Technical Report

- **Project**: DigiLightRail
- **WP**: WWW
- **Deliverable**: T2.5
- **Producer**: HASLab / INESC TEC
- **Title**: Architecture and functional specification of the EVEREST tool
- **Summary**: *This report (.....)*

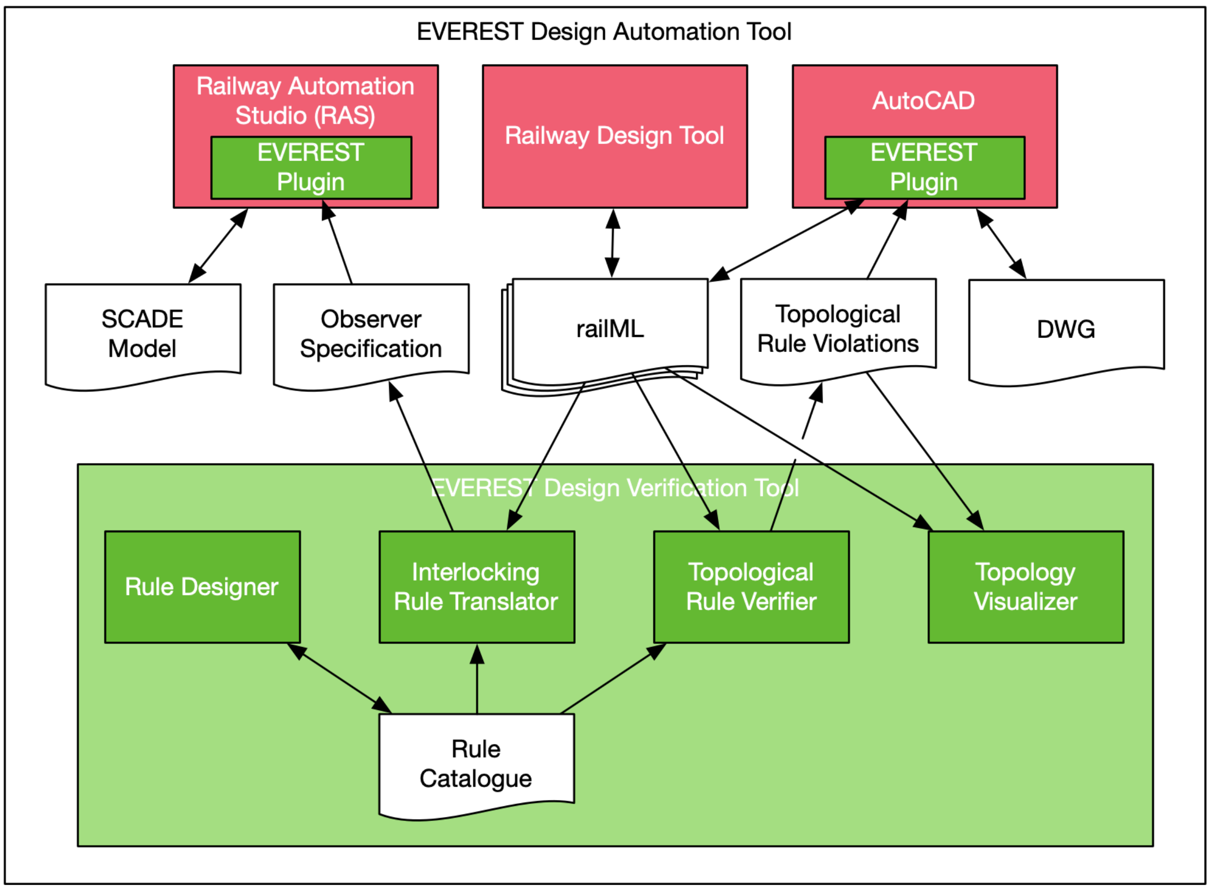
Introduction

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.

