Technical Report

- **Project**: DigiLightRail
- **WP**: WWW
- **Deliverable**: T2.6
- **Producer**: HASLab / INESC TEC
- **Title**: EVEREST tool's component specification
- **Summary**: *This report describes the EVEREST architecture and the technologies involved in its components (.....)*

Introduction

DigiLightRail aims at creating a platform for the design of command and control systems for light surface trains, 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.

The main deliverable of DigiLightRail is a design automation tool, EVEREST (**E**facec **Ve**rification of **R**ailway n**E**twork**S** **T**ool) whose goal is to automatically validate the design of railway infrastructures through user-defined infrastructure and interlocking rules. Its architecture will be a set of loosely coupled components that support the specification of rules and their verification in concrete railway topologies, including subsequent validation reporting. The main components of EVEREST 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 (IRT):** 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.

The components Topology Visualizer, Rule Designer, Topological Rule Verifier and Interlocking Rule Translator are part of the **EVEREST Design Verification Tool (DVT)**,

which provides the user a project-based interface and support for a rule catague to be shared among all projects.

This report gives details about the different technologies, as well as useful implementation details, involved in each component except for the RAS EVEREST Plugin, which is described in a companion document.

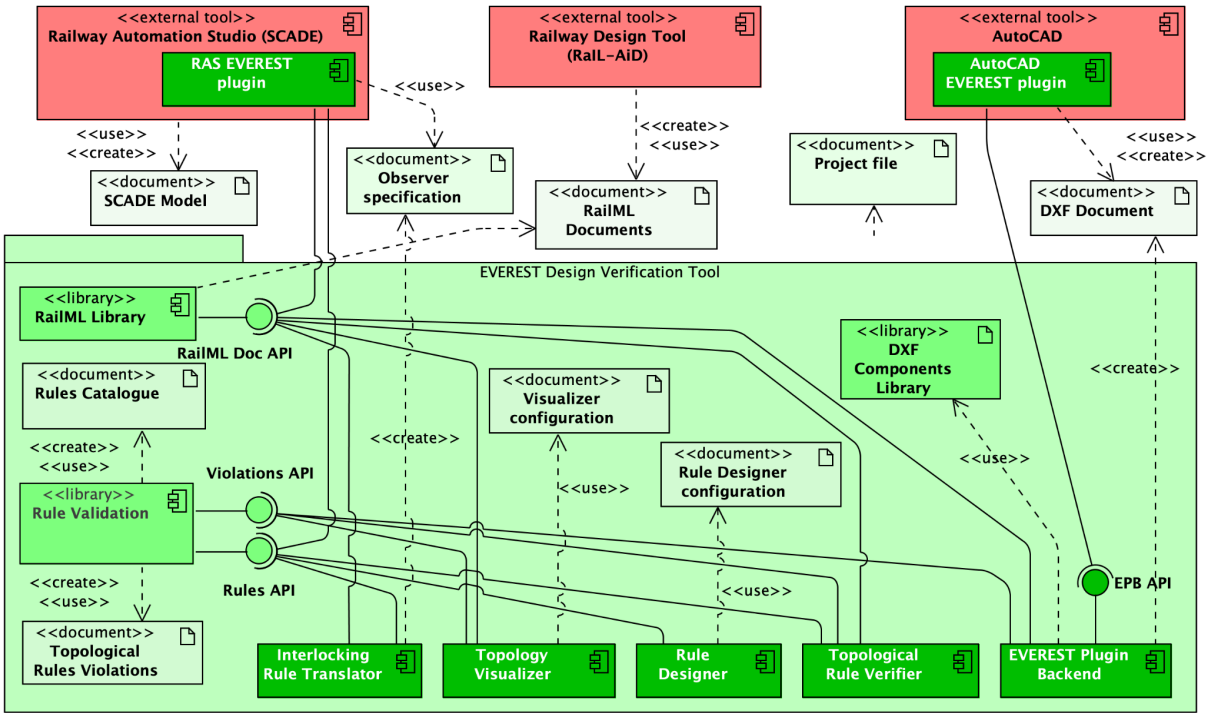
Software Architecture Specification

The EVEREST toolset accepts as input the railML models, to store information about railway model topologies, infrastructure and interlocking. As detailed in the DigiLightRail tech report T2.5 we assume some restrictions on the railML models, and, at the moment, we recommend using the RaIL-AiD editor to create and modify railML models and act as the RDT, since it obeys these restrictions. Additionally, SCADE has been selected to act as RAS in EVEREST. SCADE is able to model check interlocking rules, and is already being used in the Efacec ecosystem. Lastly, tech report T1.3 had selected the AutoCAD platform to be used to create drawings with accurate coordinates of railways and act as the TDT of EVEREST.

Figure X depicts the component diagram of the proposed EVEREST solution. The DV encompasses the main components of the platform, namely the RD, the TV, the TRV and the IRT. Moreover, a *EVEREST Plugin Backend* (EPB) provides the functionalities required by the AEP. These are developed in .NET to more seamlessly integrate with the Efacec ecosystem, and their UI relies on the *Windows Presentation Foundation* (WPF). The two plug-ins external to the DVT are implemented using different technologies. The AEP is implemented in AutoLisp, communicating with the DVT through the EPB command-line interface. The REP, whose specification is outside the scope of this document, is developed in Java to interact with SCADE's Java API, and will process *Observer specifications* created by the IRL. A *Project file* provides the overall configuration of an EVEREST project, identifying the relevant railML files and the active EVEREST rules.

All components, except the RD, interact with railML models, so an auxiliary *railML library* processes railML models and can be used to extract relevant information. This relies on the C# native XML library**,** which supports the parsing and manipulation of XML files, as well as XPath functionalities to access XML nodes easily. All these components require either handling EVEREST rules or the reported violations. Thus, a *Rule Validation Library* encapsulates rule-related functionalities, offering two APIs: one for parsing and validating EVEREST rules, implemented using ANTLR[1]; and another for creating and reading CSV files containing rule violations. Lastly, the EPB uses an asset DXF file that contains a *railML Block Library* to create DXF drawings with railML models. This relies on the netDxf[2] C# library to manipulate DXF files, create blocks and entities. The RD and the TV consider additional *configuration* files that are not expected to be modified by end users. The former contains railML typing information and expression macros, which ease the writing and validation of EVEREST rules. The latter specifies how certain elements should be drawn in the topology through a XAML description.

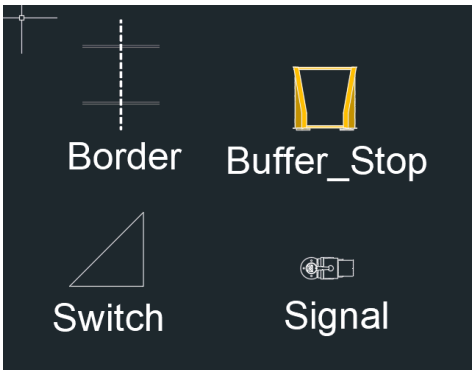
[1] https://github.com/antlr/antlr4/blob/master/doc/csharp-target.md
[2] https://github.com/haplokuon/netDxf



Library Components

railML DXF Components Library

The railML DXF components library is a set of DXF blocks representing railML infrastructure elements, such as signals, switches, and borders. The following illustration depicts examples of four of the blocks that are available in this library:



Each block of this library contains the following DXF attributes:

- **ID:** the railML `ID` attribute.
- **NAME:** the railML `NAME` attribute.
- **NETELEMENT:** the id of the net element to which the element belongs.
- **POS:** the position of the element along the net element (with a value ranging between 0 and the net element's length).
- **ERROR:** the violations resulting from a validation process, separated by the ";" delimiter.

Rule Validation Library

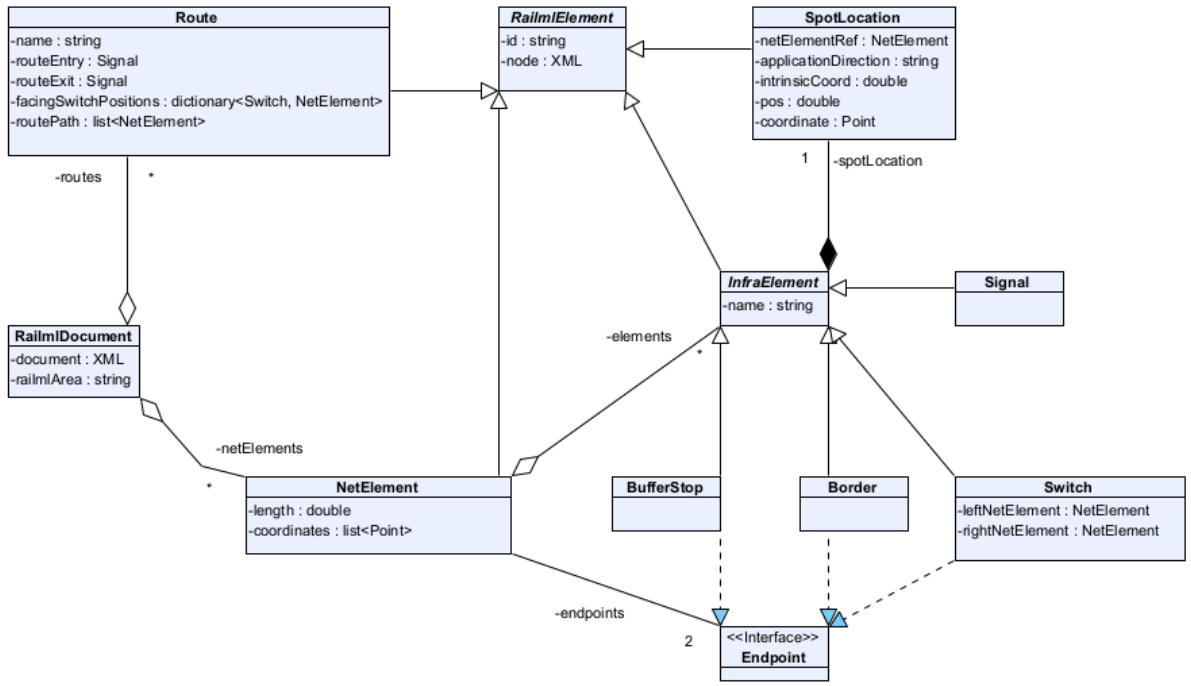
This C# library encapsulates functionalities for parsing rule-related objects, consisting in particular of two APIs, one for EVEREST rules and another for detected violations.
The rule API provides functionalities for parsing, validating, managing and traversing EVEREST rules. It uses ANTLR, a C# grammar library to implement the `CHECK-SYNTAX` functionality of the RD. ANTLR also provides a listener interface that follows the Visitor pattern, used to facilitate the traversing of generated syntax trees in the implementation of various functionalities of the RD, the TRV and the IRT.
The violations API allows the parsing and managing of sets of violations, which are encoded as CSV files. The TRV uses the API to register the violations found during the analysis (`EVALUATE-IS-RULE`), while the RD and the EPB use the API to process the violations file in order to present the violations to the users, the former in the DVT UI, and the later in AutoCAD through the EAP.

C# railML Library

This library, developed in C#, is used by most of the EVEREST DVT components. Its main goal is to process railML files, extract relevant information, and encapsulate all this information in objects.

The railML elements needed for evaluating rules, that are also extracted by this library, should be configured in the Rule Designer configuration and are extracted using a procedure specified in the Topological Rule Verifier component. Here we will focus on the extraction of the key railML elements that are necessary for the EPB, the Topology Visualizer, and to compute all the routes in a railML model (needed for various components).

The following class diagram depicts the most relevant components and attributes that are extracted by this library (concrete parser classes and all class methods were omitted for the sake of simplicity):



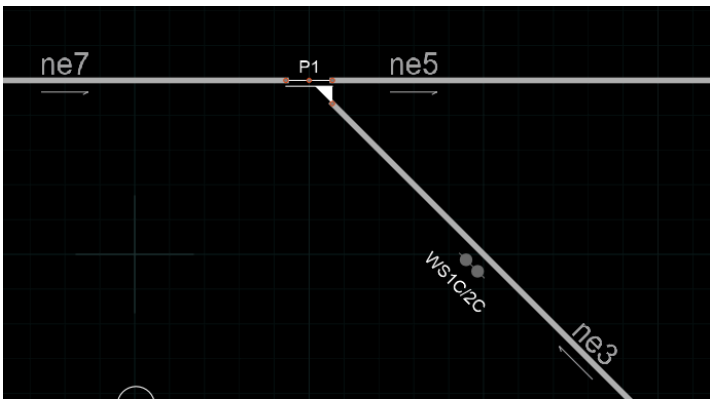
Each extracted railML element is represented by a C# class in the library, and every element inherits base properties from the `RailmlElement` abstract class. This class contains two attributes: the railML `id` and the XML node of the railML element. If the railML element does not contain an unique `id` attribute, then the library will create it. The library will extract information from the following railML elements:

- `functionalInfrastructure` (inside the `infrastructure` subschema)
- `topology` (also inside the `infrastructure` subschema)
- `assetsForInterlocking` (inside the `interlocking` subschema)

The `functionalInfrastructure` elements inherit their properties from the `InfraElement` abstract class that contains the railML `name` and the `spotLocation` as attributes. The set of extracted elements includes the elements of type `switchIS`, `signalIS`, `border`, and `bufferStop`. To distinguish endpoint elements (namely switches, borders, and buffer stops) these follow a `Endpoint` marker interface. For switches the library needs to extract additional information, namely the ids of the `netElements` of their left and right branches.

Regarding `interlocking` elements, the library extracts at least their `route` element `names`, their `entry` and `exit` elements, the `facingSwitchPositions` as a dictionary that maps `switch` ids to `netElement` ids, and computes the `route` path, represented as a sequence of `netElements`.

The `facingSwitchPositions` are helpful to compute the `route`'s path. This dictionary indicates the `netElement` the `route should` follow after reaching a `netElement` facing a switch. For example, consider the following picture that depicts three `netElements` (`ne3`, `ne5` and `ne7`) and a `switch P1`.



If a `route` reaches `netElement ne7`, which faces switch `P1`, it can go to `netElement ne5` or `ne3`. The `facingSwitchPosition` dictionary will indicate which `netElement` it should follow. By contrast, if a `route` reaches `ne3`, which does not face `P1`, it can only go to `netElement ne7`.

The following algorithm is used to extract the sequence of `netElements` of a `route`:

1. Extract the `route`'s entry element and its `netElement`.
2. Get the `netElement` entry spot location according to the `route`'s direction.
3. Add the `netElement` to the `route`'s path.
4. Check if the `route`'s exit `signal` is located along the `netElement`.
    a. If true, terminate the `route` extraction.
    b. If false, proceed to the next step.
5. Check if the opposite spot location contains an endpoint element, such as a `border` or a `bufferStop`.
    a. If true, terminate the `route` extraction.
    b. If false, proceed to the next step.
6. Due to the restrictions imposed on railML models, at this step the opposite spot location necessarily contains a `switch` element.
    a. If the `netElement` is facing the `switch` proceed to `netElement` indicated in the `facingSwitchPositions`.
    b. Otherwise, the next `netElement` is the unique `netElement` that can be reached by the switch.
7. Go back to step 2.

The described three groups of objects are stored as dictionaries with their `ids` as keys in a `RailmlDocument` object. When a user creates a `RailmlDocument` instance with a railML file, the instance uses the C# XML library to parse the relevant information of the received file.

Main Components

AutoCAD EVEREST Plugin

The EVEREST plugin of DigiLightRail's architecture is an AutoCAD extension that includes an AutoCAD plugin, developed in AutoLisp, and a Backend (EPB), developed in C#, that bridges between railML models and DWG drawings.

The AutoCAD plugin contains a set of AutoLisp commands to interact within AutoCAD. Typically, a plugin command receives the required user input, converts the DWG drawing into a DXF format and executes the EPB with all this data as parameters. It then waits for the program to finish its execution and presents the result in the DWG drawing. The result is usually a generated DXF file that can be imported into the DWG drawing. Not every command of the plugin requires service from the EPB.

The EPB can also process multiple railML files in the same drawing. Therefore, to keep the traceability between the DWG elements and the originating railML files, the EPB stores as an `XRecord` the mapping between a railML file and a unique area name. This area name is prefixed to every element id imported from a railML file.

The EPB processes the input parameters sent by the AutoCAD plugin's commands. Moreover, it uses the netDxf library to read and manipulate the received DXF files and create

the resulting drawings. Finally, if the EPB finds an error during its execution, it generates a log file containing the description of that error reported to the user by the AutoCAD plugin.

