ERTMSFormalSpecs User Guide

Version 1.0.4

1 Table of contents

Revision history

Version	Date	Name	Description	Paragraphs
0.2.1	15/12/2010	Stanislas Pinte	First version	All
0.2.2	15/12/2010	Laurent Ferier	Document revision	All
0.2.3	06/01/2011	Svitlana Lukicheva	Added in/out variables	5.3, 6.4
0.3.1	28/01/2011	Laurent Ferier	Document revision according to release 0.3	All
0.3.2	21/02/2011	Svitlana Lukicheva	Added chapter "Reports"	8
0.4.0	04/03/2011	Laurent Ferier	Document revision according to release 0.4	All
0.5.0	05/04/2011	Laurent Ferier	Document revision according to release 0.5	All
0.6.0	13/05/2011	Svitlana Lukicheva, Laurent Ferier	Document revision according to release 0.6	All
0.7.0	17/10/2012	Svitlana Lukicheva, Laurent Ferier	Document revision according to release 0.7	All
1.0.0	15/01/2014	Luis Marcos, Laurent Ferier, James Oakey	Document revision according to release 1.35	All
1.0.1	26/03/2015	Svitlana Lukicheva	Interfaces documented, tables of figures, equations and index of tables moved at the end	All
1.0.2	22/04/2015	James Oakey	Model updates documented	7

1.0.3	24/04/2015	Laurent Ferier	Integrated the technical design document as chapter 13 and 14	13 and 14
1.0.4	01/07/2015	Laurent Ferier	Minor changes	

2 Introduction

This document is a User's Guide to the ERTMSFormalSpecs Workbench (EFSW). It details the features and functions of the ESFW.

2.1 References

Index	Title	Date
[1]	EFSW Technical design	15/12/2010
[2]	Subset-026 System requirements Specification V3.4.0	06/01/2015

2.2 Terms and definitions

Term	Definition
BL	Baseline
EFSW	ERTMSFormalSpecs Workbench
EFS	ERTMSFormalSpecs
EFSM	ERTMSFormalSpecs Model
EFST	ERTMSFormalSpecs Test
EVC	European Vital Computer
BNF	Backus-Naur Form

3 ERTMSFormalSpecs Workbench Introduction

3.1 The language

The ERTMSFormalSpecs language is an ad-hoc language, developed by *ERTMS Solutions* for the sole purpose of modelling ERTMS specifications.

The EFS model is made up of a data dictionary containing requirements, traceability information, types, functions, procedures, variables, and all the rules and tests organized hierarchically. These represent the behaviour of the trainborne or trackside systems, as described in ERA's Subsets.

The EFS model for trainborne system is a non-trivial artefact (estimated at 2000 rules, 500 variables, 1500 tests, 500 pages of specifications), far too large and complex to be managed without ad hoc tools.

3.2 The workbench

The ERTMSFormalSpecs Workbench (EFSW) is a graphical tool designed to develop, maintain, and document model-based development for the Subset-026, Subset-027, Subset-034, Subset-076 and Optional Documents related with the ETCS/ERTMS system, such as "*DMI Start and Stop Conditions*". It is a desktop application, running on the *Microsoft Windows platform*. EFSW has also been successfully used to model trackside systems, such as Interlocking or RBC.

EFSW provides high-level features:

- **Requirement analysis tool,** used to manage the requirements related to the system and manage traceability between requirements, model and tests (Section 4).
- **Model browser,** used to define the system's structure and dynamics (Section 5).
- Test browser and execution environment, used to test the model and animate it (Section 6).
- **A specific tool** to automatically translate tests as described in Subset-076 (Section 7).

EFSW provides all features and functions required to support these high-level features, explained in details in the following sections.

3.3 Workbench installation

The complete EFS environment is distributed in a setup file which automatically installs the tool in the installation directory

C:\Program Files (x86)\ERTMSSolutions\ERTMSFormalSpecs

The path where the EFS environment is installed can be changed during the setup phase.

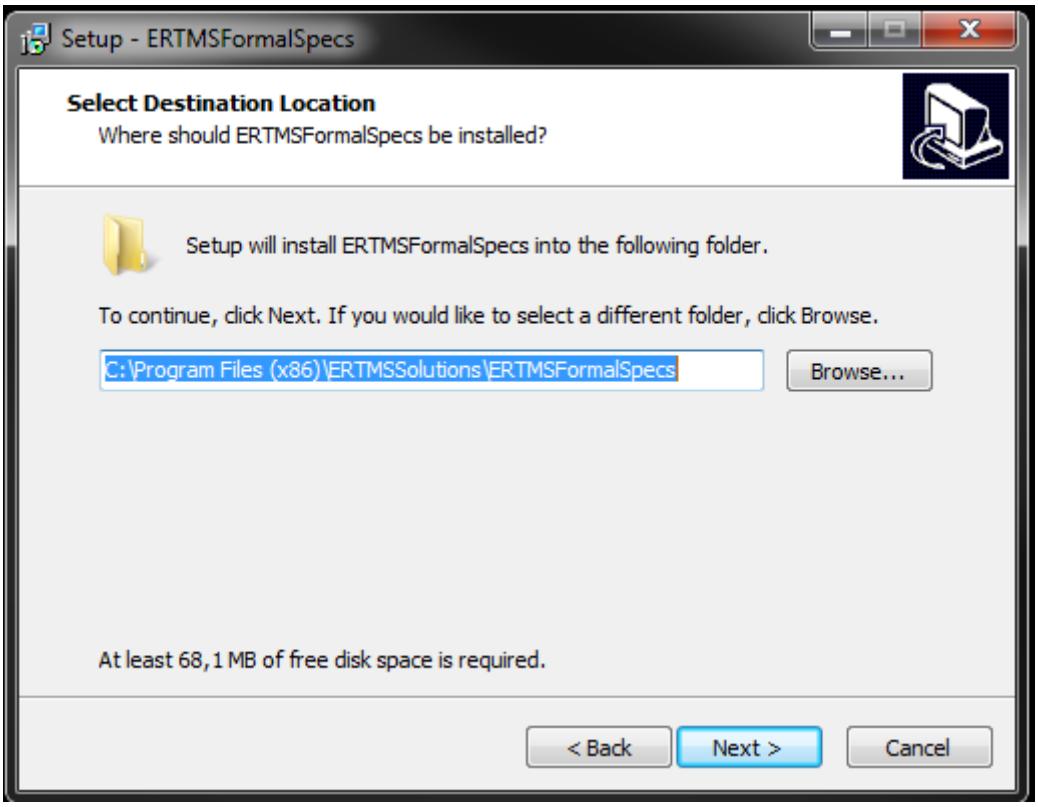

Figure 1: Path selection during installation

The files related to the model are placed in the Documents\ERTMSSolutions folder.

The installation process creates icons on the windows start menu to launch EFSW and access the user guide of the ERTMSFormalSpecs Workbench. Table 1 indicates the minimum requirements:

Architecture	64 bits
Operating System	Windows 7
Framework	.NET 4.0 or higher

Table 1: Minimum requirements for installing EFSW

The installer also checks that the required .NET framework 4.0 is installed on the target machine before performing the installation.

3.4 Initial workbench state

The initial state of EFSW main window when it is launched is empty, as Figure 2 shows.

Figure 2: Initial EFSW main window status.

3.4.1 Opening a EFS file

The EFS files are stored on:

${Documents}\ERTMSFormalSpecs

Where the ${Documents} references the user's Document folder.

This directory holds, among other documents, the model and test files used to describe the trainborne system. To start modelling the trainborne system the user is required to open one of these files, as Figure 3 and Figure 4 depict.

Figure 3: Open an EFS file.

To open a file, go to: File -> Open, select an .efs File and click on Open. Files with the .efs extension are the root files describing the model. They rely on other files to describe the system

- **Files with the extension .efs_ns** to describe the namespaces used in the model
- **Files with the extension .efs_tst** to describe the tests used to verify the model

Such files are located in directories below the .efs file they refer to.

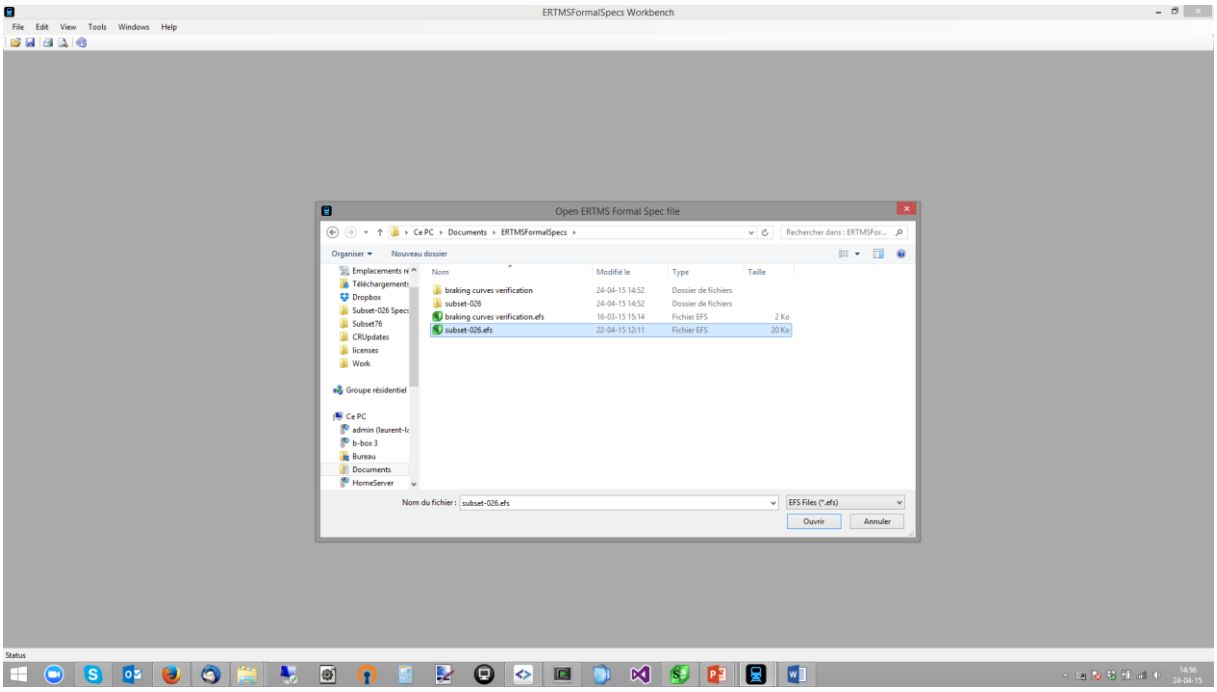

Figure 4: EFS file selection.

When an EFS file is loaded, specification browsers (see Chapter 4), data dictionary browsers (see Chapter *5*) and the test browser (see *Chapter 6*) are opened automatically, as displayed in the following figures.

The data dictionary browser is displayed by default (as Figure 5 depicts)

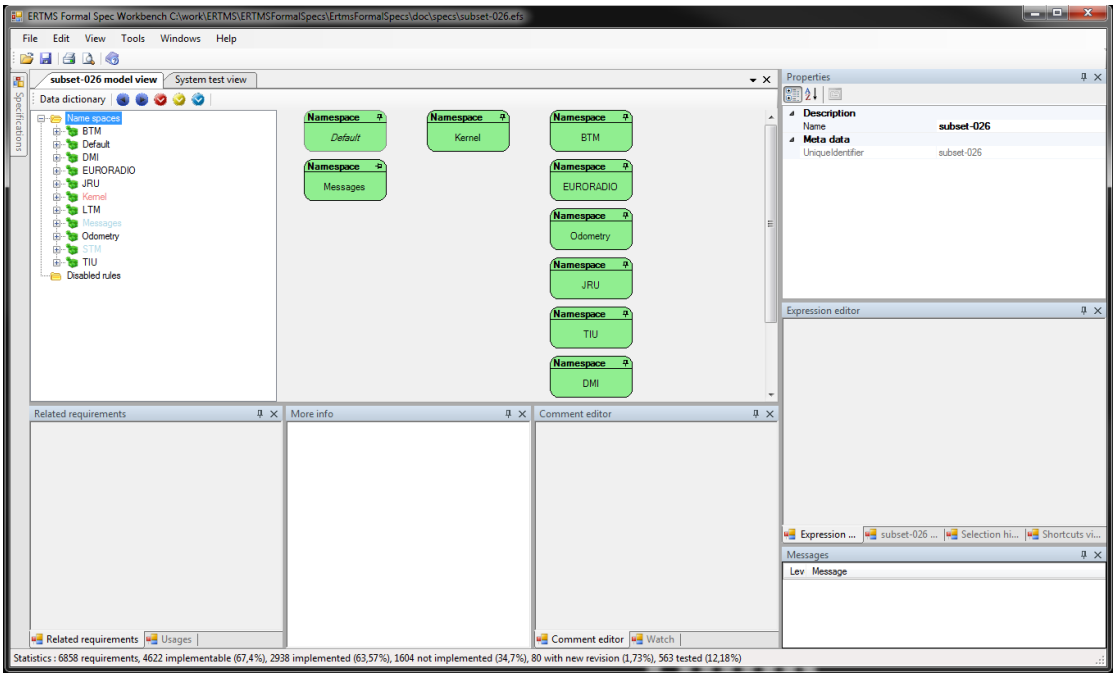

Figure 5: Data dictionary browser

The test browser is the second view displayed in the main part of EFSW (Figure 6):

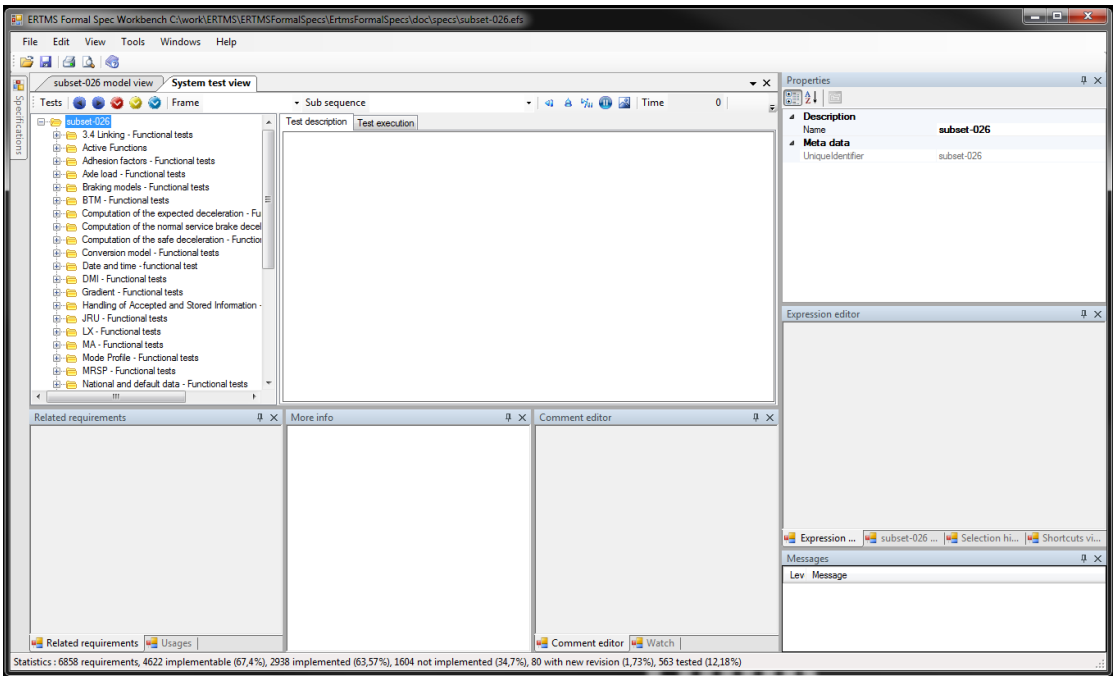

Figure 6: Test browser and execution environment

The Specification browser can be seen by expanding the left part of the EFWS (Figure 7):

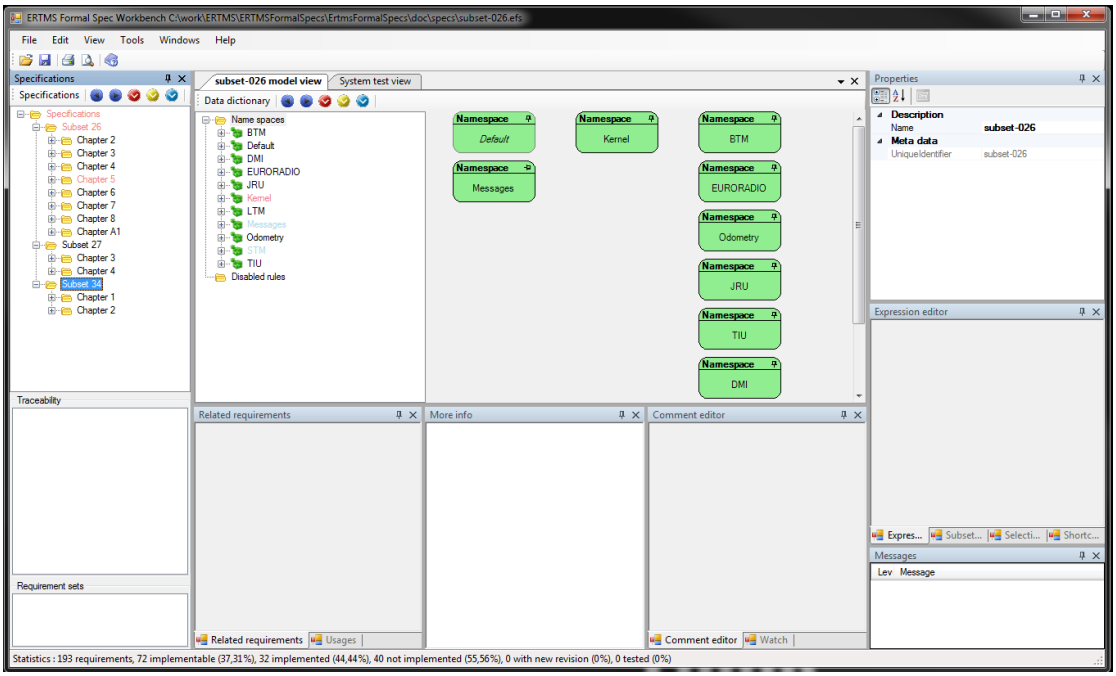

Figure 7: Subset-026 Specification browser

Several files can be opened at the same time. The resulting modelled EFS system is the combination of all those files. This allows, for instance, to add project specific behaviour to a given system, or to dissociate specific tests from the model. For instance, braking tests with specific train data are located in a separated (test) model, named “braking curves verification.efs”.

3.5 Messages on the ERTMSFormalSpecs Workbench:

3.5.1 Messages window

EFSW provides information about the status of the selected item in the Message window (lower right part of the Workbench main window). Figure 8 presents a typical example of the message window’s contents.

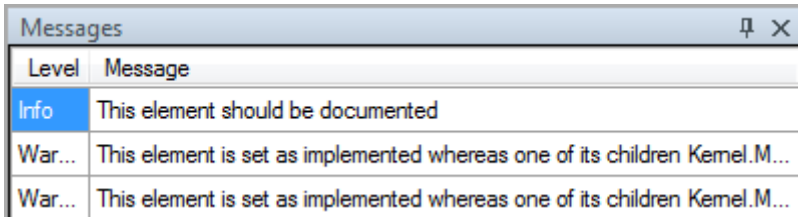

Figure 8: Message window showing messages related with one of the requirements.

Messages are composed of a Level and the informative message. The Level column is related with the type of the message:

- **Info:** additional information, such as a search result or an indication related to a requirement.

- **Warning:** indication related to an element which may become a problem and provoke an Error.
- **Error:** indication related with an element which will cause the system not to work.

3.5.2 Messages colours:

EFSW uses different colours to identify the elements which have generated a Message, Figure 9 depicts a situation where several messages are highlighted on the data dictionary browser:

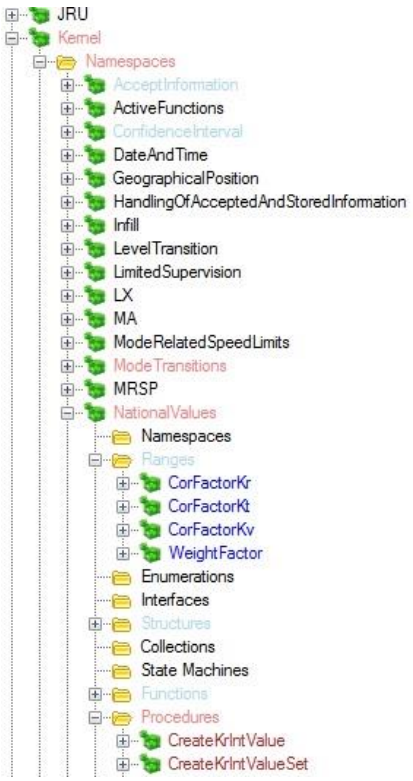

Figure 9: Representation of different colours used on EFSW.

Table 2 presents the colours used to denote the presence of a message. They are ordered in increasing precedence order, meaning that if there is a choice to display a colour, the system will select the colour the furthest in the table. For instance, if Light blue and Orange can be selected, the system will colour the node in Orange.

Colour	Level	Meaning
Light blue	Info	There is at least one Info message on the hierarchical tree under this element.
Dark blue	Info	The current element is directly associated with an Info message.
Rose	Warning	There is at least one Warning message on the hierarchical tree under this element.
Brown	Warning	The current element is directly associated with a Warning message.
Orange	Error	There is at least one Error message on the hierarchical tree under this element.
Red	Error	The current element is directly associated with an Error message.

Table 2: Messages and colours representation

Figure 10 depicts a model element containing a single message of type Info.

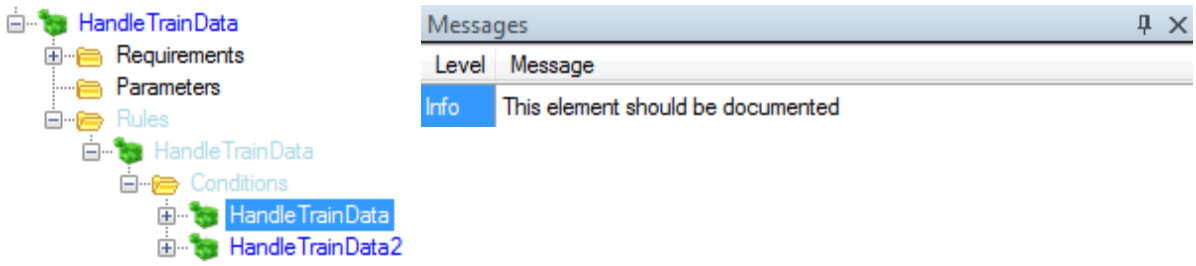

Figure 10: Representation of the information messages on EFSW

Figure 11 depicts how the warning messages are represented in the EFS data dictionary browser and the associated explanation in the messages view window. The warning message indication in the data dictionary tree is done following the colour representation of Table 2.

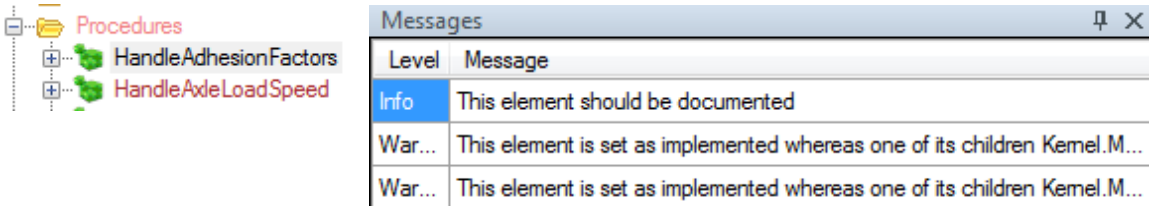

Figure 11: Representation of the warning messages on EFSW.