In this report, we provide the functional description of each of the above components. Although loosely coupled, the first four components will be packaged in a standalone application called **EVEREST Design Verification Tool (DVT)**. The RAS EVEREST Plugin will be specified in a companion tech report.


EVEREST Design Automation Tool
Railway Automation Studio (RAS)
EVEREST Plugin
Railway Design Tool
AutoCAD
EVEREST Plugin
SCADE Model
Observer Specification
railML
Topological Rule Violations
DWG
EVEREST Design Verification Tool
Rule Designer
Interlocking Rule Translator
Topological Rule Verifier
Topology Visualizer
Rule Catalogue

railML® restrictions

railML (Railway Markup Language) models are central to the EVEREST Design Automation Tool. For more information about railML please read the DigiLightRail tech report T1.4. In order to automate the processing, EVEREST makes a number of assumptions about its input railML models, that will be produced by a Railway Design Tool (such as RaIL-AiD), which are in line with the expected practice at Efacec and its partners, in particular:

- The model consists of a single network that contains all the declared topology elements (`netElements` and `netRelations`).
- Every `netElement` should have two endpoints represented by infrastructure elements (such as switches or buffer stops).
- A `screenPositioningSystem` coordinate system must be defined, and topology elements have `visualization` positions assigned on that system.
- The ids of elements in references are correctly typed, in the sense that they point to entities of the appropriate type.

Moreover, the project will focus on the latest stable version of the railML at the time of development, railML 3.1.

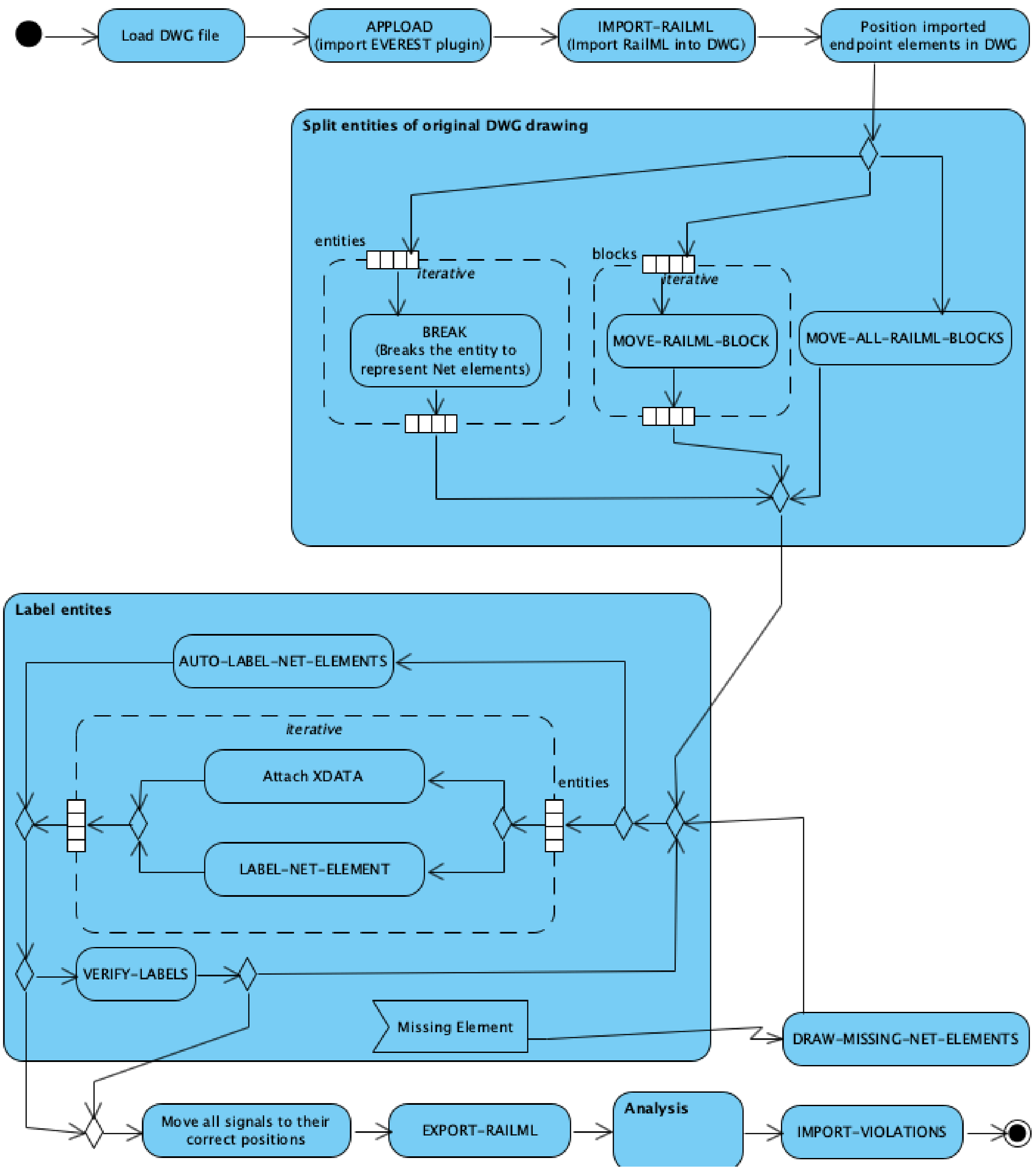
All functionalities described below that handle railML models assume that the provided model satisfies these assumptions.

