Technical Report

- **Project**: DigiLightRail
- **WP**: WWW
- **Deliverable**: T3.2
- **Producer**: HASLab / INESC TEC
- **Title**: Prototyping the EVEREST tool
- **Summary**: *This report (.....)*

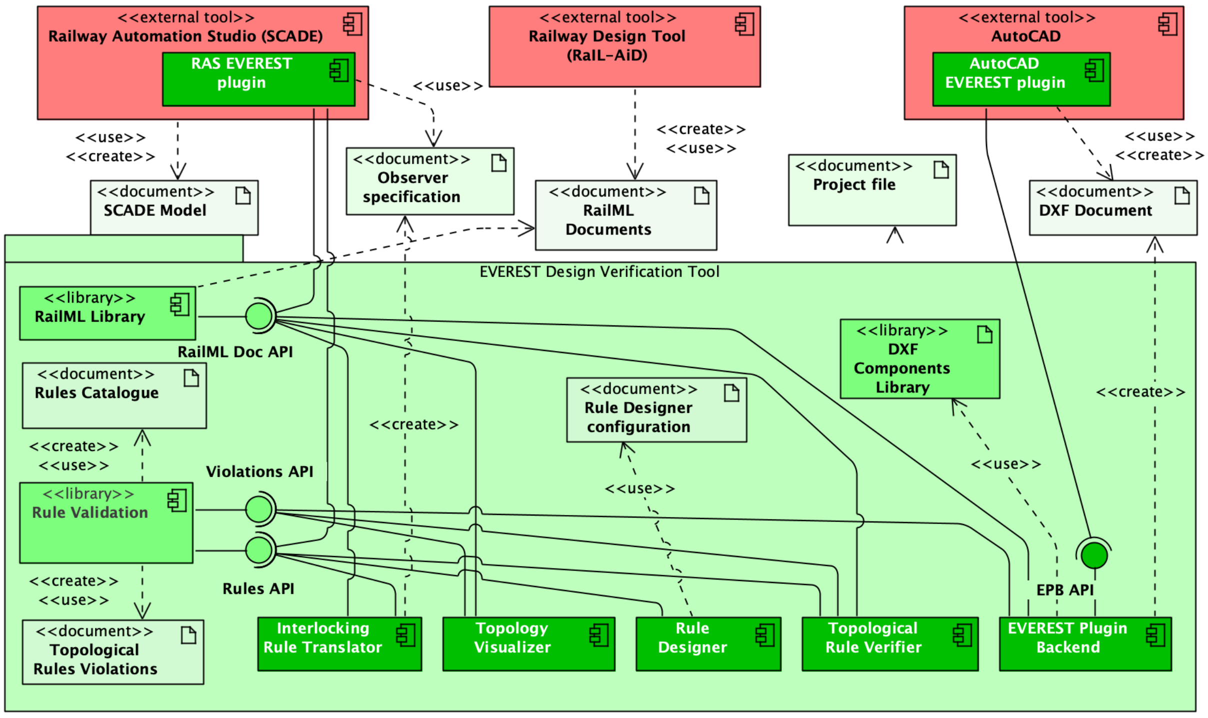
Introduction

Project DigiLightRail aims at designing and implementing a platform for the design of command and control systems for light surface trains (vulg "*metros*" in the French terminology), allowing for the configuration of *automatic train protection* (ATP) systems and the basic configuration of an entire *cyber-physical system of systems* (CPSoS) control and command system.

One of the main deliverables of DigiLightRail is a design automation tool, EVEREST (**E**facec **Ve**rification of **R**ailway n**E**twork**S** **T**ool). The goal of this tool is to automatically validate the design of railway infrastructures through user-defined infrastructure and interlocking rules. This tool's architecture, as shown in the picture below, is a set of loosely coupled components. These components will support the specification of rules, their verification against concrete railway topologies, and subsequent reporting of validation. The components are the following:

- **Rule Designer (RD):** offers a UI to define a set of rules that a railML model should follow.
- **Topological Rule Verifier (TRV):** grants the possibility to test a railML model against a set of user-defined infrastructure rules.
- **Topology Visualizer (TV):** provides a UI that can display railML topologies as graphs as well as infrastructure rule violations.
- **Interlocking Rule Translator (IRL):** translates interlocking rules to propositional logic formulas that specify observers that can later be incorporated into SCADE models using the RAS EVEREST Plugin.
- **RAS EVEREST Plugin (REP):** a Railway Automation Studio (RAS) plugin that creates SCADE observers from the propositional logic formulas resulting from interlocking rules.
- **AutoCAD EVEREST Plugin (AEP):** an AutoCAD's extension that allows users to import railML documents as DWG entities, enrich the railML model with topological information, and depict topological rule violations in a drawing.

On a technical level, these components will be implemented as described in the component diagram below.