Figure 12 depicts the error messages representation in the data dictionary hierarchical tree and the linked messages to the highlighted model item. Table 2 describes the colours used for the model nodes and elements.

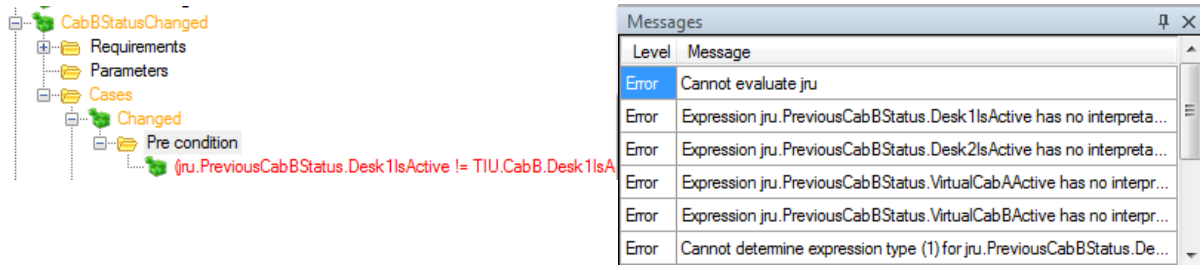

Figure 12: Representation of the error messages on EFSW.

3.5.3 Messages navigation buttons

The buttons, and can be used to navigate to respectively the next error, warning or information message, according to the selected node in the tree.

3.6 ERTMSFormalSpecs Workbench properties

All the elements in EFSW have several metadata associated to them – properties – which are displayed in the Property view, on the upper right side of EFSW main window, Figure 13 illustrates the location of the property window.

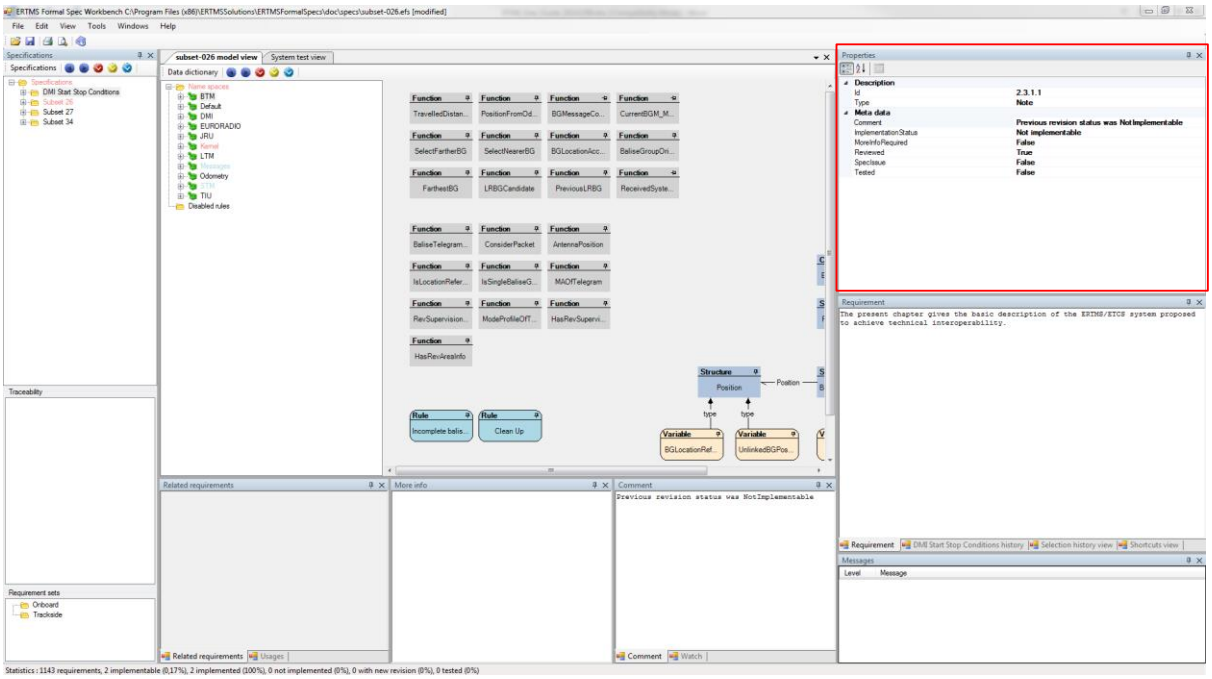

Figure 13: Properties location on EFSW main window.

Figure 14 presents a typical content of the property window for a specification paragraph. Specific properties will be explained when describing each component of the EFS model.

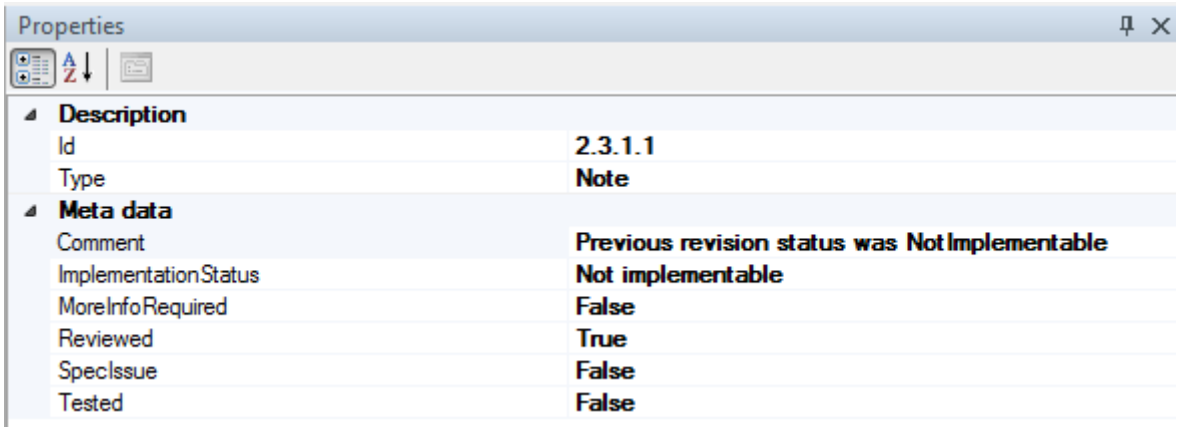

Figure 14: Detailed view of the properties window of an EFSW element.

Most of the elements in EFSW contain some of the following properties.

- **Comment**: additional information giving an explanation of the current selected element of the hierarchical tree.
- **Name**: the given name of the current selected item of the hierarchical tree.

- **Unique identifier**: represents the unique name given to the current selected element of the hierarchical tree on the namespace where it belongs.
- **Needs requirement**: indicates that this part of the model requires a requirement to be linked to it.
- **Implemented**: indicates the status of implementation of the current node of the hierarchical tree. When this flag is set to true means that it has been fully implemented.
- **Verified**: gives the information related to the revision status of the current node of the hierarchical browser tree. When set to true, means that the current selected element has been reviewed.

4 Specification browser

The specification browser is used to display the requirements to be modelled. In the case of the trainborne system, it contains requirements from several Subsets, and also optional documents.

4.1 Opening the specification browser

The specification browser is automatically opened when launching EFSW and it is reduced on the upper left corner on EFSW main window.

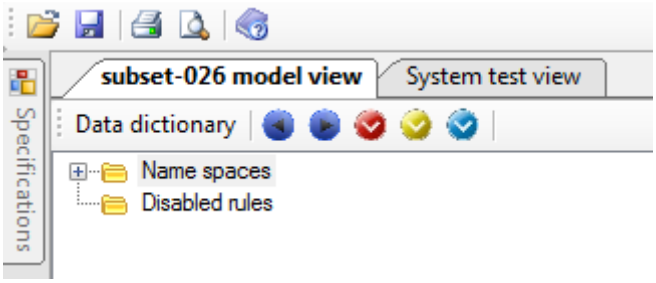

Figure 15: Location of the Specification browser when launching

In case the model window tab is closed clicking on the close button located on the top right corner of the specifications tab, it can also be re-opened using the available option on View\Show specification view menu. Selecting this option with a specification browser opens an additional specification browser in EFSW. Figure 16 depicts how to re-open the specification window.

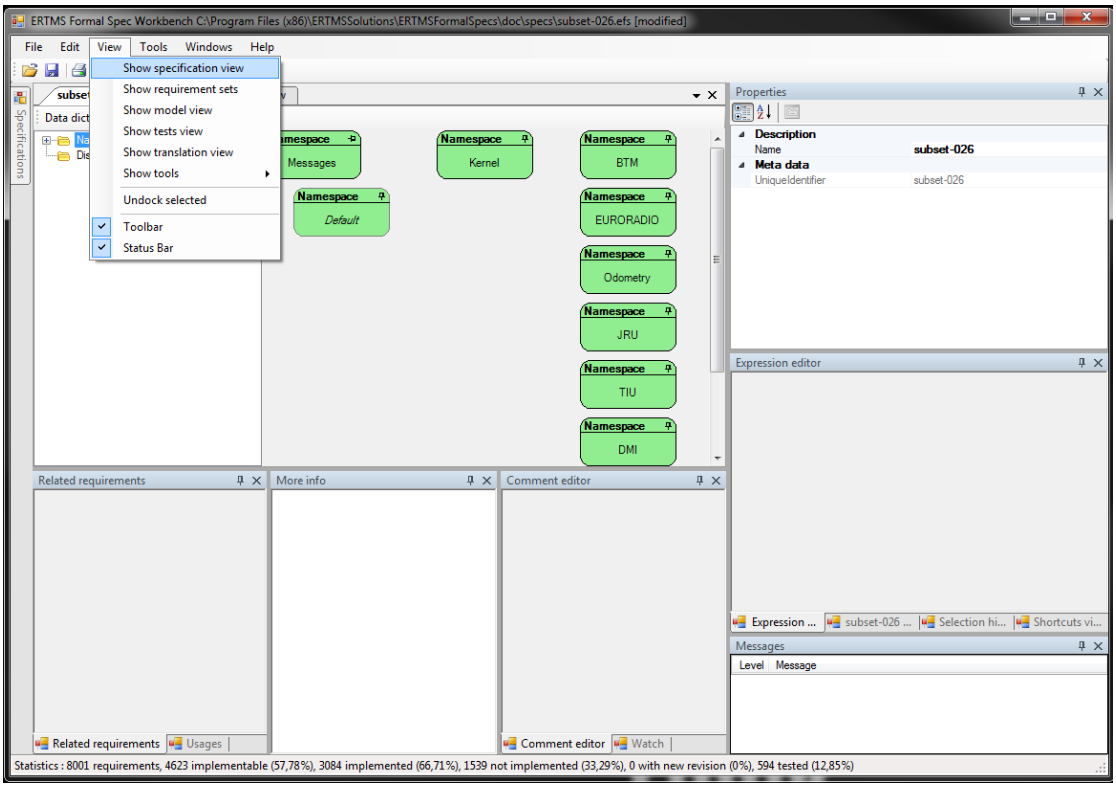

Figure 16: Opening the specifications browser clicking the view contextual menu.

Figure 17 displays the general overview of EFSW when the specification browser is pinned up as part of the main window. The specification browser is located on the left side of EFSW main window.

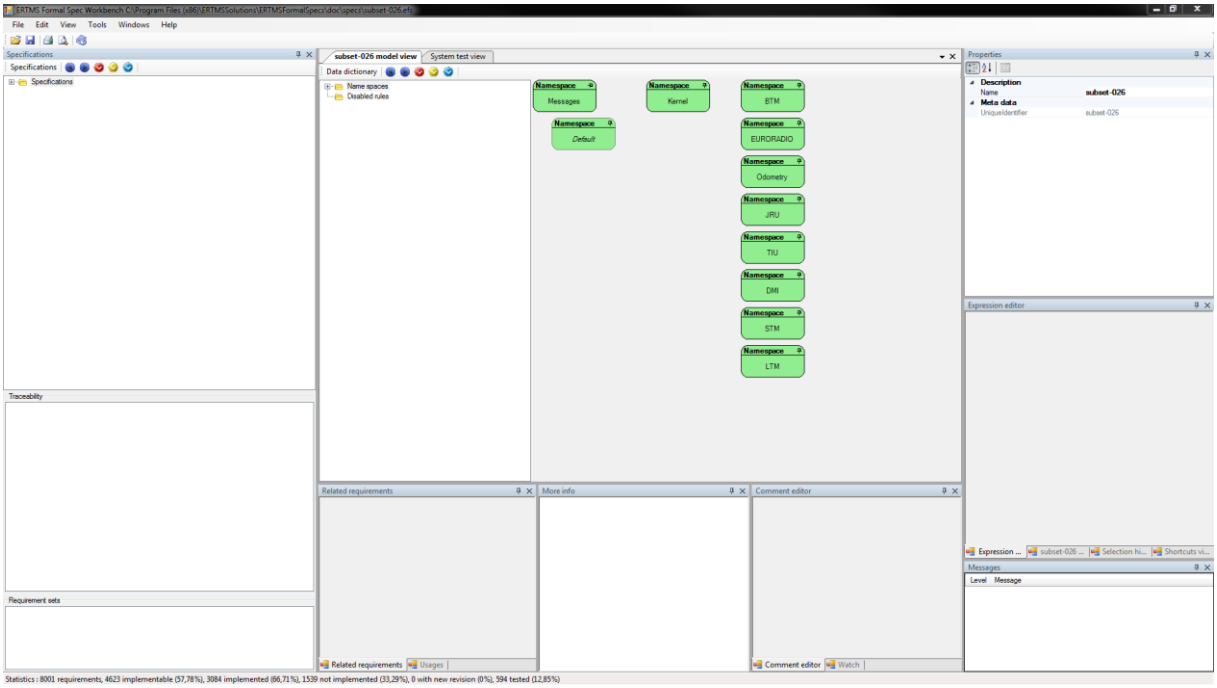

Figure 17: General overview of the specification browser.

4.2 Specification browser description

4.2.1 Overview

The specification browser is decomposed into several parts, as Figure 17 indicates.

- **The specification view**: on the left side of the main window, contains the hierarchical tree specification browser, traceability information and the requirement sets tabs.
- **The hierarchical tree view**: on the upper left side, displays the input specifications in hierarchical order. It faithfully mirrors the content of the selected Subset as depicted by Figure 18.

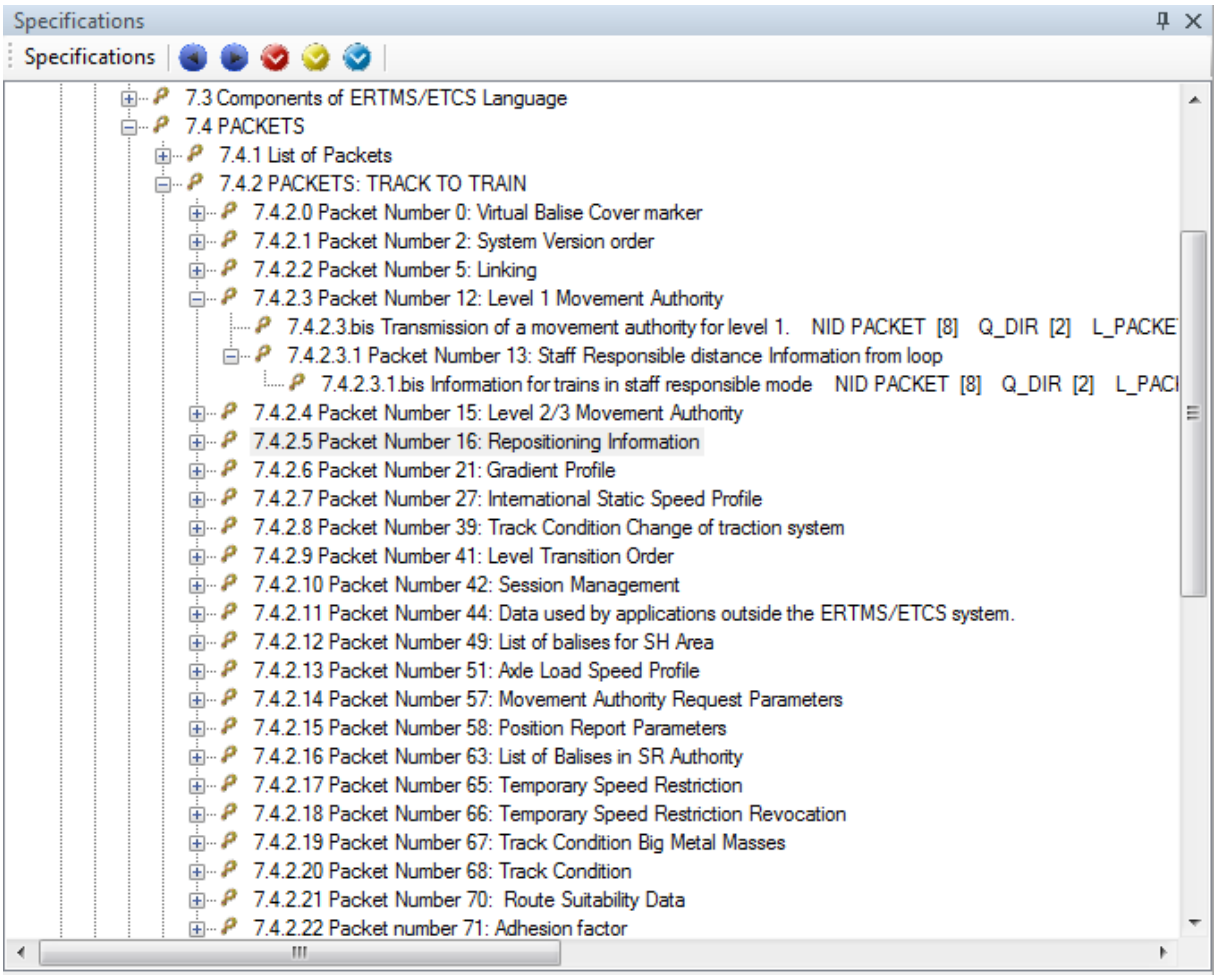

Figure 18: Selected Subset Specification view.

- **The middle left part of the specification browser window**: displays the traceability. See Section 4.4 for further details about the traceability.

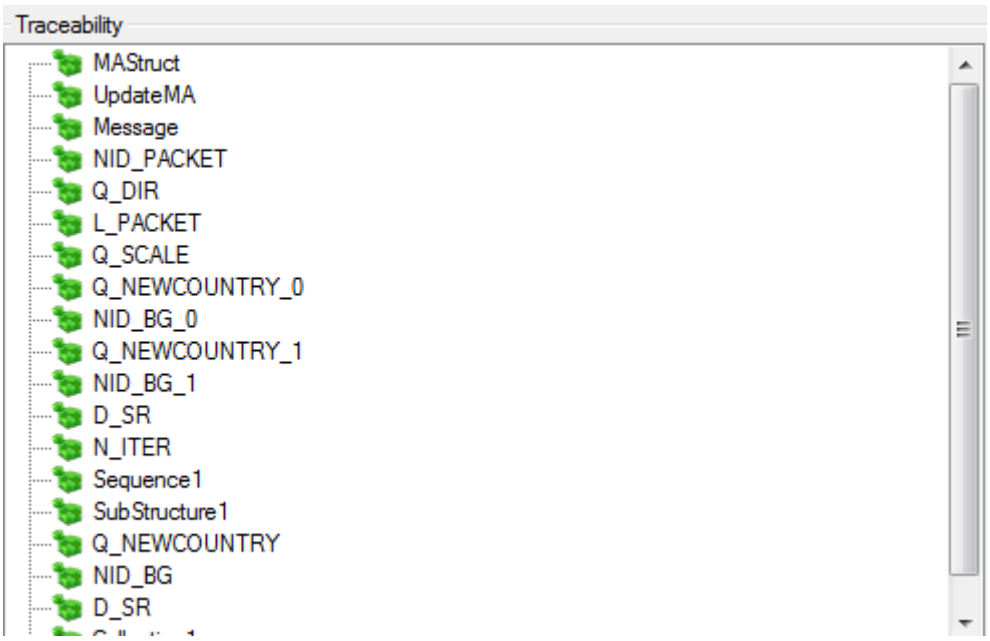

Figure 19: Traceability section on the specification browser.

- **The requirement sets**: in the lower left part, indicates to which set of requirement the current selected requirement belongs. In this case, the requirement belongs to two requirement sets: Onboard & Trackside.

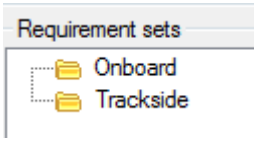

Figure 20: Requirement sets section on the specification browser.

For further details about the requirement sets, see Section 4.2.2.

- **The right side of the main window**: contains the properties, messages and different tabs containing the requirement information, historical information, selection history and shortcuts view.
- **Properties**: located on the upper right corner. See Section 4.2.2.
- **Middle right side**: contains several different tabs.
- **Requirement**: allows editing the requirement text of the selected requirement.

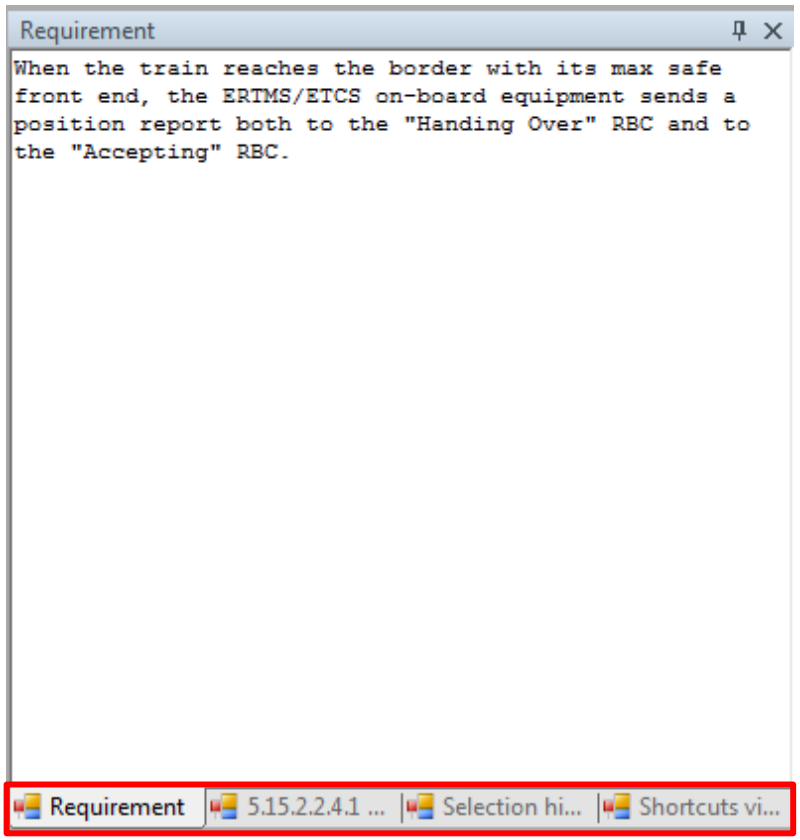

Figure 21: Detailed view of the middle right side of the main window.

- **Messages**: located on the lower right corner. For further details see Section 3.5.

4.2.2 Requirements classification

Requirements are classified according to their role in the document. Some parts of the ERA subsets are divided into sub-requirements:

- In tables, each line of the table is an individual sub-requirement. The identifier of these sub-requirements is composed of the identifier of the table and an extra number for the row. (For instance Figure 23 shows a requirement coming from [2], Chapter 4, paragraph 4.7.2 “DMI versus Mode table”) and adding before the requirement number the keyword.
- Table for the table header.
- Entry for each table line.
- When a requirement is further split into several requirements, a number is added to the end of the requirement number.