EVEREST Workflow

The starting point of the EVEREST workflow is creating the railML topological model - possibly composed of multiple railML files - and the DWG technical drawing, in a *Railway Design Tool* and *AutoCAD*, respectively. At this point, the railML model misses some critical topological information necessary for the design validation, namely the lengths of `netElements` and the position of the remaining infrastructure elements along those.

Users can enrich the railML model with this information by using the *EVEREST plugin* for AutoCAD and performing the following steps (also depicted below in an activity diagram). The EVEREST plugin commands shown in bold will be described with more detail in the following section.

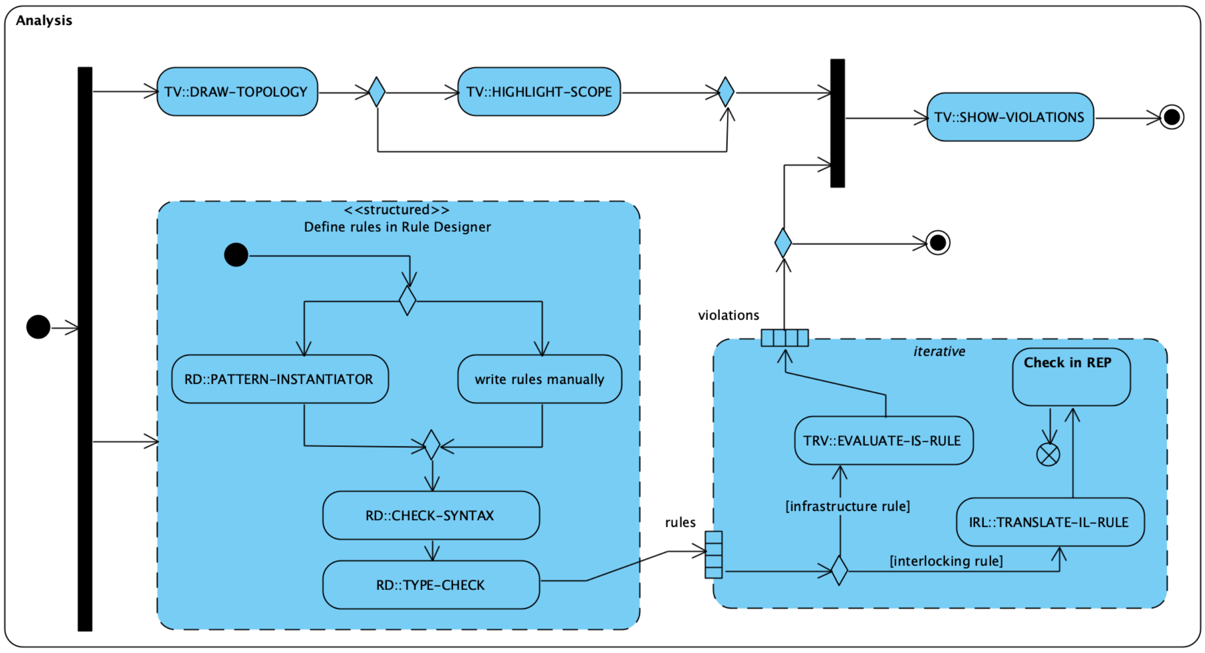
1. Launch AutoCAD and load both the DWG file and the *EVEREST plugin* with the standard `APPLOAD` command.
2. Import the topology elements into the drawing (**`IMPORT-RAILML`**).
3. Position the imported endpoint elements in the drawing.
4. Split entities of the original drawing into the `netElements` defined in the railML model. This step includes the following approaches:
    a. Use AutoCAD's `BREAK` command.
    b. Execute the **`MOVE-RAILML-BLOCK`** command for each railway block
    c. Execute the **`MOVE-ALL-RAILML-BLOCKS`** command.
5. Label each `netElement` with its corresponding id. This step includes the following approaches:
    a. Manually attach Xdata.
    b. Use the **`LABEL-NET-ELEMENT`** command for each `netElement`.
    c. Use the **`AUTO-LABEL-NET-ELEMENTS`** command.
6. If the original technical drawing misses any entities representing `netElements`, users can use the **`DRAW-MISSING-NET-ELEMENTS`** command.
