

Technical Report

- **Project:** DigiLightRail
- **WP:** WWW
- **Deliverable:** T5.4
- **Producer:** HASLab / INESC TEC
- **Title:** Testing the EVEREST tool - Unit testing
- **Summary:** *This document reports on the unit testing of the libraries that comprise the EVEREST Design Verification Tool.*

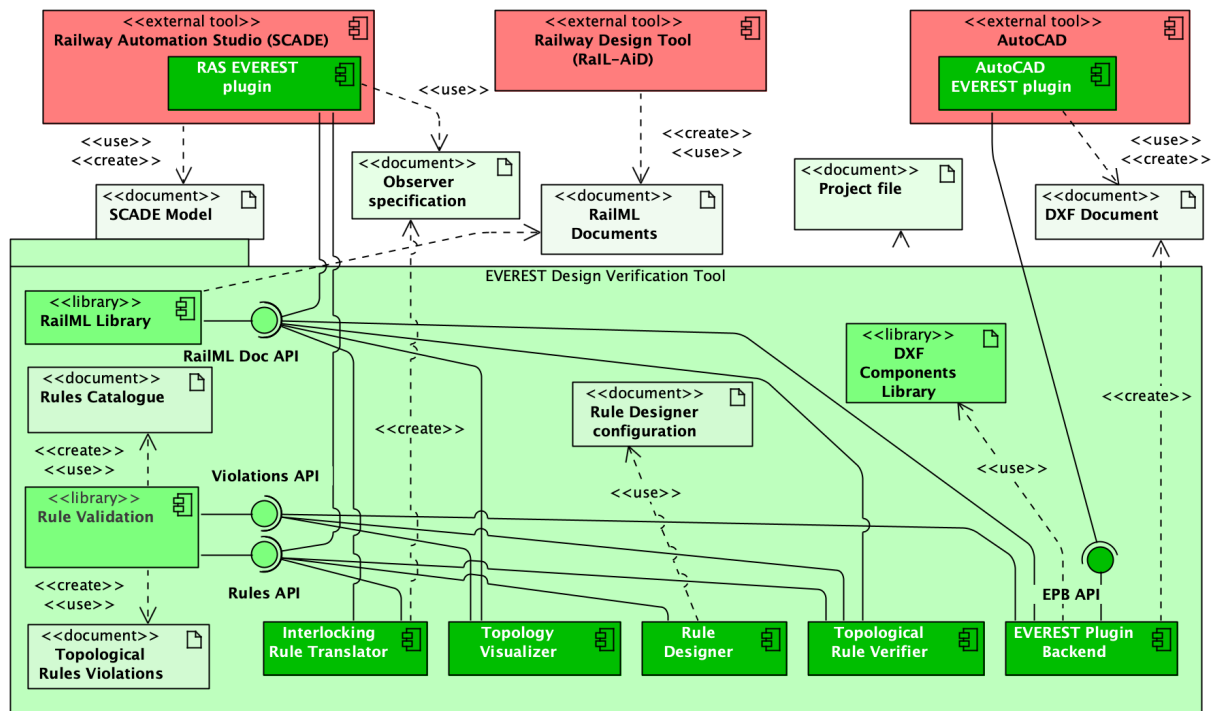
Introduction

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 (Efacec Verification of Railway nEtworKs Tool). The goal of this tool is to automatically validate the design of railway infrastructures through user-defined infrastructure and interlocking rules. This tool's architecture, as shown in the picture below, is a set of loosely coupled components. These components will support the specification of rules, their verification against concrete railway topologies, and subsequent reporting of validation. The components are the following:

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

On a technical level, these components are implemented as described in the component diagram below (see report T2.6 for details).



In this document we report on the unit testing of the libraries that comprise the **EVEREST Design Verification Tool**. The interaction of the Design Verification Tool with the remaining elements of the EVEREST environment will be addressed by integration testing (see report T6.3).