Figure 22 and Figure 23 depict the strategy of classification used on EFSW:

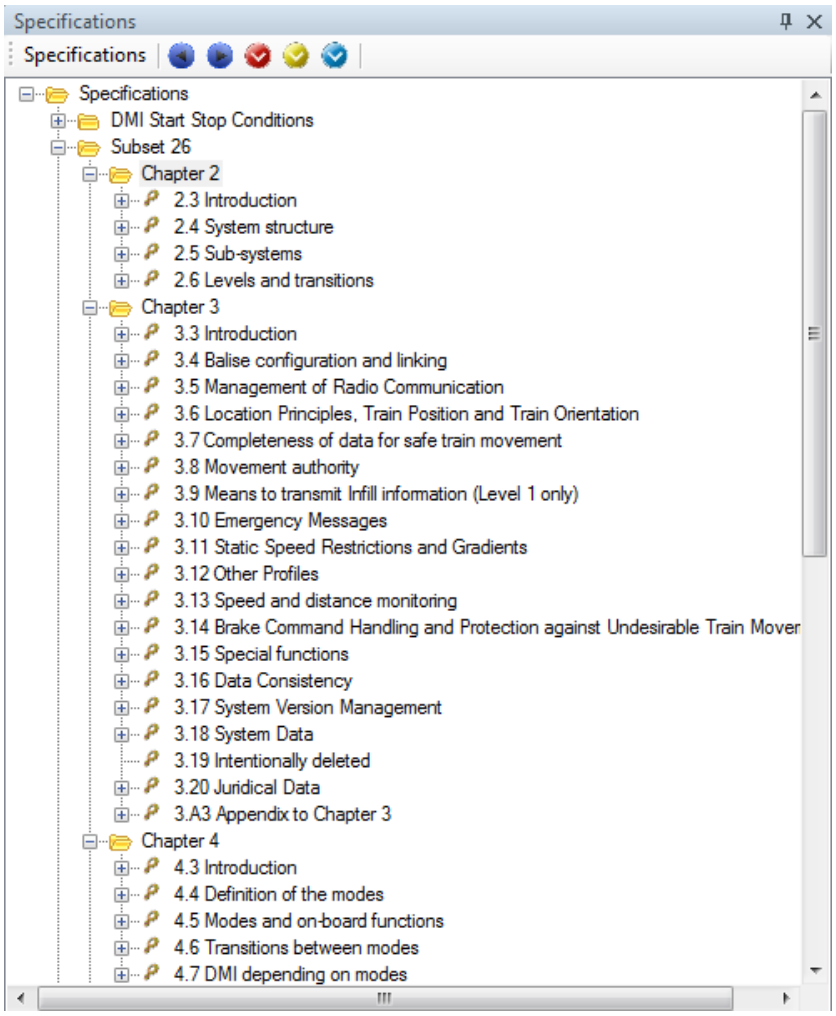

Figure 22: Overview of the given names for tabluar requirements

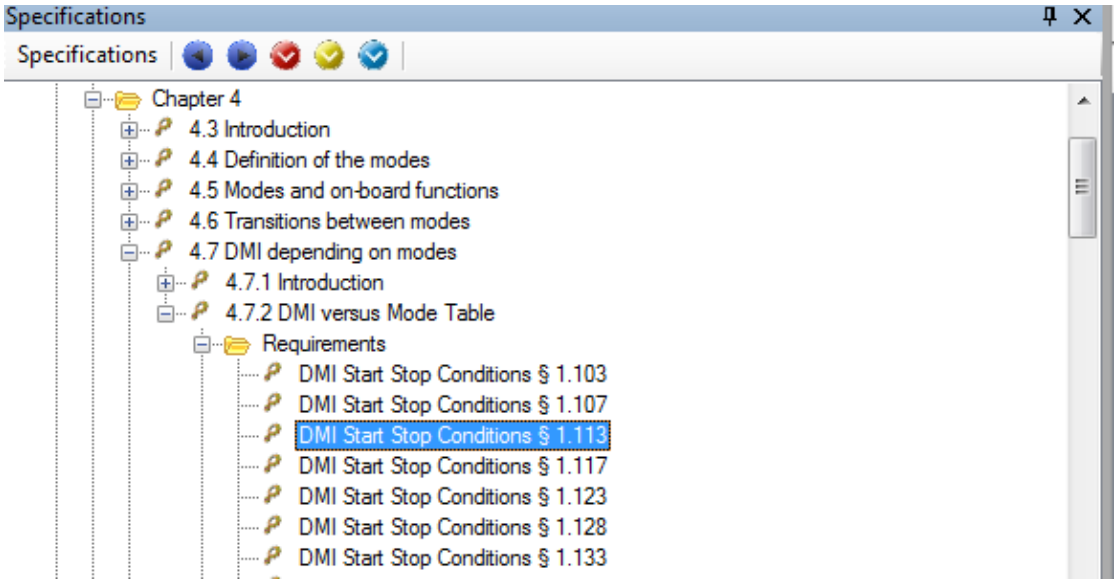

Figure 23: Handling of tabular requirements in specifications browser

4.2.3 Specifications properties

The specification properties summarize the selected requirement's incorporation into the model and test database, and track problems discovered in the specification.

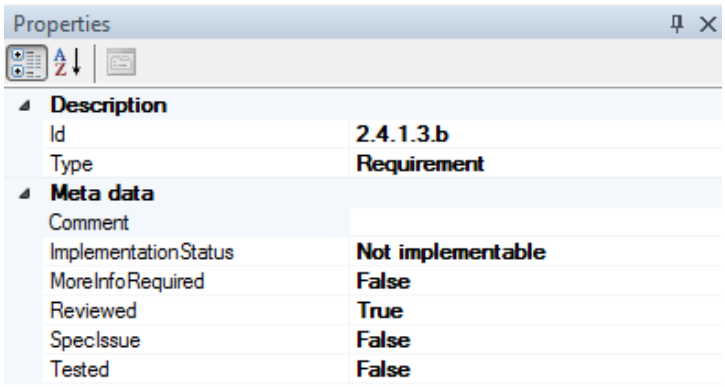

Figure 24: Specifications properties view

The properties of specification elements are:

- **Id**: the unique identifier of the requirement in the document where it was defined.
- **Type**: the purpose of the selected requirements for the modelling purposes. The possible types are:
  - **Title**: the names of the different sub-sections which compose a chapter.
  - **Note**: comments present on the specifications.
  - **Definition**: the definition of a concept.
  - **Requirement**: a requirement of the specifications which need be modelled.
  - **Table Header**: similar to title, tables are split with each line becoming a new sub-requirement (of type Requirement).
- **Implementation status**: the current status of the model related to this requirement.
  - **Implemented**: the requirement has been completely modelled. There should be traceability information between this requirement and the related model elements.
  - **Not implemented**: an implementation is required for this requirement, but it has not (yet) been created.
  - **Not implementable**: the requirement cannot be modelled for various reasons, for instance because it is a requirement which constraints the environment in which the model is meant to exist.
  - **New version available**: the requirement has been modelled for a previous version, but it has changed with a newer version and the model has not been reviewed to take that change into account.
- **More info required**: the requirement is not precise enough and should provide more information. Usually, there is a comment attached to requirement with "More info required" to indicate the kind of information which is required.
- **Reviewed**: the content of the requirement has been reviewed and matches the source text (Word document, paper …).
- **Spec issue**: there is an issue related with the requirement. Usually, a comment is provided to explain the issue encountered when modelling this requirement.
- **Tested**: tests have been created to ensure that the model correctly implements the requirement's expectations. When a requirement is set to tested, it should hold traceability information to the corresponding tests.

4.2.3.1 Properties of Requirement Document

The properties of a Requirement Document give the information to identify it in a unique way. Figure 25 displays the properties of a Subset or optional document.

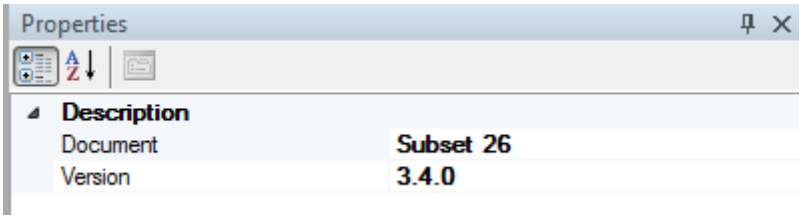

Figure 25: Subset or Optional Document properties

- **Document**: represents the name of the selected Requirement Document.
- **Version**: indicates the version of the selected Subset or Optional Document.

4.2.3.2 Properties of a Chapter on a Requirement Document

The properties of a Chapter are limited to the chapter number in the Requirement Document. Figure 26 depicts the properties of a chapter.

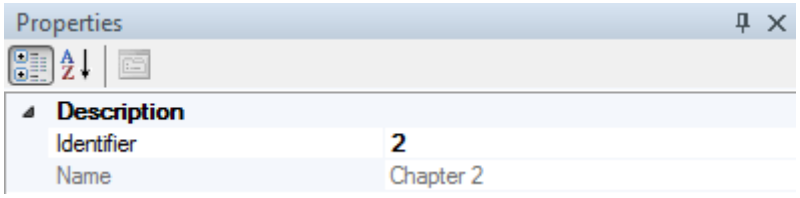

Figure 26: Chapter properties.

- **Identifier**: unique identifier of the Chapter in the document to which it belongs.

4.2.3.3 Properties of a Paragraph

The properties of a Paragraph are all the Specification properties described above. Figure 27 illustrates these properties.

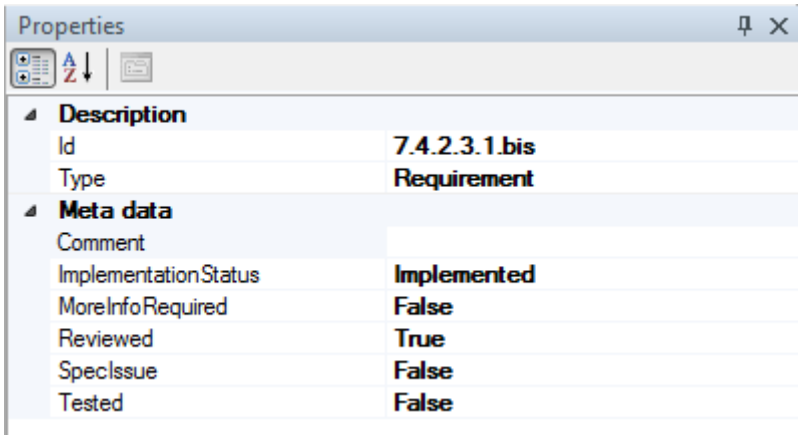

Figure 27: Properties contents

4.3 Requirement sets

Requirements can be classified in **requirement sets**, which allow management of several requirements as a group. For instance, for the onboard unit, two main requirement sets are defined

- **Scope**, which identifies the part of the system the requirement is related to. The scope holds the following requirement sets
  - **Trackside**: requirements related to the trackside
  - **Onboard**: requirements related to the on board unit
  - **Rolling stock**: requirements related to the rolling stock.
- **Functional block**, which splits the system into a set of functions
  - Levels and transitions.
  - Recording of juridical data.
  - Management of radio communication
  - Acceptance of received information.
  - Speed and distance monitoring commands.
  - Train interface
  - …

A requirement can be classified on one or more, categories. For example a requirement can be linked to the **Onboard** requirements and linked to the **Train interface** requirement set.

4.3.1 Managing the requirement sets

The **Requirement sets** can be seen by: View -> Show requirement sets. Figure 28 depicts how to display the **Requirement sets**.

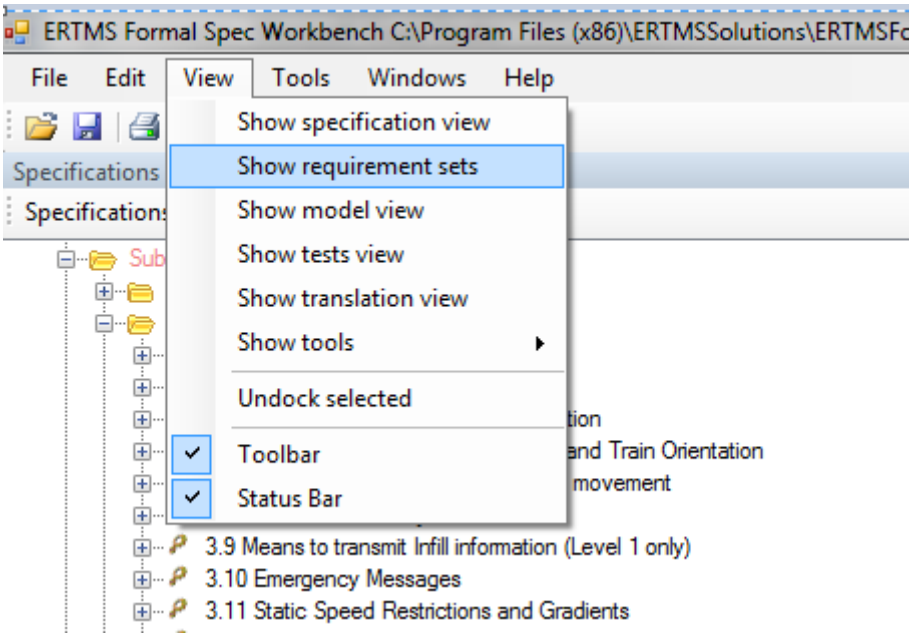

Figure 28: Selecting the requirements sets.

After clicking on the highlighted option, EFSW displays the Requirement sets window for the selected Subset. Figure 29 illustrates the result of clicking on this option.

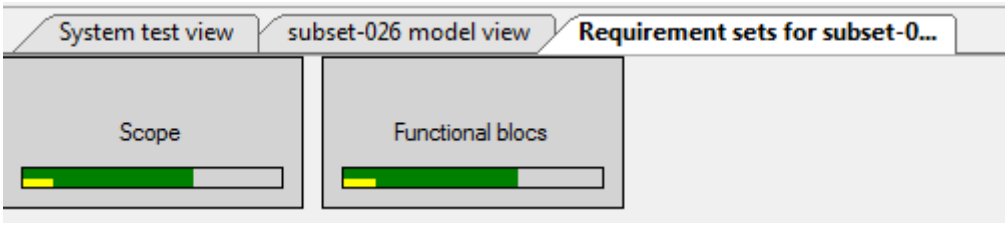

Figure 29: Requirement sets window for the selected Subset.

Figure 29 depicts the two highest levels of the requirement sets, and on their representation are displayed two progress bars.

- **Green progress bar:** the percentage of each requirement set that has been modelled.
- **Yellow progress bar:** the percentage of each requirement set that has been tested.

Double clicking on the **Scope** Requirement set opens a new window containing the first series of requirement sets. See Figure 30.

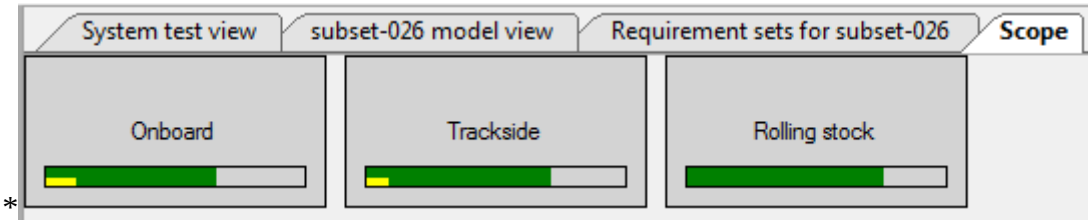

Figure 30: Representation of the possible scopes.

Right-clicking on a **Scope** and selecting the “Select paragraphs” option marks the paragraphs related to the selected requirement set with an info message. See Figure 31.

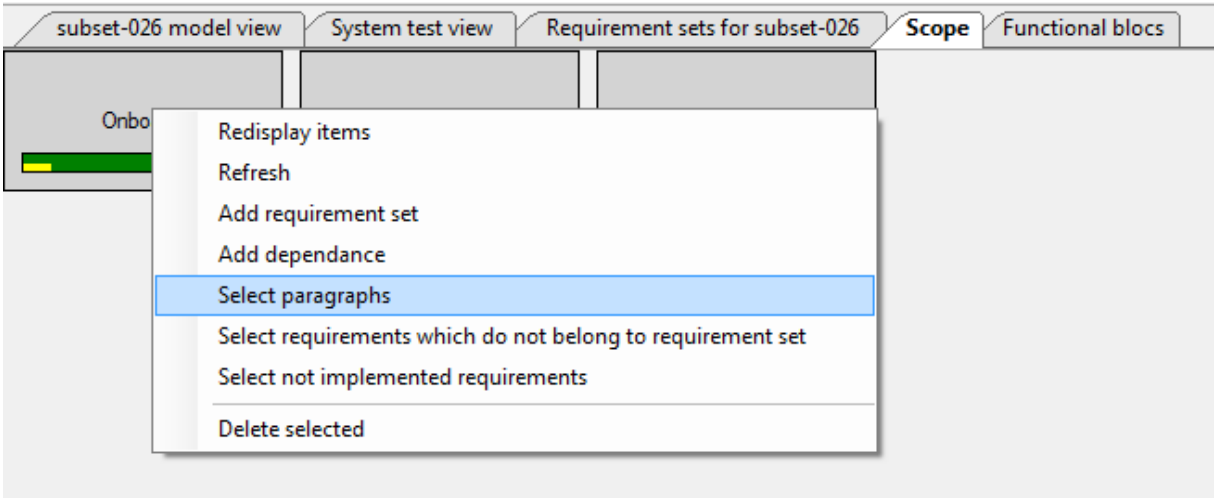

Figure 31: Selecting the paragraphs contained in a requirement set.

The paragraphs contained in the selected Scope are displayed in dark blue on the Specification browser.

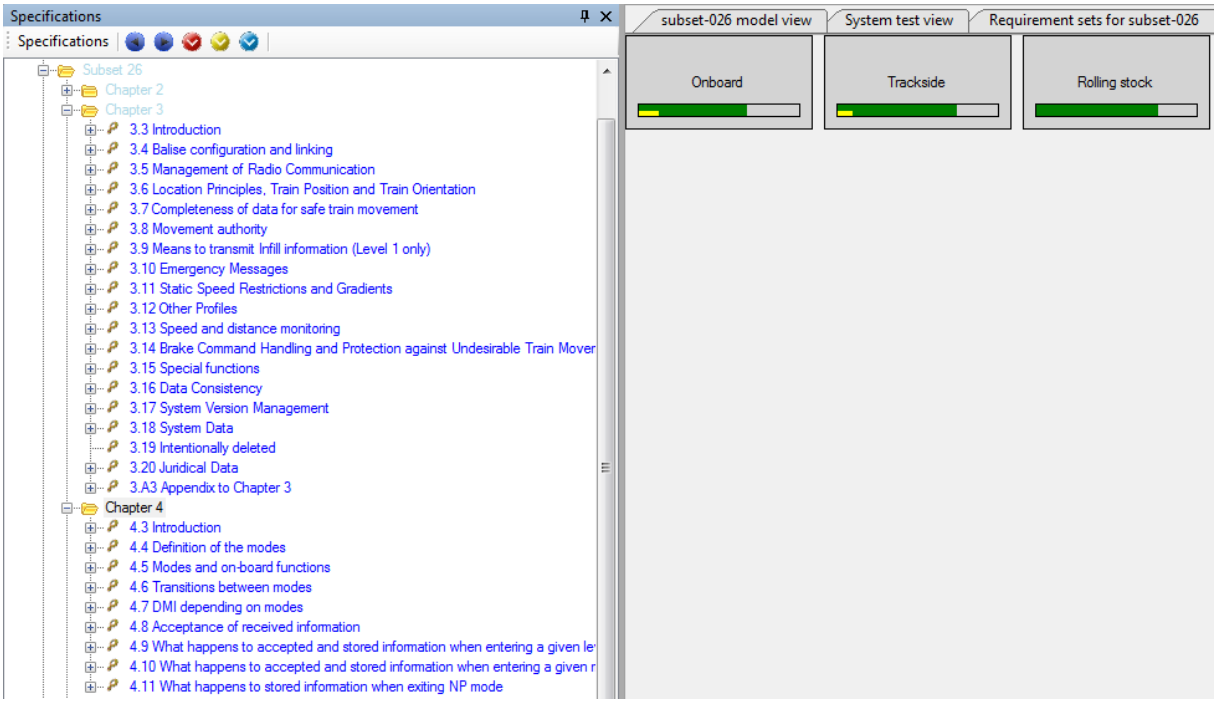

Figure 32: Paragraphs related to the requirement sets

A requirement can be linked to a **Scope** by drag & dropping the requirement on the desired **Scope**. EFSW verifies automatically if the selected requirement is already present in the **Scope** and in that case, it is not added.

Double clicking on a **requirement set** shows the sub requirement sets. For instance, double clicking on the **Functional block** requirement set shows all the **requirements sets** that are of the type "**Functional block**". Figure 33 shows the **Functional blocks** that have been defined for the onboard model.

Figure 33: Requirement sets nesting.

4.3.2 Requirement sets properties:

Figure 29 displays the two highest levels of the categories used to classify the **requirement sets**. When clicking on one of them, **scope** or **functional blocks**, a corresponding properties window is displayed, see Figure 34 .

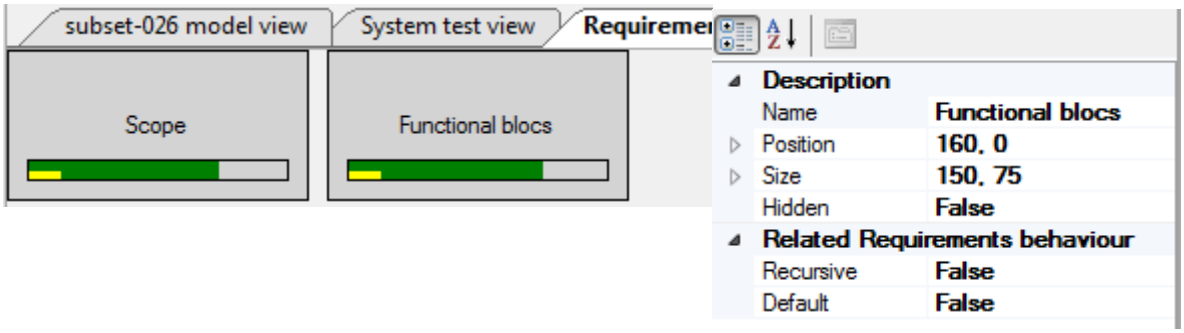

Figure 34: Requirement sets properties window

Figure 35 shows a detailed view of the requirement sets window.

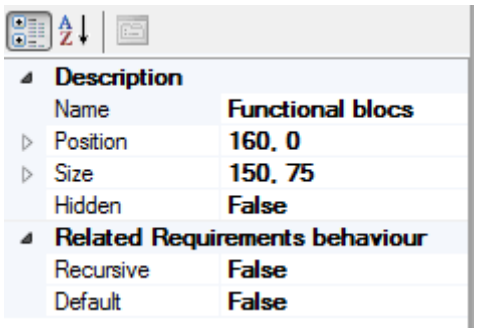

Figure 35: Detailed view of the requirement sets properties window

The properties of a requirement set are the following

- **Recursive:** when this flag is set to true a requirement added to this requirement set also (implicitly) adds all its sub-requirements to the requirement set.
- **Default:** new requirements are automatically added to all requirement sets where the Default flag is set to true.

4.4 Requirements Traceability:

The **Traceability** tool is located on the middle part of the Specifications tab, as Figure 36 displays.