7. Users can optionally execute the **`VERIFY-LABELS`** command to check if all net elements were correctly labelled.
8. Move the remaining, non-endpoint topological elements (such as `signals`) to their correct positions.
9. Execute the **`EXPORT-RAILML`** command to generate more accurate railML files (containing the lengths of `netElements` the updated positions of other infrastructure elements) for validation.
10. Analyse the resulting railML model in the EVEREST DVT.
11. Execute the **`IMPORT-VIOLATIONS`** command to load the violations that result from that analysis.


Load DWG file
APPLOAD (import EVEREST plugin)
IMPORT-RAILML (Import RailML into DWG)
Position imported endpoint elements in DWG
Split entities of original DWG drawing
entities
iterative
BREAK (Breaks the entity to represent Net elements)
blocks
iterative
MOVE-RAILML-BLOCK
MOVE-ALL-RAILML-BLOCKS
Label entites
AUTO-LABEL-NET-ELEMENTS
iterative
Attach XDATA
LABEL-NET-ELEMENT
entities
VERIFY-LABELS
Missing Element
DRAW-MISSING-NET-ELEMENTS
Move all signals to their correct positions
EXPORT-RAILML
Analysis
IMPORT-VIOLATIONS

Users can perform the infrastructural validation of railML models with the EVEREST DVT, a tool that, as mentioned above, integrates several components that support the specification of design rules, their evaluation against railML topological models, and subsequent visualization of violations. This process typically involves the following steps:

1. Use the *Topology Visualizer* component to import a railML model, previously modified by AutoCAD EVEREST plugin to include lengths and positions, by executing the **`DRAW-TOPOLOGY`** command.
2. Optionally, validate the existing routes using the **`HIGHLIGHT-SCOPE`** command.