IMPORT-RAILML

Functionality	**`IMPORT-RAILML`**		
Description	*Imports a railML model into the drawing, positioning it at a given origin and using a given scale factor.*		
Input 1	`railml`	`xml`	*name of the railML file with the model to be imported. This input value is provided with a file picker.*
Input 2	`point`	`3D`	*a three-dimensional point that marks the origin to draw the imported model. Users can provide this input value by selecting any point in the drawing with AutoCAD's cursor. (NB: 3D points are represented by AutoLisp lists of three reals.)*
Input 3	`scale`	`real`	*a scale factor represented by a real value to scale the imported model.*

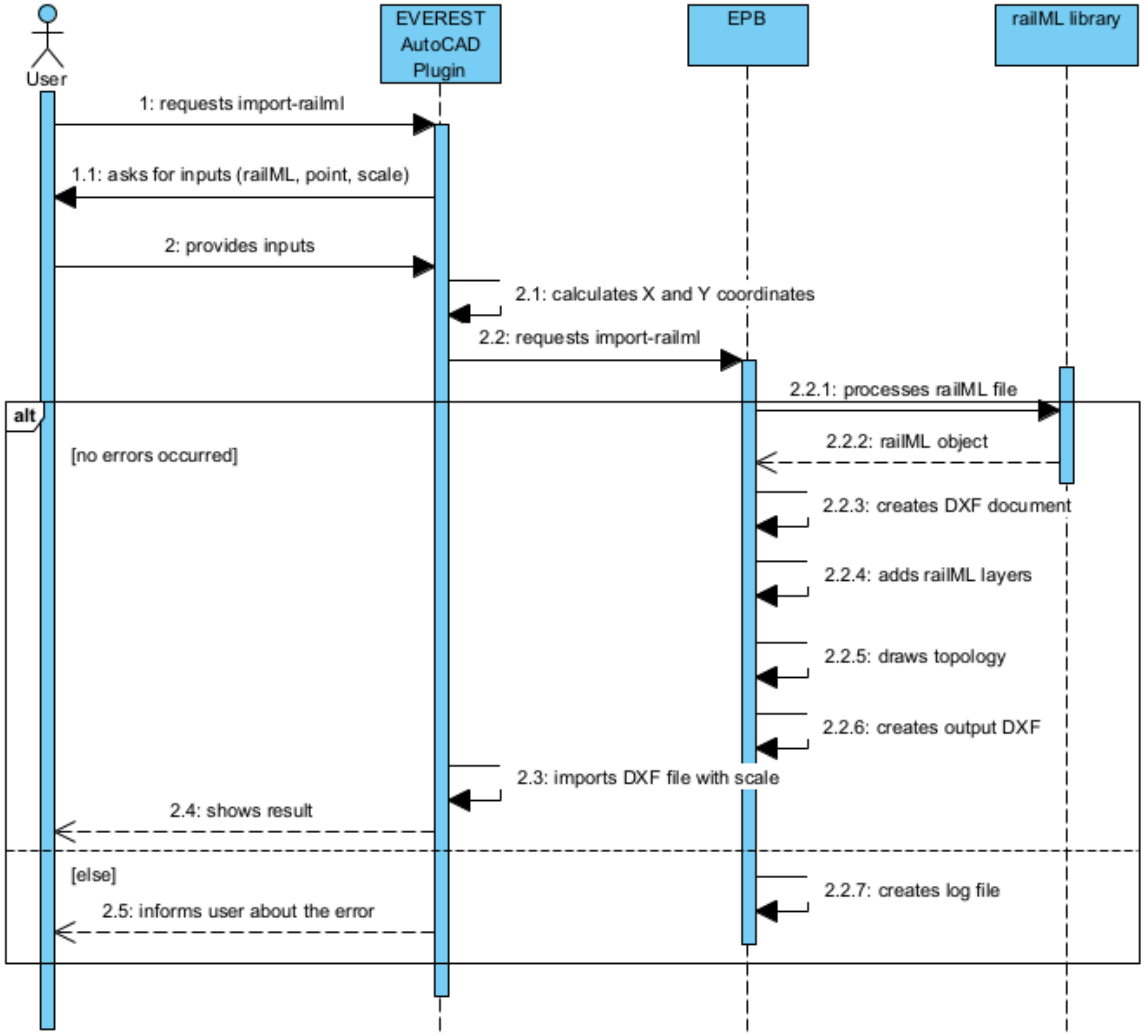
The AutoCAD plugin starts by asking for a railML file containing the logical infrastructure model to draw, the origin point to draw this model and its scale. Then it computes the true X and Y coordinates of the origin by applying the scale factor to the received point. Next, it invokes the EPB, sending both the railML file name and the (X, Y) coordinates to draw the logic model.

The EPB uses the underlying C# railML library to parse the railML file and extract the net elements and all infrastructure elements from it, creating a DXF document with the assistance of the netDxf library containing all the block definitions of the DXF railML block library. Next, it adds two layers to the drawing: the "`railml_net_elements`" that will contain all net elements and the "`railml_elements`" that will contain all infrastructure elements.

The program adds an `LWPolyline` entity for each net element of the railML model. These are drawn according to the positions defined by visualization elements following a `screenPositioningSystem` in the railML plus the received (X, Y) coordinates. Moreover, the program adds a text label with the net element's id approximately at the mid-point of the drawn `LWPolyline`.

For the infrastructure elements, the program performs a mapping from the railML tags of these elements of the model to the available blocks of the DXF railML block library. These are represented by block references and drawn according to the positions defined by visualization elements following a `screenPositioningSystem` plus the received (X, Y) coordinates. The block `NAME`, `ID`, and `NETELEMENT` attributes receive the appropriate railML values, whereas the `POS` and `ERROR` attributes are assigned an empty string value.

Finally, after the described process of the EPB, the AutoCAD plugin imports, with the scale factor, the created DXF drawing containing the logical infrastructure model into the DWG drawing. The following diagram illustrates all this process:



MOVE-RAILML-BLOCK

Functionality	**`MOVE-RAILML-BLOCK`**		
Description	*Moves an endpoint railML block to the nearest position of an entity and breaks that entity at that position.*		
Input 1	`railml-block`	`str`	*name of the railML block to move. Users can provide this input by selecting the desired block with AutoCAD's cursor*
Input 2	`entity`	`entity`	*an entity representing the netElement to break. Users can provide this input by selecting the desired entity with AutoCAD's cursor. (NB: entity objects are represented by AutoLisp association lists of DXF group codes and property pairs.)*

The command starts by asking the user for the entity to break and the endpoint block that will break that entity. Then, it calls the `vlax-curve-getclosestpointto` method to compute the point of the selected entity that stands closest to the railML block origin point. Finally, it uses the `vla-move` method to move the block to the closest point and uses the AutoCAD's `BREAK` command to break the selected entity at that point.

MOVE-ALL-RAILML-BLOCKS

Functionality	**`MOVE-ALL-RAILML-BLOCKS`**		
Description	*Moves all endpoint blocks to the nearest entities of a layer and breaks those entities at those positions.*		
Input 1	`layer`	`str`	*name of the layer that contains the entities to break. Users can provide this parameter by selecting any entity of the desired layer.*

This command is an iterative version of the `MOVE-RAILML-BLOCK` command. It moves all railML blocks representing railML entities, such as switches, borders and buffer-stops located at the endpoints of net elements. This command assumes that every entity to break belongs to the same DWG layer, provided as input.

The command asks the user for the layer name containing all entities to break. Then it loops through all railML blocks to move. For each railML block, it loops through all entities of the selected layer and computes their closest point and distance to the current block with the `vlax-curve-getclosestpointto` and `distance` methods. Then, the command moves the current railML block to the entity with the minimum distance and breaks that entity and the calculated closest point. The following listing presents the pseudo-code of the described process:

```
var layer-name = input("Select layer name")
var layer-entities = ssget(layer-name)

for block in railml-blocks do
	var min-dist = MAX_VALUE
	var selected-entity = nil
	var min-point = nil

	for entity in layer-entities do
		var closest-point = vlax-curve-getclosestpointto(entity, block)
		var dist = distance(closest-point, block.point)

		if dist < min-dist do
			min-dist = dist
			selected-entity = entity
			min-point = closest-point

	vla-move(block, block.point, min-point)
```

```
BREAK(selected-entity, min-point, min-point)
```

LABEL-NET-ELEMENT

Functionality	**LABEL-NET-ELEMENT**		
Description	*Labels a net element entity in the drawing with its corresponding net element's Id.*		
Input 1	`entity`	`entity`	*an entity representing a railML net element. Users can provide this input by selecting the desired entity with AutoCAD's cursor. (NB: entity objects are represented by AutoLisp association lists of DXF group codes and property pairs.*
Input 2	`netElement`	`str`	*net element's id.*

This command labels a net element entity in the drawing with its corresponding net element's Id by attaching an Xdata string value with the name "`NET_ELEMENT_ID`" to that entity.

AUTO-LABEL-NET-ELEMENTS

Functionality	**AUTO-LABEL-NET-ELEMENTS**		
Description	*Labels all net element entities in the drawing with their corresponding net element's Id.*		
Input 1	`railml`	`xml`	*railML model, which should have been previously imported with the* **IMPORT-RAILML** *command.*

This command is an automatic version of the `LABEL-NET-ELEMENT` command that attempts to label all entities in the DWG drawing representing net elements with their corresponding net element's Id, taking into consideration the railML elements at their endpoints.

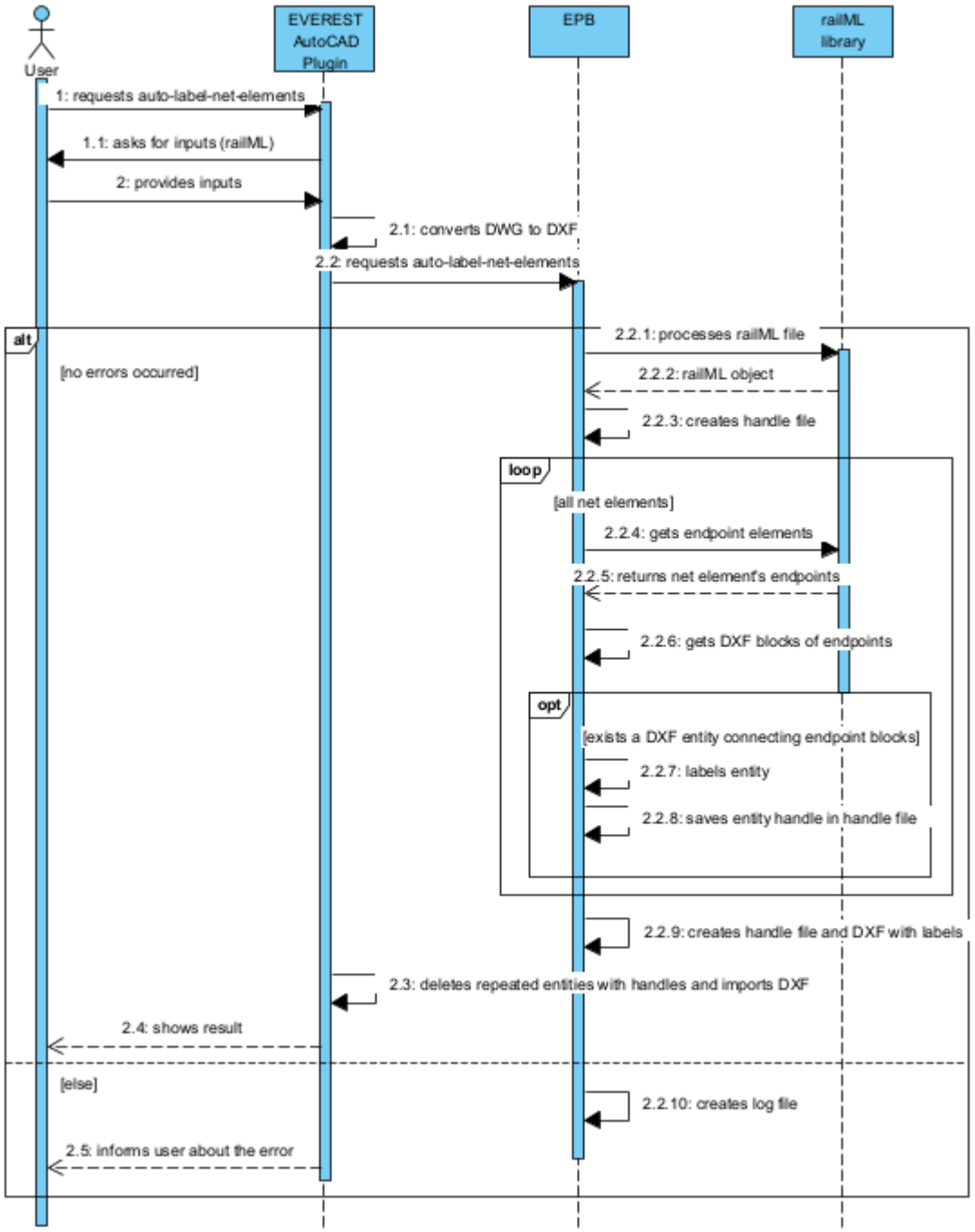
The command asks the user for the railML file holding the railway model imported with the `IMPORT-RAILML` command. Then it converts the DWG drawing into a DXF file format and invokes the EPB with this converted file and the railML file as parameters.

The EPB loops through every net element of the railML topology and checks, for each net element, if there is one entity in the DXF drawing that connects two railML blocks representing the net element's endpoints. In that case, the program draws a copy of the matched entity and introduces an Xdata label in the DXF drawing. This Xdata will be added to the original element in the DWG when the DXF is imported. Moreover, it stores in a text file the DXF handle of the matched entity. This value will be used later by the AutoCAD plugin to delete repeated entities.

The EPB performs the following steps to extract the drawing's entity that represents a net element:

1. Gets the railML element's ids representing the net element's endpoints.
2. Finds the railML blocks in the drawing that represent the endpoint elements.
3. For each entity, it checks if all endpoints of that entity intersect the railML blocks representing the net element's endpoints.
4. An entity intersects a railML block if one of its endpoints is located inside a circular region with its centre equals the block's insertion point. The region's radius is a value that can be user configured (default being five units). This command will be less precise with larger radius values.

The EPB's result consists of two files: a DXF file containing all labelled entities and a text file containing the DXF handles of the old entities labelled in the process. The AutoCAD plugin then collects all entities with the DXF handles of the DWG drawing and removes them to prevent duplicated entities. Then, it imports all entities of the EPB DXF output file to the DWG drawing. The following diagram presents the overall process of this command:



DRAW-MISSING-NET-ELEMENTS

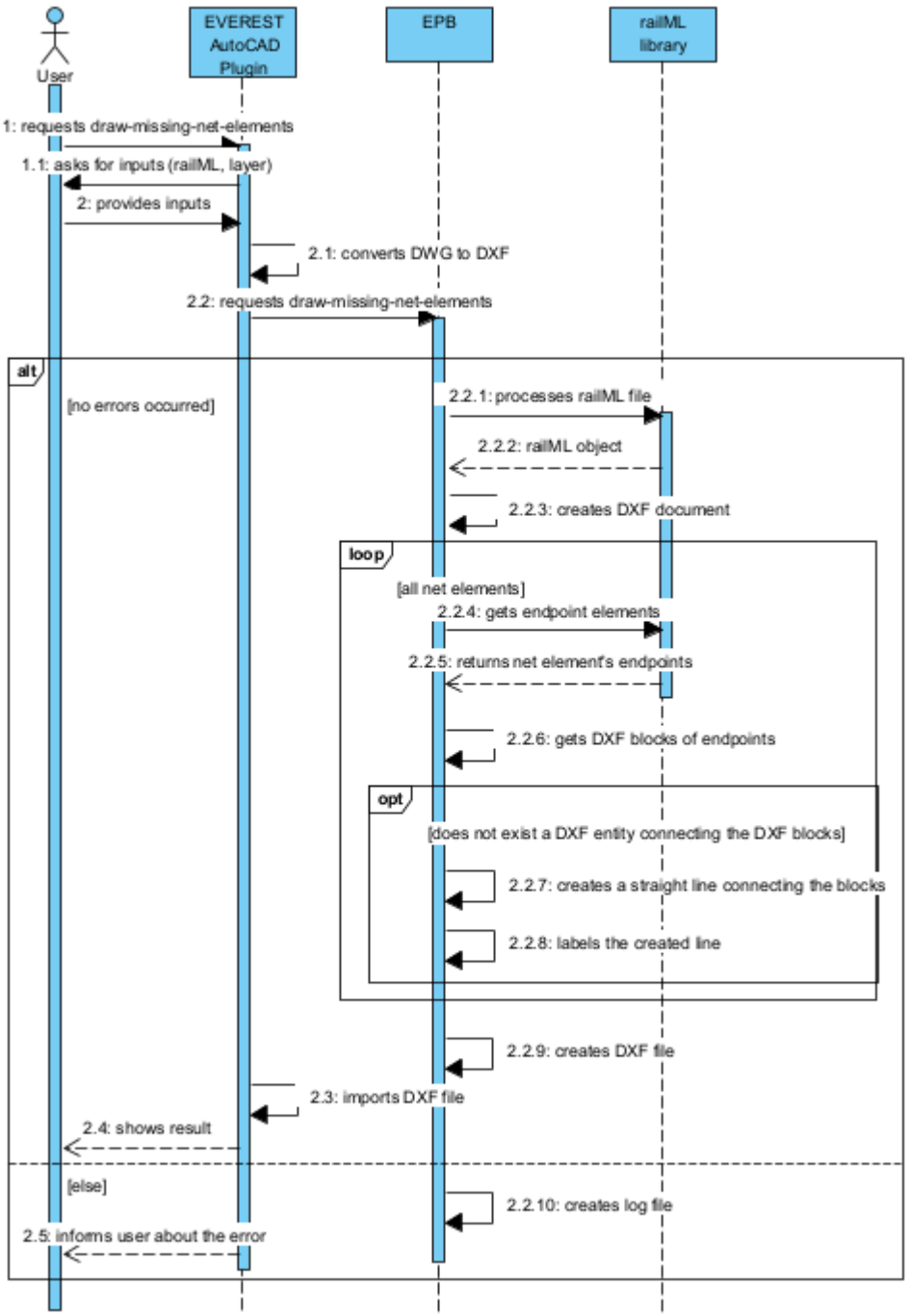
Functionality	**`DRAW-MISSING-NET-ELEMENTS`**		
Description	*Draws any missing net elements of the drawing and labels these with their corresponding net element's Id.*		
Input 1	`railml`	`xml`	*railML model, which should have been previously imported with the* **`IMPORT-RAILML`** *command.*

Input 2	`layer`	`str`	*name of the layer to draw the missing net elements of the drawing. Users can provide this input value by selecting any entity of the desired layer.*

This AutoCAD plugin command asks for the same input values of the `AUTO-LABEL-NET-ELEMENTS` command and invokes the EPB with these values as parameters.

