while route R2 traverses segments ne2 and ne3. From this sequences we can compute the spots of each route and track. The entry location of scope elements is trivial to compute. For computing the dist between two locations one must consider their position in the respective net elements, as well as the length of all net elements in between (the information about net elements is contained in the tag <topology>, whose content was omitted in Fig. 7). The between function is easily computed from the respective locations and the sequence of net elements of each route or track.

Configuration file. The main goal of the EVEREST configuration is to define which railML elements should be loaded into model M, as well as the respective typing context Γ to be used in the type-checker. As explained in Section 3.2, types are tuple sets of atomic types. The user is free to chose the atomic types in the configuration file. In general there will be a single atomic type per railML tag, but for tags with multiple parent elements it might be a good idea to further discriminate to increase the precision and usefulness of

```
"id", "type", "macro definition"
"route", "{[route]}"
"signalIS", "{[signalIS]}"
"signalIL", "{[signalIL]}"
"routeEntry", "{[route,routeEntry]}"
"routeExit", "{[route,routeExit]}"
"isVirtual", "{[signalIL,bool]}"
"refersTo", "{[routeEntry,refersToSignalIL],
     [routeExit,refersToSignalIL], [signalIL,refersToSignalIS],..."
"ref", "{[refersToSignalIS,signalIS],
     [refersToSignalIL,signalIL],..."
"entrySignal", "{[route,signalIL]}", "routeEntry.refersTo.ref"
"exitSignal", "{[route,signalIL]}", "routeExit.refersTo.ref"
```

Figure 8: EVEREST configuration file excerpt

the type-checker. Figure 8 presents an excerpt of a possible configuration file. Notice how several atomic types where created for `refersTo`, so that more precise information about the type of the referred element can be obtained. If a single atomic type was used for `refersTo`, the type of expression `routeEntry.refersTo.ref` in a **`route`** context would no longer be just {⟨`signalIL`⟩}, but instead contain almost all railML elements, which would be undesirable.

The configuration file also allows the definition of *macros*. These are essentially expression abbreviations that can be used when defining rules, being replaced in-place before evaluation. Macros help tame the, sometimes, cumbersome verbosity and high level of indirection of railML. For instance, to retrieve the infrastructure view of the entry signal of a route, one would have to write `routeEntry.refersTo.ref`, which in the example configuration file we abbreviate as macro `entrySignal`. Once defined in the configuration file, they are treated as any other railML identifier, their value being calculated and assigned to M during railML processing.

Rule boilerplates. To ease the writing of rules that have similar shape, but vary in concrete parameters, such as safety distances, EVEREST supports the definition of rule boilerplates in the catalog. These have *placeholder* identifiers, marked with $, that must be assigned concrete values when instantiated. For example, a very convenient boilerplate rule is the following generalization of the rule presented above, stating that in every route an element of type A cannot be followed by an element of type B in the next d meters.

```
route :: everywhere
  (some $A implies everywhere [0..$d] no $B)
```

4.2 EVEREST AutoCAD plugin

The EVEREST AutoCAD plugin is an AutoCAD extension, whose goal is to support the positioning of signalling elements in the technical drawing, and updating a railML model with the precise length of track segments and the precise location of localized elements, information needed to verify rules.

Users can interact with the plugin by executing its AutoLisp commands. Typically, a plugin command receives the required user input, invokes the backend, and updates the technical drawing with the results. The backend is a C# console application that relies on the netDXF library[4] to process AutoCAD drawings and perform

[4]https://github.com/haplokuon/netDxf, last visited May 12, 2022.

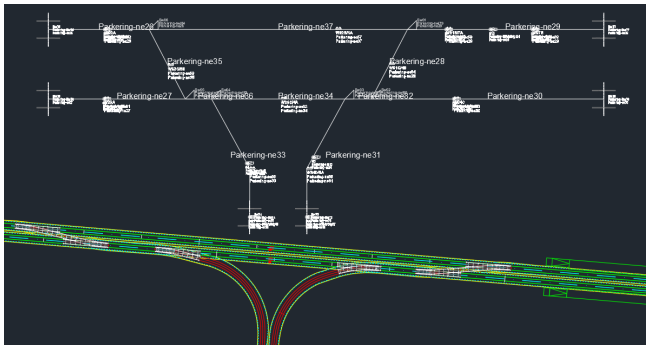


Figure 9: Signalling diagram imported into a CAD drawing

the required calculations for the plugin. Moreover, the plugin uses a block library to draw the various infrastructure elements, such as switches, borders, and signals. Each railML block defined in this library contains the following five attributes necessary to determine its precise location and depict any reported violations: `ID`, the railML identifier; `NAME`, the railML `name` attribute; `NETELEMENT`, the identifier of the net element where the element is located; `POS`, the precise location of the element along the net element, initially empty and updated after positioning; and `ERROR`, with the possible violations detected by the Verifier.