3. Define rules to validate the model using the *Rule Designer* component. Users can either use the **PATTERN-INSTANTIATOR** command to ease the rule definition process or write the rules manually.
4. Check the rules' syntax and type correctness with the **CHECK-SYNTAX** and **CHECK-TYPE** commands, respectively.
5. Interlocking rules can be translated into specifications of observer components using the **TRANSLATE-IL-RULE** command of the *Interlocking Rule Translator*, to be subsequently incorporated in SCADE models using the RAS EVEREST plugin.
6. Use the **EVALUATE-IS-RULE** of the *Topology Rule Verifier* component to check the defined infrastructure rules against the imported railML models and detect possible violations.
7. Subsequently, use the *Topology Visualizer* to illustrate the infrastructure violations with the **SHOW-VIOLATION** command.
8. Additionally, infrastructure violations can also be depicted in the AutoCAD technical drawing by using the *EVEREST plugin* **IMPORT-VIOLATIONS** command.



…

The EVEREST DVT will also provide a unified, project-level interface to support this workflow. Here, each project is essentially composed of a set of relevant railML files and the set of rules (selected from a general rule catalogue, where all specified rules will be saved), that are active for that project (for each project the user will be able to select which rules are to be evaluated). Basic support for the versioning of rules is also provided in the rule catalogue, allowing the release of new versions. The *Rule Designer* component will also be configurable using a textual file that describes the supported railML elements and attributes, and macros available to simplify the usage of such entities.

EVEREST AutoCAD Plugin

The EVEREST plugin of DigiLightRail's architecture is an AutoCAD extension that includes a set of AutoLisp commands. This component allows the importing of railML models into DWG drawings as sets of DWG entities. Besides, it provides functionality for exporting railML models with updated topological information based on a DWG drawing. Last but not least, it can be used to depict rule violations in the drawing as a text value attached to entities that reference a railML element.

The operations presented below act at the railML file level, although a project may have its topological model split among multiple such files. Thus, the defined operations may be called multiple times with distinct inputs. To avoid conflicts, the traceability between elements and respective railML files must be preserved in the meta-data of the DWG file.

Functionalities

Users can interact with this environment in AutoCAD through an AutoLisp plugin. This plugin implements a set of AutoLisp commands that users can execute in AutoCAD. The functionalities implemented by these commands are the following:

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 (possibly part of) 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.*
Observations	*This command shall draw every net element in the model as an `LWPolyline` entity, paired with a text label indicating its id, in a layer called "`railml_net_elements`". Furthermore, it imports each infrastructure element of the railway model, such as switches, signals and borders, as references to blocks defined in a dedicated DWG library of railML blocks. Each block reference contains five attributes: the `NAME` that indicates the railML element's name; the `ID` that represents the railML element's id; the `NETELEMENT` that indicates the id of the net element to which the infrastructure element belongs; the `POS` corresponding to the element's pos value, and the `ERROR` attribute, used to hold the error messages originated from subsequent validation processes. Both the `POS` and `ERROR` attributes are created with empty values. Lastly, these infrastructure elements are added in a “`railml_elements`" layer. The topology is drawn according to the*		

	positions defined in the railML model by `visualization` elements following a `screenPositioningSystem`.

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.*
Observations	*This command starts by calculating the `POS` values of all elements based on the DWG drawing and stores these values in their position attribute. Moreover, it also computes all net elements' lengths. It then updates the railML document with the updated spotLocations pos and net elements' lengths values.* *This command assumes that every netElement id was already assigned to one of the drawing entities. The process of that assignment can either be done manually, by breaking entities in the drawing with AutoCAD's* **`BREAK`** *command and attaching Xdata with Express tools, or by executing the* **`MOVE-RAILML-BLOCK`** *and* **`LABEL-NET-ELEMENT`** *commands described below.*		

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.*
Observations	*The command lists every resulting violation in the error attribute of all relevant railML elements. Users should execute this command after executing the* **`EXPORT-RAIML`** *command and validate the resulting model. The command prompts the message "No violations were found" if the model fulfilled all specified rules.*		

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 a 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.)*
Observations	*Before applying the command, users should move a railway block located at the endpoint of a net element to a position near the entity one wishes to break. The command will move this block to the selected entity (line, arc, or polyline) and break this entity at that position. The command will output the message "Error: not a block" if the user selects an entity that is not a block, and the message "Error: not a railML block" if the user selects any block that is not a railML block.*		