Then, EPB loops through all net elements of the railML model and checks if one entity in the DXF drawing connects two railML blocks representing the net element's endpoints in the same fashion as the `AUTO-LABEL-NET-ELEMENTS` command. If there is no such entity, then the program draws a straight `LWPolyline` entity connecting the two railML blocks representing the endpoints of a net element. The two vertices of the added `LWPolyline` are the insertion points of the two railML endpoint blocks. Furthermore, the added entity is also labelled with the corresponding net element's id. Thus, the result of this process is a DXF file containing the added entities.

Then, the AutoCAD plugin collects the produced DXF file and adds its content to the DWG drawing. The following diagram presents this process:



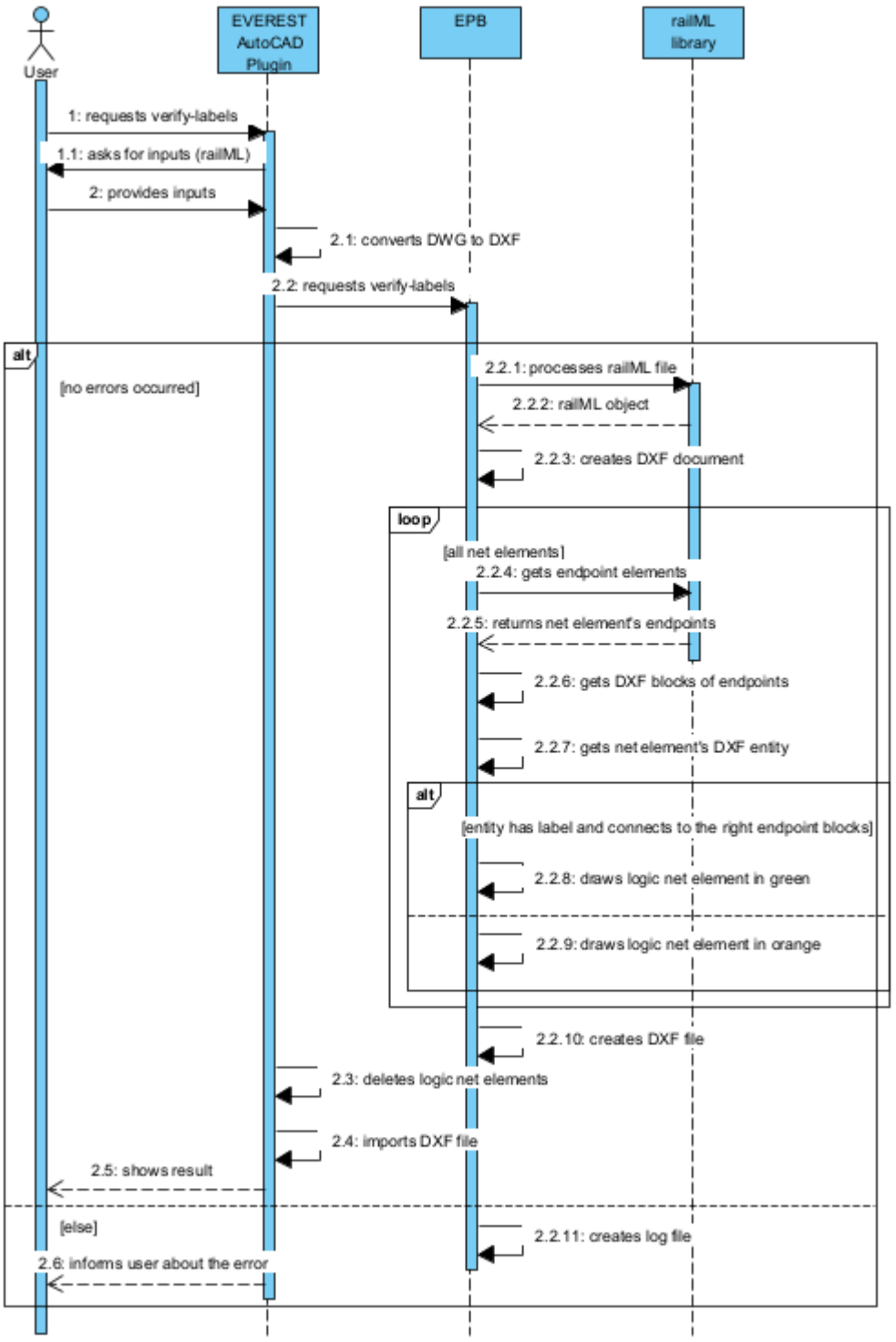
VERIFY-LABELS

Functionality	**`VERIFY-LABELS`**		
Description	*Checks if all net elements of the drawing are appropriately labelled.*		
Input 1	`railml`	`xml`	*railML model, which should have been previously imported with the* **`IMPORT-RAILML`** *command.*

First, the AutoCAD plugin asks for the railML file containing the railway model and converts the drawing to DXF file format. Then it invokes the EPB with the railML file and the converted DXF as parameters.

This command creates a DXF file containing a logical drawing with the same net elements produced with the `IMPORT-RAILML` command, but with colours indicating whether they have been correctly labelled. The EPB loops through all net elements of the railML model and checks if the DXF drawing contains any entities representing them in the same fashion as the `AUTO-LABEL-NET-ELEMENTS` command. If it did not find any entity representing a net element or if the entity has an incorrect label, then the program draws a logical net element entity with an orange colour. Otherwise, if the backend finds an entity with a correct label, it draws a logical net element entity with a green colour.

Afterwards, the AutoCAD plugin removes all entities belonging to the "`railml_net_elements`" drawn with the `IMPORT_RAILML` command, and imports the newly generated DXF file containing the same entities but with the colours indicating their labelling state. The following diagram illustrates this process:



EXPORT-RAILML

Functionality	**`EXPORT-RAILML`**
Description	*Updates the railML model with the position of all infrastructure elements and computes all net element's lengths.*

Input 1	`railml`	`xml`	*railML model, which should have been previously imported with the* **`IMPORT-RAILML`** *command.*

The command starts by computing the length of all net elements entities in the drawing. First, it selects every entity with the AutoLisp `ssget` method that contains an Xdata value named "`NET_ELEMENT_ID`". Then it uses the methods `vlax-curve-getDistAtParam` and `vlax-curve-getEndParam` to calculate the length of an entity. Finally, it stores the length of an entity by attaching an Xdata value with the name "`NET_ELEMENT_LENGTH`". The following listing presents the pseudo-code of the method `update-net-elements-lengths`:

```
var net-elements = ssget("NET_ELEMENT_ID")

for net-element in net-elements do
      var length = vlax-curve-getDistAtParam(net-element,
            vlax-curve-getEndParam(net-element))
      add-xdata(net-element, "NET_ELEMENT_LENGTH", length)
```

After computing all the net elements' lengths, the plugin calculates the "`POS`" attribute of all railML blocks of the drawing with the following steps:

1. It selects all railML blocks with the `ssget` method and, for each block, it gets the entity referring to the net element pointed by the block's "`NETELEMENT`" attribute.
2. It uses the `vlax-curve-closestpointto` to calculate the nearest point of the entity to the railML block.
3. It uses the `vlax-curve-getdistatpoint` to compute the distance between the nearest point and the entity's origin. Note that the entity's origin could be inverted in the drawing, leading to inverted "`POS`" values. However, the EPB can analyse the model and fix wrong "`POS`" values.
4. It stores the calculated distance in the block's "`POS`" attribute.

The following listing presents the pseudo-code of this process:

```
var railml-blocks = ssget("Signal", "Border", "Buffer_Stop", "Switch")

for block in railml-blocks do
      var net-element-id = get-attribute-value(block, "NETELEMENT")
      var      net-element-ent      =      get-by-xdata-value("NET_ELEMENT_ID",
net-element-id)
      var nearest-point = vlax-curve-getclosestpointto(net-element-ent, block)
      var dist = vlax-curve-getdistatpoint(net-element-ent, nearest-point)
      set-attribute-value(block, "POS", dist)
```

Afterwards, the AutoCAD plugin asks for the railML file containing the railway model, converts the DWG drawing into a DXF file format and invokes the EPB with these two parameters.

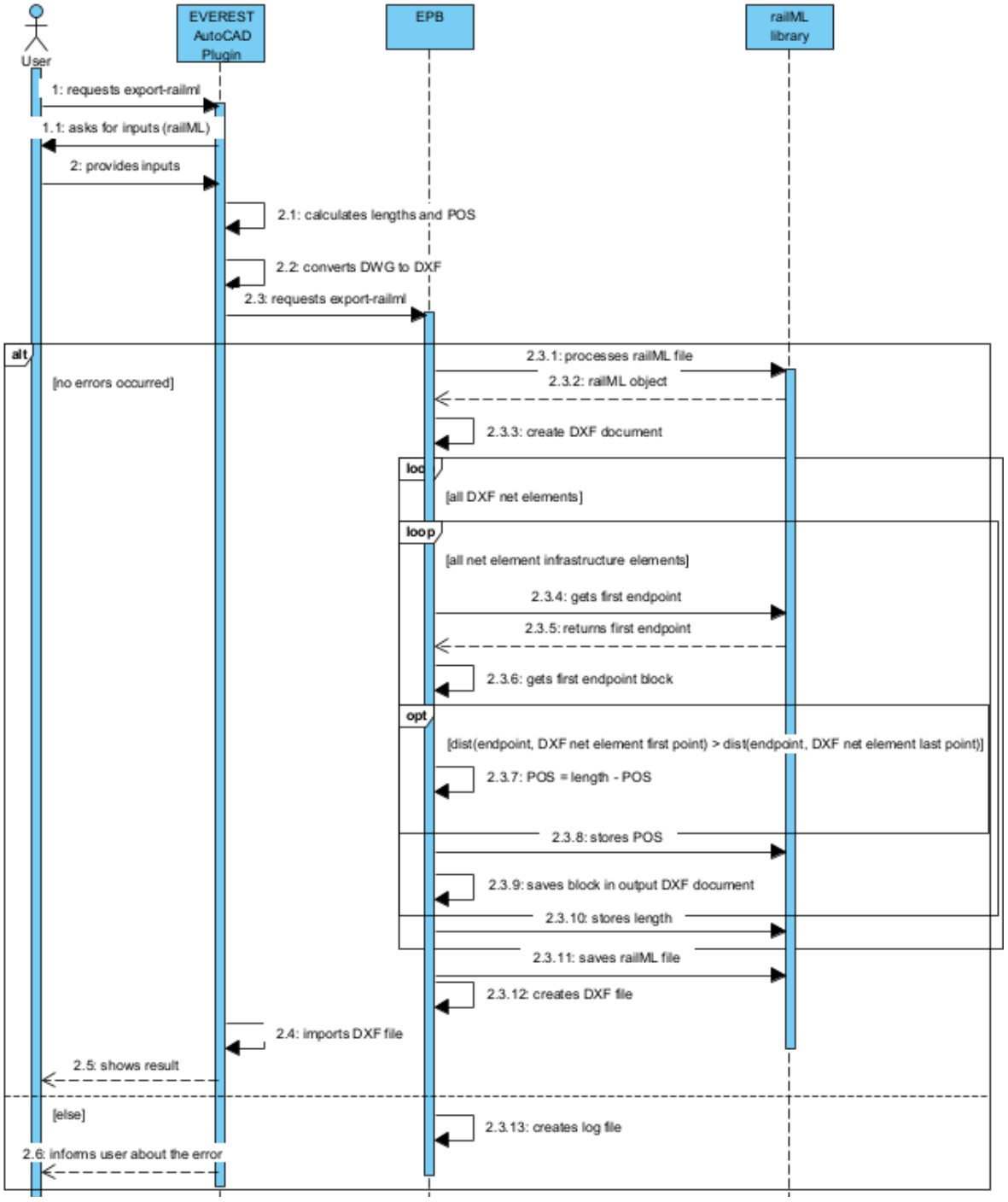
The EPB loops through all net elements entities of the received drawing and, for each entity, gets the railML blocks referring to the infrastructure elements located in the entity's net

element by using the C# railML library. Then, for each block, it checks if its “POS” value is inverted and fixes that value. The backend performs the following steps to check if the “POS” value is inverted:

1. Through the railML library, it gets the railML block referring to the net element's begin endpoint.
2. Checks if the distance in the drawing between that block and the net element's DWG entity's first point is greater than the distance in the drawing between that block and the DWG entity's last point.
3. If the latter condition is true, then the beginning of the railML net element is swapped in the respective DWG entity, and thus the retrieved “POS” value is inverted. In that case, the program sets this value equals to `entity.length - POS`.

Afterwards, it updates the received railML file with the correct “POS” attributes and the lengths of all net elements. Furthermore, it creates a DXF file containing all railML blocks with the updated “POS” attributes' values.

Finally, after the EPB’s execution, the AutoCAD plugin deletes all railML blocks to prevent duplication and imports the resulting DXF file into the drawing. The following diagram depicts the whole process of this command:



IMPORT-VIOLATIONS

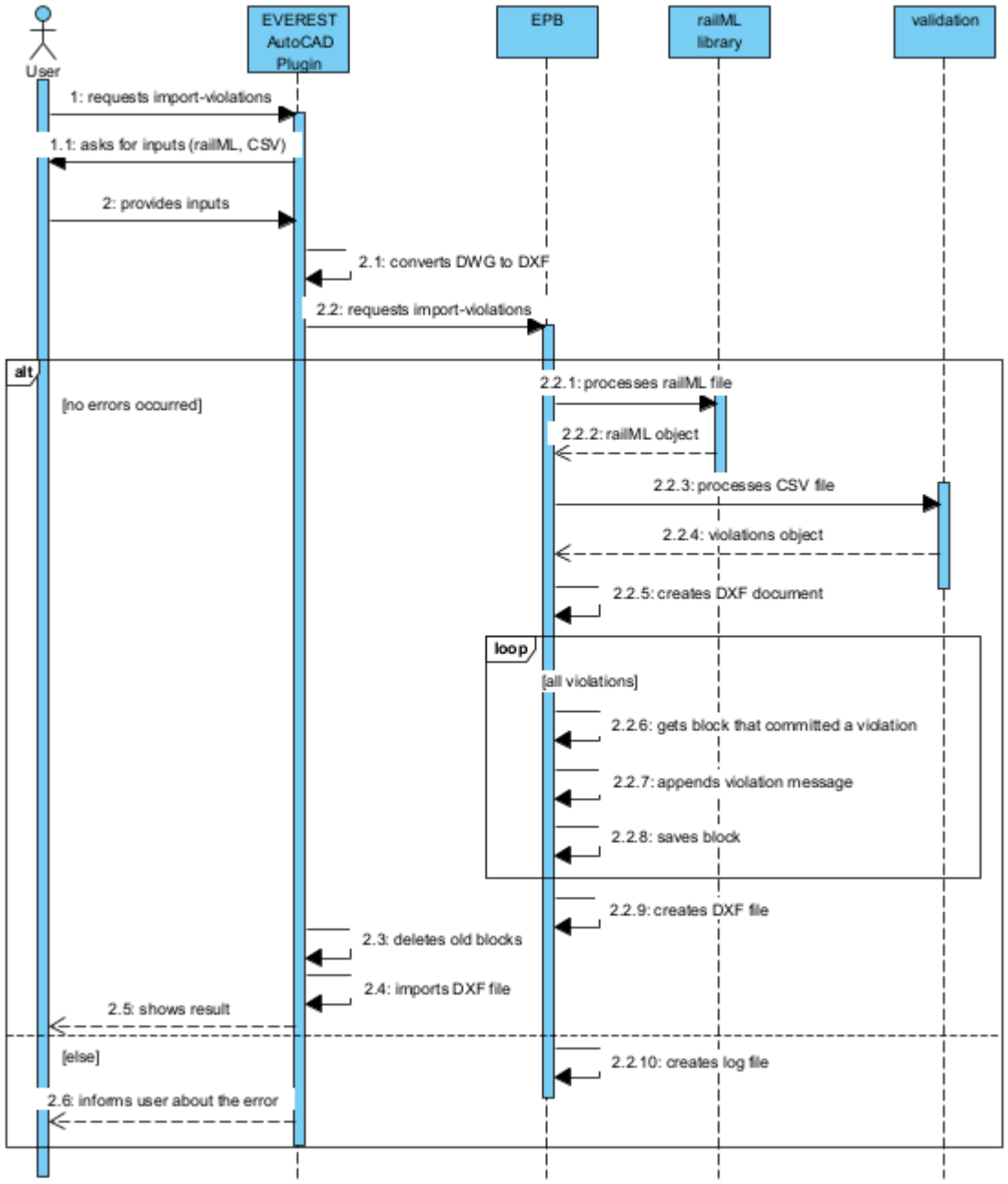
Functionality	**`IMPORT-VIOLATIONS`**		
Description	*Depicts in the drawing all violations resulting from the validation of a railML model.*		
Input 1	`railml`	`xml`	*railML model, which should have been previously imported with the* **`IMPORT-RAILML`** *command.*

Input 2	`violations`	`csv`	*name of the CSV file containing a set of violations resulted in the validation process. This input value is provided with a file picker.*

This command should be only executed after running the `EXPORT-RAILML` command and validating the railway model with the Topological Rule Verifier component. First, the AutoCAD plugin asks for the railML file containing the railway model and the CSV file holding the violations resulting from the validation process. Next, it converts the drawing into a DXF file format and invokes the EPB with all this data as parameters.