Scope

Not all functionalities of the EVEREST Design Verification Tool are amenable to unit testing, since some do not have inputs or outputs that can be easily encoded and tested programmatically. In particular our unit testing strategy focuses on the following components:

- **Rule Designer: CHECK-SYNTAX and CHECK-TYPE.** These functionalities are essential to guarantee the correct definition of EVEREST rules. The third functionality of RD, **INstantiate-PATTERN**, is trivially implemented and will not be subjected to exhaustive testing.
- **Topological Rule Verifier: EVALUATE-IS-RULE.** This is the only functionality of the TRV and one of the main functionalities of the Design Verification Tool.
- **Topology Visualizer: none.** All functionalities of the TV have a graphical outcome that is not amenable to be unit tested.
- **Interlocking Rule Translator: TRANSLATE-IL-RULE.** This is the only functionality of the IRT and one of the main functionalities of the Design Verification Tool.
- **RAS EVEREST Plugin: none.** The REP is outside the scope of the Design Verification Tool, being only fed observer specifications by the IRT. It will be the subject of integration testing (report T6.3).
- **AutoCAD EVEREST Plugin: none.** The AEP is outside the scope of the Design Verification Tool, relying internally on the EPB component. It will be the subject of integration testing (report T6.3).

Besides the high-level functionalities, the testing of the low-level libraries must also be taken into consideration. In particular:

- **railML library:** the key functionality is the extraction of the abstract view of the topology from railML as described in report T2.6. Thus, we performed direct unit tests for the spots and distance functions. This library is also indirectly tested through the unit testing of the TRV, IRT and EPB.
- **Rule Validation library:** directly addressed by the testing of the RD.
- **EVEREST Plugin Backend library:** most functionalities will be tested during integration testing of the AEP (document T6.3).

Strategy and technologies

The decision to focus on high-level functionalities allows us to define the input of the unit tests as external text files (rather than programmatically). This allows members of the team other than the developers to specify the tests, according to best software testing practices. In particular, the inputs of the various operations include:

- Rule Designer configuration files (patterns are not tested in this process)
- railML files, already populated with the element's lengths and positions
- Identifiers of specific elements of those railML files
- EVEREST expressions and rules (both infrastructure and interlocking)

The table below shows a summary of the inputs fed to each operation under test and the expected output. They are described in more detail in the following section.

Operation	Input	Output
spots (of a scope element)	railML file, scope element	All spot locations along the scope element, their direction and position
spots (of an element)	railML file, infrastructure element	The element's spot location (network element, orientation and position)
distance	railML file, scope element, two spots in the scope element	The distance between the two spots along a scope, or "NA" if unreachable
CHECK-SYNTAX	RD configuration file, EVEREST rule	"OK", or syntax error code and location
CHECK-TYPE (expression)	RD configuration file, scope type, syntax-checked EVEREST expression	"OK" and type, or type error code and location
CHECK-TYPE (rule)	RD configuration file, syntax-checked EVEREST rule	"OK", or type error code and location

EVALUATE-IS-RULE (expression)	RD configuration file, railML file, scope element, spot in the scope element, type-checked EVEREST expression	“OK” and tuple set, or runtime error code and location
EVALUATE-IS-RULE (rule)	RD configuration file, railML file, type-checked EVEREST rule	“OK” and Boolean, or runtime error code and location
EVALUATE-IS-RULE (violations)	RD configuration file, railML file, valid failing EVEREST rule	Violation element
TRANSLATE-IL-RULE	RD configuration file, railML file, type-checked EVEREST rule	“OK”

For the tests that output an error code, the following table presents the possible result codes, along with their descriptions. Codes 1 through 4 denote typing errors, code 5 syntax errors, and codes 6 through 9 runtime errors.

Code	Message
0	OK
1	Irrelevant expression error
2	Invalid expression arity
3	Wrong type, not a bool
4	Wrong type, not a number
5	Syntax error
6	Invalid infrastructure error
7	Not a bool singleton
8	Not a number singleton
9	Division by zero

The components that are being tested in this report are all developed in .NET, which has robust mechanisms to support the deployment and reporting of unit tests. In addition, we require the following components to support the definition of test cases and the retrieval of reports:

- NUnit: This library allows the development of parameterized tests that can receive input from CSV files.

- Fine Code Coverage: Extension for Visual Studio to perform code coverage while executing the create unit tests.