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.*
Observations	*Before applying the command, users need to move all railway blocks located at the endpoints of net elements (signal blocks are not considered by this operation) to the respective positions near the lines or polylines one wishes to break. The command will move these blocks to the nearest entities (lines, arcs, or polylines) and break those entities at that position. As requirements, all entities to be broken must belong to the same layer and users need to move the blocks to a close enough location, so that the command is able to select the correct entity to break.*		

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.*
Observations	*n/a*		

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.*
Observations	*The command assumes that every net element contains two endpoints represented by railML infrastructure elements, represented in the drawing by railML blocks. Users should execute* `MOVE-RAILML-BLOCK` *or the* `MOVE-ALL-RAILML-BLOCKS` *commands to position all railML blocks correctly before running this command.*		

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.*
Observations	*This command draws a straight line connecting two railML block references representing the endpoints of a net element that could not be found in the drawing. The command also labels the drawn lines with their corresponding net element's Id as the* `LABEL-NET-ELEMENT` *command.*		

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.*
Observations	*This command checks for labels indicating the net element's Id. Then, it paints with a green colour every net element in the* `railml_net_elements` *layer imported with the* `IMPORT-RAILML` *command. Any missing or incorrect label is reported in orange.*		

Rule Language

Rules are needed to express required conditions for the safety of the system. Two type of rules can be identified:

- *Infrastructure* rules that express static conditions on the topology of the rail network, for instance guaranteeing adequate distances between speed signals along a route. These will be verified by the EVEREST DVT.

- *Interlocking* rules that express necessary *invariant* conditions to guarantee the safety of the routes which will be established by the system. For example, to ensure that all TVD sections along a route are correctly locked when the route entry signal has the aspect *proceed*. These will be translated to SCADE observers and checked with the RAS.

An excerpt of the proposed syntax for EVEREST rules is presented 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 / scadeMacro( (expr),* ) / 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
placeholder := $id
const       := number / string / true / false
```

The distinction between infrastructure and interlocking rules is introduced by the keyword **`everytime`**, which denotes a dynamic invariant.

Rules may refer to railML elements and attributes (`railMLId`) that will be automatically populated from railML infrastructure and interlocking subschemas. The concrete set of elements and attributes to be considered will be defined in the *Rule Designer* configuration file. All these entities will be represented by mathematical relations, more specifically, sets of tuples of railML identifiers. For example, we will have sets `signalIS`, `signalIL`, `switchIS`, or `route`, to denote the set of signals at the infrastructure level, signals at the interlocking level, switches at the infrastructure level, and routes, respectively. Notice that sets can also be seen as unary relations, whose tuples contain only one entity. Attributes and children elements will give origin to binary relations. For example, we will have a binary relation `isVirtual` that associates each interlocking signal with (at most) one boolean that indicates whether that signal is virtual or not. Other examples, derived from children elements instead of attributes, are the binary relation `routeEntry`, that associates each route with the respective route entry element, and the `refersTo` binary relation that, among others, relates interlocking signals or route entry elements with an auxiliary reference element, from which the respective referred entities can be accessed via the `ref` attribute.

Relational logic operators (inspired by the well-know Alloy[1] formal specification language) can be used in rules to compose these basic relations in order to quickly obtain from a given railML entity other, somehow related, entities. The fundamental relational operator is composition (`.`), that can be used to chain together relations. For example, to determine the entry signal of a route `r` we can use the expression `r.routeEntry.refersTo.ref`. Given the verbosity and high level of indirection of railML, the rule language will also support the usage of railML macros (`railMLMacro`), which will be auxiliary relations defined in the *Rule Designer* configuration file. For example, we could define a macro called `entrySignal` (a binary relation) that is defined as `routeEntry.refersTo.ref`, to simplify the computation of the entry signal of a route. Sets and relations computed using such relational operators can then be checked for equality or inclusion. It is also possible to check the cardinality of sets and relations using multiplicity operators. These basic atomic formulas can then be composed using standard First-order Logic connectives and quantifiers.

All rules are implicitly quantified over the scope defined in the rule (which can be a `route` or a `track`) and the rule is to be checked against all entities of that type. railML entities that are associated to a particular entity of the scope will be implicitly projected when evaluating the rule. To distinguish such scope-dependent elements, below they will be presented underlined. For instance, if the `scope` is `route`, <u>`entrySignal`</u>, the macro binary relation that identifies the signal that marks the entry point of a route, will be implicitly projected, and will become a (singleton) set that directly contains the entry signal. However, if the scope is `track`, `routeEntry` keeps its original unprojected value as a binary relation.

Spatial formulas quantify over the (spot) locations of an entity of the scope. Such is the case of **`everywhere`** formulas - that check a property from the current location along the entity - or **`before`** formulas - which check a property up to the current location along the entity. When

[1] http://alloytools.org

no spatial quantifiers are used, the rule applies to the location at the beginning of the scope entity. Elements that are located along the `netElements` in the scope entity (i.e., those that have a spot location) are implicitly projected to the location under evaluation. To distinguish those location-dependent elements, they will be presented in italic. For instance, signals have a location assigned, so when the set *`signalIS`* is used inside **`everywhere`**, at every spot the formula will be checked it will contain only the signals at that particular spot.

Spatial operators, like **`everywhere`** or **`somewhere`**, can further be parametrized with a range distance constraint along the scope entity. That constraint will determine the exact linear section of the entity to which the rule applies. In the case of **`everywhere`** it will restrict the surrounding area where the inner formula must always be true, and in the case of **`somewhere`** the area where it should be true.

Some infrastructure rule examples follow below, together with the respective formalization in the proposed language.

- All speed sections along a route have the respective signal at most 10m before.