Then, the EPB processes the CSV file and extracts the railML blocks in the drawing that contain violations, appending any violation into the `ERROR` attribute of the corresponding railML block. The result is a DXF containing all railML blocks with the violations held by the received CSV file.

Finally, the AutoCAD plugin deletes all railML blocks to prevent duplicates and imports the resulting DXF file into the drawing. The following diagram illustrates the whole process of this command:



EVEREST Design Verification Tool

The EVEREST DVT provides a unified interface for most EVEREST functionalities (except those triggered by the external plugins, the AEP and REP), and can be used to

- manage the catalogue of EVEREST rules;
- manage an EVEREST project.

The DVT keeps track of a catalogue of rules, written using the Rule Designer. Basic versioning is also provided for this catalogue of rules, supporting two operations:

- **Duplicate Rule**, which creates a fresh rule cloned from a pre-existing one;
- **New Version**, which creates a new version for an existing rule, rendering the previous version deprecated.

Internally, this makes use of an additional configuration for the Rule Designer. This *Rule Designer configuration* consists of two CSV files, one containing the definitions of rule patterns and another with railML expression macros and the type of railML entities to be extracted from railML files (for more information about the types see the specification of the *Rule Designer* component). These configuration files are not expected to be modified by the end-users of the tool. An example of the pattern configuration follows. Each line consists of a pattern identifier, its description and the respective specification.

```
noNeighbours, “Wherever a $X element appears, there is no other $X for $Y meters along the route.”, route :: everywhere (some $X implies everywhere [0..$Y] no $X)
```

An example of the railML entities configuration file is presented below. For railML entities each line describes its identifier and its type. For macros, we also have the definition

```
routeEntry, {<route,routeEntry>}
refersTo, {<routeEntry,refToSignalIL>,<signalIL,refToSignalIS>,...}
ref, {<refToSignalIL,signalIL>,<refToSignalIS,signalIS>,...}
entrySignal, {<route,signalIL>}, (routeEntry.refersTo).ref
```

An EVEREST project is essentially a set of railML files, representing different zones of the topological model and a set of active rules selected from the catalogue. New projects are created from the DVT menu, initially containing no railML files nor active rules. railML files are added through the DVT menu, at which point the user can activate EVEREST rules in the rule catalogue, to be verified through the Topological Rule Verifier and Interlocking Rule Translator. This information is stored in an *EVEREST project file* that specifies the set of railML files that belongs to each project, and the set of active rules from the catalogue. If a project points to a rule that has become deprecated due to the release of a new version, a warning is raised. For each project, a set of violations last detected by the Topological Rule Verifier is also preserved. Although a project may be composed of multiple railML files, the Topology Visualizer, Topological Rule Verifier and Interlocking Rule Translator act on a single railML file at a time. The DVT manages the needed calls to `EVALUATE-IS-RULE` and `TRANSLATE-IL-RULE`, and allows the user to select which railML file (and associated violations) should be depicted by the Rule Visualizer at each moment.

Topology Visualizer

This component provides a UI in the DVT that:

- allows users to visualize the network's topology of multiple railML models (`DRAW-TOPOLOGY`);
- provides a railML route table that users can interact with to highlight tracks and routes in the topology drawing (`HIGHLIGHT-SCOPE`);
- illustrates any violations that resulted from the EVEREST validation process. All these violations are depicted in a violations table (`SHOW-VIOLATION`).

This component is based on WPF, and its UI is a window containing a sidebar with the routes and violations and a `Canvas` containing the railway topology. It relies on the C# railML

library to process input railML files and draw its topology, and on the Topological Rule Verifier component to compute all violations. Mockups of the UI for the various operations of this functionality are provided below.

Finally, it uses XAML resource files to draw infrastructure elements that resemble the railML blocks of the DXF library. To generate one of these resources, we need to perform the following steps:

1. In AutoCAD, convert the desired railML block to PDF.
2. Convert the PDF file into an SVG format. Several tools, such as Inkscape[3] and Adobe Illustrator[4], can assist in this step.
3. Use the SvgToXaml[5] program to convert the SVG file into a XAML resource file.

DRAW-TOPOLOGY

Functionality	**`DRAW-TOPOLOGY`**		
Description	*Draws the network's topology of a railML model.*		
Input 1	`railml`	`xml`	*railML model containing the network's topology. This input value is provided with a file picker.*

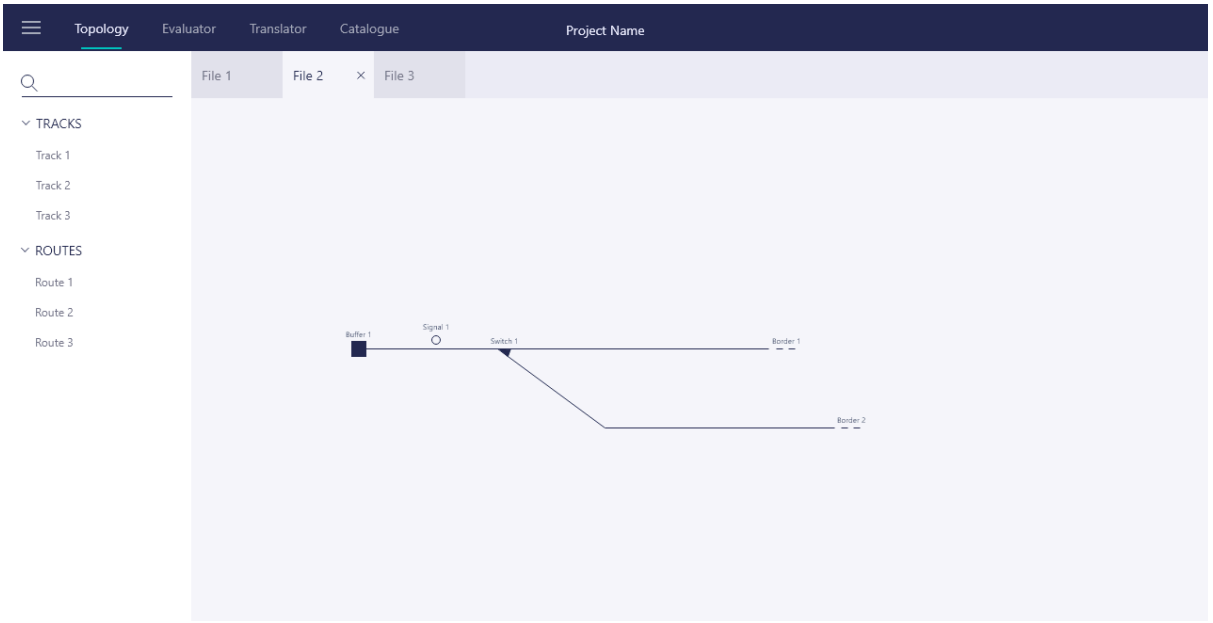
First, the visualizer uses the C# railML library to process the input railML file and extract its relevant information. Next, for each net element, it draws a `Line` shape with the element's visualization data following `screenPositioningSystem` coordinates.

For the infrastructure elements, it performs a mapping between the elements' types (such as border, switch, and signal) and the generated XAML resource files of the DXF railML block library. Then, it draws each of these elements as an `Image` containing the mapped resource file and an invisible `Rectangle` overlay that is displayed when users interact with these elements.

The XAML resources assigned to topological elements can be customized in the TV configuration file. Only elements that are directly placed alongside the net elements (such as signals and train detection elements) can be assigned new XAML drawings. Moreover, elements of the same type can be drawn differently by being assigned specific id prefixes, which are specified in this configuration file.

Finally, the visualizer adds a label for each route containing the route's name. When the user clicks on a route's label, the visualizer highlights all net elements and all infrastructure elements of the route path. The following illustration depicts a possible result of this functionality:

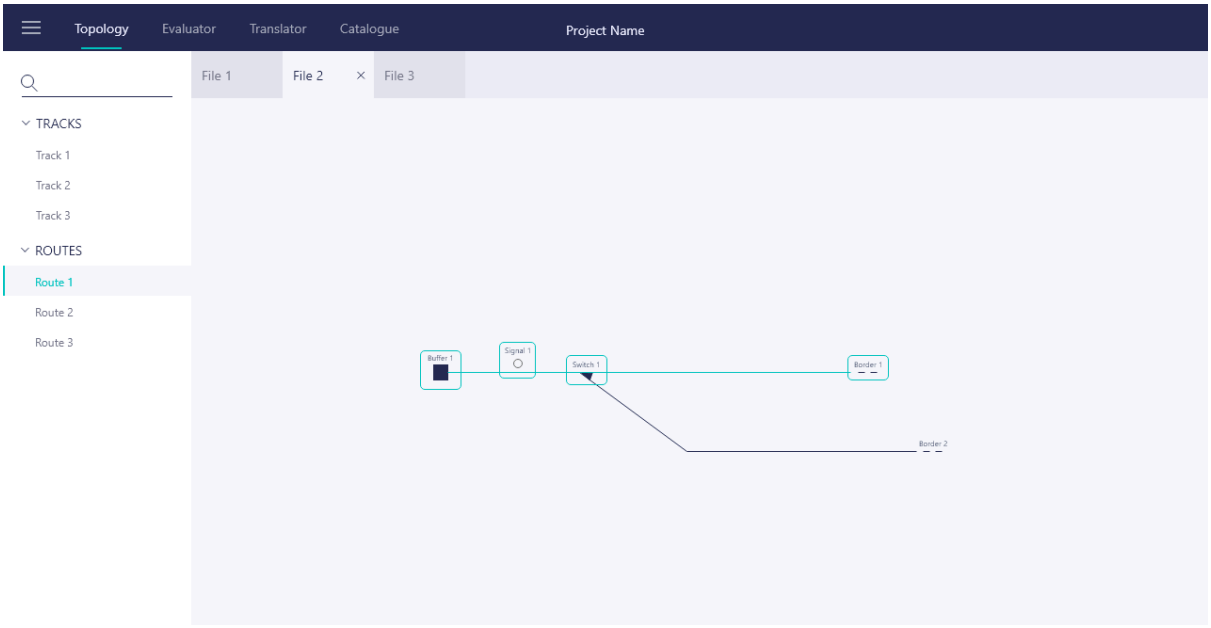
[3] https://inkscape.org/pt/
[4] https://www.adobe.com/products/illustrator.html
[5] https://github.com/BerndK/SvgToXaml



HIGHLIGHT-SCOPE

Functionality	**`HIGHLIGHT-SCOPE`**		
Description	*Highlights a railML route/track in the drawn network topology.*		
Input 1	`railml`	`xml`	*railML model containing a topology and a list of routes. This input value is provided with a file picker.*
Input 2	`scopeId`	`str`	*id of the route/track to be highlighted. This input value is provided by selecting any of the available routes/tracks of the route/track table.*

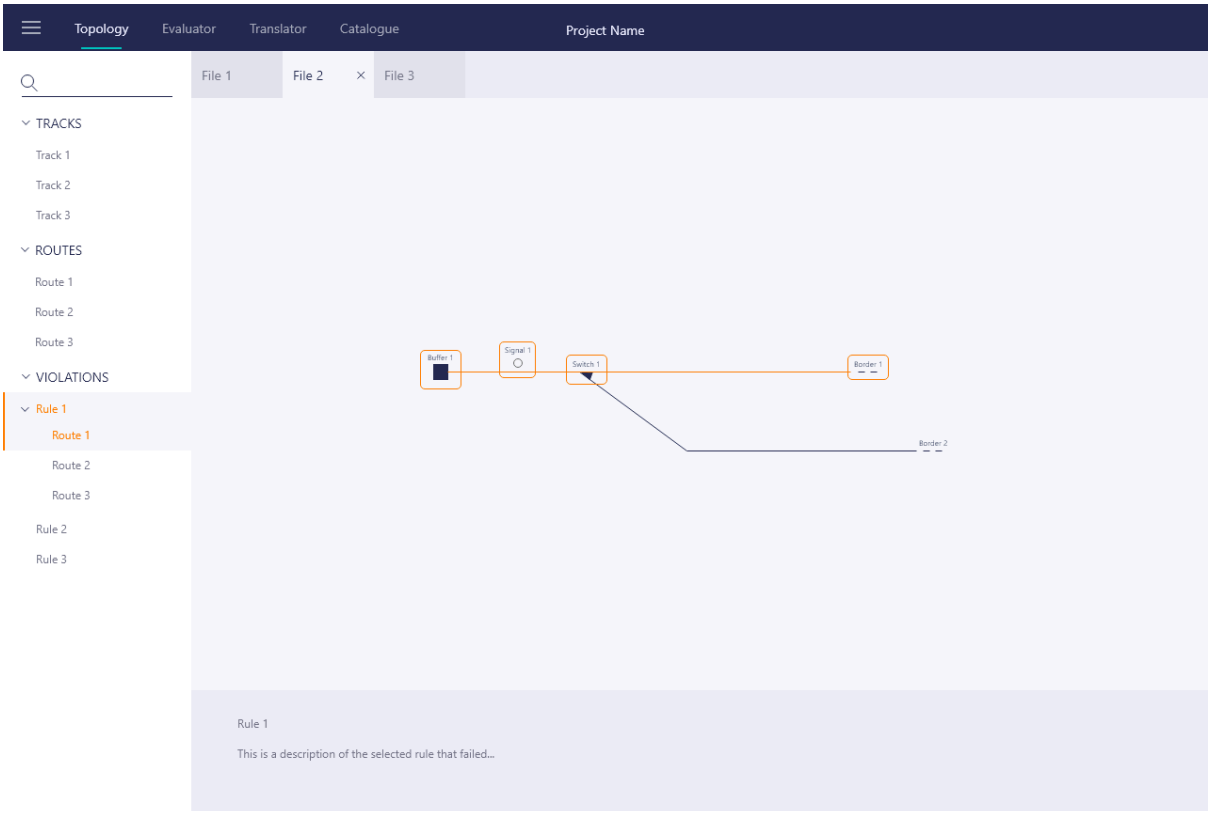
In this functionality, the visualizer presents the railML topology and the route/track table in the same fashion as the `DRAW-TOPOLOGY` functionality. Then, when the user selects the scope element (route or track) in the corresponding table it highlights all net elements and infrastructure elements belonging to that element. The following illustration depicts a possible result of this functionality:



SHOW-VIOLATION

Functionality	**`SHOW-VIOLATION`**		
Description	*Depicts a selected violation that resulted from the rule validation process.*		
Input 1	`railml`	`xml`	*railML model containing a topology to depict the validation process violations. This input value is provided with a file picker.*
Input 2	`violations`	`csv`	*CSV data containing a set of violations resulted in the validation process. This input value passed from the validation process or selected with a file picker.*
Input 3	`violation-id`	`id`	*violation to be highlighted identified by the name of the rule and the element of its scope that caused the violation. This input value is provided by selecting any of the available violations of the violations table.*