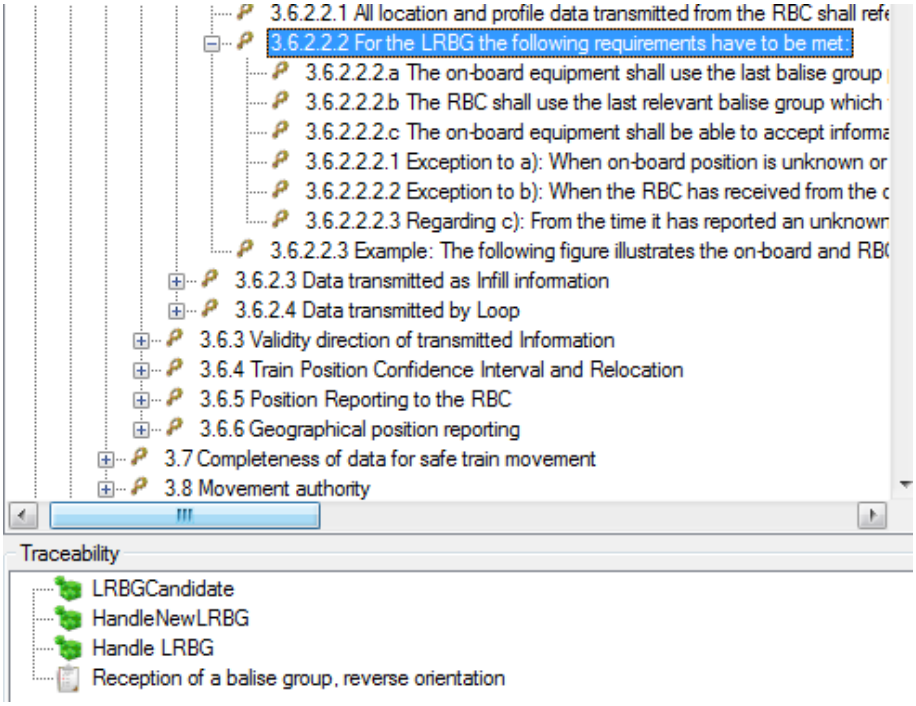

Figure 36: Location of the Traceability window on the Specifications tab.

The **Traceability** is used to link requirements to model elements or to tests. Figure 36 shows that requirement 3.6.2.2.2 is related to three model elements and one test.

Linking with model element is described in Chapter 5, whereas traceability to tests is described in Chapter 6.

4.5 Tools related to the specification view

Figure 37 shows the actions which can be executed on the specification.

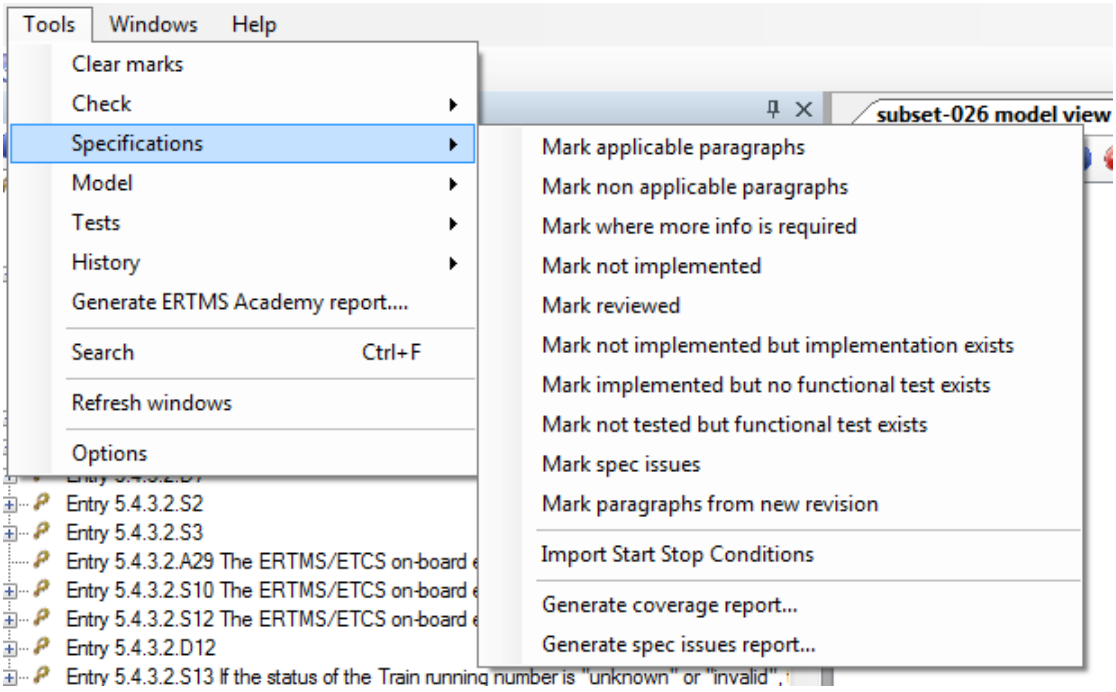

Figure 37: Actions related to the Specification tree.

4.5.1 Search for applicable and non-applicable paragraphs

Not all the requirements require implementation in the model. EFSW can be used to find the requirements which require an implementation using the Mark applicable action, which marks them with an info message.

In the same way, the Mark non applicable action allows to mark paragraphs which do not require a model implementation.

4.5.2 Search for paragraphs where more information is required

The paragraphs requiring more information to be implemented can be found using the Mark where more info is required action. This marks all requirements for which the **More Info Required** property has been set to true.

4.5.3 Search for not implemented requirements

Requirements for which no implementation has been provided can be found using the Mark not implemented action. This marks all requirements whose implementation status is Not Implemented. The implementation of a requirement is complete when

- **The Implementation status** is Implemented.
- **All model elements** associated with this requirement are also marked as Implemented.

4.5.4 Search for requirements that have not been verified

Requirements that have not been submitted to the review process can be found using the Mark not reviewed action. This marks all requirements which have not been marked as reviewed. This ensures that the requirements, as encoded in the tool, correspond to the requirement from the original source (paper, word document…).

4.5.5 Search for requirements have not been implemented but implementation exists

The Mark not implemented but implementation exists action allows to show all requirements for which an implementation is available (they are linked to at least a model element), but the status is still not implemented.

4.5.6 Search for implemented requirements without functional tests

The requirements that are implemented but not yet covered by a functional test can be found using the Mark implemented but no functional test action. This marks all requirements that are implemented but are not yet covered by functional tests.

4.5.7 Search for requirements which are incomplete or erroneous

If a requirement contains an error, or is incomplete, it is marked as spec issue. The Mark spec issues action highlights all the spec issues that have already been detected.

4.5.8 Search for requirements from a new revision

If a paragraph was changed in a version update to its Specification, but its implementation has not been revised, it can be found with the Mark paragraph from new revision action.

4.6 Processes related to requirements

Among the possible actions which can be performed over a requirement, the following are particularly important to the modelling process: Review the requirements and Update the status of a requirement after implementation.

4.6.1 Review the requirements

Requirements must be reviewed for several reasons

- **The conversion process** between the selected Specification file and XML file is manual, hence error prone.
- The **type** and **scope** of each requirement must be set manually.
- **Requirements** expressed in the selected Specification file are not directly ready for modelling: some requirements may be too small and must be merged with their neighbours; some others are too big and should be split to improve traceability.

As soon as requirements have been reviewed, the reviewer changes the Reviewed status of the requirement as **True**. Figure 38 depicts the available values for the **Reviewed** property.

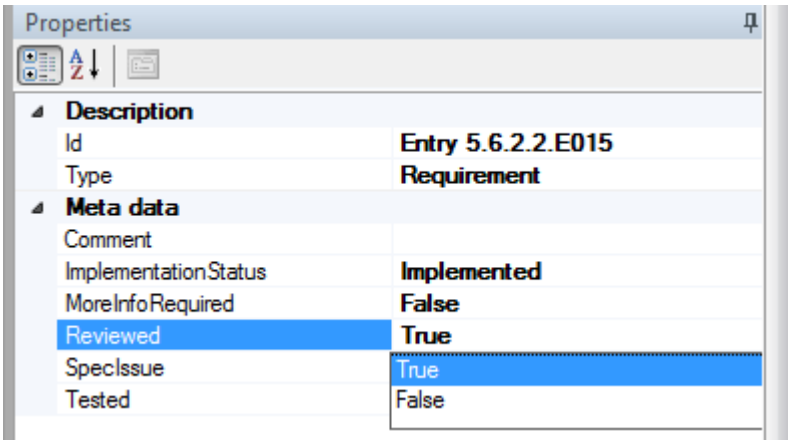

Figure 38: Requirement has been reviewed

The action presented in Section 4.5.4 allows detecting the requirements that still require reviewing.

4.6.2 Create the model for the requirement

Once the requirement has been modelled, the engineer changes the Implementation status of the requirement to **Implemented**. The action presented in Section 4.5.3 allows detecting the requirements that still require some modelling.

4.6.3 Update the model of an Implemented requirement

As soon as a model related to a requirement changes, the implementation status of the requirement must be manually changed from **Implemented** to the proper current status**.** Additionally, when the text of a requirement changes, the status is automatically set to new **revision available**

5 ERTMSFormalSpecs Model

The EFSM is used to model the specifications described on Chapter 4. The model contains several items:

- **Types:** structure the model**.**
- **Variables:** provide the system's state**.**
- **Functions:** factorize common computations on the system.
- **Procedures:** allow sequenced operations. Otherwise, the model described in EFS is purely functional.
- **Rules:** provide the system dynamics by checking a set of preconditions before applying changes on the system's state.
- **State machines:** describe the system's dynamic behaviour according to the inputs.

Figure 39 shows a typical view of EFSW when opening a model.

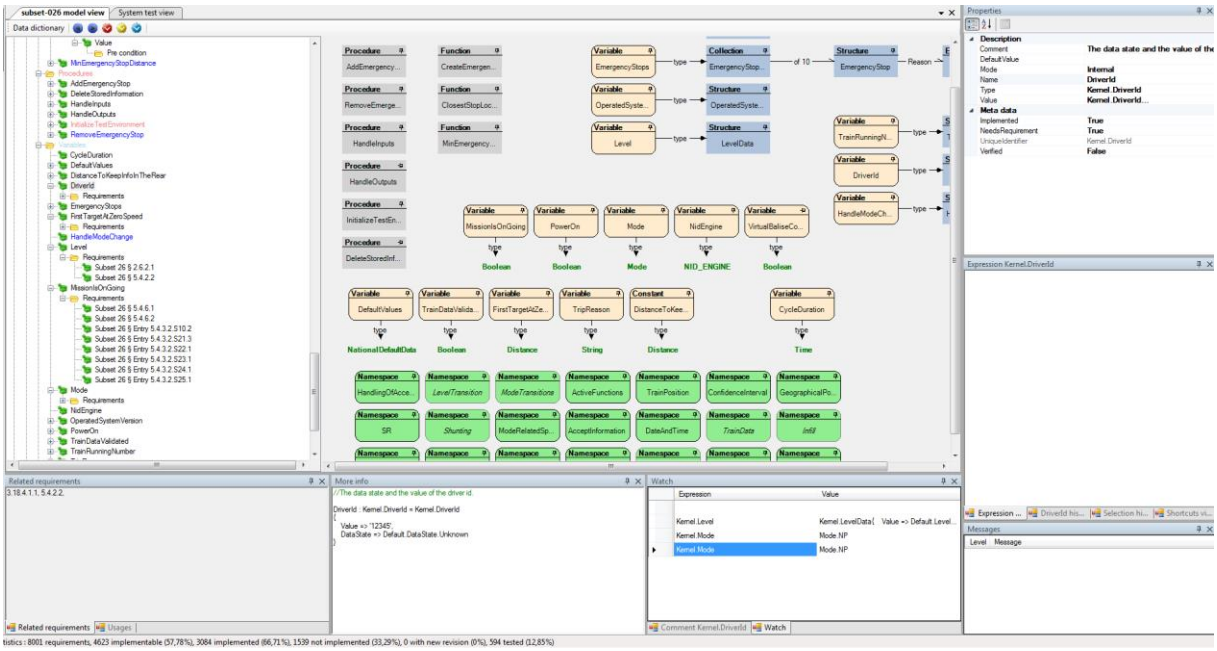

Figure 39: Data dictionary main window view.

5.1 Opening the data dictionary browser

The data dictionary main window is automatically opened, but also could be closed clicking on the close button on EFSW main window view. Figure 40 illustrates the location of the close button on EFSW main window.

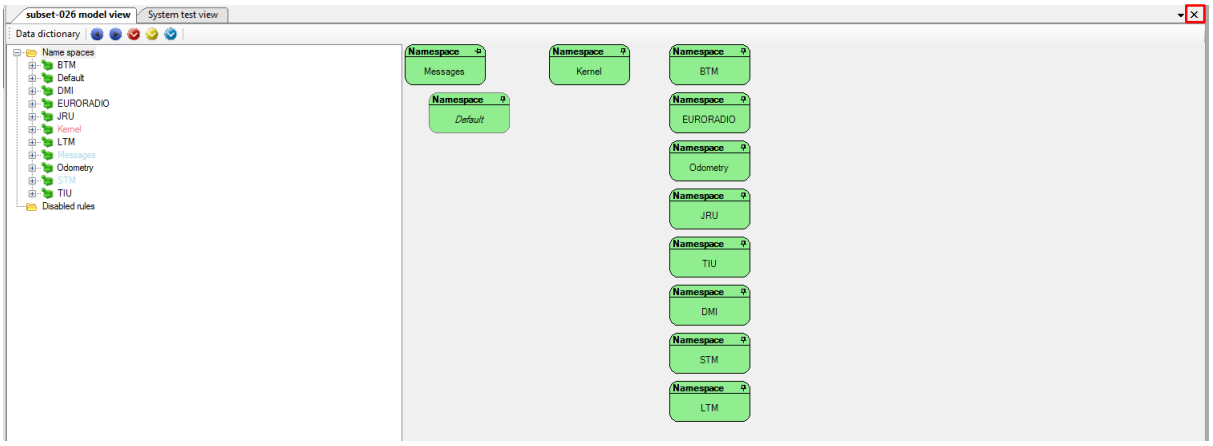

Figure 40: Location of the close button on the model data dictionary window

The same window can be re-opened using the View\ Show model view contextual menu. Figure 41 depicts how to re-open the model view window.

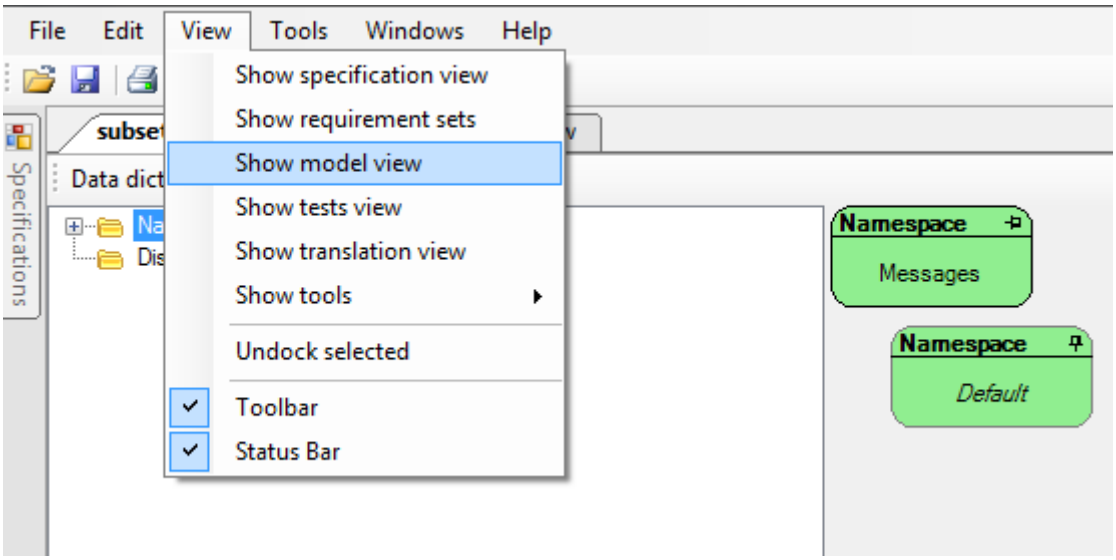

Figure 41: Re-opening the model view window

If several EFS files have been opened, a dialog box is provided to select the EFS file on which the data dictionary browser should be opened as Figure 42 displays.

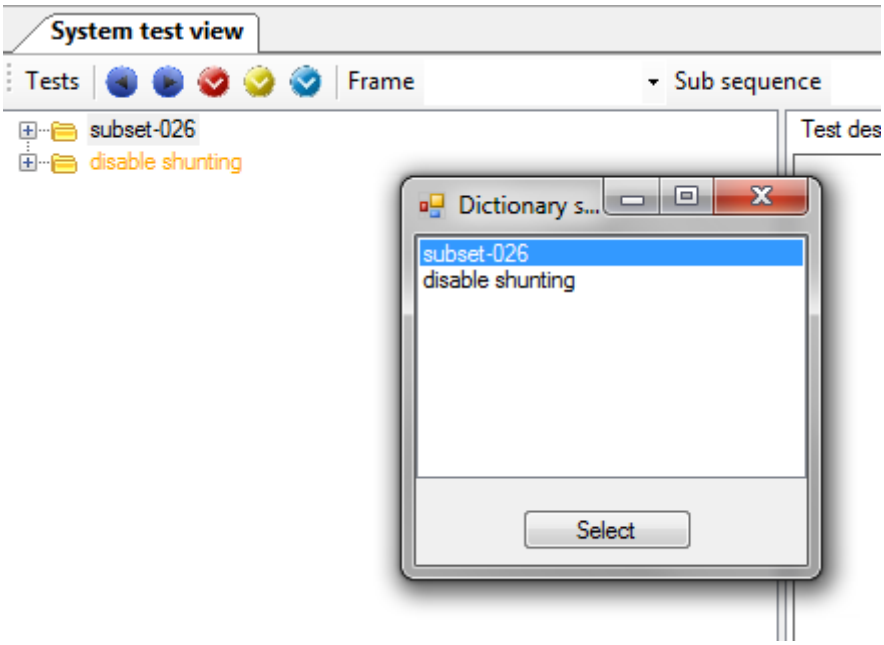

Figure 42: Dialog box for selecting the model to open.

5.2 Data dictionary browser description

5.2.1 Overview

The data dictionary browser is decomposed into several parts highlighted on red. Figure 43 displays the parts of the data dictionary browser.

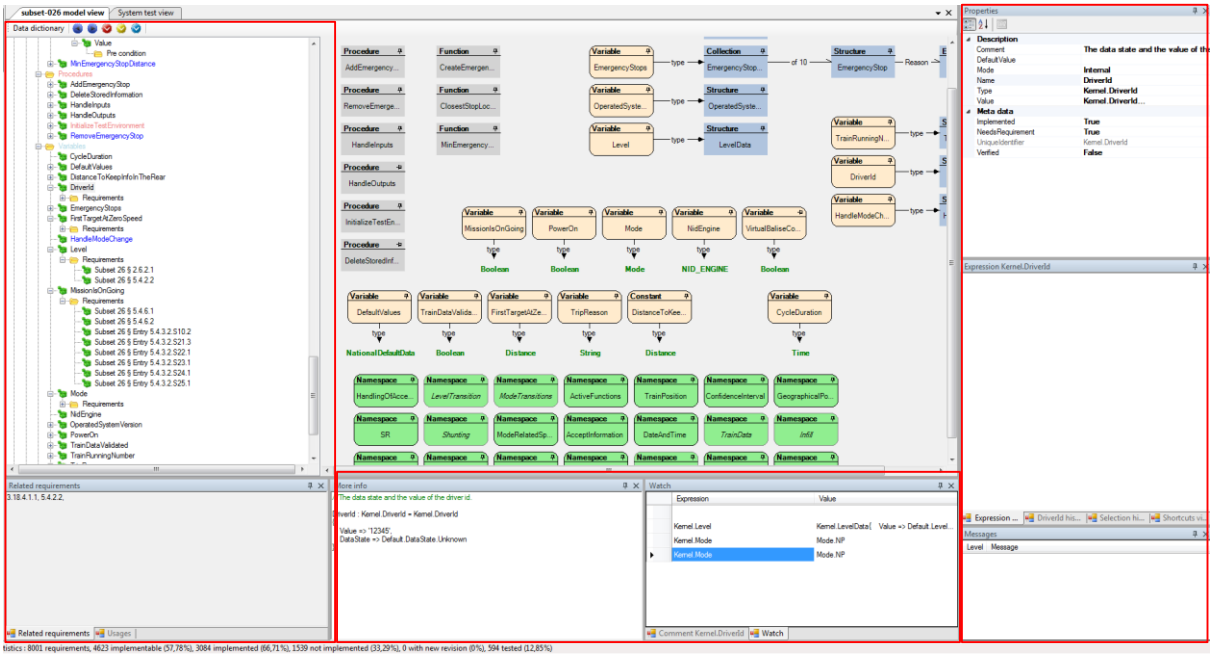

Figure 43: Representation of the different components of the data dictionary.

- The left side contains the **hierarchical tree view** which allows selecting an element in the model. To see the hierarchical tree in detail see Figure 44**.**

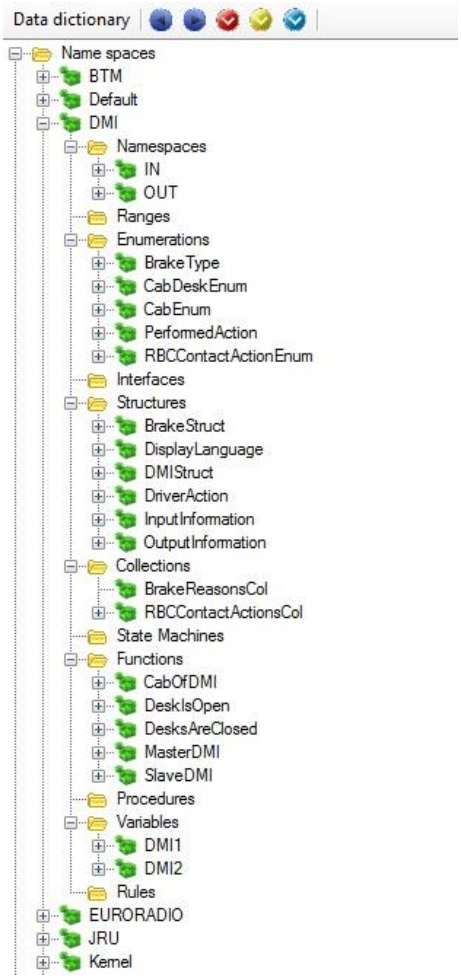

Figure 44: Hierarchical tree of the model elements.

- The upper right corner contains a **property view,** with the details of the element selected in the tree view. For further details about properties, see Section 3.6.
- The lower right corner contains a **messages view,** where the messages provided by the Workbench on the selected element (see Section 3.5) are displayed.