<<external tool>> Railway Automation Studio (SCADE)
RAS EVEREST plugin
<<external tool>> Railway Design Tool (RaIL-AiD)
<<external tool>> AutoCAD
AutoCAD EVEREST plugin
<<use>>
<<create>>
<<document>> SCADE Model
<<document>> Observer specification
<<document>> RailML Documents
<<document>> Project file
<<document>> DXF Document
EVEREST Design Verification Tool
<<library>> RailML Library
RailML Doc API
<<document>> Rules Catalogue
<<library>> DXF Components Library
<<document>> Rule Designer configuration
Violations API
<<library>> Rule Validation
Rules API
EPB API
<<document>> Topological Rules Violations
Interlocking Rule Translator
Topology Visualizer
Rule Designer
Topological Rule Verifier
EVEREST Plugin Backend

The components Topology Visualizer, Rule Designer, Topological Rule Verifier and Interlocking Rule Translator are part of the *EVEREST Design Verification Tool*, which provides the user a project-based interface and support for a rule catague to be shared among all projects.

In this document, we report on the prototyping of the EVEREST Tool according to the specifications of technical report 2.6, except for the RAS EVEREST Plugin, which is outside the scope of this project.

Prototyped components and functionalities

All the components and functionalities of the EVEREST environment were prototyped except for the IRL, since at the time the prototype started to be developed this component was not yet fully specified. The EVEREST Design Verification Tool was also prototyped, where the various prototype components were integrated. The support for the EVEREST rules catalogue is implemented, as well as the configuration file with support for expression macros (railML typing information was still not considered in the configuration file). However, the catalogue still does not allow the creation of new versions of a rule nor the cloning of rules. The notion of project is implemented, with a set of rules selected as active, but only a single railML model (i.e., a single topology area) is supported. Moreover, the developed tool's UI is still preliminary.

The prototyped components and their functionalities were the following:

- **RD:** this component's prototype includes the creation of EVEREST rules. The prototype only implements the following subset of the proposed grammar:

```
rule        := scope :: structural
```

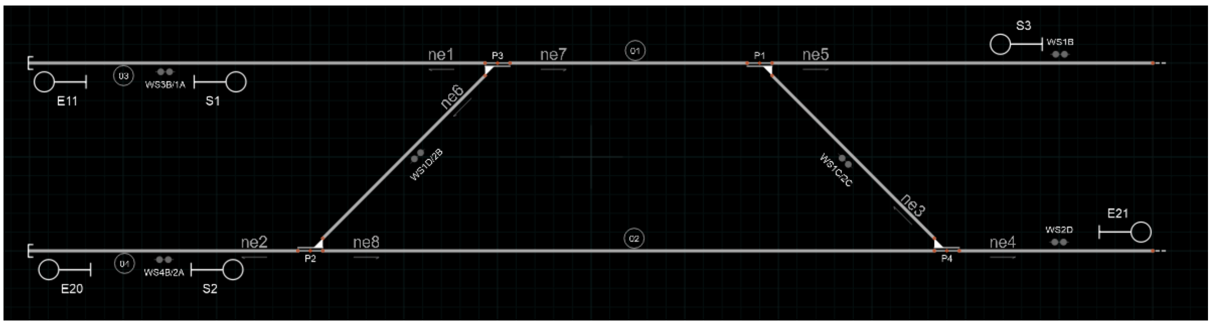
```
structural  := fol / spatial
spatial     := everywhere structural
fol         := expr in expr / expr = expr
             / structural binFormOp structural
             / not structural
             / sexpr
binFormOp   := and / or / implies / iff
expr        := var / railMLId / const
             / expr binExprOp expr
binExprOp   := .
scope       := route
railMLId    := id
var         := id
const       := number / string / true / false
```

For this subset prototype parses and checks the syntax of any created rule against the grammar (`CHECK-SYNTAX`) and performs a simple type check (`CHECK-TYPE`) according to a simplified version of the specified algorithm. In particular, the implemented type checking verifies the comparison between different types and the access of invalid properties with the **`.`** operator. Lastly, the prototype for this component does not yet support the creation of rules with patterns (`INSTANTIATE-PATTERN`).