  ```
  route :: everywhere all s: speedSection |
    somewhere [-10..0] some t: signalIS |
      t.isSpeedSignal.refersToBeginOfSpeedSection.ref = s
  ```

- All signals located somewhere in the network are the entry point of a route.

  ```
  track :: everywhere all s: signalIS |
           some r : route | r.entrySignal.refersTo.ref = s
  ```

- Every route entry is a virtual sign located at the beginning of the route.

  ```
  route :: entrySignal.isVirtual and
           entrySignal.refersTo.ref in signalIS
  ```

- After every signal along a route there is no switch in the next 20m.

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

Interlocking rules may also refer to elements that are used in the context of the SCADE verifier, and that should be treated as black-box by the EVEREST DVT components. The identifiers of such elements (`scadeMacro`), identified by a # mark, are parameterized macros to be later expanded by the RAS EVEREST plugin. They can only be used if the rule begins with the everytime keyword, otherwise a syntactic error is raised. As an example of an invariant, we can specify that everytime the aspect of the entry signal of a route has the aspect proceed all TVD sections of the route are locked to the respective route.

```
route :: everytime #aspect(entrySignal) = #proceed() implies
              all t : hasTVDSection.ref | #locked(t)
```

In the Rule Designer configuration file it will also be possible to define rule patterns using the same language, but where identifiers marked with a `$` denote `placeholders` to be later instantiated when adding rules to the catalogue. For example, we could write a generic pattern that states that everywhere along a route the distance between elements of type X is at least Y, as follows:

```
route :: everywhere (some $X implies everywhere [0..$Y] no $X)
```

Rule Designer

This component will help design both types of rules defined above, checking them for syntactic and type correctness. Support for the instantiation of rules from boiler plates for useful specification patterns will also be provided.

Functionalities

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.*
Observations	*For a syntactically incorrect rule such as* `route :: entrySignal.isVirtual & some entrySignal.refersTo.ref` *An error (with the proper location) should be returned since in the right-hand side of the intersection operator we have a formula instead of an expression, as required by the grammar.*		

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.*
Observations	*For a syntactically correct rule such as* `route :: entrySignal in speedZone`		

	A type error (with the proper location) should be returned since `entrySignal` does not return `speedZone` elements. *Note that placeholders `$id` and macros `#id` are not expected to be type checked.*

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.*
Observations	*Patterns are rules with placeholders, which are identifiers marked with the special symbol $, as exemplified above. Given the pattern to be instantiated, this operation will interact with the user of the tool to obtain the elements in the model that will be used to create the instantiated rule. We foresee that this interaction will be supported by a, to be designed, dialogue window.*		

Topology Visualizer

This component provides a UI that allows users to visualize the network's topology of a railML model. Furthermore, it provides a railML route table that users can interact with to highlight tracks and routes in the topology drawing. Lastly, it can illustrate any violations that resulted in a validation process.

Functionalities

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 passed by the DVT or selected with a file picker.*
Observations	*Draws all net elements and all infrastructure elements of a railML model. This operation uses the positions defined by `visualization` elements following a `screenPositioningSystem`.*		

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 passed by the DVT or selected 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.*
Observations	*Highlights each net element that encompasses the selected scope element and every infrastructure element involved in that element.*		

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 passed by the DVT or selected with a file picker.*
Input 2	`violations`	`csv`	*CSV data containing a set of violations resulted in the validation process. This input value is passed by the DVT 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.*
Observations	*Highlights in the topology the elements relevant to the selected violation.*		

Topological Rule Verifier

Topological rules express static conditions over the topology of the rail network that are needed to guarantee safe operation. This component is responsible for verifying that the topology of the railML model is consistent with the rules defined with the help of the Rule Designer component. The component generates a list of found violations.

Functionalities

Functionality	`EVALUATE-IS-RULE`
Description	*Evaluates an infrastructure rule over a railML topological model, returning the found violations.*

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* `#` *mark (which refer to SCADE elements)*
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.*

Observations	*This component effectively implements the semantics of the topological rules.*
	
	How the elements involved in a violation are identified from the rule remains to be defined.
	
	The constraints assumed to hold on a valid railML model are listed in the beginning of this document.
	
	For an infrastructure `rule`
	
	`(“switch-free zone”,`
	`route ::` **`everywhere`** `(`**`some`** ***`signalIS`*** **`implies everywhere`**`[<20m]` **`no`** ***`switchIS`***`))`
	
	if signals `s1` *and* `s2` *broke the rule within route* `r1` *of the* `railml` *model, then the following entry would be added to the output CSV.*
	
	`“1”,“switch-free zone”,route,“r1”,“s1,s2”`		

Interlocking Rule Translator

This component is responsible for translating interlocking rules to propositional logic formulas that specify observers to be later incorporated in SCADE models by the RAS EVEREST plugin. Interlocking rules express necessary conditions for the safety of the system and are designed in the Rule Designer component (see above).

Functionalities

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*

Observations	*This component translates the specified rule into propositional logic formulas. It assumes nothing about the SCADE model over which the rule will be verified; rule identifiers marked with* `#` *are macros that should later be expanded into SCADE identifiers by the RAS EVEREST plugin.*
	
	For example, given a railML model with two routes, `R1` *and* `R2`*, with route entry signals* `S1` *and* `S2`*, respectively, and TVD sections* `{T1,T2}` *and* `{T2,T3}`*, respectively, an interlocking* `rule`
	
	`(“all locked”,`
	`route :: everytime #aspect(entrySignal) = #proceed() implies`
	`        all t : hasTVDSection.ref | #locked(t))`
	
	the following propositional logic formulas would be generated by this translator:
	
	`(#aspect(R1,S1) = #proceed(R1) implies #locked(R1,T1)) and`
	`(#aspect(R1,S1) = #proceed(R1) implies #locked(R1,T2))`
	
	`(#aspect(R2,S2) = #proceed(R2) implies #locked(R2,T2)) and`
	`(#aspect(R2,S2) = #proceed(R2) implies #locked(R2,T3))`		