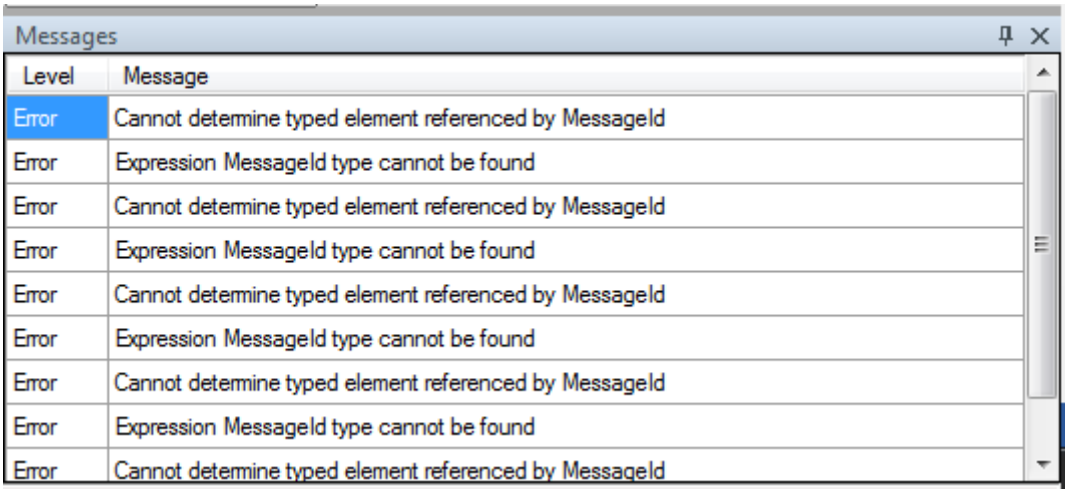

Figure 45: Message view of a selected element from the model.

- **More info:** located in the lower middle part. This view provides a textual description of the current item using a pseudo-code representation. See Figure 46 for an example.

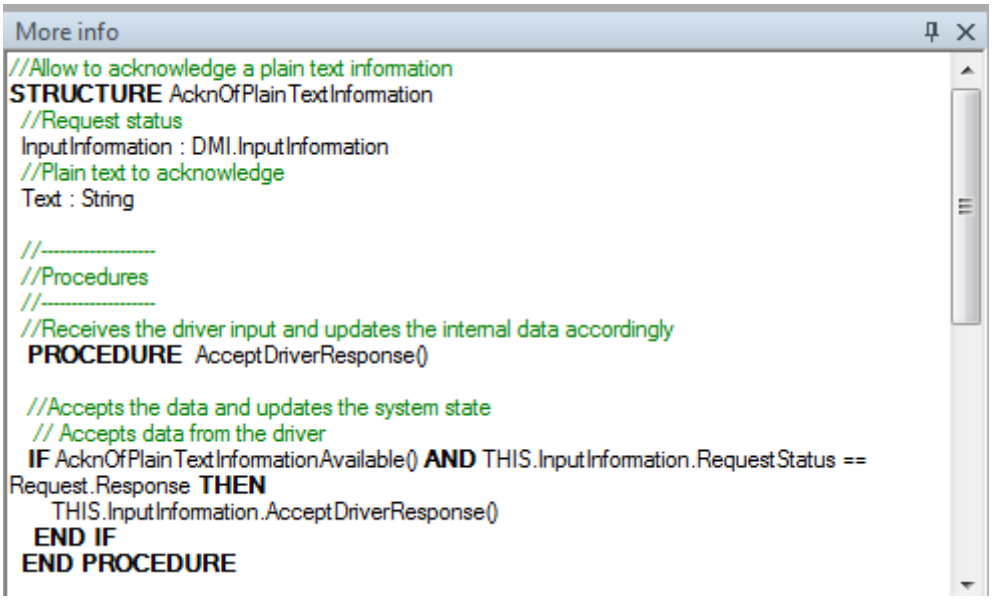

Figure 46: Representation of the More Info window.

- **Expression:** located on the middle right part. It is used to display and edit the expression related to the selected element, such as Cases, Preconditions, Actions and Expectations. Figure 47 depicts an example of the Expression window.

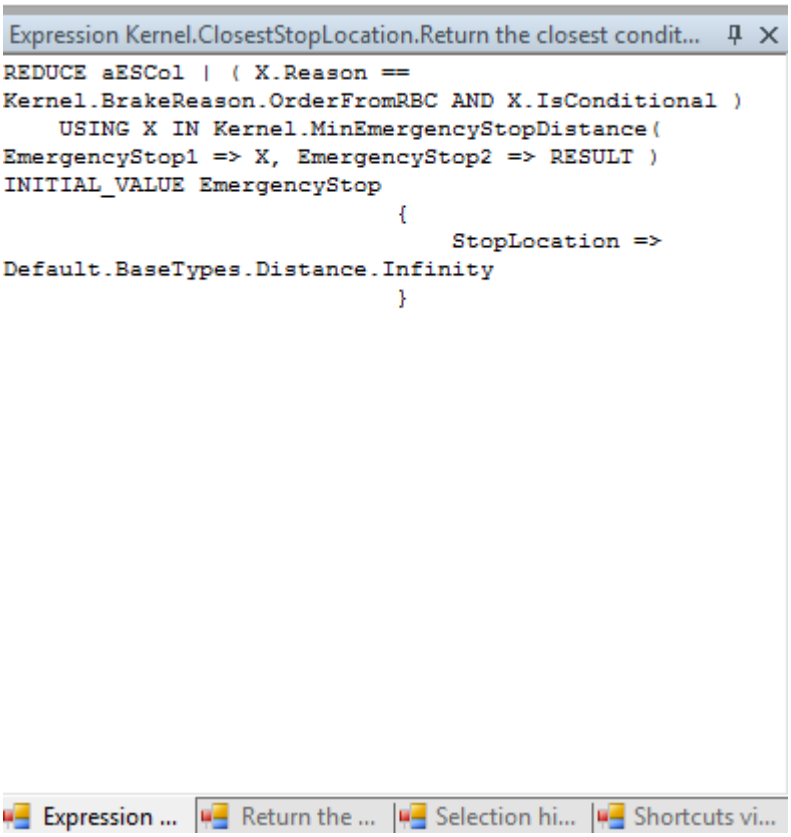

Figure 47: Representation of the Expression editor.

- **Comment:** provides meta-information related to the selected item. This window is used to display and edit the selected item's comment. The comment can also be accessed through the properties of the element (see Section 3.6).

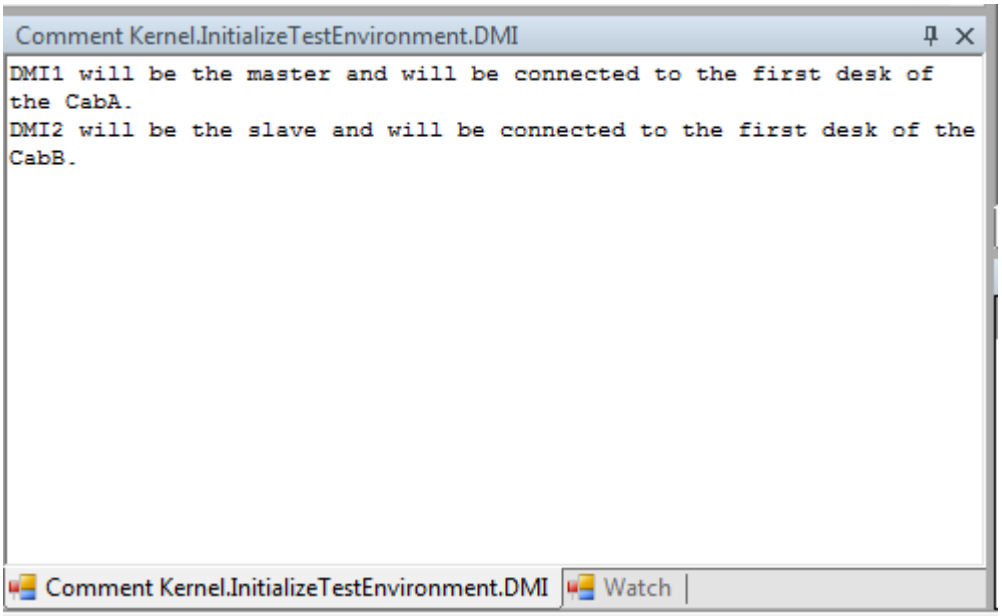

Figure 48: Representation of the comment section.

- **Watch:** during execution of a sequence, the watch window provides the current value of a list of variables and expressions the user wishes to track. Variables can be drag-and-dropped onto the watch window to add their visualization to the list. Double clicking on the Expression section of the Watch window opens an Expression editor that allows the user to edit the expression. See Figure 50.

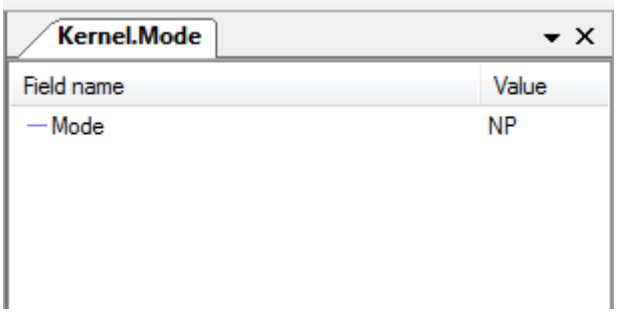

Figure 49: Representation of the Display window

Figure 50: Expression editor.

- **Usage:** the usage view displays all model elements or tests where the current item is used. The usage view is available for
  - **Types:** provides the variables that are instances of the selected type (represented by the symbol).
  - **Variables:** provides the locations where the selected variable is either read (represented by the symbol) or set (represented by the symbol).
  - **Functions:** provides the locations where the selected function is called (represented by the symbol).
  - **Interfaces**: provides the structures and interfaces that implement the selected interface (represented by the symbol).
  - **Redefinitions**: provides the elements redefined by the current structure element (represented by the symbol).

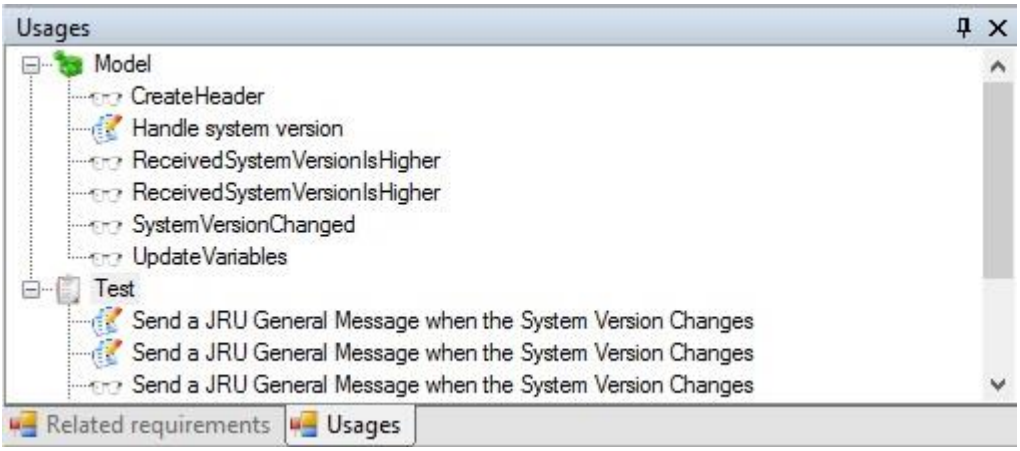

Figure 51: Usage representation.

- **Related Requirements:** displays the requirements linked to the selected model element.

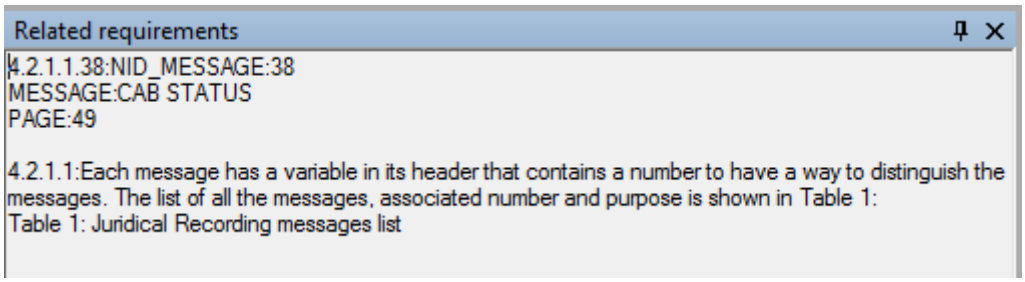

Figure 52: Representation of the Related Requirements on EFSW.

5.2.2 Common properties

The properties of model elements are as follows:

- **Default value:** the value of the selected model element when an instance is created. The default value of a variable takes precedence over the default value of its type.
- **Implemented:** the implementation status of the selected item. When this metadata is set to true the selected model element has been fully implemented.
- **Verified:** the revision status of the selected element in the hierarchical tree. If the flag is set to true, it means that has been reviewed and the implementation accepted, otherwise it must be set to false.

5.3 The model

5.3.1 Data types

The data types which are described in this section are the base data types present in EFSW. According to the needs of the modeller new EFS types can be created; the created types will inherit from one of the base data types and have specific properties depending on the base type chosen.

5.3.1.1 Operations performed on the ERTMSFormalSpecs Model for all data types

The operations which can be performed over all the base types are the following:

- Create new
- Delete existing

5.3.1.1.1 Add a new element to the hierarchical tree

To create a new element, select the proper folder in which the element should be created, right-click to open a contextual menu and select Add to add the new element. For instance, Figure 53 shows how to add a new range in the model.

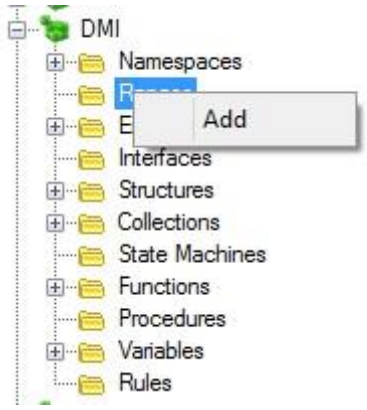

Figure 53: Add a new element on the hierarchical tree

The newly created element is added with a default name, which should be modified using the property view (see 5.2.2), along with all the properties specific to that element. The base type of the added element depends on the folder the element was added to so it is impossible, for example, to add a function to the Ranges folder.

5.3.1.1.2 Delete an existing element from the hierarchical tree

Additionally elements can be deleted from the model through EFSW. Select the element to be deleted, right-click on it and choose the Delete option in the contextual menu. The element immediately disappears from the hierarchical tree.

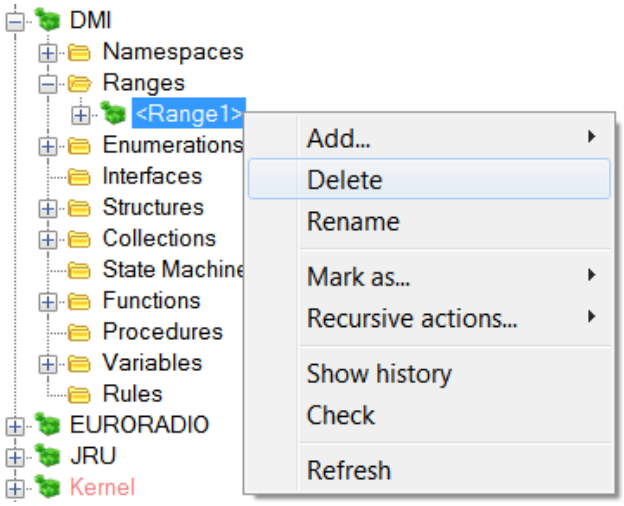

Figure 54: Delete an existing element from the hierarchical tree

5.3.1.2 Range data type

Ranges are used for numerical representations with an upper and lower bound. Each range must have a precision, the numerical separation between two consecutive values in the range. There are two types of precision:

- **Integer:** the minimal separation is a natural unit[1].

[1] The natural unit is 1.

- **Floating point:** the minimal separation is a fraction of a natural unit[2].

In addition to the bounds, **ranges** can assign a special meaning to values for the model, these are the **special values**.

Ranges have the properties described below. Figure 55 shows the properties of ranges[3]:

- **Max value:** the high limit of the possible values of the range.
- **Min value:** the low limit of the possible values of the range.
- **Precision:** separation between two consecutive values of the range. The possible options for this property are:
  - **Integer**
  - **Floating point**

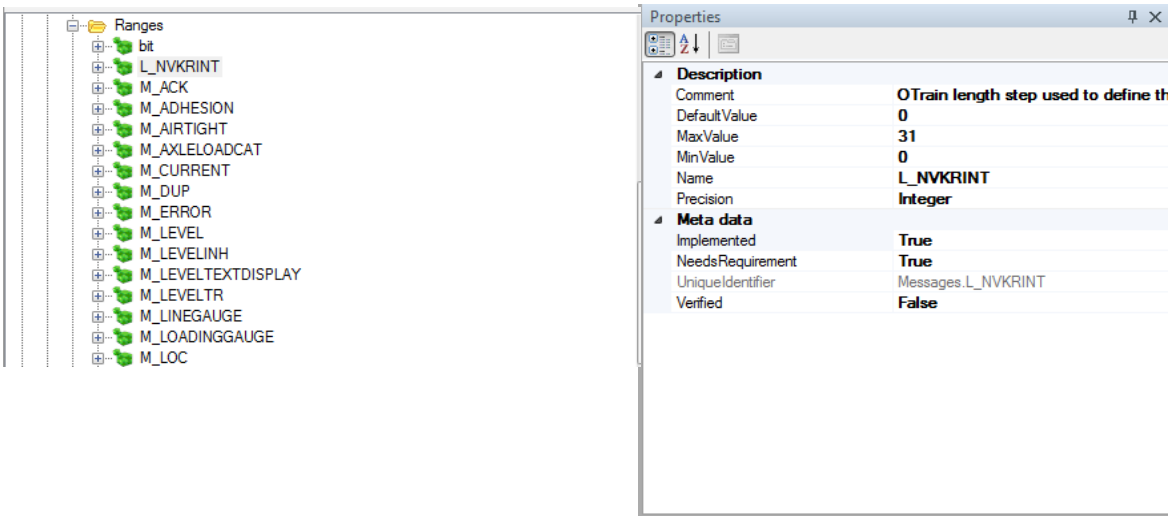

Figure 55: Range properties

5.3.1.3 Enumeration data type

Enumerations specify a set of discrete values identified by name. Normally the possible discrete values defined for an enumeration are represented by chains of characters and each value can also have a unique integer number associated to it.

An enumeration can contain sub-enumerations, representing subgroups of the enumeration's values. For instance the Mode enumeration (representing the modes available to the EVC on a train) has a sub-enumeration "Mode.ModeProfile", which contains all the possible modes that can be requested by a mode profile. Figure 56 illustrates the properties of an enumeration.

- **Values:** the collection of possible values that can be given to the selected enumeration.

[2] A fraction of the natural unit is given by: 1/X; where X > 1
[3] For further information about ranges, see Section 5.3.1.2.

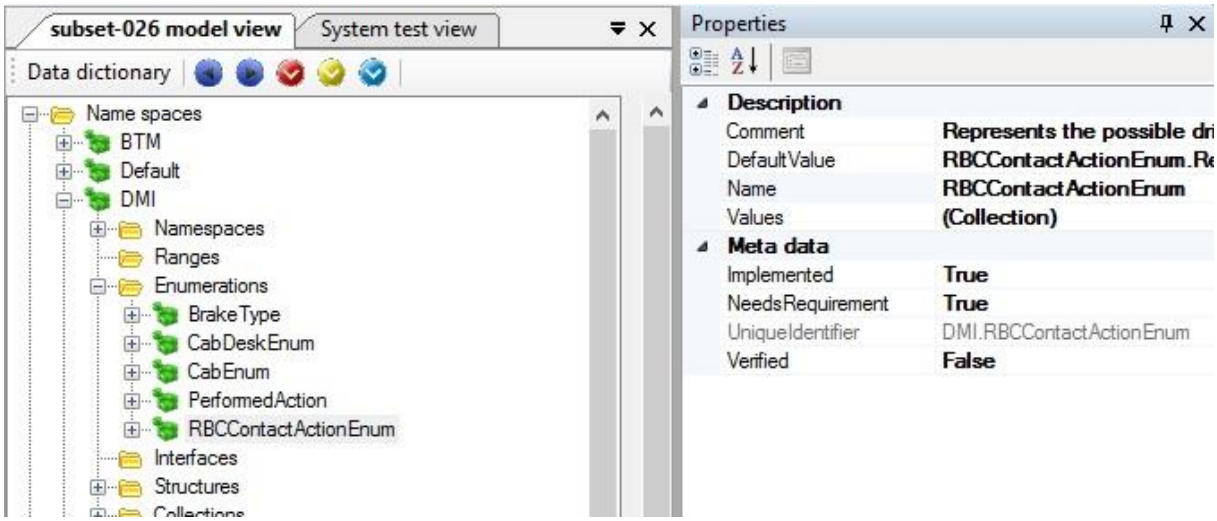

Figure 56: Enumeration properties

5.3.1.4 Interface data type

Interfaces are an abstract type[4] (that means that they cannot be instantiated) allowing to define a set of fields shared between structures (described in Section 5.3.1.5). A structure (or even an interface) indicates that it complies with an interface by indicating this in "Interfaces" folder.

The advantage of using interfaces is that one can define an operation (procedure or function) common to all structures implementing that interface only once, by indicating that the operation uses an instance of that interface instead of duplicating the operation for each structure which implement the interface.

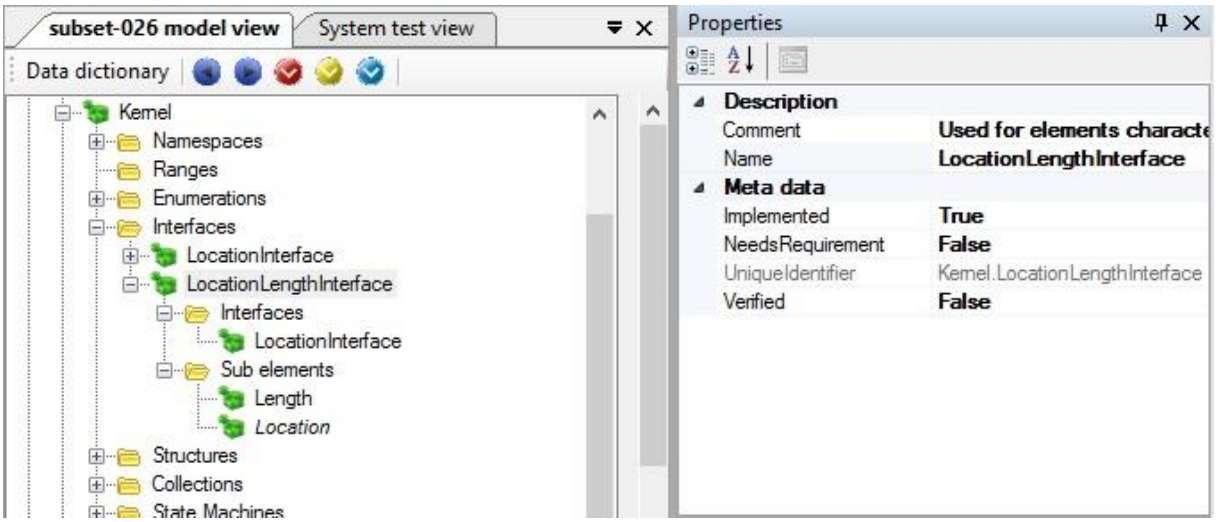

Figure 57: Interface properties

[4] This indicates that one cannot create an instance of the corresponding type. For instance, if I1 is an interface, one cannot write the following statement: A <- I1 { Id => 123 }, because this would create an instance of that interface.

5.3.1.5 Structure data type

The EFS **Structure** matches the C-like *struct*. All the structures present on EFSW can contain different sub-elements with different types, including other structures (called sub-structures), rules, procedures…

The presence of **procedures** and **rules** in structures provide dynamic behavior (see Sections 5.3.2.3 and 5.3.2.5 for further details). Figure 58 shows the properties of structures.