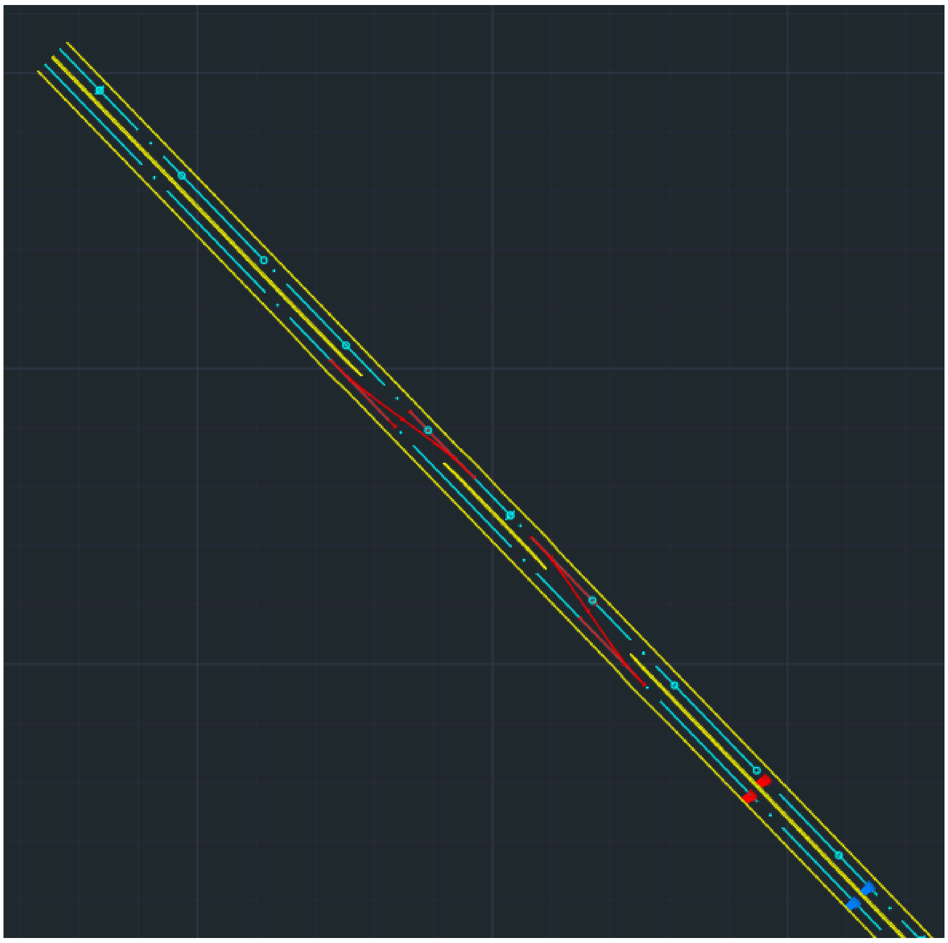
- **TRV:** the evaluation of EVEREST rules and the generation of violations in a CSV file format (`EVALUATE-IS-RULE`) was implemented for the language subset supported by the prototype RD, described above. For this reason, this component can only generate violations for the **`route`** scope. Moreover, it still does not report additional railML elements that may be involved in the violation.
- **TV:** this prototype implements the drawing of a railML model's topology (`DRAW-TOPOLOGY`). It also supports the highlighting of routes and tracks (`HIGHLIGHT-SCOPE`), as well as violations (`SHOW-VIOLATION`). Nevertheless, this prototype can only represent four railML elements: `signalIS`, `switchIS`, `border` and `bufferStop`.
- **IRL:** no prototype was created for this component.
- **REP:** this component is outside the scope of this project.
- **AEP:** this prototype encompasses an AutoLisp script and a C# program and performs a bridge between the EVEREST tool and AutoCAD. The plugin's prototype offers a list of commands for importing railML models (`IMPORT-RAILML`), updating railML elements' coordinates (`EXPORT-RAILML`) and reporting violations in AutoCAD drawings (`IMPORT-VIOLATIONS`). Moreover, the prototype allows railML block movement and breaking of DWG entities (`MOVE-RAILML-BLOCK` and `MOVE-ALL-RAILML-BLOCKS`), labelling (`LABEL-NET-ELEMENT` and `AUTO-LABEL-NET-ELEMENTS`), drawing missing entities (`DRAW-MISSING-NET-ELEMENTS`), and verification of improper labelling (`VERIFY-LABELS`). Lastly, this prototype can only represent four railML elements: `signalIS`, `switchIS`, `border` and `bufferStop`.

Hjallese use case

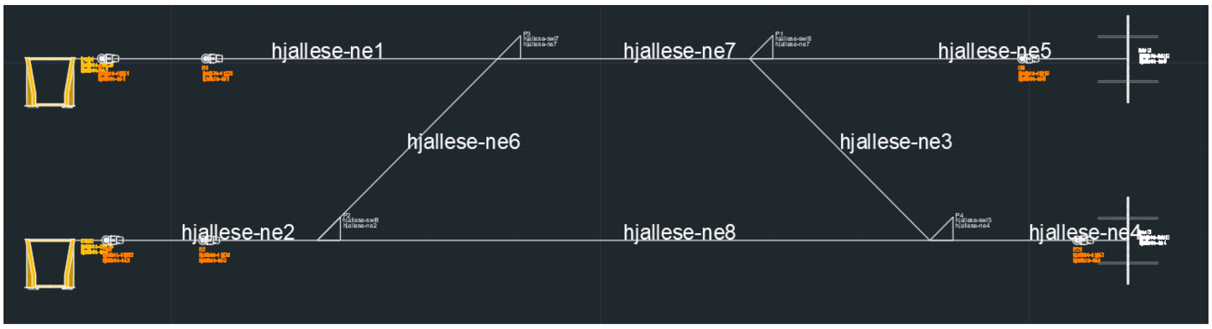
To test the prototype components and functionalities, we developed a use case based on the Hjallese station, for which both a AutoCAD drawing and railML model exist. The railML model contains eight net elements, four switches, two borders, two buffer stops and six signals (three of them virtual). The topological model is depicted below:



The goal of the use case is to show how to update the railML elements' positions and lengths using the EVEREST plugin and define rules to evaluate the model. The AutoCAD drawing used in this use case is given below, showing only relevant layers. In particular, the topological information is represented in the drawing by two cyan lines and two red blocks.