The main functionalities of the plugin are the following.

railML import. To start the element positioning process, the signalling diagram in a railML file is imported into the technical drawing. This CAD signalling diagram is drawn in a user-defined position and layer, using the block library to draw each net element. It serves as a guide for the technical designers to correctly place (by just moving) the various infrastructure elements in their precise locations. The result for our running example is presented in Fig. 9.

Identification of net elements. To compute the lengths of track segments and the precise locations of elements, it is necessary to identify in the technical drawing the lines corresponding to the various net elements of the signalling diagram. To help with that task, the plugin provides commands to break lines (typically, in the technical drawing a single line corresponds to a track spanning various net elements) and label the resulting segments with the corresponding identifier. Moreover, users can run a verification command to check if all net elements are already correctly labelled, with the feedback being depicted by different colors in the CAD signalling diagram: a green net element indicates a correctly labelled segment was found, a red color indicates an incorrectly labelled segment, and orange a missing label. A net element is correctly labelled if the expected elements are already placed in its extremities (typically switches, buffer stops, or borders). There is also a command to draw missing net elements in the technical drawing (as straight lines between the elements defining the extremities).

Computation of lengths and positions. After the labelling process and positioning of all infrastructure elements in the technical drawing, users can execute a command that updates the railML model with the precise length of all net elements and location of the localized elements, which is stored in the `pos` attribute of the respective

<spotLocation> tags. This is done through AutoCAD functions that determine the length of an arbitrary line and the distance (along the line) between its begin point and any other point in it.

Visualization of violations. Rule violations detected during verification are imported and depicted in the technical drawing, by storing the respective descriptions in the `ERROR` attribute of relevant elements and changing the color of their attributes to orange. Figure 1b shows precisely the result of invoking this command, with a switch violating the example rule given in Section 2, requiring that no switches appear in the 20 meters following a signal.

5 EVALUATION

Our evaluation aimed to answer the following research questions:

RQ1 Is the rule language expressive enough to specify typical infrastructure constraints of railway network models?

RQ2 Is the infrastructure verifier efficient enough to evaluate the rules in realistic network models?

5.1 Rule expressiveness

To evaluate RQ1 we specified a catalog of rules relevant in the design of railway networks for light rail solution, an application area where EFACEC has vast experience in providing signalling systems. We found that most of the rules fall into one of four categories:

Existence of elements. In this category we have rules requiring certain elements to be present (or absent) in the network. Some variants include requiring that if a certain element is present another related element should also be present. Typically they are specified with the **track** scope, since we want to check them independently of the defined routes. As an example consider the following rule, requiring that every virtual signal is the exit signal of some route.

```
track :: all s : signalIL |
  s.isVirtual implies some exitSignal.s
```

Here, `exitSignal` is the macro defined in Fig. 8 that associates a **route** with the respective `signalIL`. Expression `exitSignal.s` determines the set of routes of which signal `s` is the exit signal, which the rule requires to be non-empty if `s` is virtual. This is a degenerate rule that does not use the modal spatial operators, since it is not concerned with the actual location of elements. Many rules in this category are degenerate in this sense. To evaluate RQ2 we used 6 rules in this category (#1 to #6), this being rule #2.

Order of elements. In this category we have rules requiring that elements appear in certain order. For example, the first switch after entering an area must be preceded by a train detection element, to ensure the switch is in a *Track Vacancy Detection* (TVD) section. Since entry tracks have a `border` element at the beginning, this rule could be specified as follows.

```
track :: some border implies everywhere
  (some switchIS implies
    somewhere ]..0[ some trainDetectionElement)
```

Here we use the **somewhere** operator with a (open-ended) negative range to check that somewhere before a switch there is the required train detection element. Such negative ranges are very convenient in the specification of these category of rules. To evaluate RQ2 we used 2 rules in this category (#7 and #8), this being rule #7.