This functionality uses the C# railML library and the XAML resource files to present the topology and routes in the same fashion as the `DRAW-TOPOLOGY` functionality. Next, it uses the validation component to process the received CSV file containing the violations resulting in the validation process. Then, it adds each violation to the violations table. Any click performed on an element of this table will highlight the scope element of the violation (route or net element) and all its infrastructure elements in orange. Furthermore, any infrastructure elements that committed any violation are drawn with a warning icon. Finally, when the user selects a violation it highlights the scoped element and all infrastructure elements of that violation in the drawn topology. The following illustration depicts a possible result of this functionality:



The EVEREST Rule Language

The proposed syntax for the EVEREST rule language is presented in the BNF below.

```
rule        := scope :: (structural | invariant)
invariant   := everytime structural
structural  := fol | spatial
spatial     := everywhere [range] structural
             | nowhere [range] structural
             | somewhere [range] structural
             | structural until [range] structural
fol         := multOp expr | expr in expr | sexpr = sexpr
             | sexpr numComp sexpr
             | qtOp (var : expr),+ | structural
             | structural binFormOp structural
             | not structural
             | sexpr
qtOp        := all | some
binFormOp   := and | or | implies | iff
multOp      := one | lone | some | no
sexpr       := expr | scadeVar | scadeMacro( (sexpr),* ) | sexpr binNumOp
sexpr
expr        := var | railMLId | railMLMacro | placeholder | const
             | expr binExprOp expr | expr binNumOp expr
             | unExprOp expr
binExprOp   := . | | | & | -> | \
```

```
binNumOp    := + | - | * | /
numComp     := < | > | <= | >=
unExprOp    := ~ | ^
range       := ([ | () [expr] .. [expr] (] | ))
scope       := route | track
railMLId    := id
railMLMacro := id
var         := id
scadeMacro  := #id
scadeVar    := ` (char | { expr } )* `
placeholder := $id
const       := number | string | true | false
```

For `spatial` formulas we will adopt a point-wise semantics, meaning the formula will only be evaluated at spot locations along the scope element where some element is positioned in the railML model. In abstract terms, a `spotLocation` is a tuple with a `netElement` id, a position in that `netElement`, and a direction (either normal or reverse, when compared to direction from the endpoint 0 to the endpoint 1 of the respective `netElement`). To define the semantics we assume the existence of the following artifacts:

- a function `spots` that, for each element with at least one `spotLocation` and for each scope element (either a `route` or `track`), returns all its `spotLocations`[6];
- a function `entry` that given a scope element returns its entry `spotLocation`;
- a function `distance` that, given a scope element and two of its `spotLocations`, returns the distance between them;
- a predicate `between` that checks, for a particular scope element, if one `spotLocation` is between two other `spotLocations`.

We also assume that it is possible to check if a number (for example, a distance) belongs to an interval range using the $\in$ operator.

Rule semantics are defined over a model M, a partial function that associates each `var`, `railMLId`, or `railMLMacro` to the respective value: a unary relation with a single tuple (a singleton set) in the first case and an arbitrary relation extracted from the railML model in the other cases. A relation is just a set of tuples of identifiers of railML entities or constants (numbers, Booleans or strings). Expression $M^{\oplus v \mapsto \{t_1,...t_n\}}$ shall denote the overriding of the value of v in M by tuple set $\{t_1,...t_n\}$ (either insertion or replacement).

In our notation:

- the fact that a rule R holds in a model M will be denoted by ${}_M[\![R]\!]$; in detail, a rule R holds in model M iff the respective formula holds at the first position of all entities of the appropriate scope.
- the fact that a formula φ holds in a model M at position l of scope element s will be denoted by ${}_M[\![\varphi]\!]_{s,l}$.

[6] Notice that, for example, switches are located at different spot locations in the different `netElements` they connect.

Below we define the semantics of formulas by induction, for a kernel of the language. Semantics of the other logical operators can be derived from this kernel, and omitted ranges represent the unbounded range `[0..[` by default. We also assume that, at this point, placeholder identifiers have already been instantiated, otherwise their semantics would be undefined. Notice that the semantics of numeric expressions is only well-defined when they denote a singleton element (a relation with a single unary tuple, also known as a *scalar*). Likewise for Boolean expressions used as formulas. For example some Boolean attributes are optional in railML. If the user accesses the Boolean attribute of an element, and that attribute is not present, the semantics of the resulting formula will not be well-defined.

The kernel of the language for which the formal semantics is defined does not include interlocking rules, meaning no `everytime` and no SCADE macros, because the latter have an opaque semantics (this implies that gramarwise we consider that the non terminal `sexpr` can only be an `expr`).

Below, the following (possibly subscripted) identifiers

- v stand for `var` elements
- i for `railMLId` or `railMLMacro`
- a and b for `expr`
- φ and Ψ for `fol`/`spatial`/`structural`
- s for scope ids
- x and y for other railML ids and constants
- t for tuples of railML ids
- l for spot locations
- n and m for numerical constants
- u for string constants
- S for scope (either *route* or *track*)

Pattern matching over tuples is allowed. For instance, $t = \langle x \rangle$ means t is a singleton tuple whose single component is x. Besides the standard set operators, given relations T and U we have the following operators, to compute the composition, the transitive closure, and the converse:

$$T \bullet U = \{ \langle x_1,...,x_{n-1},y_2,...,y_m \rangle \mid \langle x_1,...,x_n \rangle \in T \wedge \langle y_1,...,y_m \rangle \in U \wedge x_n = y_1 \}$$
$$T^+ = T \cup T \bullet T \cup T \bullet T \bullet T \cup T \bullet T \bullet T \bullet T \cup \ldots$$
$$T^\circ = \{ \langle x_2,x_1 \rangle \mid \langle x_1,x_2 \rangle \in T \}$$

The formal semantics of the EVEREST rule language is as follows:

$$[\![S :: \varphi]\!]_M \triangleq \forall \langle x \rangle \in M(S) : [\![\varphi]\!]_{M,x,\texttt{entry(x)}}$$

$$[\![\textbf{everywhere } r\ \varphi]\!]_{M,s,l} \triangleq \forall\, l' \in \texttt{spots}(s) \mid \texttt{distance}(s,l,l') \in [\![r]\!]_{M,s,l} \rightarrow [\![\varphi]\!]_{M,s,l'}$$
$$[\![\textbf{nowhere } r\ \varphi]\!]_{M,s,l} \triangleq \forall\, l' \in \texttt{spots}(s) \mid \texttt{distance}(s,l,l') \in [\![r]\!]_{M,s,l} \rightarrow \neg [\![\varphi]\!]_{M,s,l'}$$
$$[\![\textbf{somewhere } r\ \varphi]\!]_{M,s,l} \triangleq \exists\, l' \in \texttt{spots}(s) \mid \texttt{distance}(s,l,l') \in [\![r]\!]_{M,s,l} \wedge [\![\varphi]\!]_{M,s,l'}$$
$$[\![\varphi \textbf{ until } r\ \Psi]\!]_{M,s,l} \triangleq \exists\, l' \in \texttt{spots}(s) \mid \texttt{distance}(s,l,l') \in [\![r]\!]_{M,s,l} \wedge [\![\Psi]\!]_{M,s,l'} \wedge \forall\, l'' \in \texttt{spots}(s) \mid \texttt{between}(s,l,l',l'') \rightarrow [\![\varphi]\!]_{M,s,l''}$$

$_M[\![\texttt{[}a\ \texttt{..}\ b\texttt{]}]\!]_{s,I} \triangleq [n,m]$ where $_M[\![a]\!]_{s,I} = \{\langle n\rangle\} \wedge {}_M[\![b]\!]_{s,I} = \{\langle m\rangle\}$

$_M[\![\texttt{[}a\ \texttt{..}\ b\texttt{)}]\!]_{s,I} \triangleq [n,m)$ where $_M[\![a]\!]_{s,I} = \{\langle n\rangle\} \wedge {}_M[\![b]\!]_{s,I} = \{\langle m\rangle\}$

$_M[\![\texttt{[}a\ \texttt{..}\ \texttt{)}]\!]_{s,I} \triangleq [n,\infty)$ where $_M[\![a]\!]_{s,I} = \{\langle n\rangle\}$

$_M[\![\texttt{(}a\ \texttt{..}\ b\texttt{]}]\!]_{s,I} \triangleq (n,m]$ where $_M[\![a]\!]_{s,I} = \{\langle n\rangle\} \wedge {}_M[\![b]\!]_{s,I} = \{\langle m\rangle\}$

$_M[\![\texttt{(}\ \texttt{..}\ b\texttt{)}]\!]_{s,I} \triangleq (\texttt{-}\infty,\texttt{n}]$ where $_M[\![b]\!]_{s,I} = \{\langle n\rangle\}$

$_M[\![\texttt{(}a\ \texttt{..}\ b\texttt{)}]\!]_{s,I} \triangleq (n,m)$ where $_M[\![a]\!]_{s,I} = \{\langle n\rangle\} \wedge {}_M[\![b]\!]_{s,I} = \{\langle m\rangle\}$

$_M[\![\texttt{(}\ \texttt{..}\ \texttt{)}]\!]_{s,I} \triangleq (\texttt{-}\infty,\infty)$

$_M[\![\textbf{not}\ \varphi]\!]_{s,I} \triangleq \neg {}_M[\![\varphi]\!]_{s,I}$

$_M[\![\varphi\ \textbf{and}\ \Psi]\!]_{s,I} \triangleq {}_M[\![\varphi]\!]_{s,I} \wedge {}_M[\![\Psi]\!]_{s,I}$

$_M[\![\textbf{all}\ v\ \texttt{:}\ a\ \texttt{|}\ \varphi]\!]_{s,I} \triangleq \forall\ t \in {}_M[\![a]\!]_{s,I} : {}_{M \oplus v \mapsto \{t\}}[\![\varphi]\!]_{s,I}$

$_M[\![a\ \textbf{in}\ b]\!]_{s,I} \triangleq {}_M[\![a]\!]_{s,I} \subseteq {}_M[\![b]\!]_{s,I}$

$_M[\![\textbf{some}\ a]\!]_{s,I} \triangleq |{}_M[\![a]\!]_{s,I}| > 0$

$_M[\![\textbf{lone}\ a]\!]_{s,I} \triangleq |{}_M[\![a]\!]_{s,I}| < 2$

$_M[\![a]\!]_{s,I} \triangleq x = \textbf{true}$ where $_M[\![a]\!]_{s,I} = \{\langle x\rangle\}$

$_M[\![a\ \texttt{<}\ b]\!]_{s,I} \triangleq n < m$ where $_M[\![a]\!]_{s,I} = \{\langle n\rangle\} \wedge {}_M[\![b]\!]_{s,I} = \{\langle m\rangle\} \wedge$

$_M[\![a\ \texttt{|}\ b]\!]_{s,I} \triangleq {}_M[\![a]\!]_{s,I} \cup {}_M[\![b]\!]_{s,I}$

$_M[\![a\ \textbf{\&}\ b]\!]_{s,I} \triangleq {}_M[\![a]\!]_{s,I} \cap {}_M[\![b]\!]_{s,I}$

$_M[\![a\ \texttt{\textbackslash}\ b]\!]_{s,I} \triangleq {}_M[\![a]\!]_{s,I} \setminus {}_M[\![b]\!]_{s,I}$

$_M[\![a\ \texttt{.}\ b]\!]_{s,I} \triangleq {}_M[\![a]\!]_{s,I} \bullet {}_M[\![b]\!]_{s,I}$

$_M[\![a\ \texttt{->}\ b]\!]_{s,I} \triangleq {}_M[\![a]\!]_{s,I} \times {}_M[\![b]\!]_{s,I}$

$_M[\![\texttt{\~{}}\ a]\!]_{s,I} \triangleq {}_M[\![a]\!]_{s,I}{}^{\circ}$

$_M[\![\texttt{\^{}}\ a]\!]_{s,I} \triangleq {}_M[\![a]\!]_{s,I}{}^{+}$

$_M[\![a\ \texttt{+}\ b]\!]_{s,I} \triangleq \{\langle n + m\rangle\}$ where $_M[\![a]\!]_{s,I} = \{\langle n\rangle\} \wedge {}_M[\![b]\!]_{s,I} = \{\langle m\rangle\}$

$_M[\![a\ \texttt{-}\ b]\!]_{s,I} \triangleq \{\langle n - m\rangle\}$ where $_M[\![a]\!]_{s,I} = \{\langle n\rangle\} \wedge {}_M[\![b]\!]_{s,I} = \{\langle m\rangle\}$

$_M[\![a\ \texttt{*}\ b]\!]_{s,I} \triangleq \{\langle n \times m\rangle\}$ where $_M[\![a]\!]_{s,I} = \{\langle n\rangle\} \wedge {}_M[\![b]\!]_{s,I} = \{\langle m\rangle\}$

$_M[\![a\ \texttt{/}\ b]\!]_{s,I} \triangleq \{\langle n \div m\rangle\}$ where $_M[\![a]\!]_{s,I} = \{\langle n\rangle\} \wedge {}_M[\![b]\!]_{s,I} = \{\langle m\rangle\}$

$_M[\![i]\!]_{s,I} \triangleq \{\, t \mid t \in M(i)\!\downarrow\! s \wedge ((t = \langle x\rangle \wedge x \in \mathrm{dom}(\texttt{spots})) \rightarrow I \in \texttt{spots}(x))\,\}$ [7]

$_M[\![v]\!]_{s,I} \triangleq M(v)$

$_M[\![n]\!]_{s,I} \triangleq \{\langle n\rangle\}$

$_M[\![u]\!]_{s,I} \triangleq \{\langle u\rangle\}$

$_M[\![\textbf{true}]\!]_{s,I} \triangleq \{\langle \textbf{true}\rangle\}$

$_M[\![\textbf{false}]\!]_{s,I} \triangleq \{\langle \textbf{false}\rangle\}$

Each rule is implicitly quantified over its scope. This scope can be either a `route` or a `track` and the rule is to be checked against all entities of that type. Thus, the value of railML entities or macros that are associated with a particular entity of the scope will need to be projected for that element when evaluating the rule. The projection of a relation to a particular scope element is denoted by ↓. For example, when the scope is **`route`** and we are implicitly quantifying over all routes r, expression $M(\texttt{routeEntry})\!\downarrow\! r$ that project the value of railML entity `routeEntry`, that associates routes with the respective route entry element, for a particular model *M,* we will obtain an unary relation (a set), computed as follows:

[7] The meaning of *M(i)↓s is explained in the sequel.*

$$M(\texttt{routeEntry})\downarrow r = \{\langle r \rangle\} \bullet M(\texttt{routeEntry})$$

On the other hand, if the scope is **`track`** and we are implicitly quantifying over all tracks t, projecting the value of `routeEntry` will have no effect:

$$M(\texttt{routeEntry})\downarrow t = M(\texttt{routeEntry})$$

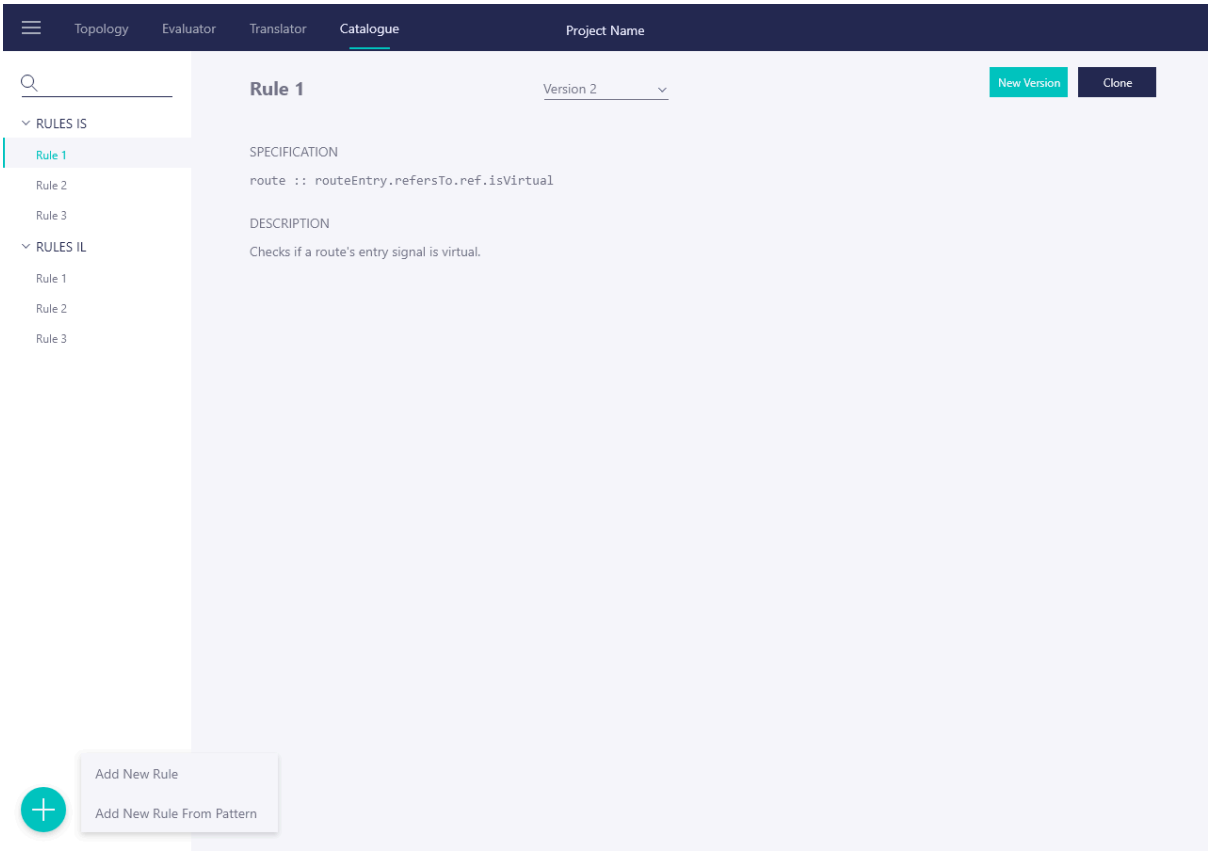
Besides projecting over the scope element, this set also needs to be filtered to contain only the elements that appear at each location the rule is checked (for the case of elements that have a spot location). That is precisely the goal of the set comprehension in the semantic rule for `railMLId` or `raimMLMacro` identifiers.