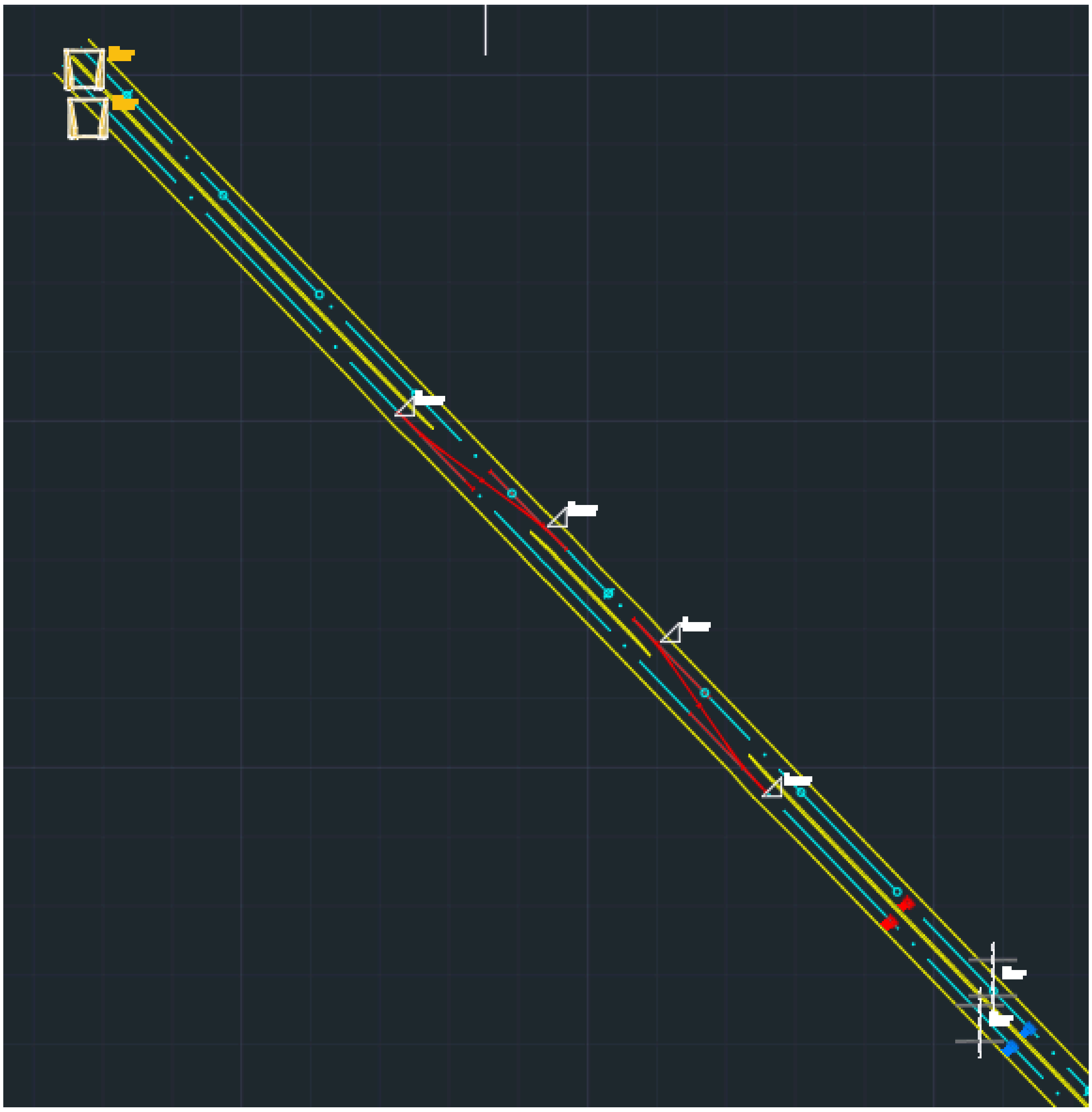
We start by using the plugin to import the railML model into the drawing. The result is given in the following picture, with an abstract view of the topological model and the elements that must be positioned in the technical drawing:



Next, we move all endpoint railML blocks (borders, switches and buffer stops) to their appropriate positions in the technical drawing. In this step, we can either execute

- the command `MOVE-RAILML-BLOCK` for each block or
- the `MOVE-ALL-RAILML-BLOCKS` command