For instance, consider the testing of **CHECK-SYNTAX**. The test input is a CSV file that contains four columns: configuration file, rule, an integer value indicating the syntax correctness and the expected rule position where the syntax error will occur.

```
"config.csv","route :: routeEntry.refersTo in signalIL","0","-1"
"config.csv","route :: routeExit.refersTo.ref = 1","0","-1"
"config.csv","route :: routeExit","5","9"
"config.csv","route :: routeEntry.refersTo.ref.isVirtual =", "5","42"
"config.csv","route :: routeEntry.refersTo.ref.approachSpeed = 0","0","-1"
```

Below is an illustration of the test explorer provided by Visual Studio to depict the results from these unit tests.

The screenshot shows the Visual Studio Test Explorer interface. On the left, a tree view lists tests: EVEREST Tester (10), EVEREST (10), RailmlTest (5), ValidationTest (5), and TestSyntax (5). The 'TestSyntax' test is expanded, showing several sub-tests. One sub-test, 'TestSyntax("rou...", 41', is highlighted with a red 'X' icon, indicating a failure. The right pane shows the 'Test Detail Summary' for this failed test. It includes the source file 'ValidationTest.cs' at line 24, a duration of 41 ms, and a message: 'Expected: False But was: True'. A stack trace is also visible, pointing to 'ValidationTest.TestSyntax(String rule, Boolean expected)' at line 27.

While below is an illustration of the coverage information provided by Fine Code Coverage for the same tests.

Coverage

Summary

Risk Hotspots

Rate & Review

Log Issue/Suggestion

Buy me a coffee

Collapse all

Expand all

Filter:

▼ Name	▼ Covered	▼ Uncovered	▼ Coverable	▼ Total	▼ Line coverage	
⊕ EVEREST Tester	118	0	118	201	100%	<div></div>
⊖ RailIML	549	107	656	4275	83.6%	<div></div>
Border	4	0	4	185	100%	<div></div>
BufferStop	4	0	4	185	100%	<div></div>
InfraElement	16	5	21	185	76.1%	<div></div>
InfraElementParser	9	0	9	259	100%	<div></div>
NetElement	71	15	86	141	82.5%	<div></div>

The coverage information is also applied to the source code, so that execution paths not covered by the test suite can be easily identified.

```

12  {
13      public static List<Violation> Evaluate(RailmlDocument doc, string rule)
14      {
15          AntlrInputStream inputStream = new(rule);
16          RailmlRuleLanguageLexer lexer = new(inputStream);
17          CommonTokenStream tokenStream = new(lexer);
18          RailmlRuleLanguageParser parser = new(tokenStream);
19          EvaluationVisitor visitor = new(doc);
20          RailmlRuleLanguageParser.RailmlRuleContext railmlRule = parser.railmlRule();
21          _ = visitor.Visit(railmlRule);
22          return visitor.Violations;
23      }
24
25      public static void CheckType(RailmlDocument doc, string rule)
26      {
27          AntlrInputStream inputStream = new(rule);
28          RailmlRuleLanguageLexer lexer = new(inputStream);
29          CommonTokenStream tokenStream = new(lexer);
30          RailmlRuleLanguageParser parser = new(tokenStream);
31          TypeVisitor visitor = new(doc);
32          RailmlRuleLanguageParser.RailmlRuleContext railmlRule = parser.railmlRule();
33          _ = visitor.Visit(railmlRule);
34      }
35
36      public static ParserResult CheckSyntax(string rule)
37      {
38          return Parse(rule);
39      }
40
41      public static ParserResult Parse(string rule) { ... }

```

Results and coverage

railML Library

As mentioned, we defined three test suites for the railML library. The first suite verifies all spot locations extracted across a scope element. The suite is defined in a CSV file, [available](#) in the repository along with the railML files. It has 12 unit tests and contains the following fields:

- **railML file:** railML filename of the area to test.
- **Scope element:** identifier of a scope element (track, route or network) on which to evaluate the rule.
- **Spot locations:** list of the spot location tuples (net element id, orientation and position) expected for the scope element.