Rule Designer

The DVT Integrates this component in a UI to support users in writing EVEREST rules and checking their syntax and type correctness. Namely it allows users to:

- create a rule manually or with the assistance of patterns (`INSTANTIATE-PATTERN`);
- check syntax (`CHECK-SYNTAX`) and type (`CHECK-TYPE`) errors when creating rules, relying on the Validation component.

It is based on WPF, and its UI is a window containing a list of all EVEREST rules - split among those for infrastructure and interlocking - and a panel that allows users to check rule details and either make a new version of a rule or clone it. The following illustration depicts a mockup of the UI of this component.

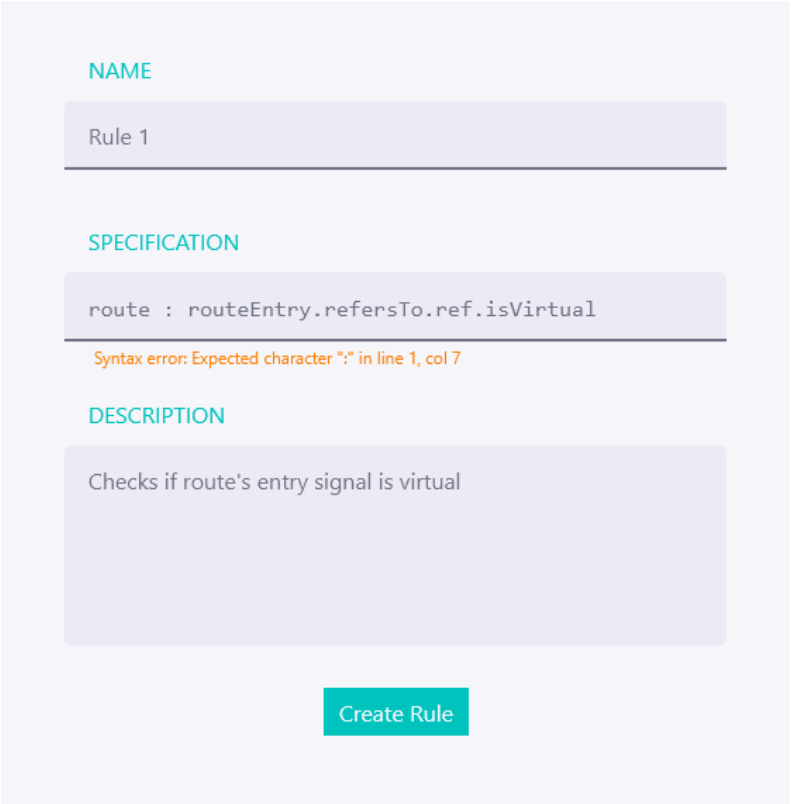


CHECK-SYNTAX

Functionality	**`CHECK-SYNTAX`**		
Description	*Checks that a rule is syntax-correct.*		
Input 1	`rule`	`rule`	*rule to be checked*
Output 1	`errors`	`(loc,str)`*	*list of parsing errors and the respective localization in the rule*

In this functionality, the Rule Designer collects the railML rule `RichTextBox` and sends this text to the Validation component of the rule validation library. Then, the Validation component uses the ANTLR library to parse the received text and check its syntax correctness. The ANTLR library provides a listener that indicates both the text's line and position that failed the specified grammar. In this case, the Rule Designer presents in the output text box the line and the position of the text where the error occurred.

The following illustration depicts the UI mockup for a possible result of this functionality.



CHECK-TYPE

Functionality	**`CHECK-TYPE`**		
Description	*Checks that a rule is well-typed.*		
Input 1	`rule`	`rule`	*rule to be type-checked already syntactically checked*
Output 1	`errors`	`(loc,str)`*	*list of type errors and the respective localization in the rule*

The typing system for the rules implemented by this functionality is defined in a typing context Γ, a partial function that assigns a type to every `railMLId` or `railMLMacro` identifier. This context will be determined based on information contained in the Rule Designer configuration. Type checking is also performed on interlocking rules, but since SCADE macros have an unknown type the type checking algorithm will be limited if they are used. However, it assumes that the parameters of macros must be singleton sets (scalars) and that they also return scalars.

To define the EVEREST rule language typing system we follow the approach of the Alloy typing system, namely in regarding a type *T* itself as a relation of tuples of primitive types. A primitive type is either **`number`**, **`bool`**, **`string`**, or one of the railML entities such as `signalIL` or `route`. We also have the special type *, to be used in SCADE macros and that represents an unknown type. This type relation is in a sense an upper-bound for the value of the expression being typed, and the main goal of the type system is precisely to infer this upper-bound statically for expressions and report an error if it ends up being empty (meaning

the expression will always denote an empty relation, which most likely is due to a specification error). Since types are relations, we can manipulate them using the same relational operators used to define the semantics. Another goal of the typing system is to check that operators are only applied to relations of compatible arity.

The fact that a rule R is well-typed in a context Γ will be denoted by $\Gamma \vdash R$. Likewise, the fact that a formula φ is well-typed in a context Γ for scope S will be denoted by $\Gamma \vdash_S \varphi$. The fact that an expression a is well-typed in a context Γ for scope S, and has type T and arity n will be denoted by

$$\Gamma \vdash_S a \subseteq T^n$$

Again we will assume that the type of an identifier can be projected to the type of the scope using operator $\downarrow$. For example, the type of `routeEntry` in the scope **`route`** would be

$$\Gamma(\texttt{routeEntry})\downarrow\textbf{route} = \{\langle \textbf{routeEntry} \rangle\}^1$$

while in the scope Track it would be

$$\Gamma(\texttt{routeEntry})\downarrow\textbf{track} = \{\langle \textbf{route},\textbf{routeEntry} \rangle\}^2$$

The typing rules for the kernel language are defined as follows. In the typing rules, the special unknown type ***** is assumed to match any other type. For example, if we have $\Gamma \vdash_S a \subseteq \{\langle * \rangle\}^1$ then it is also true that $\Gamma \vdash_S a \subseteq \{\langle \textbf{number} \rangle\}^1$ or $\Gamma \vdash_S a \subseteq \{\langle \textbf{bool} \rangle\}^1$.

$$\Gamma \vdash S \texttt{ :: [}\textbf{everytime}\texttt{]}\ \varphi \quad \Leftrightarrow \quad \Gamma \vdash_S \varphi$$

$$\begin{array}{lll}
\Gamma \vdash_S \textbf{everywhere}\ r\ \varphi & \Leftrightarrow & \Gamma \vdash_S r \wedge \Gamma \vdash_S \varphi \\
\Gamma \vdash_S \textbf{somewhere}\ r\ \varphi & \Leftrightarrow & \Gamma \vdash_S r \wedge \Gamma \vdash_S \varphi \\
\Gamma \vdash_S \textbf{nowhere}\ r\ \varphi & \Leftrightarrow & \Gamma \vdash_S r \wedge \Gamma \vdash_S \varphi \\
\Gamma \vdash_S \varphi\ \textbf{until}\ r\ \Psi & \Leftrightarrow & \Gamma \vdash_S r \wedge \Gamma \vdash_S \varphi \wedge \Gamma \vdash_S \Psi
\end{array}$$

$$\begin{array}{ll}
\Gamma \vdash_S [a\ ..\ b] & \Leftrightarrow \quad \Gamma \vdash_S a \subseteq \{\langle \textbf{number} \rangle\}^1 \wedge \Gamma \vdash_S b \subseteq \{\langle \textbf{number} \rangle\}^1 \\
\Gamma \vdash_S [a\ ..\ b) & \Leftrightarrow \quad \Gamma \vdash_S a \subseteq \{\langle \textbf{number} \rangle\}^1 \wedge \Gamma \vdash_S b \subseteq \{\langle \textbf{number} \rangle\}^1 \\
\Gamma \vdash_S [a\ ..\ ) & \Leftrightarrow \quad \Gamma \vdash_S a \subseteq \{\langle \textbf{number} \rangle\}^1 \\
\Gamma \vdash_S (a\ ..\ b] & \Leftrightarrow \quad \Gamma \vdash_S a \subseteq \{\langle \textbf{number} \rangle\}^1 \wedge \Gamma \vdash_S b \subseteq \{\langle \textbf{number} \rangle\}^1 \\
\Gamma \vdash_S (\ ..\ b] & \Leftrightarrow \quad \Gamma \vdash_S b \subseteq \{\langle \textbf{number} \rangle\}^1 \\
\Gamma \vdash_S (a\ ..\ b) & \Leftrightarrow \quad \Gamma \vdash_S a \subseteq \{\langle \textbf{number} \rangle\}^1 \wedge \Gamma \vdash_S b \subseteq \{\langle \textbf{number} \rangle\}^1 \\
\Gamma \vdash_S (\ ..\ ) &
\end{array}$$

$$\begin{array}{lll}
\Gamma \vdash_S \textbf{not}\ \varphi & \Leftrightarrow & \Gamma \vdash_S \varphi \\
\Gamma \vdash_S \varphi\ \textbf{and}\ \Psi & \Leftrightarrow & \Gamma \vdash_S \varphi \wedge \Gamma \vdash_S \Psi \\
\Gamma \vdash_S \textbf{all}\ v : a \mid \varphi & \Leftrightarrow & \Gamma \vdash_S a \subseteq T^1 \wedge \text{all} \\
\Gamma \vdash_S a\ \textbf{in}\ b & \Leftrightarrow & \Gamma \vdash_S a \subseteq T^n \wedge \Gamma \vdash_S b \subseteq U^m \wedge n = m \wedge \mathsf{T} \cap \mathsf{U} \neq \varnothing \\
\Gamma \vdash_S \textbf{some}\ a & \Leftrightarrow & \Gamma \vdash_S a \subseteq T^n \\
\Gamma \vdash_S \textbf{lone}\ a & \Leftrightarrow & \Gamma \vdash_S a \subseteq T^n \\
\Gamma \vdash_S a & \Leftrightarrow & \Gamma \vdash_S a \subseteq \{\langle \textbf{bool} \rangle\}^1 \\
\Gamma \vdash_S a < b & \Leftrightarrow & \Gamma \vdash_S a \subseteq \{\langle \textbf{number} \rangle\}^1 \wedge \Gamma \vdash_S b \subseteq \{\langle \textbf{number} \rangle\}^1
\end{array}$$

$$
\begin{array}{lcl}
\Gamma \vdash_S a \mid b \subseteq (T \cup U)^n & \Leftrightarrow & \Gamma \vdash_S a \subseteq T^n \wedge \Gamma \vdash_S b \subseteq U^m \wedge n = m \\
\Gamma \vdash_S a \,\&\, b \subseteq (T \cap U)^n & \Leftrightarrow & \Gamma \vdash_S a \subseteq T^n \wedge \Gamma \vdash_S b \subseteq U^m \wedge n = m \\
\Gamma \vdash_S a \setminus b \subseteq T^n & \Leftrightarrow & \Gamma \vdash_S a \subseteq T^n \wedge \Gamma \vdash_S b \subseteq U^m \wedge n = m \\
\Gamma \vdash_S a \text{ -> } b \subseteq (T \times U)^{n+m} & \Leftrightarrow & \Gamma \vdash_S a \subseteq T^n \wedge \Gamma \vdash_S b \subseteq U^m \\
\Gamma \vdash_S a \,.\, b \subseteq (T \bullet U)^{n+m-2} & \Leftrightarrow & \Gamma \vdash_S a \subseteq T^n \wedge \Gamma \vdash_S b \subseteq U^m \wedge n+m > 2 \\
\Gamma \vdash_S \sim a \subseteq (T^\circ)^2 & \Leftrightarrow & \Gamma \vdash_S a \subseteq T^2 \\
\Gamma \vdash_S \hat{} \; a \subseteq (T^+)^2 & \Leftrightarrow & \Gamma \vdash_S a \subseteq T^2
\end{array}
$$

$$
\begin{array}{lcl}
\Gamma \vdash_S a + b \subseteq \{\langle \textbf{number} \rangle\}^1 & \Leftrightarrow & \Gamma \vdash_S a \subseteq \{\langle \textbf{number} \rangle\}^1 \wedge \Gamma \vdash_S b \subseteq \{\langle \textbf{number} \rangle\}^1 \\
\Gamma \vdash_S a - b \subseteq \{\langle \textbf{number} \rangle\}^1 & \Leftrightarrow & \Gamma \vdash_S a \subseteq \{\langle \textbf{number} \rangle\}^1 \wedge \Gamma \vdash_S b \subseteq \{\langle \textbf{number} \rangle\}^1 \\
\Gamma \vdash_S a \,/\, b \subseteq \{\langle \textbf{number} \rangle\}^1 & \Leftrightarrow & \Gamma \vdash_S a \subseteq \{\langle \textbf{number} \rangle\}^1 \wedge \Gamma \vdash_S b \subseteq \{\langle \textbf{number} \rangle\}^1 \\
\Gamma \vdash_S a * b \subseteq \{\langle \textbf{number} \rangle\}^1 & \Leftrightarrow & \Gamma \vdash_S a \subseteq \{\langle \textbf{number} \rangle\}^1 \wedge \Gamma \vdash_S b \subseteq \{\langle \textbf{number} \rangle\}^1
\end{array}
$$

$$
\begin{array}{lcl}
\Gamma \vdash_S \#i(a_1, \ldots, a_k) \subseteq \{\langle * \rangle\}^1 & \Leftrightarrow & \Gamma \vdash_S a_1 \subseteq T_1{}^1 \wedge \ldots \wedge \Gamma \vdash_S a_k \subseteq T_k{}^1 \\
\Gamma \vdash_S \text{”} c_1\{a_1\}\ldots\{a_k\}c_{k+1}\text{”} \subseteq \{\langle * \rangle\}^1 & \Leftrightarrow & \Gamma \vdash_S a_1 \subseteq T_1{}^1 \wedge \ldots \wedge \Gamma \vdash_S a_k \subseteq T_{k+1}{}^1
\end{array}
$$

$$
\begin{array}{lcl}
\Gamma \vdash_S i \subseteq T^n & \Leftrightarrow & \Gamma(i) \downarrow S \subseteq T^n \wedge n > 0 \\
\Gamma \vdash_S v \subseteq T^1 & \Leftrightarrow & \Gamma(v) \subseteq T^1 \\
\Gamma \vdash_S n \subseteq \{\langle \textbf{number} \rangle\}^1 & & \\
\Gamma \vdash_S s \subseteq \{\langle \textbf{string} \rangle\}^1 & & \\
\Gamma \vdash_S \textbf{true} \subseteq \{\langle \textbf{bool} \rangle\}^1 & & \\
\Gamma \vdash_S \textbf{false} \subseteq \{\langle \textbf{bool} \rangle\}^1 & &
\end{array}
$$

If during type inference a type is found to be empty or the side conditions on arities fail, an error is raised at the respective sub-expression. For example, rule

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

has an error at the inclusion operator because the type of the left hand side expression is

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

because

$$
\begin{array}{ll}
\Gamma(\texttt{routeEntry})\downarrow\texttt{route} & \subseteq \{\langle \textbf{routeEntry} \rangle\}^1 \\
\Gamma(\texttt{refersTo})\downarrow\texttt{route} & \subseteq \{\langle \textbf{routeEntry,routeEntryRef}, \ldots \rangle\}^2 \\
\Gamma(\texttt{ref})\downarrow\texttt{route} & \subseteq \{\langle \textbf{routeEntryRef,signalIL}, \ldots \rangle\}^2
\end{array}
$$

while the type of the right hand side expression is

$$\Gamma \vdash_{\text{route}} \texttt{speedSection} \subseteq \{\langle \textbf{speedSection} \rangle\}^1$$

Since the intersection of the types is empty it would never be possible for the inclusion to be true, meaning the rule is meaningless.