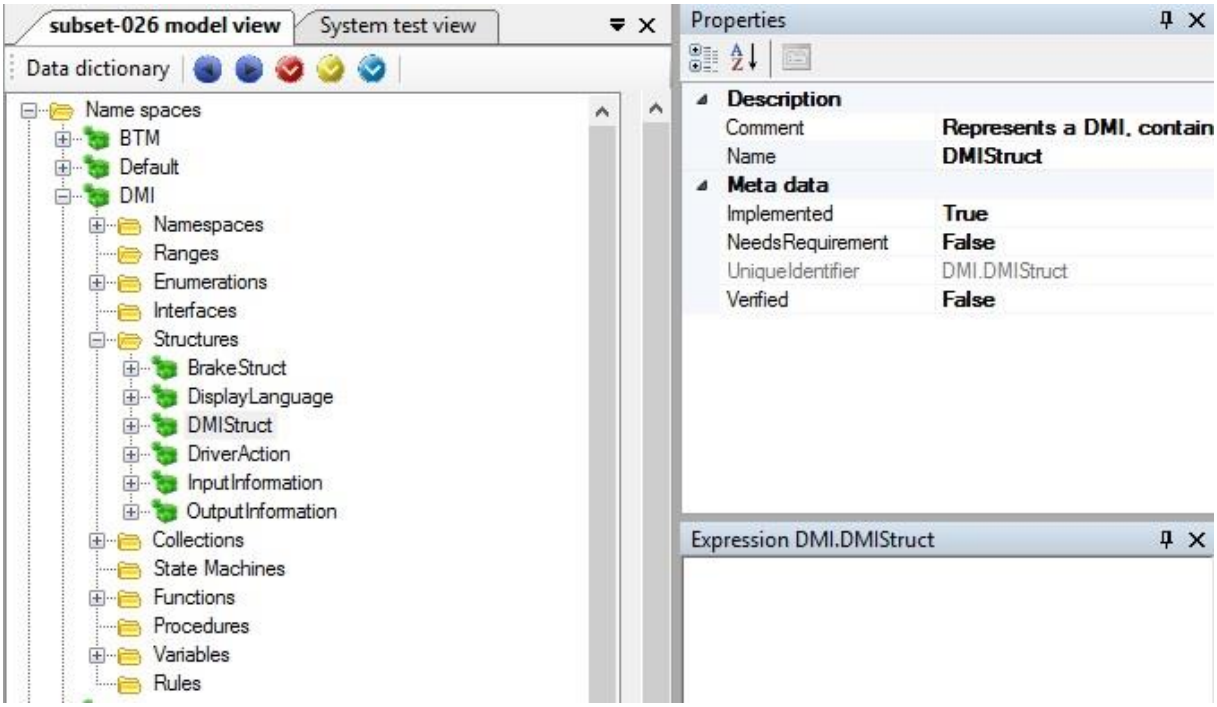

Figure 58: Structure properties

Structures implementing one or several interfaces (see Section 5.3.1.4) must implement all the structure elements defined by the interfaces. These elements can be generated automatically by selecting “Generate inherited fields” from the contextual menu, as depicted on Figure 59: Automatic generation of inherited fields. This contextual menu is only present for structures implementing one or several interfaces.

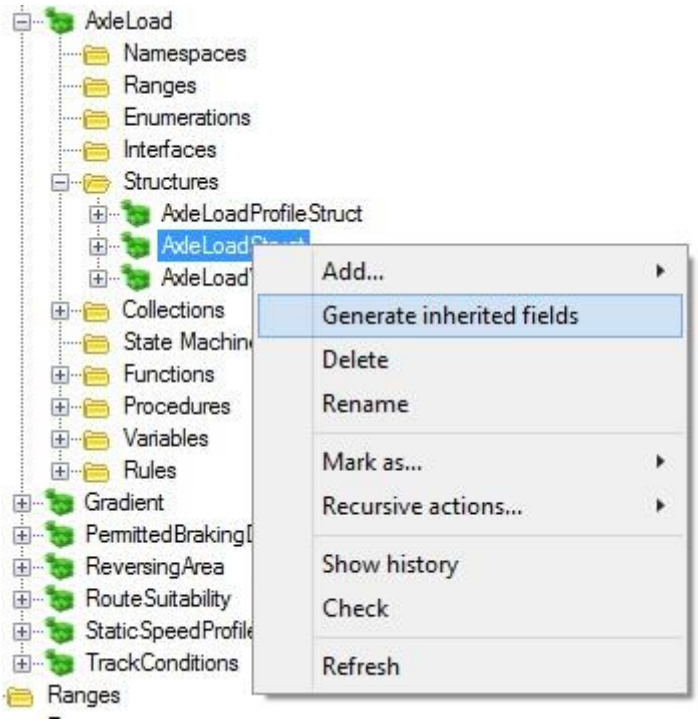

Figure 59: Automatic generation of inherited fields

5.3.1.6 Collection data type

EFSM **Collections** allow listing and grouping of several elements of the same type. In general, collections in EFSW are empty by default. To use them, at least one element needs to be added to the collection.

In order to add a new value to a collection a dynamic component of the EFSM is used, such as a procedure or rule; for further details about these components see Section 5.3.2.3 and Section 5.3.2.5.

Collections have a fixed maximum size. Figure 60 shows the properties of a collection.

- **Max size:** the largest number of elements that can be allocated to the selected collection.
- **Type:** the kind of elements to be allocated to the selected collection. Only elements which match the type of the collection can be allocated to it.

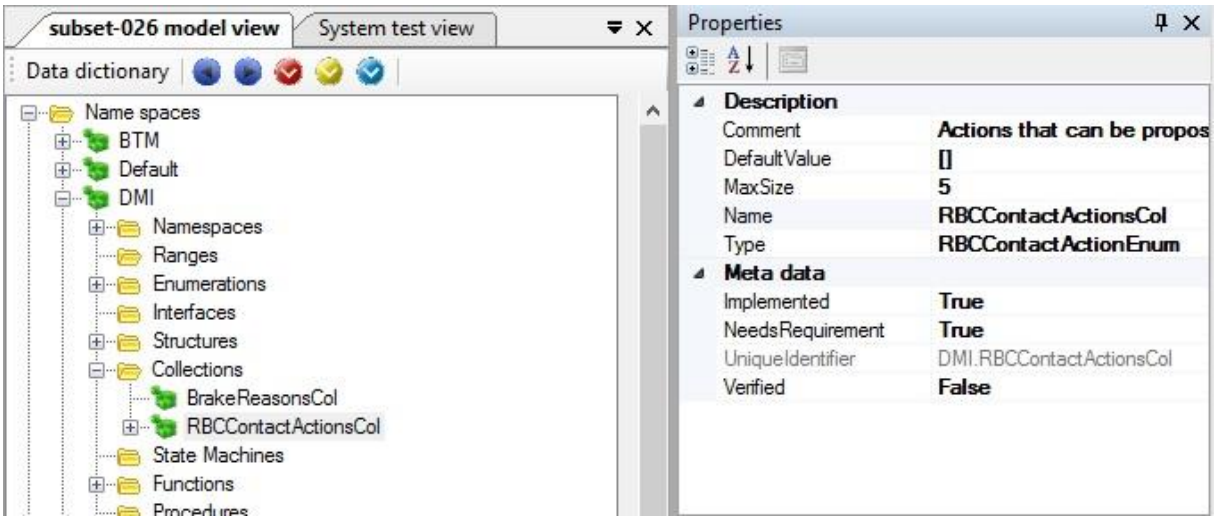

Figure 60: Collection properties

5.3.1.7 State machine data type

EFSW **state machine** matches the traditional concept of finite state machine. It has an initial state,[5] a number of possible states and a set of **rules** (see Section 5.3.2.5 for further details about **rules**).

All state machines can be graphically represented by a state diagram. The state diagram representation of the state machine is available through the View state diagram action in the contextual menu, as depicted by Figure 62. Figure 61 illustrates in detail the properties of a state machine:

- **Initial state:** indicates which the initial state of the selected state machine is.

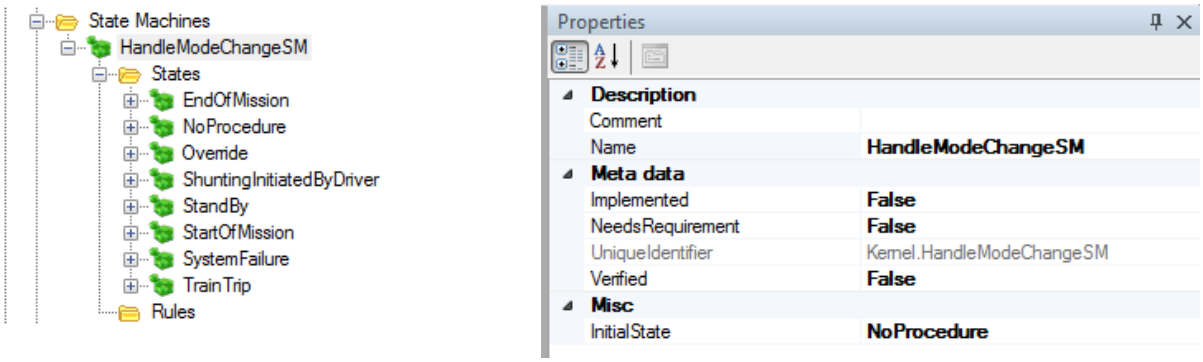

Figure 61: State machine properties

[5] The Initial State is a particular case of a State.

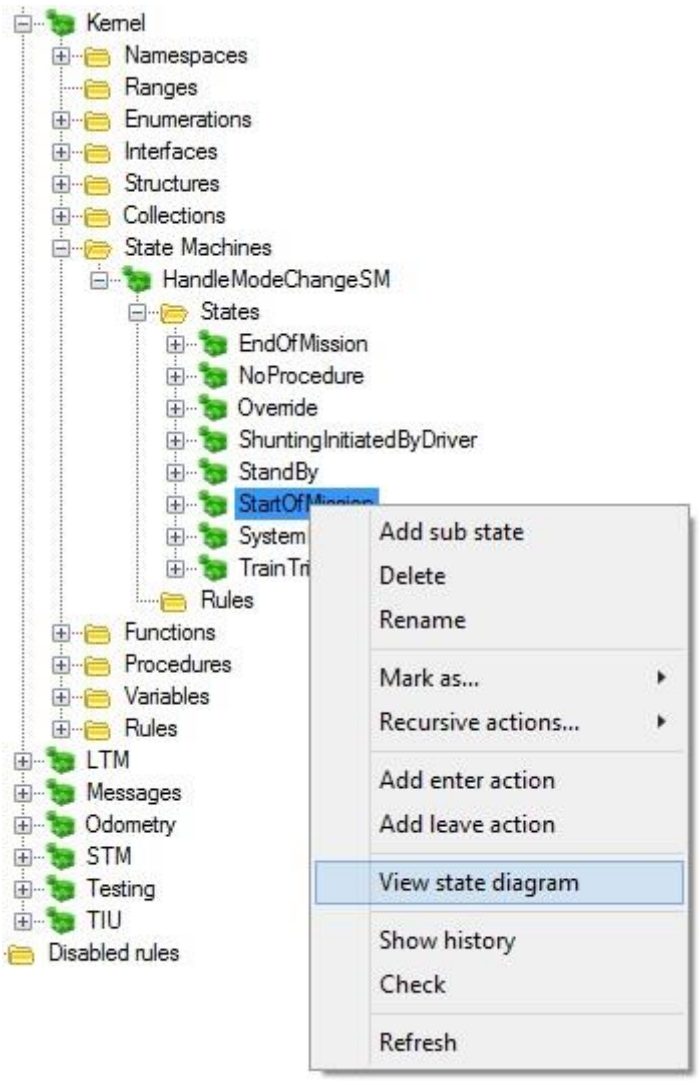

Figure 62: Opening the state diagram view.

A state is the basic block used to build a **State Machine**. It is characterized by a name which identifies it in the current **state machine**, and each state can be decomposed into further sub-states.

The second building block for state machines is a transition. Transitions are inferred from the model using the following pattern: a rule which is either defined in state S1 or which requires the state machine to be in state S1 in its preconditions, and modifies the state to a new value S2 is a transition from S1 to S2. This transition is displayed as an arrow from state S1 to state S2 in the state diagram. That arrow may take one of the two colors:

- **Black:** the rule is defined in a state of the state machine.
- **Purple:** the rule is defined in the model, outside any state of the state machine.

This has the advantage of differentiating transitions which are explicitly described in the state machine (for instance the transition from S0 to S1, or from S1 to D2 in Figure 63) from transitions defined externally, elsewhere in the specification (for instance, the transition named “After D11” from S24 to S10).

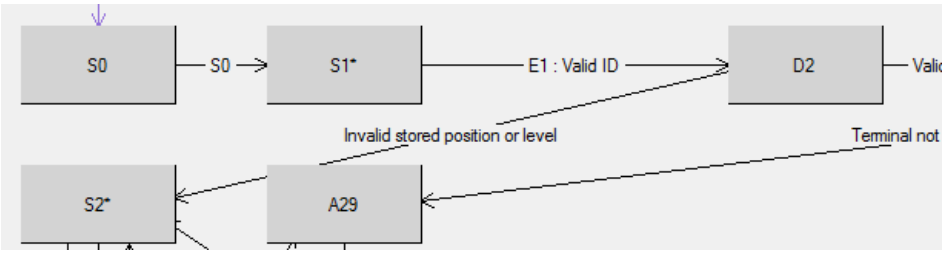

Figure 63 : Internal transition in a state diagram

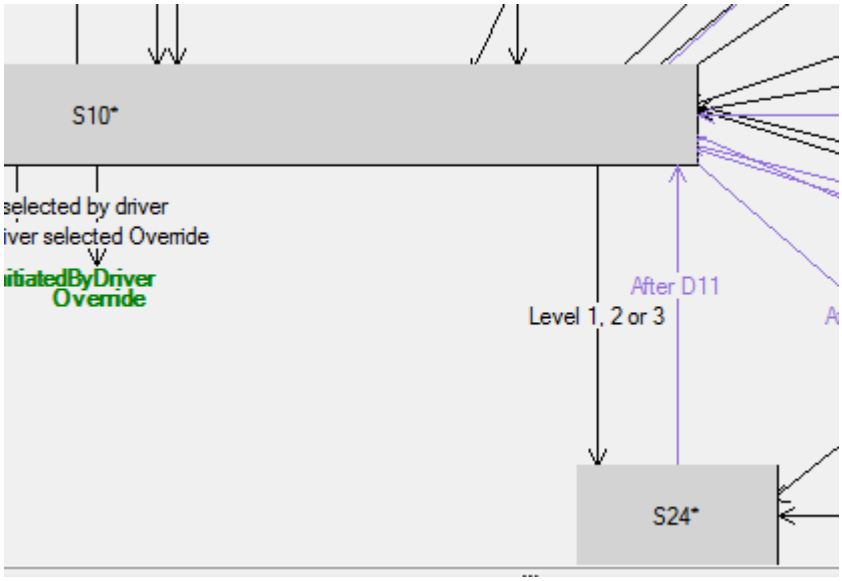

Figure 64 : External transition in a state diagram

5.3.1.7.1 State diagrams

The state diagram of a state machine described on EFSW can be visualized using state diagram view.

5.3.1.7.1.1 Display a state diagram

To display a state diagram, select the corresponding state machine in the data dictionary window, right click on the state machine to access its contextual menu, and select View state diagram, as depicted on Figure 62: Opening the state diagram view.

Based on the state machine information and on the rules of the model, EFSW builds the state diagram of the state machine and displays it in a new window. Figure 65 depicts an example of a state diagram.

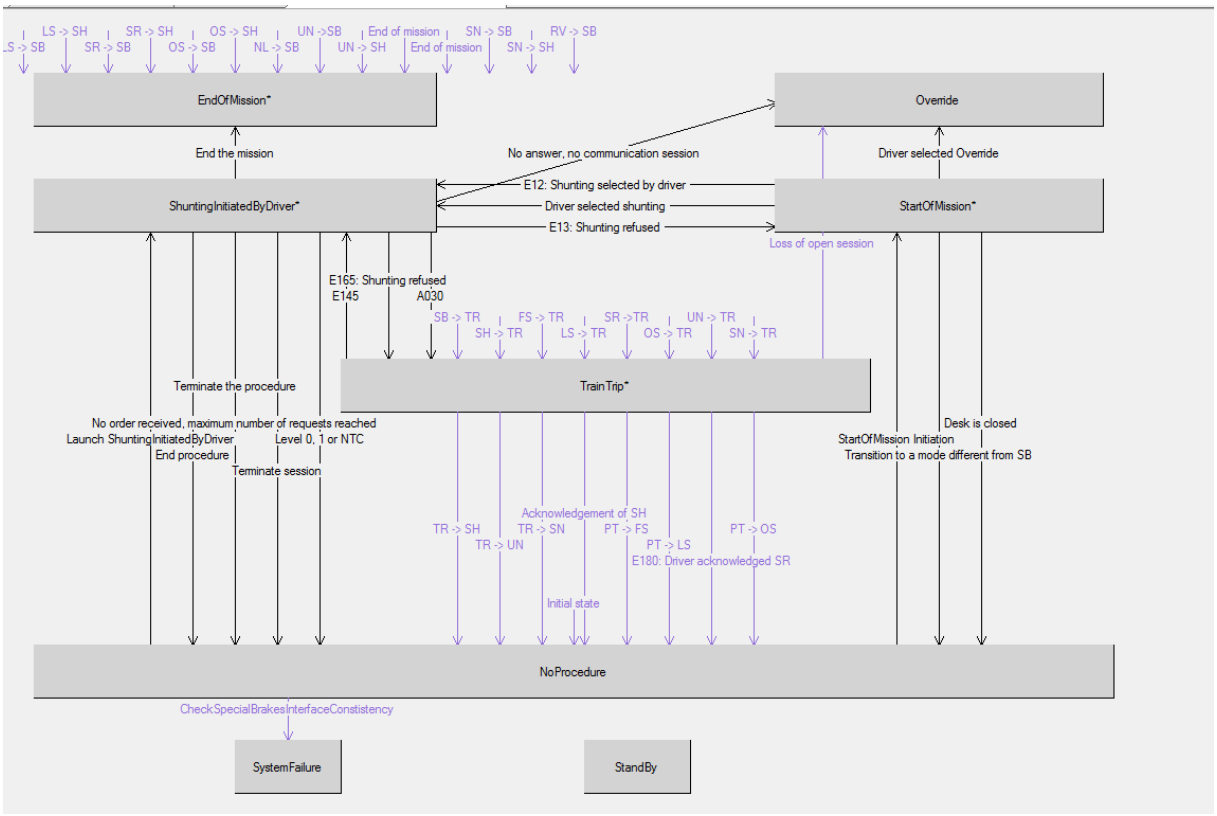

Figure 65: State diagram view

Double clicking on a state opens the sub-state machine related to that state. For instance, Figure 66 shows the sub state diagram of the *StartOfMission* state.

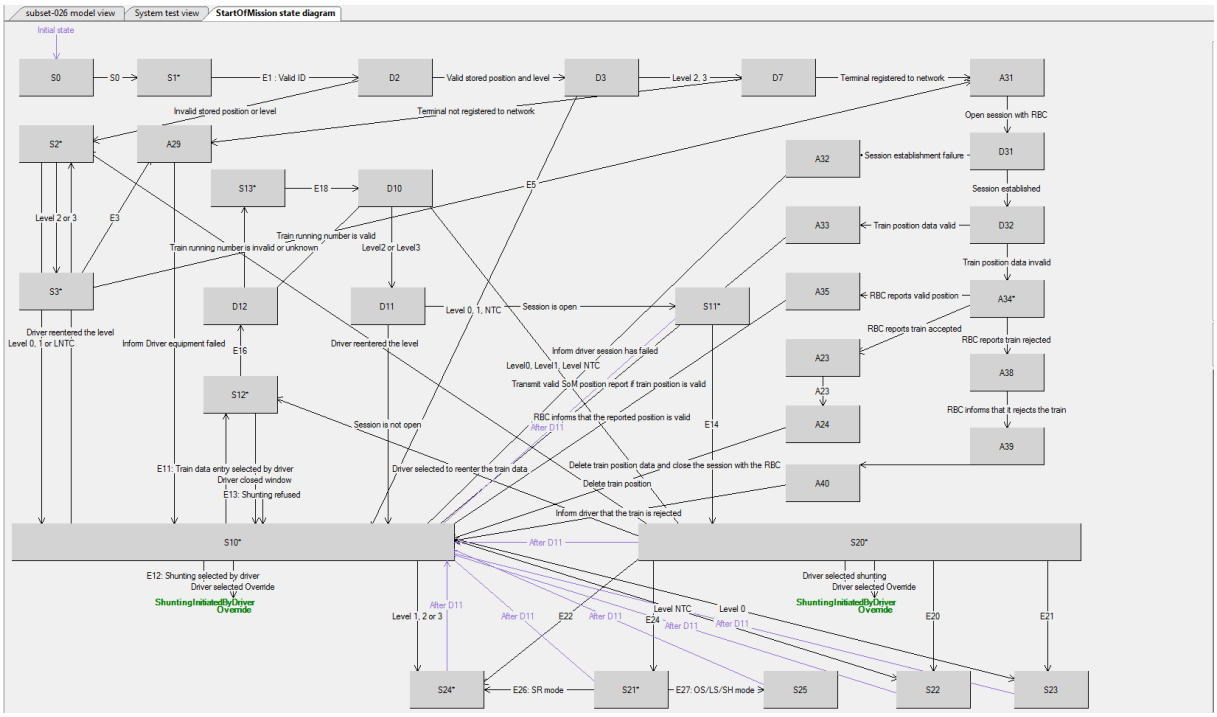

Figure 66: Sub-states of the state StartOfMission.

5.3.1.7.2 Manipulations of a state diagram

New states and new rules can be added in the state diagram using the graphical view, using the contextual menu.

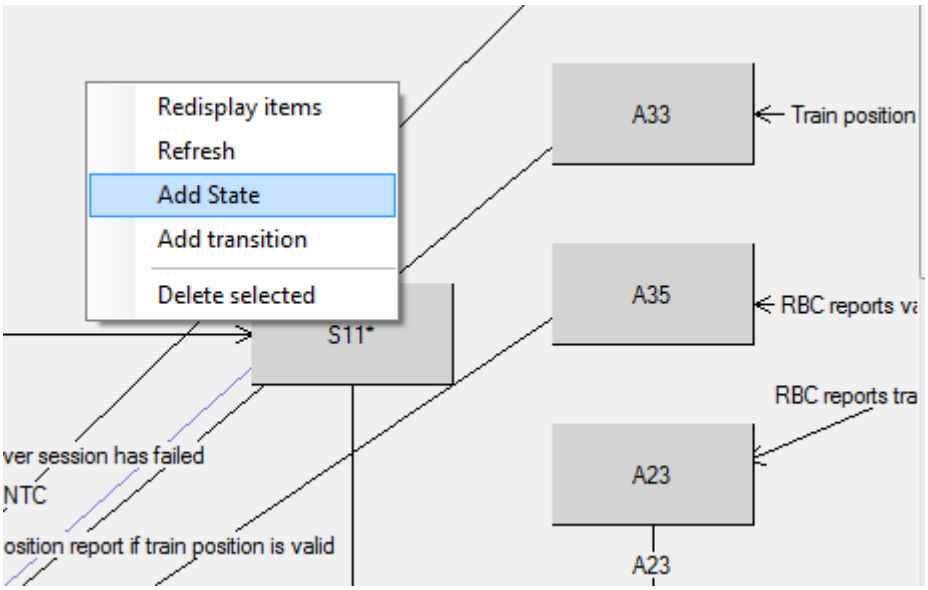

Figure 67: Contextual menu for a state diagram

5.3.1.7.2.1 Add a new state

To add a new state, select Add state in the state diagram window contextual menu. This creates a new state in the state machine and updates the state diagram accordingly.