Distance between elements. This is a very common category. Most rules follow the boilerplate presented in the previous section, that requires a minimum distance between two kinds of elements. As an example, we have a slight (more realistic) variation of the rule presented in Section 2, requiring a minimum distance of 50 meters between every signal and a facing switch.

```
route::everywhere (some signalIS implies
  everywhere [0..50] no (switchIS &
    facingSwitches.refersTo.ref))
```

Here expression `facingSwitches` is a macro that obtains the interlock view of the facing switches of a route (of type `switchIL`). By composing with `refersTo.ref` we obtain the respective infrastructure view (of type `switchIS`). And finally, by intersecting with set `switchIS` (which is implicitly project for each location) we restrict the resulting set to those switches actually located in the 50 meters following a signal. Some rules require enforcing, not minimum, but maximum distances between the occurrence of certain elements. For example, in networks with an *Automatic Train Protection* (ATP) system every (non-virtual) signal is required to have a *balise* (a transponder) nearby. This could be specified as follows.

```
track::everywhere (some s: signalIL |
  not s.isVirtual implies somewhere [-1..0[ some balise)
```

To evaluate RQ2 we used 6 rules in this category (#9 to #14), the above rules being #9 and #12, respectively.

Element coverage. In this category we have rules requiring the network to be covered by certain elements. The most relevant of this rules is one requiring every track to be correctly covered by TVD sections. Specifying this rule directly is not easy since currently the railML model does directly associate TVD sections with tracks. However, it can be specified indirectly by a combination of three different rules requiring: 1) every buffer stop to be a demarcator of one TVD section; 2) every train detection device to be a demarcator of two TVD sections; 3) with the exception of the last or first train detection devices in an area that demarcates exactly one TVD section. This exceptional case can be specified as follows.

```
track :: everywhere (all t : trainDetectionElement |
    #(hasDemarcatingTraindetector.ref.t) = 1 implies
    ((somewhere [0..[ some border) and
     (everywhere ]0..[ no trainDetectionElement) or
     (somewhere ]..0] some border) and
     (everywhere ]..0[ no trainDetectionElement)))
```

A train detection device in an area is the last one if it is in a track with some border afterwards, and, obviously, no more train detection devices. Likewise for the first train detection device. For RQ2 we used these 3 rules (#15 to #17), the above rule being #17.

Discussion. So far the rule language proved expressive enough to specify all the example constraints provided by the signalling engineering team at EFACEC. Due to the modal spatial operators and implicit scope and location projection, many rules can be specified with a very terse and rather readable style. The main issue with readability stems from the verbosity and redundancy of the railML

Table 1: Area size (quantity of elements)

	A	B	C	D	E
tracks	1	8	10	12	33
routes	3	5	4	10	46
signalsIS	4	4	6	10	32
switchesIS	0	4	4	6	18
borders	0	2	0	6	2
bufferStops	2	2	4	0	10
trainDetectionElements	3	6	6	11	30
balises	0	0	0	7	17

schema, namely the two different views for each element and the high level of indirection connecting related elements. As the example coverage rule showed, the way information is stored in railML sometimes also does not allow a direct encoding of a constraint, requiring an alternative formulation that was not always trivial to define. Note that the linear nature of the language semantics does not allow the specification of rules that require reasoning about arbitrary paths in the network, for example requiring a minimal distance between two signals along any path (not necessarily in a route or track). However, such constraints did not occur so far in the projects verified with EVEREST.

5.2 Rule verification efficiency

To evaluate the efficiency of the Verifier we selected a set of 5 network models of real areas from a light railway project for which EFACEC was contracted to develop the signalling system. These areas vary in size and diversity of elements. Table 1 presents the respective number of tracks, routes, and other elements. The names or areas are anonymized, D being our running example area. Area E is an example of a complex area, corresponding to a depot area with several tracks for parking vehicles and maintenance activities. The remaining areas are less complex and encompass the typical light rail stations. All the measurements were performed in an off-the-shelf PC laptop with a 2.6GHz Intel i7 CPU and 32GB RAM.