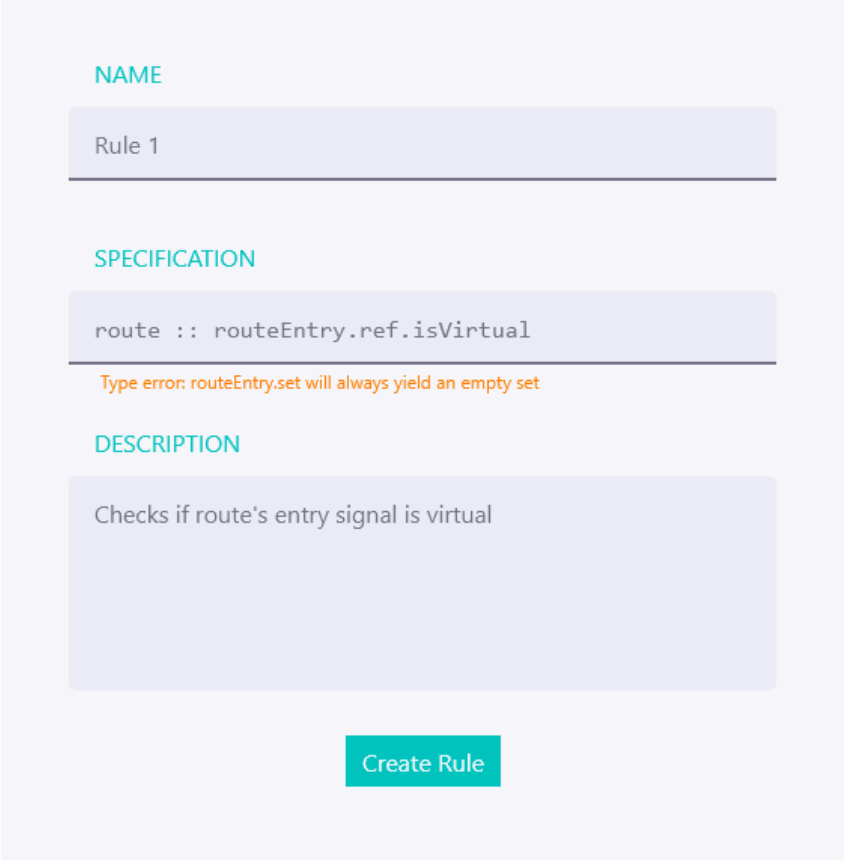
As explained above, we assume the expression parameters of SCADE variables to be singletons (unary relations), so, for example, the following rule would raise a type error, since `refersTo` is a binary relation.

```
route :: everytime “varname_{refersTo}” < 10
```

but the following would not give origin to any type error, since the return value of the macro is unknown (possibly, it could be a number).

```
route :: everytime "varname_{routeEntry}" < 10
```

The following illustration depicts the UI mockup for a possible result of this functionality.

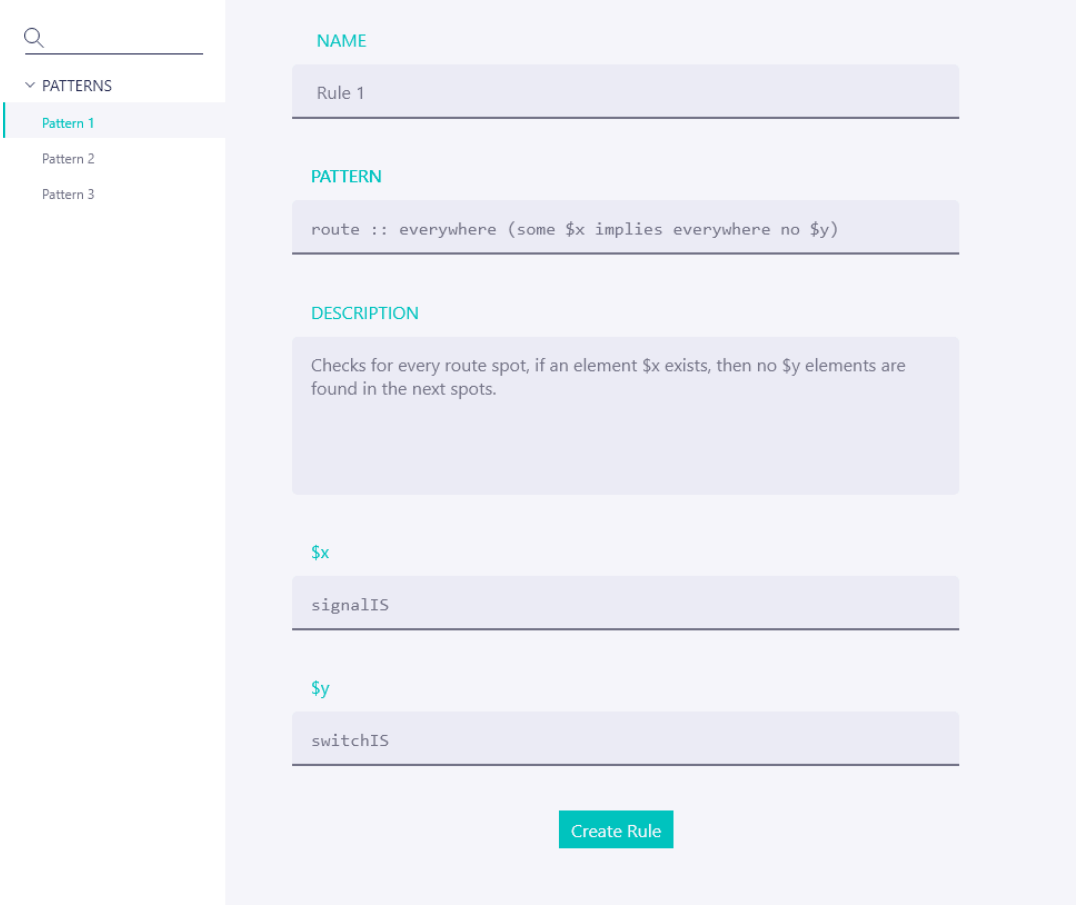


INSTANTIATE-PATTERN

Functionality	**`INSTANTIATE-PATTERN`**		
Description	*Supports the creation of rules to be checked through the instantiation of predefined pattern rules with elements of the model.*		
Input 1	`desc`	`str`	*description of the pattern*
Input 2	`pattern`	`rule`	*pattern to be instantiated, essentially a rule with placeholders*
Input 3	`help`	`(id,str)`$^+$	*description of the placeholders, to be shown in the dialog*
Output 1	`rule`	`rule`	*the instantiated rule*

In this functionality, the Rule Designer provides a UI that lists all available patterns for defining rules. To create a rule from a pattern, users need to provide a rule name, along with

the instantiation for the pattern placeholders. The following illustration depicts one possible result of this functionality:



Topological Rule Verifier

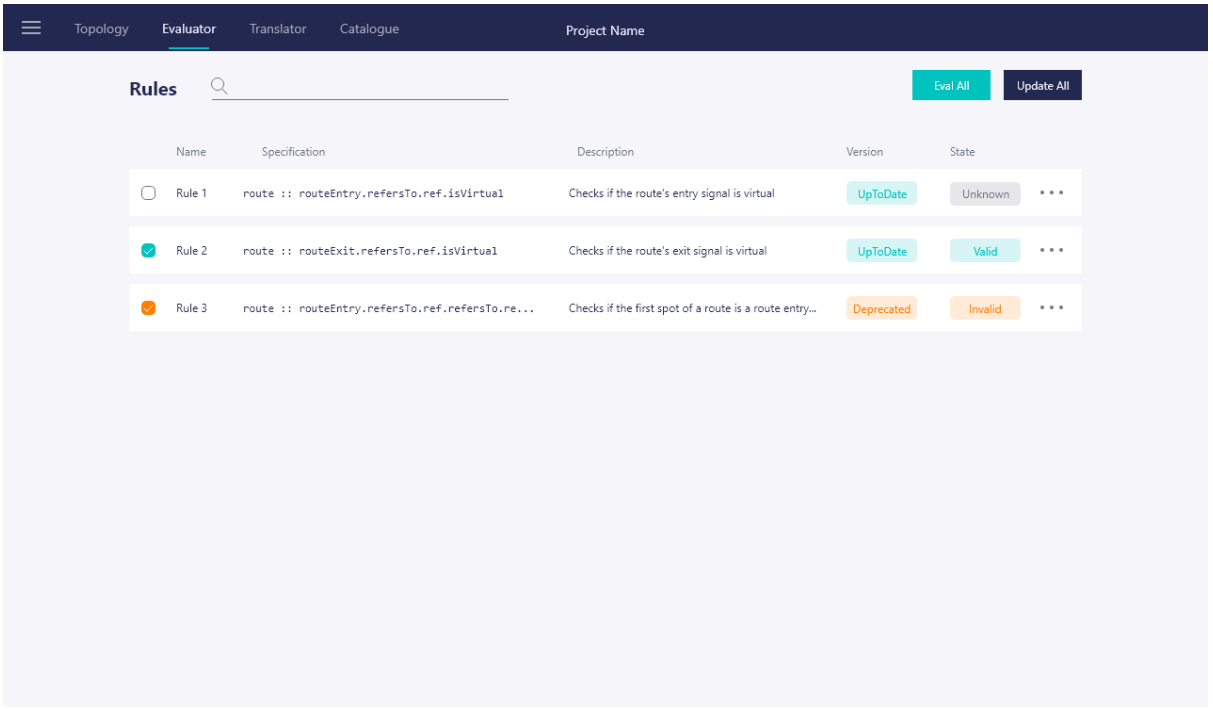
The DVT Integrates this component in a UI to

- support users in checking EVEREST infrastructure rules (`EVALUATE-IS-RULE`).

The UI is based on WPF, and shows, for each project the set of active infrastructure rules, selected from the rule catalogue. Active rules are marked as deprecated whenever a new version is created in the rule catalogue. Deprecated rules can be updated to the most recent version through the interface.

Active rules can be evaluated individually or in a batch. A rule is evaluated for all areas (i.e., railML files) of the current project. Once evaluated, they are marked as either “Valid” or “Invalid”, depending on the result of the `EVALUATE-IS-RULE`. A rule is considered invalid if it fails for any scope element of any area. Inactive rules, or active rules that have not yet been evaluated, are assigned an “Unknown” state.

The following illustration depicts a mockup of the UI of this functionality.



EVALUATE-IS-RULE

Functionality	**`EVALUATE-IS-RULE`**		
Description	*Evaluates an infrastructure rule over a railML topological model, returning the violations found.*		
Input 1	`rule`	`(str,rule)`	*the rule to be verified, assumed to be syntactically correct (**CHECK-SYNTAX**) and type-checked (**CHECK-TYPE**), and a human-readable description; the rule must be purely structural (i.e., no* **`everytime`** *properties) and not contain any SCADE macro or identifier*
Input 2	`railml`	`xml`	*railML model to be verified.*
Output 1	`violations`	`csv`	*a CSV with the violations found for the given rule, each line listing a unique id for the violation, the name of the violated rule, an identifier for the related railML file, the type of scope, an entity in that scope where that rule is violated, and additional element identifiers along the scope entity that should additionally be flagged as faulty.*

This functionality evaluates an EVEREST rule over a railML model by essentially implementing the EVEREST rule semantics defined earlier. It uses the Rule Validation module to process an EVEREST rule and the ANTLR Visitor pattern to traverse the resulting AST and implement the defined recursive semantics. The railML model over which the rule is being evaluated is processed by the C# railML Library and provided as context to the

Visitor. Found violations are then reported in a CSV file. More specifically, to evaluate a rule *S* `::` *φ*, this functionality proceeds roughly as follows:

- Calculate a map between the `ids` of railML elements containing `spotLocations` and their list of `spotLocation` objects, represented by a `Dictionary<string, List<SpotLocation>>` in C#. railML elements that are positioned at the endpoint of multiple `netElements`, such as `switches`, are assigned a unique `spotLocation` for each endpoint. Moreover, `spotLocation` objects that effectively represent the same position in the track (e.g., when two elements have the same position) are unified.
- Given this structure and the route path containing the `netElement` sequence of a `route` (that was described in the C# railML Library section), calculate the auxiliary functions `spots`, `entry`, `distance` and `between` required by the EVEREST rule semantics. Note that this functionality should only be executed after calculating `POS` attributes with the EVEREST AutoCAD Plugin since these functions depend on those attributes.
- Again supported by the C# railML Library, calculate a dictionary containing all elements and attributes of the railML model, including the macros defined in the Rule Designer configuration. This dictionary represents the model *M* of the semantics, associating to each `railMLId` a concrete tuple set. In particular, it stores a set of railML ids for each railML parent node, such as `route`, `signalIS` and `switchIS`. For child nodes, it stores a tuple containing the parent node id and the child node id. Finally, for node attributes it stores a tuple containing the node id and the attribute's value, if defined. In C#, tuples have been encoded as lists. Three possible entries of this dictionary are the following:

  ```
  “route” -> {[route1],[route2],...}
  “routeEntry” -> {[route1,routeEntry1],[route2,routeEntry2],...}
  “isVirtual” -> {[signalIL1,false],[signalIL2,true],...}
  ```

  This dictionary only covers `functionalInfrastructure` and `interlocking` railML elements, discarding elements above this hierarchy like `signalsIS` and `switchesIL`. This structure allows the evaluation of EVEREST rules with railML attributes in a generic fashion, since the `railMLId`’s that can occur in a rule are exactly those present in this dictionary. [8]
- Given *M* and the auxiliary functions, it implements the recursive semantics defined for $_M[\![S$ `::` $\varphi]\!]$ using the Visitor pattern provided by ANTLR. The translation of the declarative semantics into a sequential procedure is mostly straightforward. Whenever the context *M* has to be enhanced (namely, on quantifications assigning values to `vars`), the dictionary is updated accordingly.
- Whenever a rule evaluates to false, a violation is reported. Since all rules effectively quantify universally over the provided `scope`, each violation is detected in the context of a particular scope *s*. Moreover, whenever a universal quantification **`all`** evaluates to false, the related element is also reported alongside the scope element. Since the AutoCAD plugin supports multiple railML files, an identifier for the originating railML file is also registered.

[8] This solution is also resilient enough to eventual changes of the railML schema in the future.

- Note that the evaluation may throw runtime (type) errors if side conditions of the semantics are not met. This will essentially occur whenever numerical/Boolean expressions fail to return a singleton value.

Finally, this functionality produces CSV files listing all violations resulting from the validation process. Each line of such CSV files follows the format:

`“violation-id”,“railml-area”,“violation-desc”,“scope”,“scope-elem-id”,“error-elem-id1,error-elem-id2”`

Interlocking Rule Translator

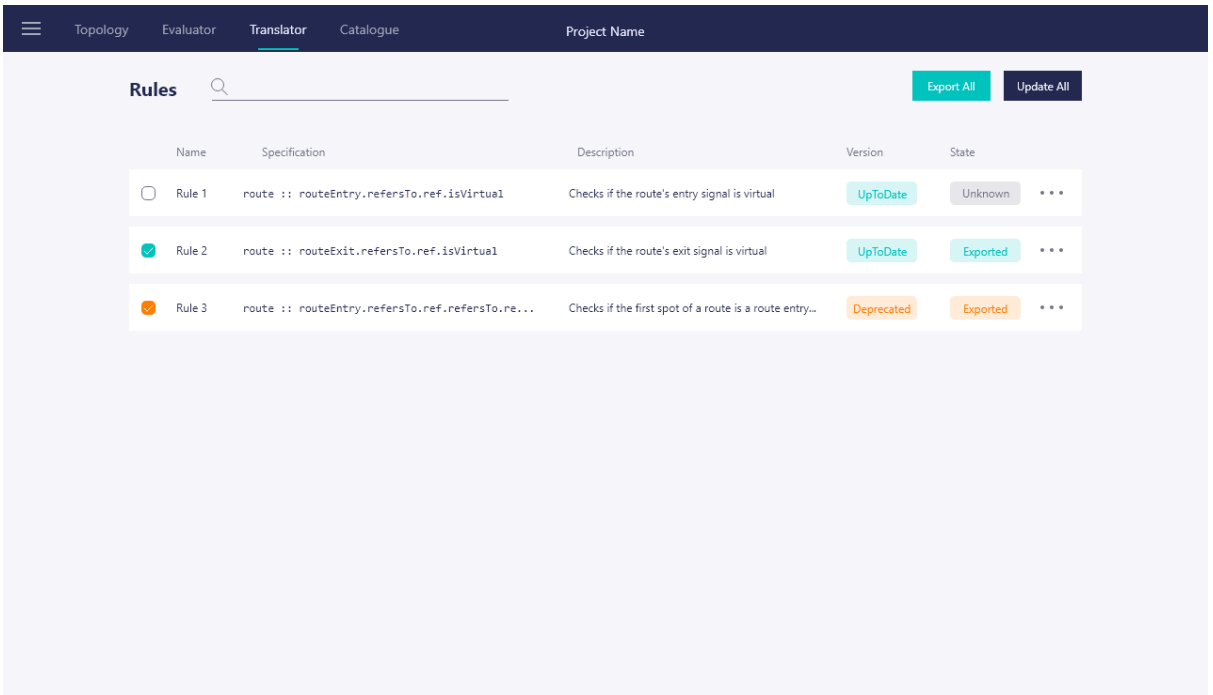
The DVT Integrates this component in a UI to:

- support users in checking EVEREST interlocking rules (`TRANSLATE-IL-RULE`).