5.3.1.7.2.2 Add a new transition

A new rule can be added by selecting Add rule in the same contextual window. This creates a new transition in to this state machine.

5.3.1.7.2.3 Selecting element in a state diagram

When a state or a transition is selected, its properties are displayed in the right side of the window and the corresponding element is selected in the Model.

5.3.1.8 Traceability

Traceability information can be added to data types by dragging a requirement to the related data type.

The **related requirements** are presented as sub-nodes of the data type on the left part of the window. They indicate the requirements that are modelled by the corresponding data type. One can add a new requirement to this list by **Drag&Drop** between the requirement in the specification view and the data type node in the data dictionary view.

5.3.2 Model elements description

5.3.2.1 Namespaces

Namespaces are used to group model elements together. Figure 68 displays a typical namespace graphical view, holding types, variables, functions, procedures and sub-namespaces. Each namespace contains

- **Sub namespaces**
- **Data types**
- **Procedures**
- **Variables**
- **Functions**
- **Rules**

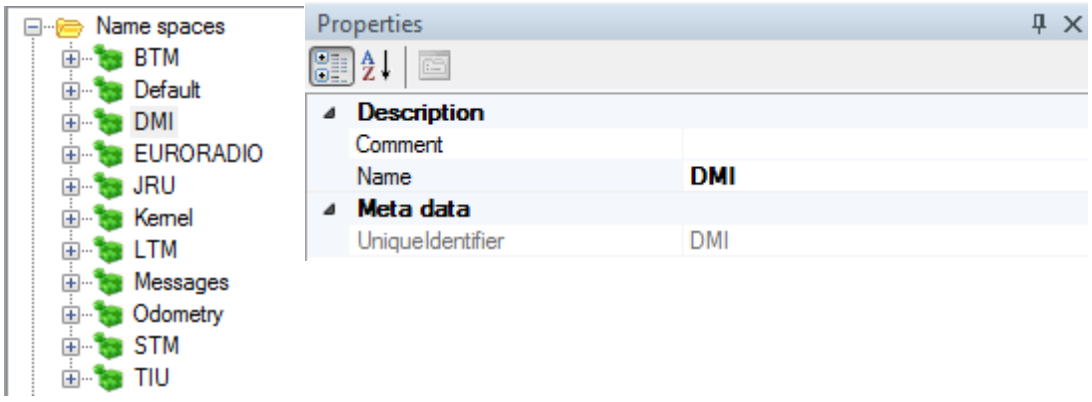

Figure 68: Namespace view

For instance Figure 69 displays the following namespaces: *BTM*, *Default*, *DMI*, *EURORADIO*, *JRU*, *Kernel, Messages, Odometry*, *STM* and *TIU*.

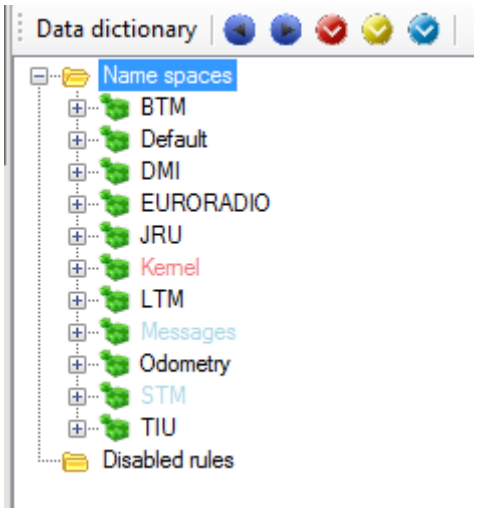

Figure 69: Available namespaces on EFSW

EFSW provides a functional view of namespaces. To access this, right-click on the desired namespace and select the Functional view option in the contextual menu. As shown in Figure 70, the functional view displays the sub-namespaces present in a namespace.

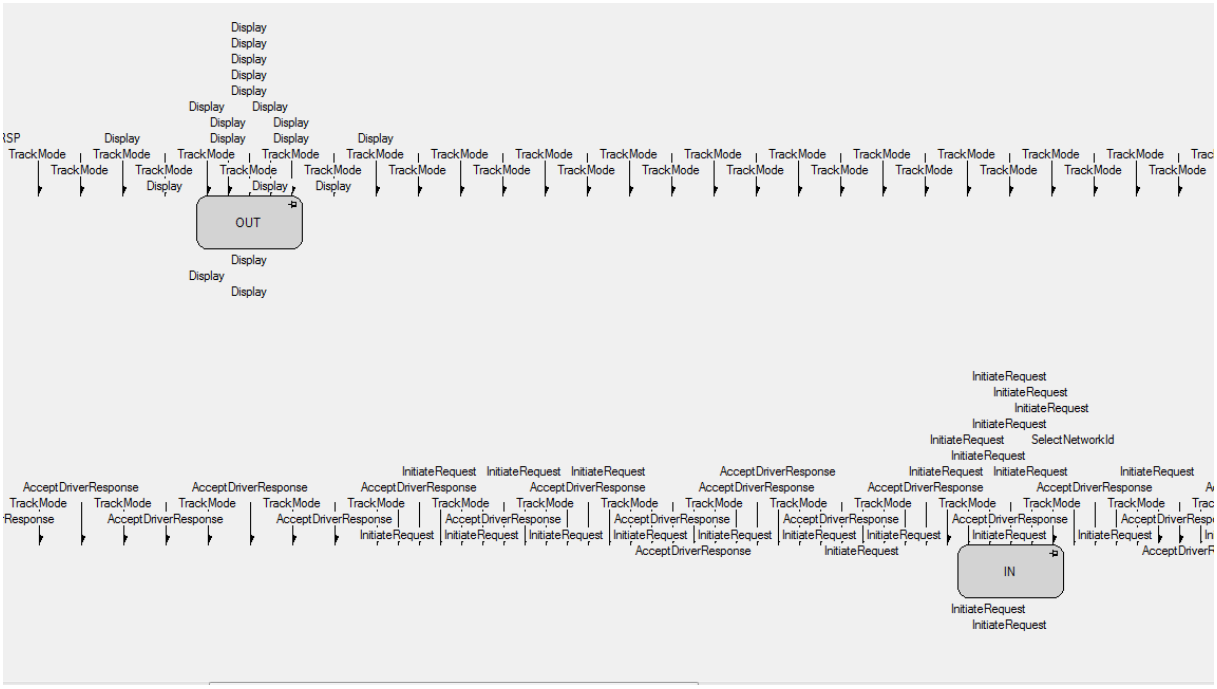

Figure 70: Functional view of the DMI namespace

5.3.2.2 Functions

Functions define functional computations, i.e. computing a value according to the parameters' values and the system's current state. A function is split in a set of mutually exclusive cases. The first case whose preconditions are satisfied is used to compute the function's value.

$$f(x) = \begin{cases} x^2+1 & \text{if } x > 0 \text{ and } x < 100 \\ 2x & \text{if } x > 100 \end{cases}$$

Equation 1: Function definition.

Figure 71 shows the properties of a function:

- **Is cacheable**: indicates whether the calculated result of the function may be stored and re-used by later function calls until the end of the cycle. When the flag is set to true the result is stored after the first call on a cycle and re-used. When it is set to false each time the function is called the result must be computed.
- **Type**: represents the kind of elements returned by the selected function as the result of its computations.

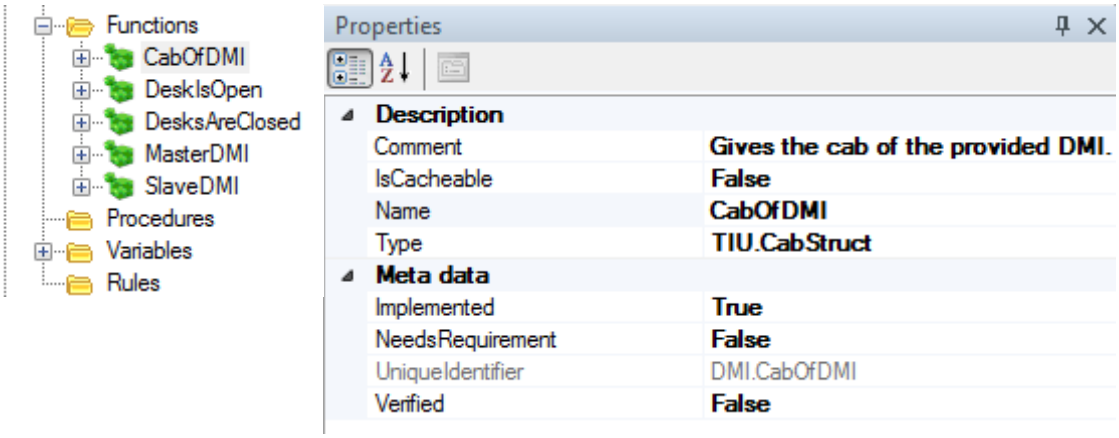

Figure 71: Function properties

Functions allow factorizing common computations in a single location. They are identified by a unique name, they have a specific return type and may have several parameters. Functions can be used to model complex functions[6] such as the one depicted in Figure 72, a graphical example of a function calculating the confidence interval of the train's position.

[6] This kind of function is mainly used in braking curve computation

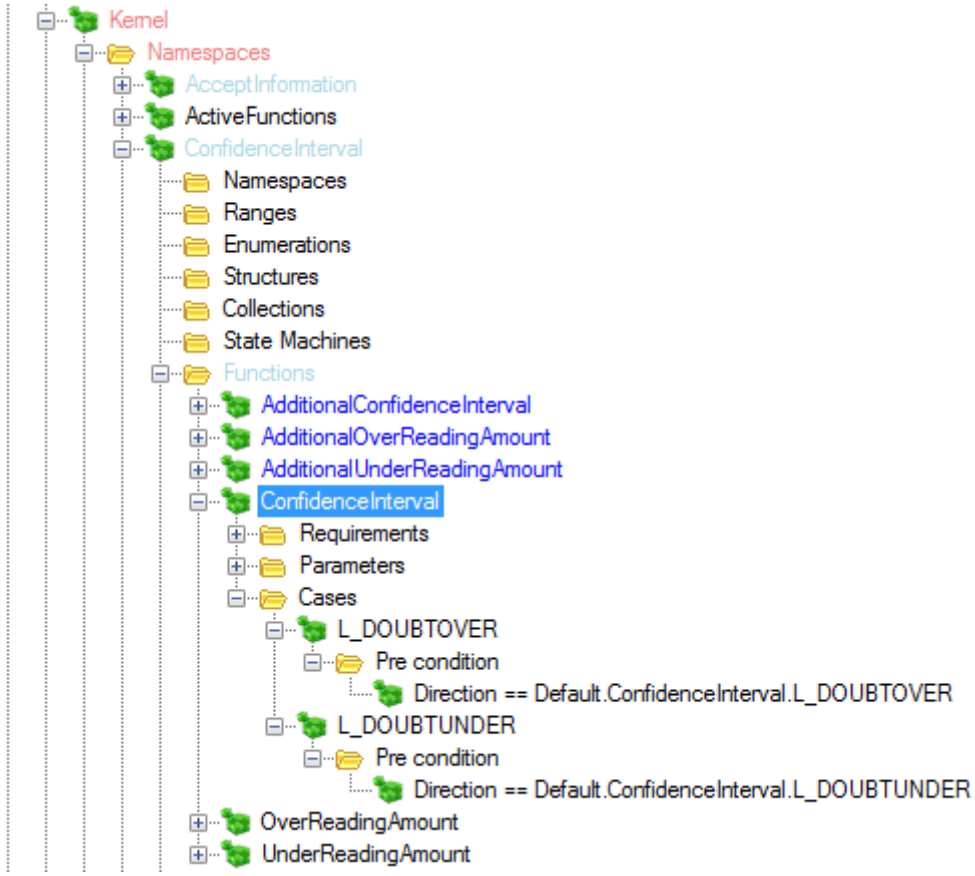

Figure 72: Example of a function

Functions manipulations (Add/Add Parameter/Add Case/Delete) are performed using the contextual menu, as in Figure 73 and Figure 74. Properties of functions are altered using the property view on the right part of the Model main window.

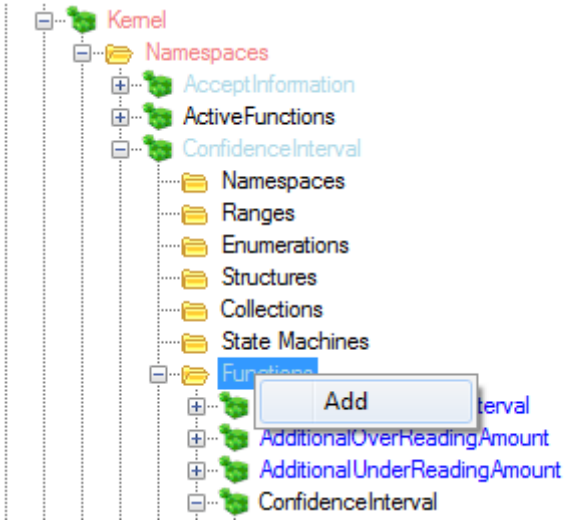

Figure 73: Contextual menu used to add a function

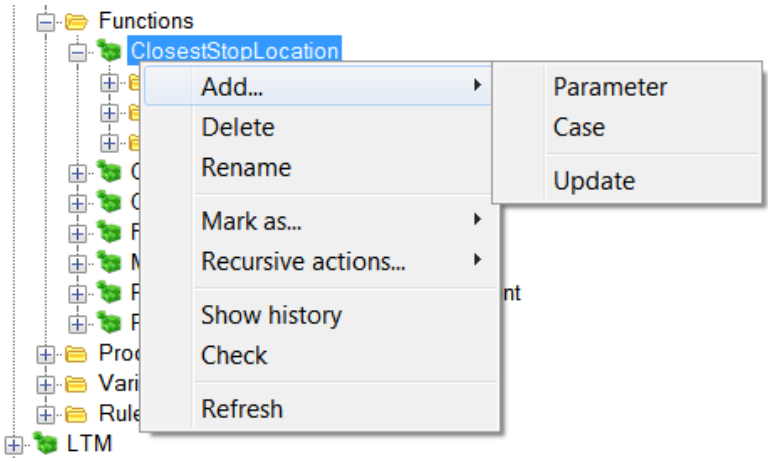

Figure 74: Contextual menu which allows adding parameters or cases to a function and deleting a function.

5.3.2.2.1 Functions graph view

EFSW provides a graphical display feature for functions whose value changes with the position or the speed of the train. This is the graph view feature. This feature is only available for certain functions: those requiring a parameter of type distance and return a speed, or functions requiring two parameters, one distance and one speed, and return an acceleration.

A function's graph is accessed by right-clicking on Display in the function's contextual menu. The graph will appear in a new window.

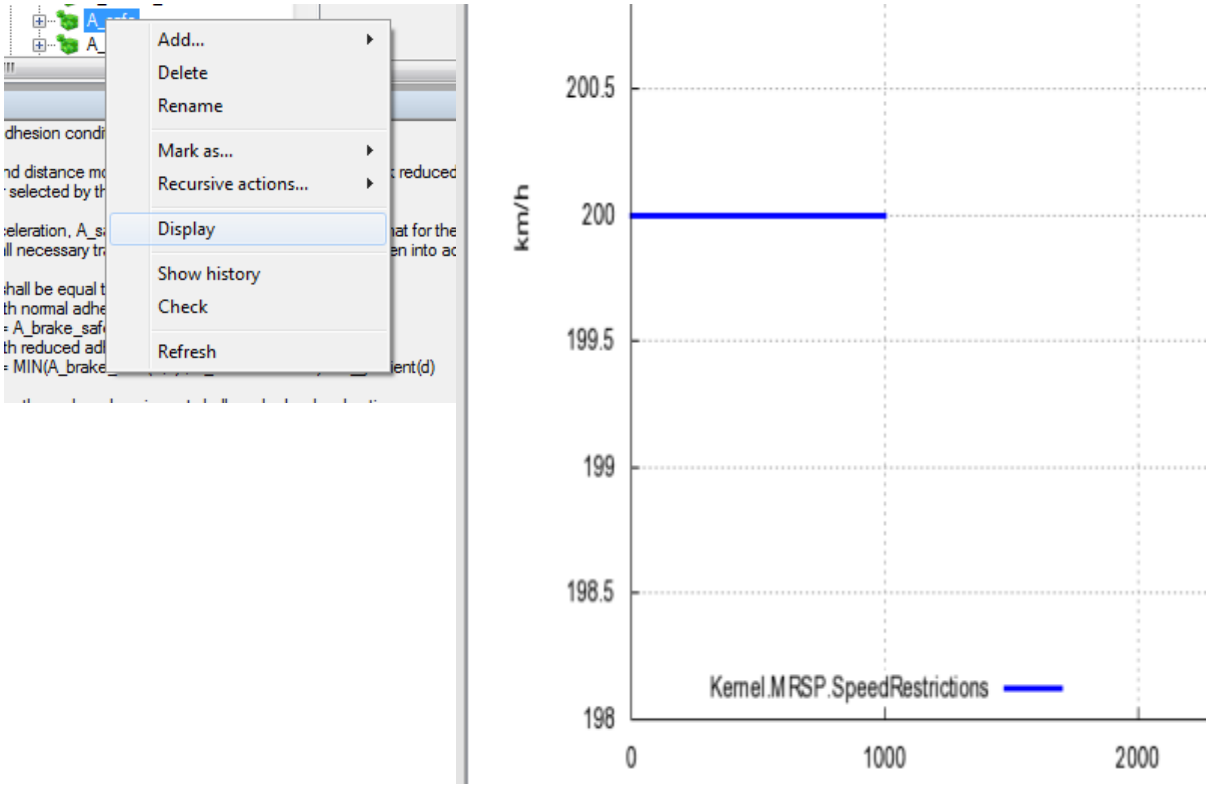

Figure 75: Graph view of the MRSP computation function

It is possible to drag&drop a function onto an open graph view. This will display the graph of the second function overlapped on the first. See Figure 76.

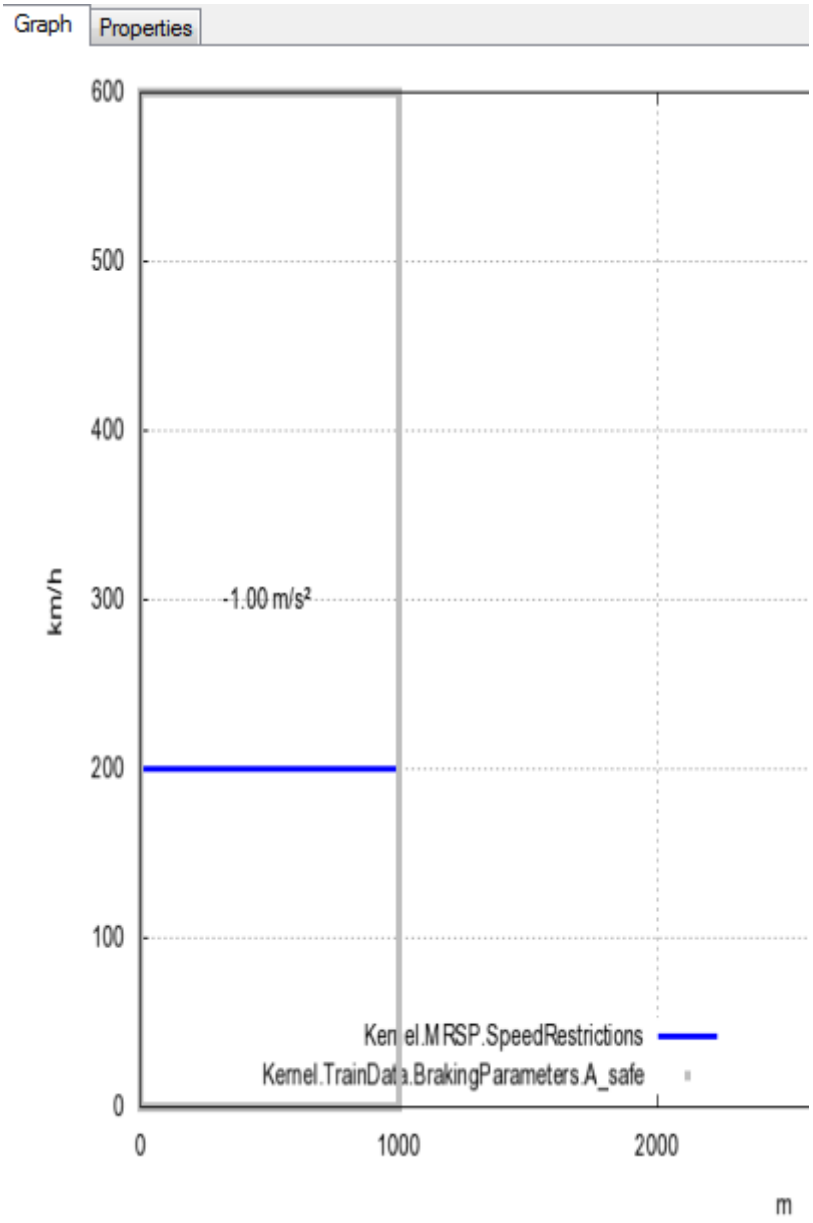

Figure 76: Overlap of the graph view of two functions

5.3.2.3 Procedures

Procedures are used to perform actions in a sequential fashion. This is the only case where imperative programming is available in EFS. In the case of procedures, the first statement is fully executed before executing the next one. All procedures contain the following:

- **Name:** identifies this procedure in the system.
- **Parameters:** formal parameters for this procedure.
- **Rules:** rules to be activated in order when the procedure is called.
- **Comment:** gives a brief description of the functionality of the current Procedure**.**

Procedures have all the common properties of model elements, as illustrated in Figure 77.

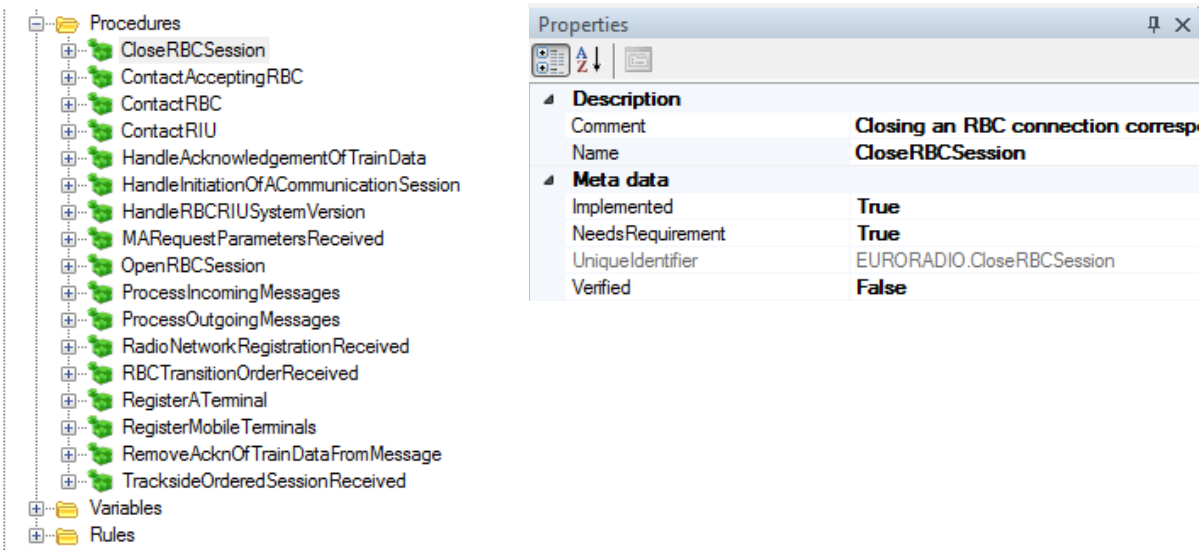

Figure 77: Procedure properties

5.3.2.4 Variables

Variables are used to store the values used to describe the system's state. **Variables** are characterized by:

- **Name:** identifies the variable in the system.
- **Type:** as defined above.
- **Default value:** which overrides the default value specified for the type.
- **Mode:**
  - **Internal:** the variable is used by the EFS model only.
  - **Out:** the variable is written by the EFS model and read by the outside world.
  - **In:** the variable is written by the outside world and read by the EFS model.
  - **In/Out:** the variable is used by both the outside world and the EFS model.
  - **Constant:** the variable is initialized at EFS start up. Its value then never changes. It pertains to the "Data Prep".
- **Value:** the current value of the variable.