There are two relevant aspects concerning efficiency. First, we have the time needed to process the railML models, to populate the abstract network model and compute the auxiliary functions spots, entry, dist, and between. This process, described in Section 4.1, is executed only once per area, when opening a railML file. Table 2 shows that this time increases with the area size, as was expected, but is still below 100ms even for the most complex area. Second, we have the time needed to actually evaluate the rules. This time also increases with the complexity of the area, as expected. All rules are verified well below 1s, and in many cases almost instantaneously, except for rule #9 in area E that took a few seconds to be checked. For the sample catalog, the total time to verify all rules in the sample areas (including the railML processing time) ranges between 11ms and 6.5s. Even scaling up for a more realistic catalog with hundreds of rules, the total time would be negligible in the context of a signalling project and, obviously, vastly more efficient than checking the rules by hand, as is currently done.

Table 2: Rule verification time (ms)

	A	B	C	D	E
railML	3	8	9	20	73
#1	0	3	4	6	183
#2	1	3	4	5	254
#3	0	0	0	0	12
#4	0	0	0	0	3
#5	0	0	0	0	8
#6	0	2	2	3	24
#7	0	0	0	1	8
#8	0	0	1	0	6
#9	0	50	15	150	5584
#10	2	1	0	9	43
#11	1	3	3	13	208
#12	0	2	1	2	22
#13	0	1	0	1	6
#14	0	2	0	8	28
#15	0	2	0	0	11
#16	1	4	6	6	20
#17	3	5	2	11	26
total	11	86	47	235	6519

6 RELATED WORK

Formal methods and railway systems. The railway community has long recognized the need for formal software development techniques. The CENELEC EN 50128 standard [1] imposes conditions for applying formal verification tools to safety-critical software in the railway domain, being highly recommended to reach the highest safety integrity level (SIL) of SIL3 / SIL4. CENELEC EN 50128 classifies tools into three different types, from T1 to T3, depending on whether they can introduce faults into the safety critical software, formal tools usually being type T2 (they may fail to identify faults but cannot introduce them themselves).

However, a relatively recent survey [8] concluded that no universally accepted formal framework has emerged, and railway companies wishing to adopt formal methods still have little guidance in the selection of the most appropriate methods and tools. Moreover, almost all approaches focus on verifying dynamic aspects of the railway. Some of these approaches are, however, built over railML network models, such as the verification of interlocking rules using model checking [9], or capacity analysis using SAT solving [13].

Railway infrastructure verification. As far as we are aware, the only work focusing on the verification of infrastructure rules over network models is the CAD-centric approach proposed by Luteberget and Johansen [14], eventually integrated and commercialized in the RailCOMPLETE® tool [20]. Here both the technical drawing and the signalling design is performed in AutoCAD, supported by a plugin. This is fundamentally different from our decoupled approach, where the technical and signalling designers can work independently with the tools they are accustomed to, synchronizing sporadically when verification is required. Likewise our approach, they use railML models as the interchangeable format (although at the time of release, railML 2 did not include interlocking information, which required a non-standard extension [5]). The verification

engine is based on the logic programming language Datalog, which supports a fragment of first-order logic. railML models are translated into Datalog facts, and infrastructure rules are expected to be programmed by users directly as Datalog rules. The authors also exploit the fact that design is incremental to more efficiently perform analysis. Our evaluation in Section 5.2 has shown that performance is not an issue with EVEREST.

Later work by the same team led to a controlled natural language (CNL) to specify infrastructure rules [12] The goal is to allow domain experts to encode rules in a more user-friendly language than Datalog. Although this language is quite rigid and much less expressive than the underlying Datalog, it still supports the encoding of many rules required by the Norwegian regulator. Our approach sits in a sweet spot between [14] and [12]: the EVEREST rule language is higher-level and more readable than programmatic rules due to its use of modal logic, but more expressive than a CNL. To simplify the specification of rules with similar patterns, EVEREST allows instead the definition of rule templates with placeholders.

Since in RailCOMPLETE both the technical drawing and the signalling design is performed in AutoCAD, the latter can become quite difficult to understand due to the overwhelming detail needed in the former. In a recent work [15], a SAT-based synthesis procedure was developed to automatically generate a more abstract schematic diagram easier to understand by signalling engineers.

The railML organization supports a validator and visualizer for railML models, railVIVID[5], but it essentially supports schema validation and has no support for verifying any kind of rules.

Railway network modelling. There is a chronic difficulty in connecting different railway IT applications. Efforts towards standardization started around 2002, resulting in the first version of railML. In 2013 the UIC (International Union of Railways) launched a group to standardize several data formats for infrastructural models in use at that point by the EU. The outcome was the RailTopoModel® (RTM) standard (IRS 30100), which proposes a multi-level topological model supporting multiple referencing systems. RTM was developed in collaboration with the railML organization, which resulted in the release of the RTM-compliant railML 3 in 2017, with a complete re-organization of the infrastructure layer. Other formats, such as ARIANE, also evolved to support RTM.