The second suite tests if the spot location of an infrastructure element is correctly extracted. The suite is defined in a CSV file, [available](#) in the repository along with the railML files. It has 35 unit tests and contains the following fields:

- **railML file:** railML filename of the area to test.
- **Infrastructure element:** identifier of the functional infrastructure element (signalIS, switchIS, ...) whose spot location will be checked.
- **Net element:** identifier of the network element of the infrastructure element's spot location.
- **Orientation:** orientation of the infrastructure element's spot location in the network element, 0 (normal) or 1 (reverse).

- **Position:** position of the infrastructure element's spot location in the network element.

The last suite checks whether the distances between two spot locations along a scope element is correctly calculated. The suite is defined in a CSV file, [available](#) in the repository along with the railML files. It has 58 unit tests and contains the following fields:

- **railML file:** railML filename of the area to test.
- **Scope element:** identifier of a scope element (track, route or network) on which to evaluate the rule.
- **Spot location 1:** first spot location tuple (net element id, orientation and position) in the scope element.
- **Spot location 2:** second spot location tuple (net element id, orientation and position) in the scope element.
- **Distance:** expected distance between the two spot locations. A positive distance means spot 2 is reachable from spot 1 in the specified orientation, a negative distance that it is reachable in the opposing orientation. "NA" is reported if they are unreachable along the scope element (e.g., have opposing orientation).

These three test suites ensured the following line coverage and branch coverage of the railML library:

▼ Name	▼ Covered	▼ Uncovered	▼ Coverable	▼ Total	▼ Line coverage	▼ Covered	▼ Total	▼ Branch coverage
+ RailML	1012	136	1148	7384	88.1%	353	430	82%
— EVEREST Tester	587	14	601	1153	97.6%	86	94	91.4%
EVEREST_Tester.UnitTests.RailmlTester	81	0	81	150	100%	16	16	100%

CHECK-SYNTAX

We defined one test suite to check the correctness of the syntax checker component of the Rule Designer, which test whether the syntax checker correctly identifies syntactically incorrect rules. In case of errors, the test also reports the position of the rule's offending token that originated the syntax error. The suite is defined in a CSV file, [available](#) in the repository along with the configuration files. It has 143 unit tests and contains the following fields:

- **Configuration file:** configuration filename, contains the valid railML attributes and their types to build EVEREST rules.
- **Rule:** the EVEREST rule to test.
- **Result code:** expected result for syntax checking, 0 (OK) or error code (5).
- **Error location:** position of the rule offending token (or -1 if the rule is syntactically correct).

As we shall see next, additional test rules are also defined to validate the type-checking component. Since in that test suite rules are assumed to be type-checked, those rules are also passed through the syntax checker as positive test cases.

Combined, these two test suites ensured both a line coverage and branch coverage of 100% in the syntax checker component, as depicted below:

▼ Name	▼ Covered	▼ Uncovered	▼ Coverable	▼ Total	▲ Line coverage	▼ Covered	▼ Total	▼ Branch coverage
— Validation	1981	204	2185	13422	90.6%	656	790	83%
Validation.SyntaxChecker	27	0	27	82	100%	10	10	100%
Validation.SyntaxVisitor	25	0	25	82	100%	8	8	100%

CHECK-TYPE

For the type checker component of the Rule Designer, as already stated we defined two test suites: one to verify the type correctness of complete EVEREST rules and another for the grammar subset consisting of expressions. The suite for expressions is defined in a CSV file, [available](#) in the repository along with the configuration files. It has 130 unit tests and contains the following fields:

- **Configuration file:** configuration filename, contains the valid railML attributes and their types to build EVEREST rules.
- **Scope type:** scope type for which to type check the rule (track, route or network).
- **Expression:** EVEREST expression to test, assumed to be syntactically correct.
- **Result code:** expected result for type checking, 0 (OK) or error code (1 through 4).
- **Expected result:** the type of the expression, or the offending token in case of errors.

The suite for rules is defined in a CSV file, [available](#) in the repository along with the configuration files. It has 93 unit tests and contains the following fields:

- **Configuration file:** configuration filename, contains the valid railML attributes and their types to build EVEREST rules.
- **Rule:** EVEREST rule to test, assumed to be syntactically correct.
- **Result code:** expected result for type checking, 0 (OK) or error code (1 through 4).
- **Expected result:** position of the rule offending token (or “OK” if the rule type checks, since rules always have type Boolean).