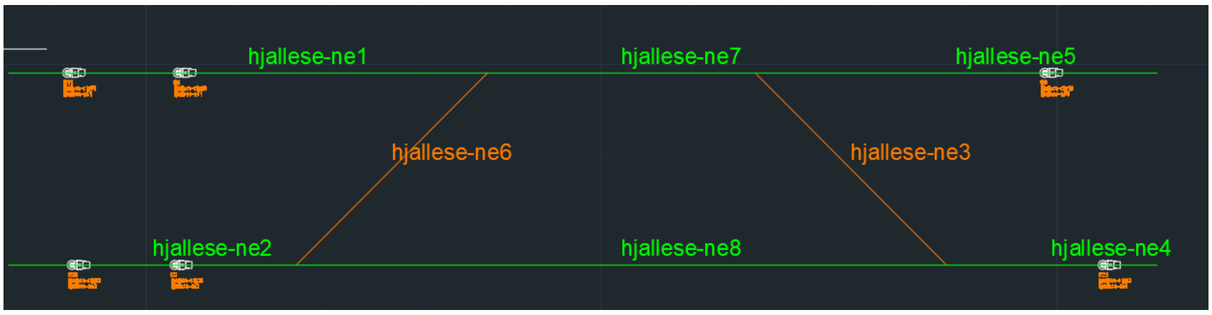
to move the railML blocks and break the drawing entities at the blocks positions. If opting for the second version, one must ensure that all entities to break belong to the same layer (in this case, we can move the bottom cyan entity to layer M16L53). At the end of this process, the drawing should look like the following picture:



The next step is to label each broken entity according to the net elements of the model. We can label each entity

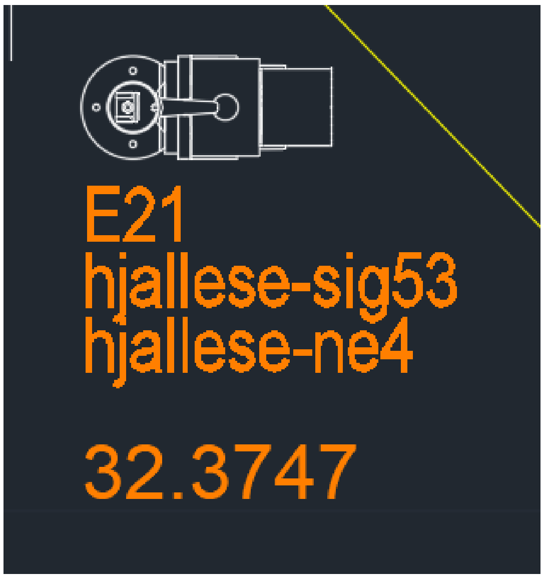
- manually with the `LABEL-NET-ELEMENT` command or
- automatically with the `AUTO-LABEL-NET-ELEMENTS` command.

Then, we can execute the command `VERIFY-LABELS` to verify if correctly labelled all broken entities. The following illustration presents one possible result of this command in the abstract topological model:

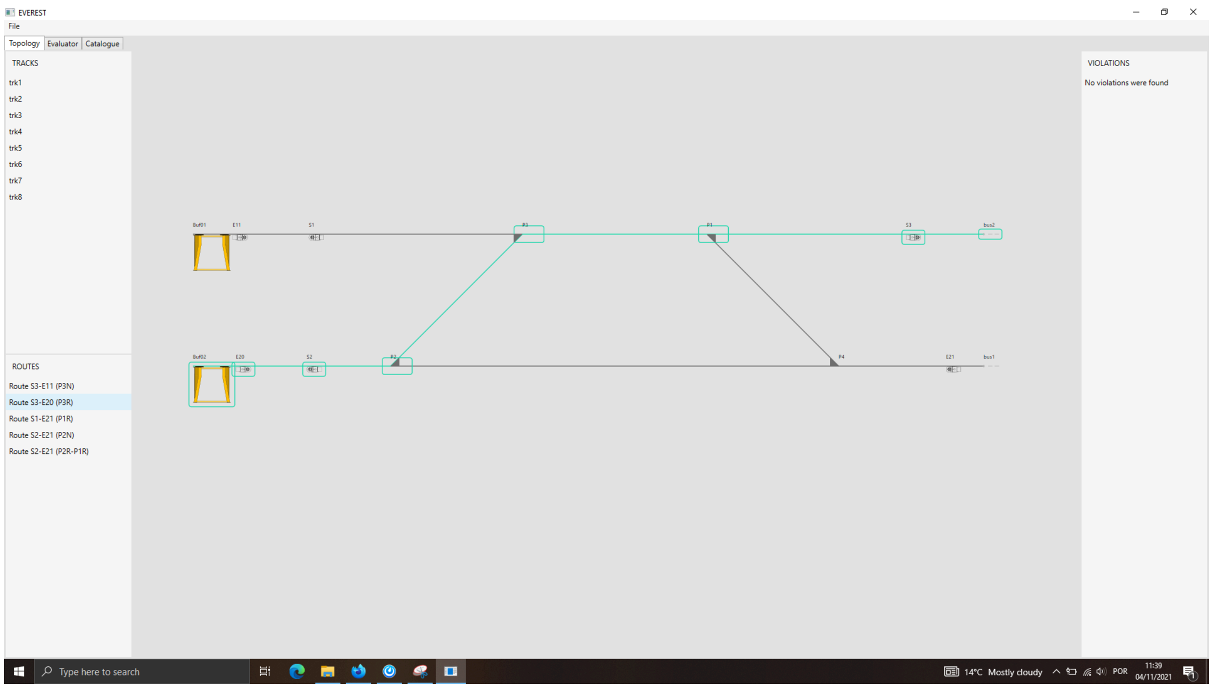


This indicates that we correctly labelled six net elements, but the net elements `hjallese-ne3` and `hjallese-ne6` are not labelled. For this case scenario, the drawing misses entities representing these two net elements. Therefore, we can execute the `DRAW-MISSING-NET-ELEMENTS` command to draw the missing entities and properly label them. If we rerun the `VERIFY-LABELS` command, all net elements should have a green colour.

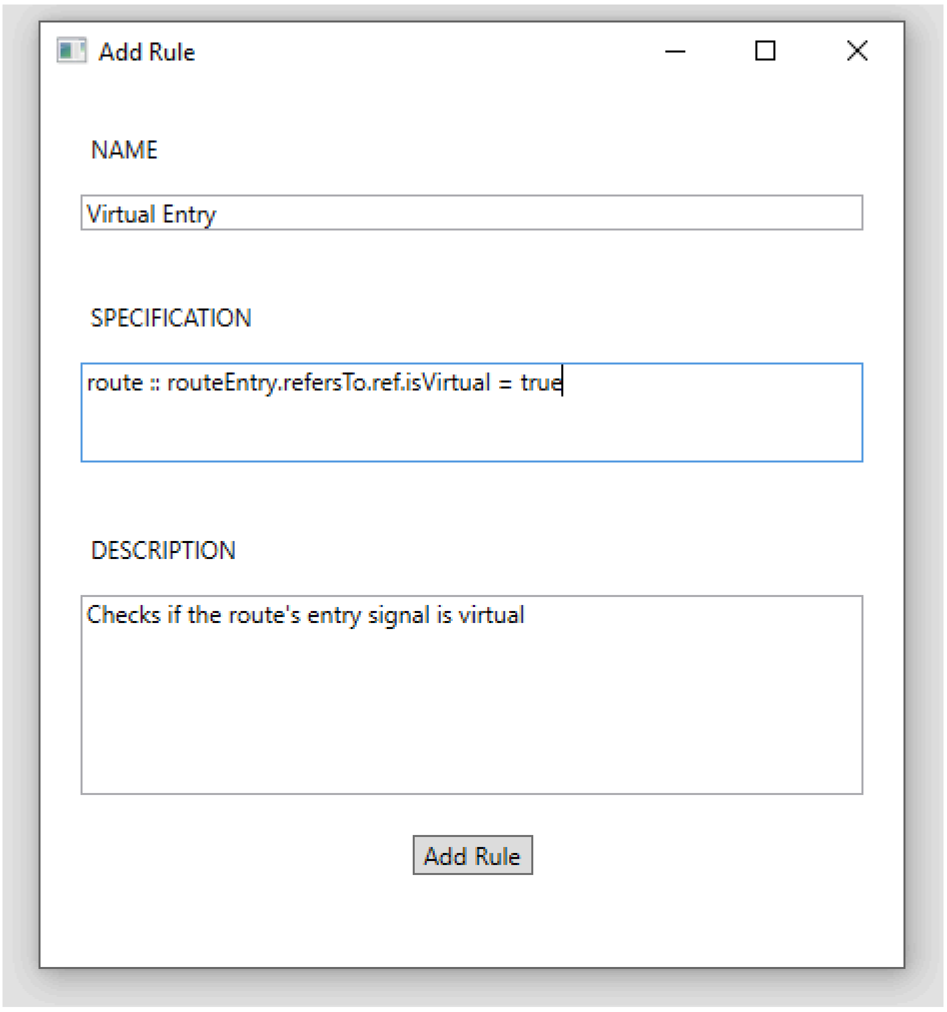
Next, we can correctly position all signals in the drawing and execute the command `EXPORT-RAILML`. This process updates the coordinates of all railML blocks and the original railML model file. Each railML block should present a position attribute like in the following picture:



We are now in the position to import the updated railML model into the EVEREST Design Verification Tool. For this we should create a new project, select a project location and add the Hjallese railML model to it. This process should present the Topology Visualizer component with the following model:



The next step is to define EVEREST rules in the Catalogue tab in order to evaluate the model. For instance, we can add a rule to check if a route's entry signal is virtual, as shown below:



NAME

Virtual Entry

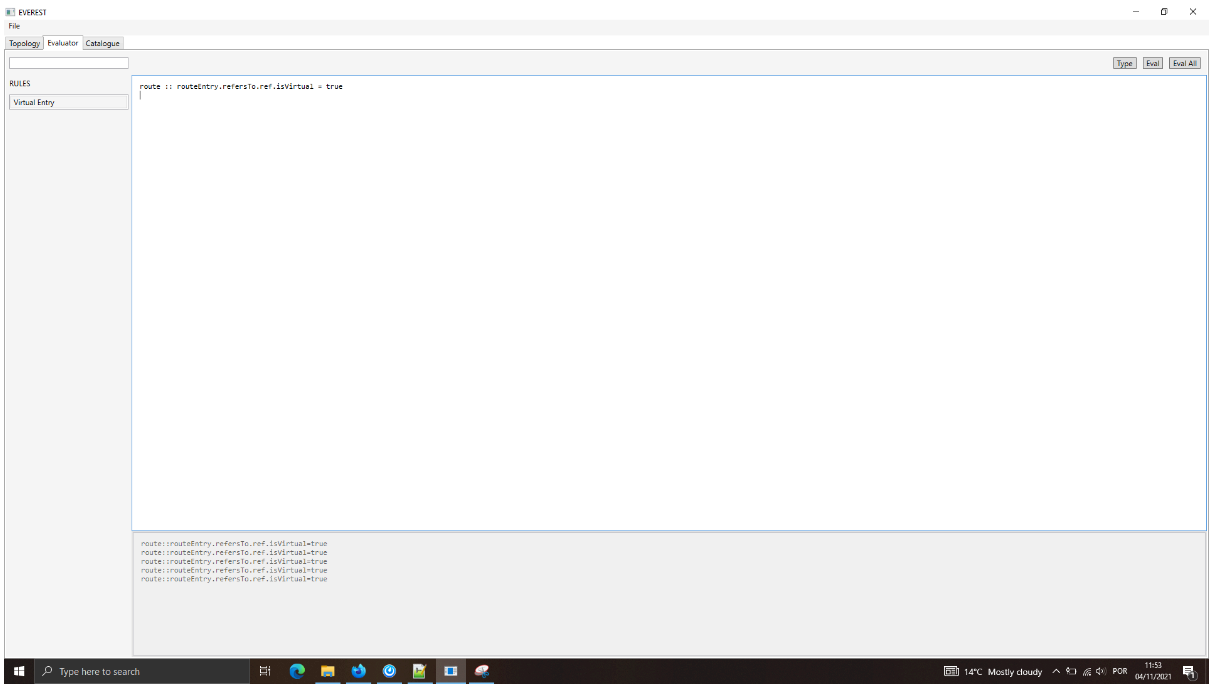
SPECIFICATION

```
route :: routeEntry.refersTo.ref.isVirtual = true
```

DESCRIPTION

Checks if the route's entry signal is virtual

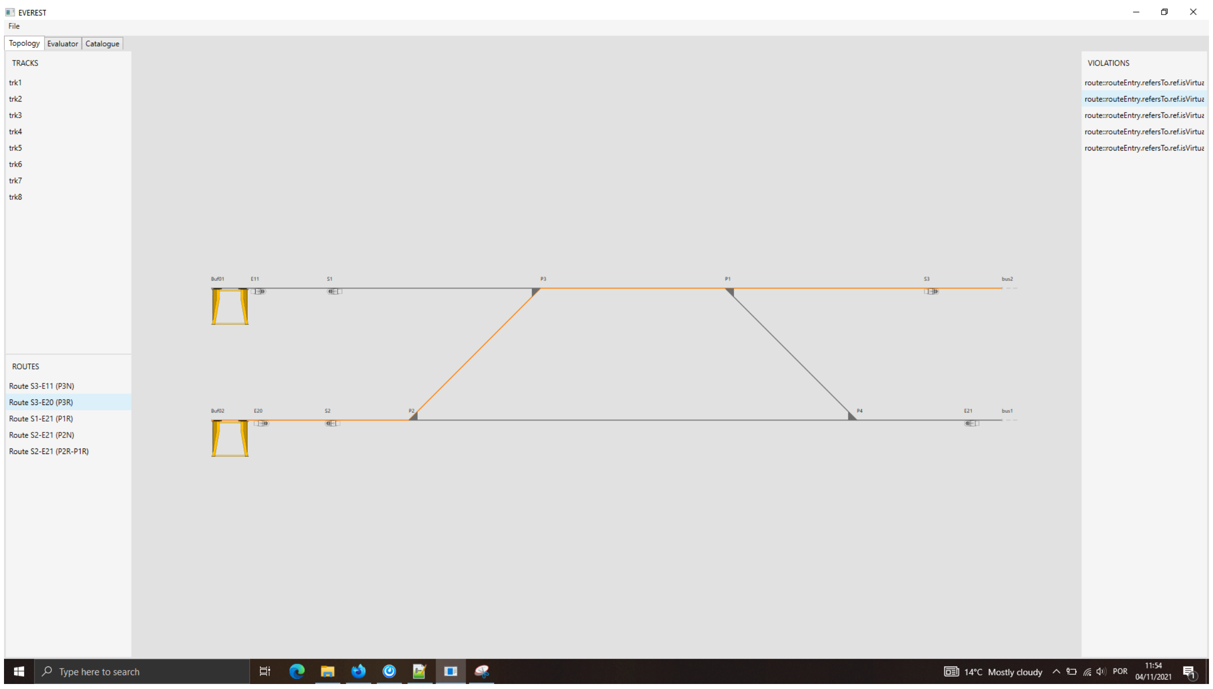
In the Evaluator tab, we can now select the added rule to evaluate the model. By clicking the Eval button, the tool should generate a violation for each model's route, since there is no entry virtual signal. The following illustration depicts the expected outcome:



```
route :: routeEntry.refersTo.ref.isVirtual = true
```

```
route::routeEntry.refersTo.ref.isVirtual=true
route::routeEntry.refersTo.ref.isVirtual=true
route::routeEntry.refersTo.ref.isVirtual=true
route::routeEntry.refersTo.ref.isVirtual=true
route::routeEntry.refersTo.ref.isVirtual=true
```