There are a few other data formats for topological models besides RTM (e.g., the IDM^{VU} standard[6] for infrastructure data management, the OpenStreetMap (OSM) file formats[7] used in the OpenRailwayMap initiative, or part of the UNISIG Subset-112 exchange format for specifying test scenarios. There is also some work on using general-purpose modelling languages in the railway domain, e.g. UML [4]. Nonetheless, EVEREST uses railML mainly because it is becoming the *de facto* standard in industry and is supported by most railway software[8], despite some current limitations that render the specifications of some rules rather indirect (see discussion in Section 5.1) or even impossible (due to lack of information).

Formal logics. Our rule language was inspired by metric interval linear temporal logic [3] and Alloy [10]. Concerning the former we just adapted the temporal modal operators to deal with distances along tracks and routes. By contrast with, e.g. [11], we do not require a fully-fledged *spacial logic* [2] because of the linear nature of railway tracks and the fact that we are only interested in spatial restrictions along those. Alloy is popular for the specification of structural restrictions and in its latest version 6 supports (non-metric) temporal modal operators [16] (which has been used to verify dynamic aspects of the Hybrid ERTMS/ETCS Level 3 railway standard [6]). Several other languages and logics can be used to specify structural restrictions, for example UML's *Object Constraint Language* (OCL) [18], but Alloy has a much simpler syntax and semantics thanks to its "everything is a relation" motto.

7 CONCLUSIONS AND FUTURE WORK

We have presented EVEREST, a toolset for automating the verification of railway network models. EVEREST is expected to have a deep impact at EFACEC due to the following key benefits: 1) support for using formal verification techniques at design phase which will increase the correctness of the produced design, and consequently, reduce the number of errors that pass to the next stages; 2) support for the development of a consistent view of the network between signalling engineers and technical designers, automating the flow of information between both views of the system, thus increasing productivity among the teams; 3) support for the definition of relevant properties to be verified, and automation of their verification, which will reduce the required effort for verification activities.

While one main goal behind the development of EVEREST was to allow the different teams to keep using the design tools that better suit their concerns, the adoption of the toolset will still impact current practices. On the one hand, technical designers will use the AutoCAD plugin to support the positioning of net elements. On the other hand, rules will have to be specified and the results of verification inspected. This opens a immediate needs to further evaluate, through empirical studies, the acceptability of the AutoCAD plugin and of the Network Visualizer representations by technical staff at EFACEC, and the expressiveness of the DSL.

On the medium to long term, we envisage a number of possibilities. This paper focuses on the verification of railway infrastructure rules, but those regarding *interlocking* must also be verified prior to deployment. These rules are dynamic, and analysis must consider the evolution of the system. We are studying ways to integrate this analysis in the EVEREST workflow. Moreover, the tool currently focuses on the identification of violations. A logical next step is to look at support for automatically fixing those violations. One possibility is to follow the ideas in [15], and explore how synthesis could be used to propose fixes for the violations found.

ACKNOWLEDGMENTS

This work is under the scope of the project DigiLightRail, being developed by EFACEC in cooperation with INESC TEC, for R&D activities. DigiLightRail is supported by the operational programme COMPETE 2020, financed by COMPETE 2020, a partnership agreement between the Portuguese government and the European Comission. The work of INESC TEC researchers is also financed by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project LA/P/0063/2020.

[5] https://www.railml.org/en/user/railvivid.html, last visited May 12, 2022.
[6] http://www.idmvu.org, last visited May 12, 2022.
[7] https://wiki.openstreetmap.org/wiki/OSM_file_formats, last visited May 12, 2022.
[8] https://www.railml.org/en/introduction/software.html, last visited May 12, 2022.

REFERENCES

[1] CENELEC CLC/TC 9X. 2011. EN 50128:2011 - Railway applications – Communication, signalling and processing systems – Software for railway control and protection systems.

[2] Marco Aiello, Ian Pratt-Hartmann, and Johan Van Benthem. 2007. *What is Spatial Logic?* Springer Netherlands, Dordrecht, 1–11. https://doi.org/10.1007/978-1-4020-5587-4_1

[3] Rajeev Alur, Tomás Feder, and Thomas A Henzinger. 1996. The benefits of relaxing punctuality. *Journal of the ACM (JACM)* 43, 1 (1996), 116–146.