Similarly to the infrastructure rules, it is based on WPF and for each project shows the set of active interlocking rules. Active rules are also marked as deprecated whenever a new version is created.

However, unlike infrastructure rules, interlocking rules are not evaluated by the DVT, but rather exported as observer specifications into the project directory, to be externally managed by the RAS and its EVEREST Plugin. Rules can be exported individually or in batch, and an observer file is created for each pair railML area/rule. Thus, active interlocking rules have three possible states: “Exported” - when the respective observer files are present in the project directory -, “Incomplete” - when only a subset of the expected files are found, possibly due to some being deleted by the user - and “Unknown” - if none of the expected files are found in the project directory, either because they have not been created or have been deleted.

The figure below presents a mockup of the UI for this functionality.



TRANSLATE-IL-RULE

Functionality	**`TRANSLATE-IL-RULE`**		
Description	*Translates an interlocking rule over a railML topological model into a set of propositional logic formulas*		
Input 1	`rule`	`(str,rule)`	*the rule to be verified, assumed to be syntactically correct (***CHECK-SYNTAX***) and type-checked (***CHECK-TYPE***), and a human-readable description; the rule must be an invariant (i.e., an* **`everytime`** *property).*
Input 2	`railml`	`xml`	*railML model to be verified.*
Output 1	`observers`	`form`*	*a set of propositional logic formulas that encode observers to be later incorporated in SCADE models*

The interlocking rule translator behaves similarly to the *Topological Rule Verifier*, but since there are uninterpreted SCADE macros in the rule it cannot fully determine the value of the rule, but instead generates a (partially evaluated) propositional logic formula where those macros still appear (with the parameters already instantiated). Likewise in the `EVALUATE-IS-RULE`, the translation requires first computing a model *M* that assigns a value to all railML ids and macros. This model is computed from a railML model using the same process described in the *Topological Rule Verifier*.

In the translation specified below (for a kernel of the language) we assume that a set of propositional formulas is returned, one for each element of the rule scope. Also **`AND`** or **`OR`** applied to a set of formulas denotes their conjunction or disjunction, respectively.

> **AND** $\{f_1, \ldots, f_k\}$ = f_1 **and** … **and** f_k
> **OR** $\{f_1, \ldots, f_k\}$ = f_1 **or** … **or** f_k

To simplify the formalization of the translation we assume that the equality checking of two expressions without SCADE macros is replaced in the kernel by the conjunction of two inclusion checks. If SCADE macros are used on either side the equality is kept. Note that in that case the compared expressions will necessarily have to denote singleton sets (scalars), and the translation will raise an exception if that is not the case for a particular model. Similar exceptions will be raised when comparing numerical expressions and when SCADE macro parameters do not denote scalars.

$$_M\llbracket S \texttt{ :: } \textbf{everytime } \varphi \rrbracket \quad \triangleq \{\, {}_M\llbracket \varphi \rrbracket_{x,\texttt{entry(x)}} \mid \langle x \rangle \in M(S) \,\}$$

$$_M\llbracket \textbf{everywhere } r\ \varphi \rrbracket_{s,l} \triangleq \textbf{AND}\ \{\, {}_M\llbracket \varphi \rrbracket_{s,l'} \mid l' \in \texttt{spots}(s) \wedge \texttt{distance}(s,l,l') \in {}_M\llbracket r \rrbracket_{s,l} \,\}$$
$$_M\llbracket \textbf{nowhere } r\ \varphi \rrbracket_{s,l} \triangleq \textbf{AND}\ \{\, \neg {}_M\llbracket \varphi \rrbracket_{s,l'} \mid l' \in \texttt{spots}(s) \wedge \texttt{distance}(s,l,l') \in {}_M\llbracket r \rrbracket_{s,l} \,\}$$
$$_M\llbracket \textbf{somewhere } r\ \varphi \rrbracket_{s,l} \triangleq \textbf{OR}\ \{\, {}_M\llbracket \varphi \rrbracket_{s,l'} \mid l' \in \texttt{spots}(s) \wedge \texttt{distance}(s,l,l') \in {}_M\llbracket r \rrbracket_{s,l} \,\}$$
$$_M\llbracket \varphi \textbf{ until } r\ \Psi \rrbracket_{s,l} \triangleq \textbf{OR}\ \{\, \textbf{AND}\ \{\, {}_M\llbracket \varphi \rrbracket_{s,l''} \mid l'' \in \texttt{spots}(s) \wedge \texttt{between}(s,l,l',l'') \,\} \cup \{\, {}_M\llbracket \Psi \rrbracket_{s,l'} \,\} \mid l' \in \texttt{spots}(s) \wedge \texttt{distance}(s,l,l') \in {}_M\llbracket r \rrbracket_{s,l} \,\}$$

$_M⟦$**not** $\varphi⟧_{s,I}$ $\triangleq$ **not** $_M⟦\varphi⟧_{s,I}$

$_M⟦\varphi$ **and** $\Psi⟧_{s,I}$ $\triangleq$ $_M⟦\varphi⟧_{s,I}$ **and** $_M⟦\Psi⟧_{s,I}$

$_M⟦$**all** v **:** a **|** $\varphi⟧_{s,I}$ $\triangleq$ **AND** $\{ {}_{M^{\oplus v \to \{t\}}}⟦\varphi⟧_{s,I} \mid t \in {}_M⟦a⟧_{s,I} \}$

$_M⟦a$ **in** $b⟧_{s,I}$ $\triangleq$ if $_M⟦a⟧_{s,I} \subseteq {}_M⟦b⟧_{s,I}$ then **true** else **false**

$_M⟦a$ **=** $b⟧_{s,I}$ $\triangleq$ x **=** y where $_M⟦a⟧_{s,I} = \{\langle x \rangle\} \wedge {}_M⟦b⟧_{s,I} = \{\langle y \rangle\}$

$_M⟦$**some** $a⟧_{s,I}$ $\triangleq$ $|{}_M⟦a⟧_{s,I}| >$ **0**

$_M⟦$**lone** $a⟧_{s,I}$ $\triangleq$ $|{}_M⟦a⟧_{s,I}| <$ **2**

$_M⟦a⟧_{s,I}$ $\triangleq$ x where $_M⟦a⟧_{s,I} = \{\langle x \rangle\}$

$_M⟦a$ **<** $b⟧_{s,I}$ $\triangleq$ n **<** m where $_M⟦a⟧_{s,I} = \{\langle n \rangle\} \wedge {}_M⟦b⟧_{s,I} = \{\langle m \rangle\}$

$_M⟦a$ **|** $b⟧_{s,I}$ $\triangleq$ $_M⟦a⟧_{s,I} \cup {}_M⟦b⟧_{s,I}$

$_M⟦a$ **&** $b⟧_{s,I}$ $\triangleq$ $_M⟦a⟧_{s,I} \cup {}_M⟦b⟧_{s,I}$

$_M⟦a$ **** $b⟧_{s,I}$ $\triangleq$ $_M⟦a⟧_{s,I} \setminus {}_M⟦b⟧_{s,I}$

$_M⟦a$ **.** $b⟧_{s,I}$ $\triangleq$ $_M⟦a⟧_{s,I} \bullet {}_M⟦b⟧_{s,I}$ \

$_M⟦a$ **->** $b⟧_{s,I}$ $\triangleq$ $_M⟦a⟧_{s,I} \times {}_M⟦b⟧_{s,I}$

$_M⟦$**~** $a⟧_{s,I}$ $\triangleq$ $_M⟦a⟧_{s,I}{}^{\circ}$

$_M⟦$**^** $a⟧_{s,I}$ $\triangleq$ $_M⟦a⟧_{s,I}{}^{+}$

$_M⟦a$ **+** $b⟧_{s,I}$ $\triangleq$ $\{\langle n$ **+** $m \rangle\}$ where $_M⟦a⟧_{s,I} = \{\langle n \rangle\} \wedge {}_M⟦b⟧_{s,I} = \{\langle m \rangle\}$

$_M⟦a$ **-** $b⟧_{s,I}$ $\triangleq$ $\{\langle n$ **-** $m \rangle\}$ where $_M⟦a⟧_{s,I} = \{\langle n \rangle\} \wedge {}_M⟦b⟧_{s,I} = \{\langle m \rangle\}$

$_M⟦a$ ***** $b⟧_{s,I}$ $\triangleq$ $\{\langle n$ ***** $m \rangle\}$ where $_M⟦a⟧_{s,I} = \{\langle n \rangle\} \wedge {}_M⟦b⟧_{s,I} = \{\langle m \rangle\}$

$_M⟦a$ **/** $b⟧_{s,I}$ $\triangleq$ $\{\langle n$ **/** $m \rangle\}$ where $_M⟦a⟧_{s,I} = \{\langle n \rangle\} \wedge {}_M⟦b⟧_{s,I} = \{\langle m \rangle\}$

$_M⟦\#i(a_1, \ldots, a_k)⟧_{s,I}$ $\triangleq$ $\{\langle \#i(x_1, \ldots, x_k) \rangle\}$ where $_M⟦a_1⟧_{s,I} = \{\langle x_1 \rangle\} \wedge \ldots \wedge {}_M⟦a_k⟧_{s,I} = \{\langle x_k \rangle\}$

$_M⟦"c_1\{a_1\}\ldots\{a_k\}c_k"⟧_{s,I}$ $\triangleq$ $\{\langle c_1 x_1 \ldots x_k c_k \rangle\}$ where $_M⟦a_1⟧_{s,I} = \{\langle x_1 \rangle\} \wedge \ldots \wedge {}_M⟦a_k⟧_{s,I} = \{\langle x_k \rangle\}$

$_M⟦i⟧_{s,I}$ $\triangleq$ $\{ t \mid t \in M(i)\downarrow s \wedge ((t = \langle x \rangle \wedge x \in \text{dom}(\texttt{spots})) \rightarrow I \in \texttt{spots}(x)) \}$

$_M⟦v⟧_{s,I}$ $\triangleq$ $M(v)$

$_M⟦n⟧_{s,I}$ $\triangleq$ $\{\langle n \rangle\}$

$_M⟦u⟧_{s,I}$ $\triangleq$ $\{\langle u \rangle\}$

$_M⟦$**true**$⟧_{s,I}$ $\triangleq$ $\{\langle$**true**$\rangle\}$

$_M⟦$**false**$⟧_{s,I}$ $\triangleq$ $\{\langle$**false**$\rangle\}$

As an example of translating a rule consider the following simple example

```
route :: everytime #macro("varname_{routeEntry}") < 10
```

To simplify let's assume we have a single route `R` with entry location 0 and that *M*(routeEntry) = {⟨R,E⟩}. The translation would work as follows.

$_M⟦$**route :: everytime** `#macro("varname_{routeEntry}") < 10`$⟧$

=

$\{ {}_M⟦$`#macro("varname_{routeEntry}") < 10`$⟧_{R,0} \}$

=

$\{ n < m$ where $_M⟦$`#macro("varname_{routeEntry}")`$⟧_{R,0} = \{\langle n \rangle\} \wedge {}_M⟦10⟧_{R,0} = \{\langle m \rangle\} \}$

=
{ n < m where $_M$⟦#macro(“varname_{routeEntry}”)⟧$_{R,0}$ = {⟨n⟩} ∧ {⟨10⟩} = {⟨m⟩} }
=
{ n < 10 where $_M$⟦#macro(“varname_{routeEntry}”)⟧$_{R,0}$ = {⟨n⟩} }
=
{ n < 10 where {⟨#macro(x_1)⟩} = {⟨n⟩} ∧ $_M$⟦“varname_{routeEntry}”⟧$_{R,0}$ = {⟨x_1⟩} }
=
{ n < 10 where {⟨#macro(x_1)⟩} = {⟨n⟩} ∧ {⟨varname_y_1⟩} = {⟨x_1⟩} ∧ $_M$⟦routeEntry}⟧$_{R,0}$ = {⟨y_1⟩} }
=
{ n < 10 where {⟨#macro(x_1)⟩} = {⟨n⟩} ∧ {⟨varname_y_1⟩} = {⟨x_1⟩} ∧ { t | t ∈ M(routeEntry)↓R } = {⟨y_1⟩} }
=
{ n < 10 where {⟨#macro(x_1)⟩} = {⟨n⟩} ∧ {⟨varname_y_1⟩} = {⟨x_1⟩} ∧ { t | t ∈ {⟨E⟩} } = {⟨y_1⟩} }
=
{ n < 10 where {⟨#macro(x_1)⟩} = {⟨n⟩} ∧ {⟨varname_y_1⟩} = {⟨x_1⟩} ∧ ∧ {⟨E⟩} = {⟨y_1⟩} }
=
{ n < 10 where {⟨#macro(x_1)⟩} = {⟨n⟩} ∧ {⟨varname_E⟩} = {⟨x_1⟩}
=
{ n < 10 where {⟨#macro(varname_E)⟩} = {⟨n⟩} }
=
{ #macro(varname_E) < 10 }

Documents

The EVEREST environment relies on several document formats to coordinate the communication between the various components.

Project file

A project file specifies an EVEREST project, and is managed by the EVEREST DVT. It is simply composed by a set of railML files that belong to the topological model of the project - that is, the different areas of the railway project - and the set of active EVEREST rules selected from the EVEREST rule catalogue.

Rule catalogue

A file storing both infrastructure and interlocking EVEREST rules that is shared among the various EVEREST projects. Rules in the catalogue have a version number assigned, so that updated versions of rules can be created, rendering old ones deprecated. They are also assumed to have already been checked for syntax and typing rules. The Rule Validation Library, used by most other components of the EVEREST DVT, manages such files.

Topological rule violations

A file storing violations to topological rules previously found by the TRV. Its format is essentially a CSV file describing the violated rule, each line listing a unique id for the violation, the name of the violated rule, an identifier for the related railML file (i.e., the area),

the type of scope, an entity in that scope where that rule is violated, and additional element identifiers along the scope entity that should additionally be flagged as faulty. The Rule Validation Library, used by other components of the EVEREST DVT, manages such files.

Rule designer configuration

Files containing auxiliary information to ease the writing and validation of EVEREST rules with the RD. It consists of two CSV files:

- a file containing the definitions of EVEREST rule patterns, to be used by the RD to instantiate concrete rules;
- a file containing railML expression macros and the type of railML entities to be extracted from railML files.

These configuration files are not expected to be modified by the end-users of the tool.

Topology visualizer configuration

Files containing additional XAML resources to represent topological elements when drawing topologies. Each entry in the configuration consists of an element type (only a subset of elements is supported, those that are placed alongside the net elements, such as signals and train detection elements), a prefix that is used to filter only certain elements of that type from their assigned id, and path to a XAML file containing a drawing.

Observer specification

The output of the IRT for the active interlocking rules, contains the specification of observers that will be integrated into the RAS by the REP. Essentially contains a set of propositional logic formulas after the expansion of EVEREST interlocking rules for the topological models under analysis.

railML models

A previous document, tech report T1.3, has already discussed the need for a standard format to encode railway topological models to be at the center of the EVEREST ecosystem. Report T1.3 has selected railML as the most robust and expressive language. These files are managed by the Railway Design Tool component of the EVEREST plugin - currently, RaIL-AiD. The EVEREST DVT processes such files with the C# railML library, which is used by most other components.

SCADE models

SCADE has been selected as the Railway Automation Studio component of the EVEREST ecosystem. SCADE is a third-party platform which has its own format to encode models. The REP (integrated in the RAS) will generate such files from the observer specification generated by the IRT.

DXF drawings

AutoCAD has been selected as the Technical Drawing Tool of the EVEREST ecosystem. AutoCAD is a third-party platform with its own formats to store drawings; in tech report T1.3, the DXF was selected as the format best-suited to be used in the interaction of EVEREST

components with AutoCAD. The AEP (integrated in AutoCAD) manipulates such files, and the EPB processes them back into the EVEREST ecosystem.

Remarks