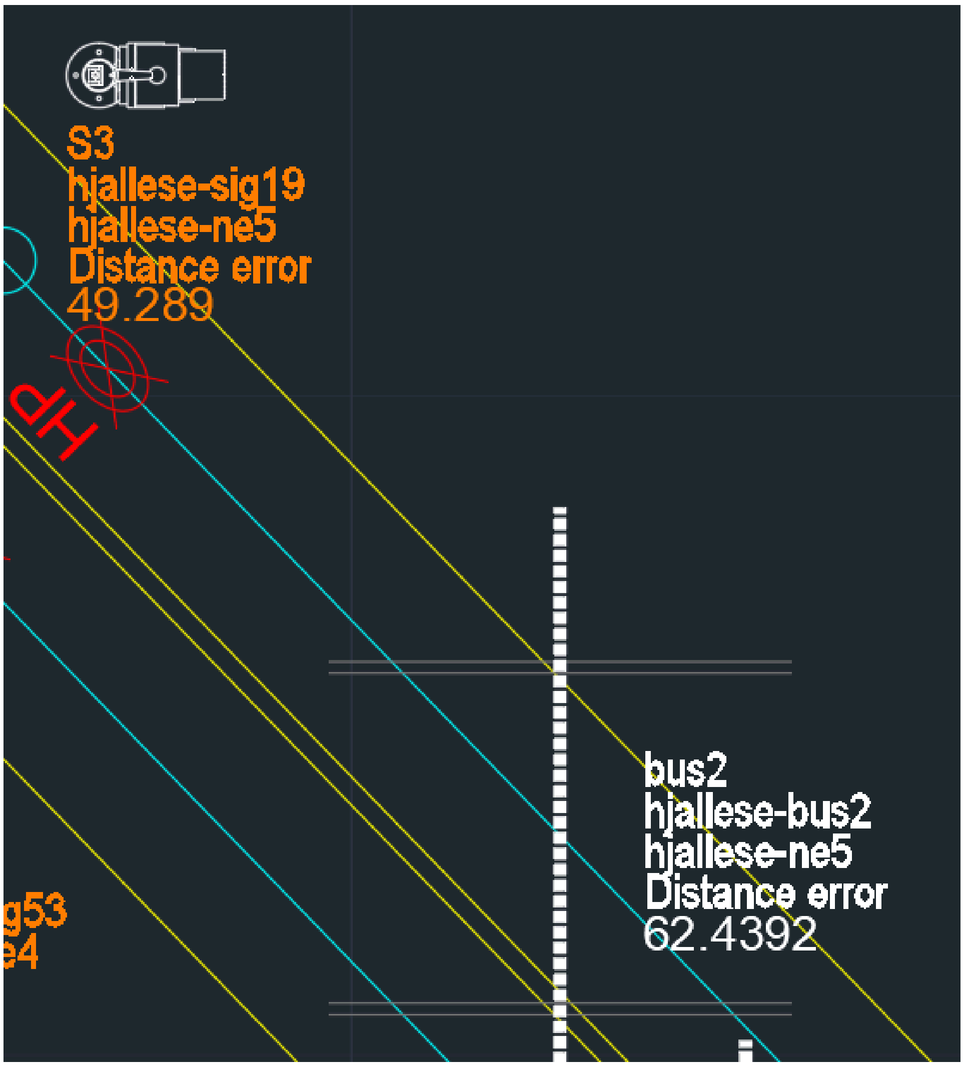
We can also check this violation graphically in the Topology Visualizer, as shown below:



Finally, we can visualize railML elements' violations in the AutoCAD drawing. Since the prototype does not support reporting of elements associated with violations, let us consider the following example violation:

```
“1”,”hjallese-”,”Distance
error”,”track”,”hjallese-ne5”,”hjallese-sig19,hjallese-bus2”
```

When we execute the `IMPORT-VIOLATION` command in AutoCAD and select the presented violation as input, the command updates the following blocks:


S3
hjallese-sig19
hjallese-ne5
Distance error
49.289
HP
g53
e4
bus2
hjallese-bus2
hjallese-ne5
Distance error
62.4392