[4] Kirsten Berkenkötter and Ulrich Hannemann. 2006. Modeling the Railway Control Domain Rigorously with a UML 2.0 Profile. In *Computer Safety, Reliability, and Security*, Janusz Górski (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 398–411. https://link.springer.com/chapter/10.1007/11875567_30

[5] Mark Bosschaart, Egidio Quaglietta, Bob Janssen, and Rob M. P. Goverde. 2015. Efficient formalization of railway interlocking data in RailML. *Inf. Syst.* 49 (2015), 126–141.

[6] Alcino Cunha and Nuno Macedo. 2020. Validating the Hybrid ERTMS/ETCS Level 3 concept with Electrum. *Int. J. Softw. Tools Technol. Transf.* 22, 3 (2020), 281–296. https://doi.org/10.1007/s10009-019-00540-4

[7] Jonathan Edwards, Daniel Jackson, and Emina Torlak. 2004. A type system for object models. In *SIGSOFT FSE*. ACM, 189–199.

[8] Alessio Ferrari, Maurice H. ter Beek, Franco Mazzanti, Davide Basile, Alessandro Fantechi, Stefania Gnesi, Andrea Piattino, and Daniele Trentini. 2019. Survey on Formal Methods and Tools in Railways: The ASTRail Approach. In *Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification - Third International Conference, RSSRail 2019, Lille, France, June 4-6, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11495)*, Simon Collart Dutilleul, Thierry Lecomte, and Alexander B. Romanovsky (Eds.). Springer, 226–241. https://doi.org/10.1007/978-3-030-18744-6_15

[9] Tim Gonschorek, Ludwig Bedau, and Frank Ortmeier. 2018. Bringing formal methods on the rail - On automatic verifying railroad interlockings from railML models. In *Safety and Reliability – Safe Societies in a Changing World Edition*. CRC Press, 741–748.

[10] Daniel Jackson. 2019. Alloy: A language and tool for exploring software designs. *Commun. ACM* 62, 9 (2019), 66–76.

[11] Sven Linker. 2015. *Proofs for Traffic Safety: Combining Diagrams and Logic*. Ph. D. Dissertation. Carl von Ossietzky Universitat Oldenburg, Oldenburg. Available from ResearchGate..

[12] Bjørnar Luteberget, John J. Camilleri, Christian Johansen, and Gerardo Schneider. 2017. Participatory Verification of Railway Infrastructure by Representing Regulations in RailCNL. In *SEFM (LNCS, Vol. 10469)*. Springer, 87–103.

[13] Bjørnar Luteberget, Koen Claessen, Christian Johansen, and Martin Steffen. 2021. SAT modulo discrete event simulation applied to railway design capacity analysis. *Formal Methods Syst. Des.* 57, 2 (2021), 211–245.

[14] Bjørnar Luteberget and Christian Johansen. 2018. Efficient verification of railway infrastructure designs against standard regulations. *Formal Methods Syst. Des.* 52, 1 (2018), 1–32. https://doi.org/10.1007/s10703-017-0281-z

[15] Bjørnar Luteberget and Christian Johansen. 2021. Drawing with SAT: four methods and A tool for producing railway infrastructure schematics. *Formal Aspects Comput.* 33, 6 (2021), 829–854. https://doi.org/10.1007/s00165-021-00566-z

[16] Nuno Macedo, Julien Brunel, David Chemouil, Alcino Cunha, and Denis Kuperberg. 2016. Lightweight specification and analysis of dynamic systems with rich configurations. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 373–383.

[17] Andrew Nash, Daniel Huerlimann, Joerg Schuette, and Vasco Paul Krauss. 2004. RailML – A standard data interface for railroad applications. In *Computers in Railways IX*. WIT Press.

[18] OMG. 2014. Object Constraint Language, version 2.4.

[19] Terence J. Parr and Russell W. Quong. 1995. ANTLR: A predicated-LL (k) parser generator. *Software: Practice and Experience* 25, 7 (1995), 789–810.

[20] RailCOMPLETE 2021. *RailCOMPLETE Tutorial v2021.0.* RailCOMPLETE. https://www.railcomplete.com/docs/2021-01-31_000%20EN%20RailCOMPLETETutorial%20v2021.0.pdf