The combination of these tests produced the following code and branch coverage:

▼ Name	▼ Covered	▼ Uncovered	▼ Coverable	▼ Total	▲ Line coverage	▼ Covered	▼ Total	▼ Branch coverage
— Validation	1981	204	2185	13422	90.6%	656	790	83%
Validation.TypeChecker	23	0	23	504	100%	0	0	
Validation.TypeVisitor	318	1	319	504	99.6%	112	142	78.8%

EVALUATE-IS-RULE

The two test suites for the Topological Rule Verifier follow the same structure as the type checking component, focusing first on the evaluation of expressions and then on full infrastructure rules. The suite for expressions is defined in a CSV file, [available](#) in the repository along with the railML and configuration files. It has 115 unit tests and contains the following fields:

- **Configuration file:** configuration filename, contains the valid railML attributes and their types to build EVEREST rules.
- **railML file:** railML filename of the area to test.
- **Scope element:** identifier of a scope element (track, route or network) on which to evaluate the rule.
- **Spot:** spot location in the scope element from where to evaluate the expression.
- **Expression:** EVEREST expression to test, assumed to be type-checked.
- **Result code:** expected result for evaluation, 0 (OK) or error code (6 through 9).
- **Expected result:** the tuple set resulting from evaluating the expression, or the offending token in case of errors.









The suite for rules is defined in a CSV file, [available](#) in the repository along with the railML and configuration files. It has 176 unit tests and contains the following fields:

- **Configuration file:** configuration filename, contains the valid railML attributes and their types to build EVEREST rules.
- **railML file:** railML filename of the area to test.
- **Rule:** EVEREST infrastructure rule to test, assumed to be type-checked.
- **Result code:** expected result for evaluation, 0 (OK) or error code (6 through 9).
- **Expected result:** the Boolean resulting from evaluating the rule, or the offending token in case of errors.

A third suite of tests focus on checking whether the violations are correctly reported by the evaluator. The suite is defined in a CSV file, [available](#) in the repository along with the railML and configuration files. It has 7 unit tests and contains the following fields:

- **Configuration file:** configuration filename, contains the valid railML attributes and their types to build EVEREST rules.
- **railML file:** railML filename of the area to test.
- **Rule:** EVEREST infrastructure rule to test, assumed to be type-checked and invalid.
- **Violation:** expected identifier of an infrastructure element causing the violation to the rule.

These three test suites achieved the following code and branch coverage:

▼ Name	▼ Covered	▼ Uncovered	▼ Coverable	▼ Total	▲ Line coverage	▼ Covered	▼ Total	▼ Branch coverage
— Validation	1981	204	2185	13422	90.6% 	656	790	83% 
Validation.EvalState	21	0	21	32	100% 	0	0	
Validation.Evaluator	35	0	35	806	100% 	0	0	
Validation.EvaluatorVisitor	524	9	533	806	98.3% 	202	214	94.3% 








TRANSLATE-IL-RULE

The tests for the Interlocking Rule Translator work differently from the other components' tests. Due to difficulties in foreseeing the exact shape of the output SCADE observer specifications, we do not define the expected outputs of the translation. Instead, the tests only check if the translator can successfully translate interlocking rules and write the

translated formulas into a text file (i.e., no errors are thrown). The output was then manually checked for correctness. The suite is defined in a CSV file, [available](#) in the repository along with the railML and configuration files. It has 57 unit tests and contains the following fields:

- **Configuration file:** configuration filename, contains the valid railML attributes and their types to build EVEREST rules.
- **railML file:** railML filename of the area to test.
- **Rule:** EVEREST interlocking rule to test, assumed to be type-checked.

The test suite made for the translator component achieved the following code and branch coverage:

▼ Name	▼ Covered	▼ Uncovered	▼ Coverable	▼ Total	▲ Line coverage	▼ Covered	▼ Total	▼ Branch coverage
— Validation	1981	204	2185	13422	90.6% 	656	790	83% 
Validation.Translator	11	0	11	683	100% 	0	0	
Validation.TranslatorVisitor	472	5	477	683	98.9% 	184	208	88.4% 
Validation.TranslatorResult	9	3	12	497	75% 	0	0	