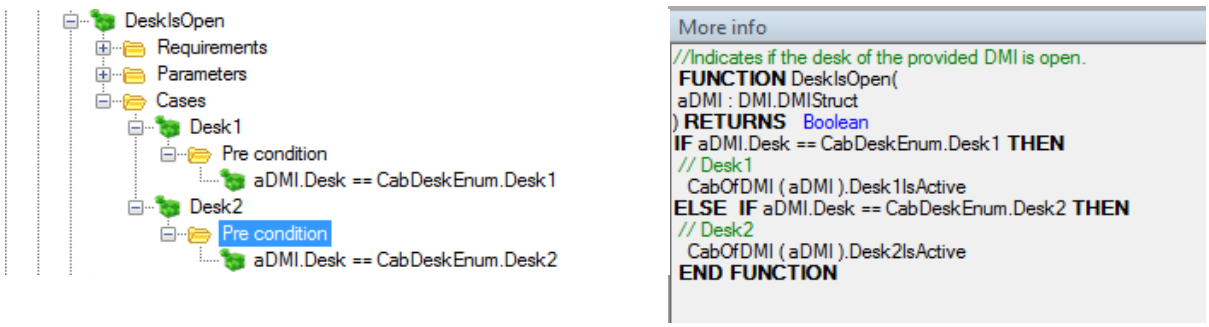

Figure 78: Variable representation in EFSW

Figure 79 shows the properties window of a variable.

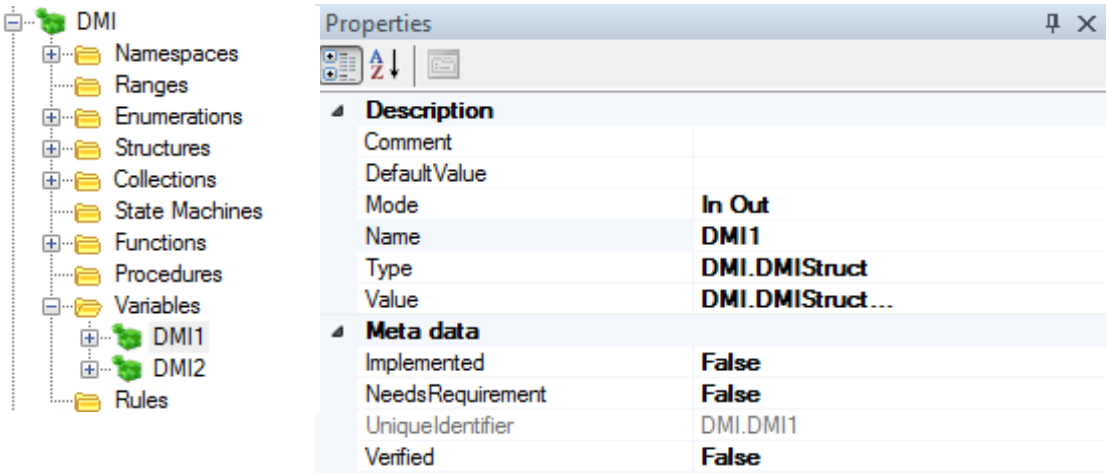

Figure 79: Variable properties

Variable manipulations (Add/Delete) are performed using the contextual menu, see Figure 80 and Figure 81. Properties of variables are altered using the property view on the right part of the Model main window.

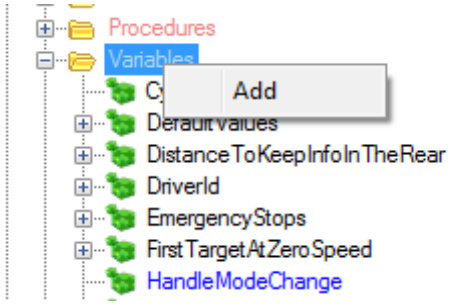

Figure 80: Contextual menu for adding a variable

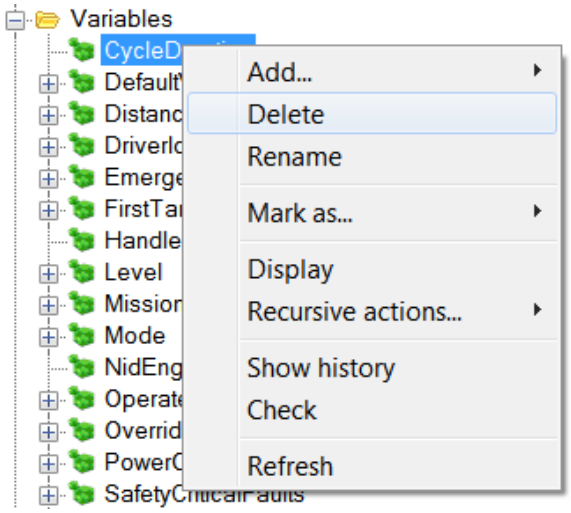

Figure 81: Contextual menu for deleting a variable

5.3.2.4.1 Sub variables

Variables in the EFSM whose base type is Structure may contain sub-variables. Right-clicking on a variable displays the contextual menu offering the possibility to Display the variable's structure. See Figure 82.

After selecting the Display option of the contextual menu, the variable's structure representation is displayed on the right side of EFSW main window. Figure 83 displays the representation of the "*Kernel.Level*" and "*Kernel.FirstTargerAtZeroSpeed*" variables.

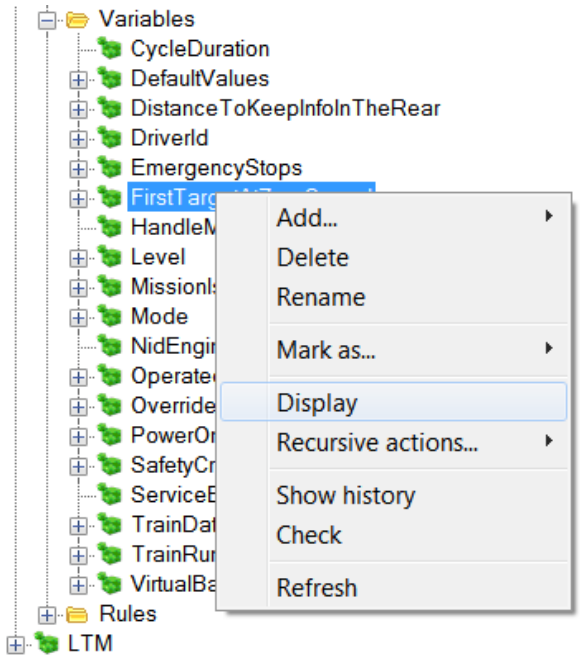

Figure 82: Contextual menu for displaying the enclosed sub-variables

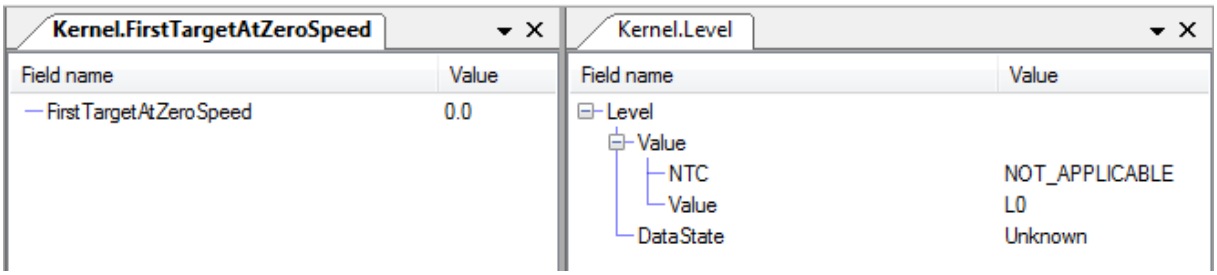

Figure 83: Representation of the sub-variables enclosed on a variable

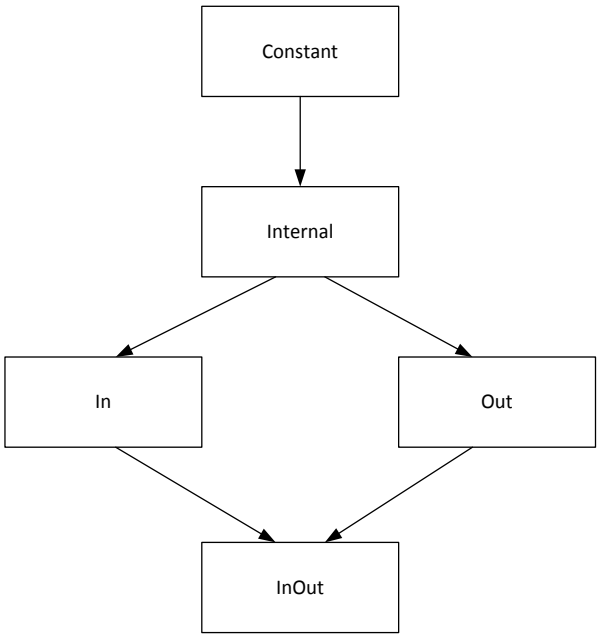

Figure 84: Order of the possible modes of a variable

The mode of a sub-variable must, at least, be higher than the mode of the variable where it is enclosed (as depicted in Figure 79). For instance, a variable whose mode is **In** can contain sub-variables with modes **In**, **Internal** or **Constant** but never **Out** or **InOut**. Figure 85 depicts an existing situation with a variable enclosed in a structure and both of them have different modes.

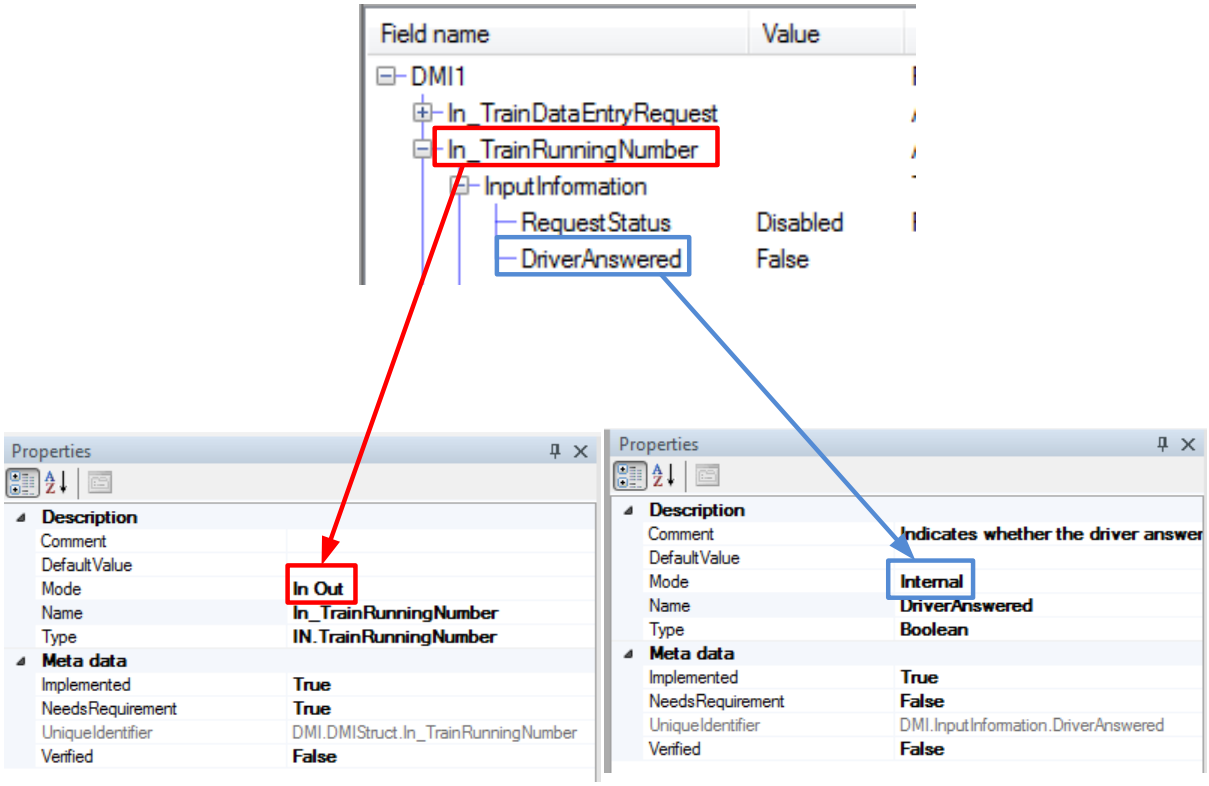

Figure 85: Variable whose mode is internal and is enclosed in an InOut variable

5.3.2.5 Rules

Rules are used to describe the system's dynamics. All the rules are made by a set of rule conditions containing preconditions, actions and sub-rules.

- **Rule conditions:** Only one of the rule's conditions is activated when a rule is activated. When several conditions can be activated, the EFS interpreter activates the first one.
- **Pre-conditions:** Each rule condition may have a list of pre-conditions that must all be true for the rule condition to be triggered. Each pre-condition is an EFS expression (which may contain variables and functions in the model) that evaluates to either True or False.
- **Actions:** Each rule condition contains a set of one or more actions which are executed when the rule is activated. An action is a statement that either calls a procedure or updates the value of a variable in the model.
- **Sub-rules:** It is possible for a rule condition to contain other rules in its hierarchical tree structure. These are called the sub-rules.

When a rule condition's preconditions are satisfied, all the actions are applied on the system, and sub rules are evaluated.

Rules manipulations (Add/Delete) are performed using the contextual menu. Reordering of rule conditions inside a rule or of rules inside a procedure can be done by the **Drag&Drop** feature of EFSW. In this case drag the rule or rule condition to be moved and drop it on the rule or rule condition which will follow it while holding down the "alt" key[7]. Properties of rules are altered using the property view on the right part of the Model main window.

The simulation performed by EFS follows a cycle, described in [1]. After inputs have been provided, an **Input Verification** phase is applied. This is followed by the **update of the internal state.** After this step, the actual **business process** is executed. The next step is the **update of the output variables**. The external world can then retrieve the output values before the final **clean up phase**. This is depicted by Figure 86.

[7] This is not only applicable for the rules or rules conditions but for all the elements on the hierarchical tree of the Model.

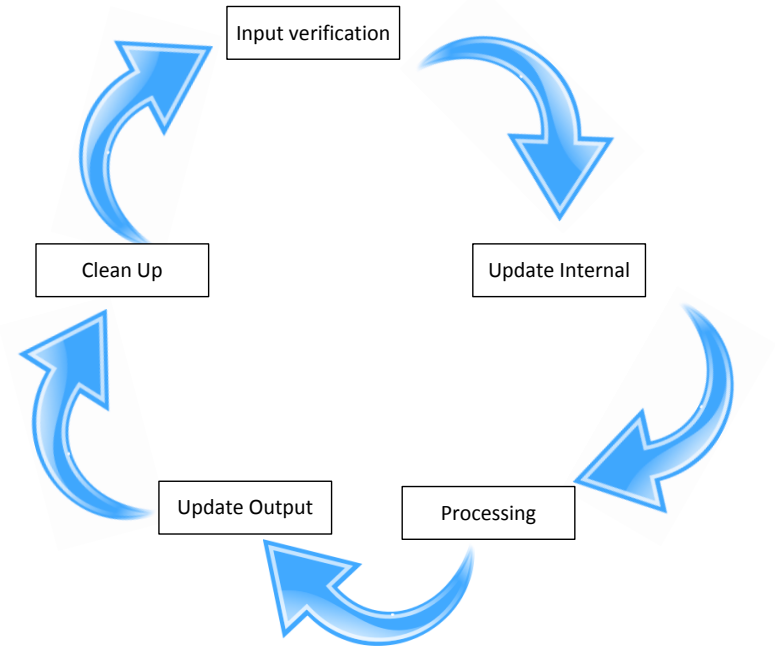

Figure 86: Diagram of the cycle on ERTMSFormalSpecs

Figure 87 presents the properties of a rule. In addition to the common model element properties a rule's priority is provided in its properties.

- **Rule priority:** This indicates the part of the activation cycle in which the rule can be activated. The priorities are the following.
  - **Input verification**
  - **Update internal variables**
  - **Processing**
  - **Update output variables**
  - **Clean Up**

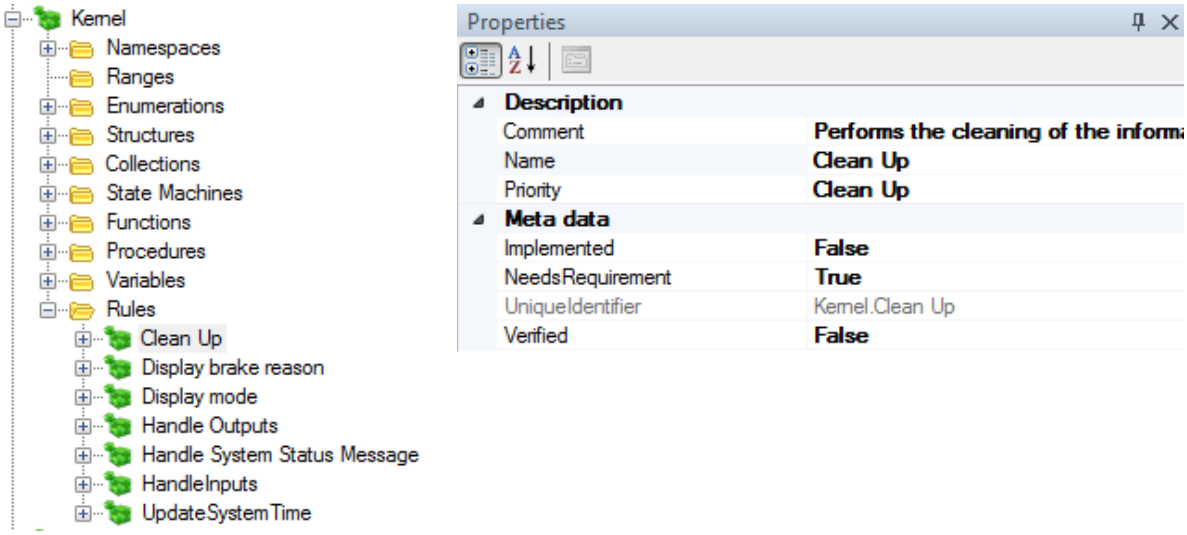

Figure 87: Rule properties

Figure 88 is an example of a rule. This rule deletes the present messages to the JRU at the end of each cycle; which means the rule is executed in the clean-up part of the cycle.

Figure 88: Rule representation

5.3.2.5.1 Specific case: a rule declared in a state

When a rule is declared in a state of a **State Machine**, it is only triggered when its pre-conditions are satisfied (as usual) but also when the current state of the **State Machine** corresponds to the state in which the rule is declared. Figure 90 displays the sub-rule related to the state StartOfMission.A40, for instance.

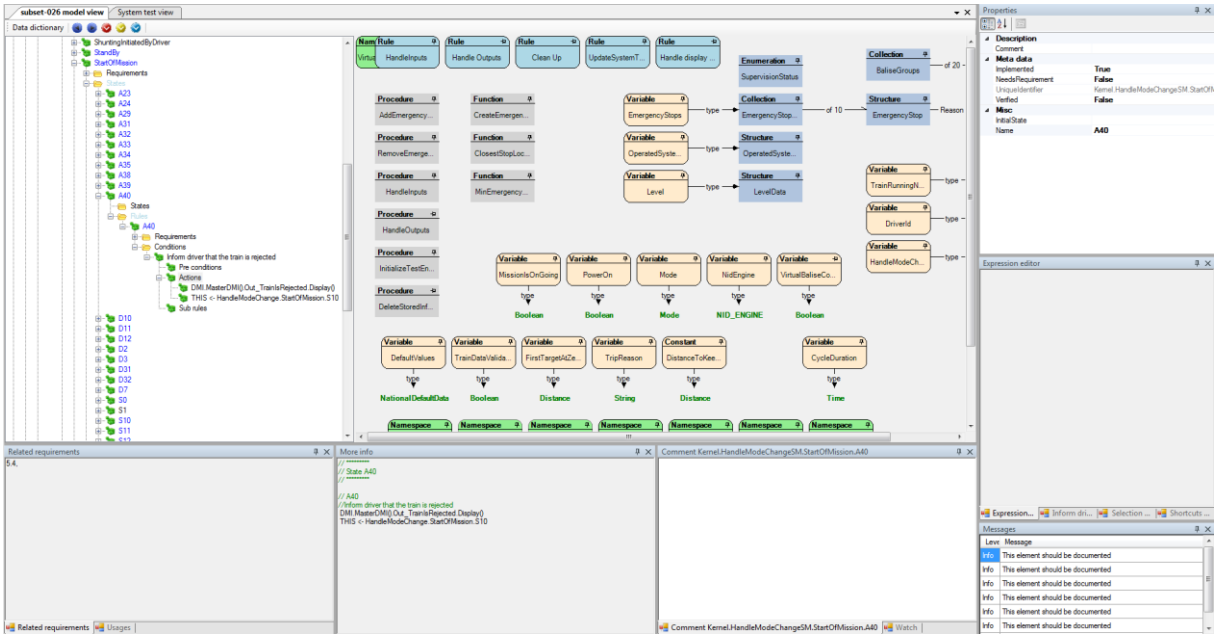

Figure 89: General view of a rule in a state of a State Machine, Start of Mission state A40

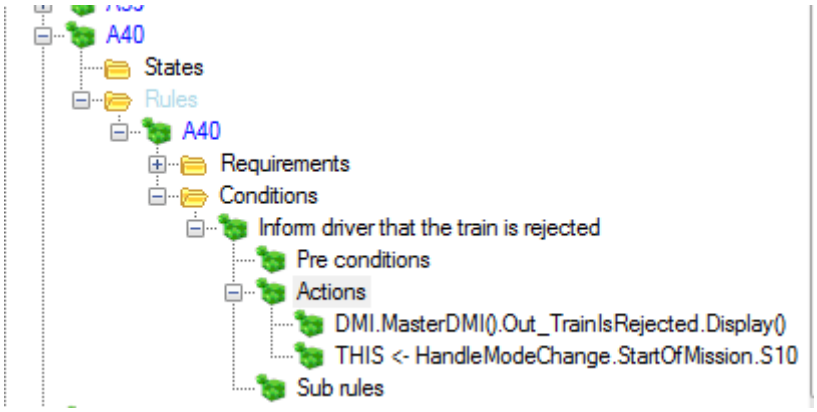

Figure 90: Detailed view of a rule in a state of a State Machine, Start of Mission state A40

The rule A40 is triggered only when its pre-conditions, if any, are satisfied (none in this case, so always true) and the current state of the state machine is A40 or a sub-state of A40. When that rule is activated, it executes two following actions:

- The State Machine **HandleModeChange** changes its state to S10.
- The procedure **DisplayTrainIsRejected** of the master DMI is called.

Please note that this also corresponds to a transition in a state diagram, and it is represented as such in the corresponding state diagram.

5.4 Selection history view

EFSW provides an easy way to navigate to one of the last accessed elements of the model. This is the Selection history. EFSW selections history is composed by a list with the name and the type of last visited elements of the model.

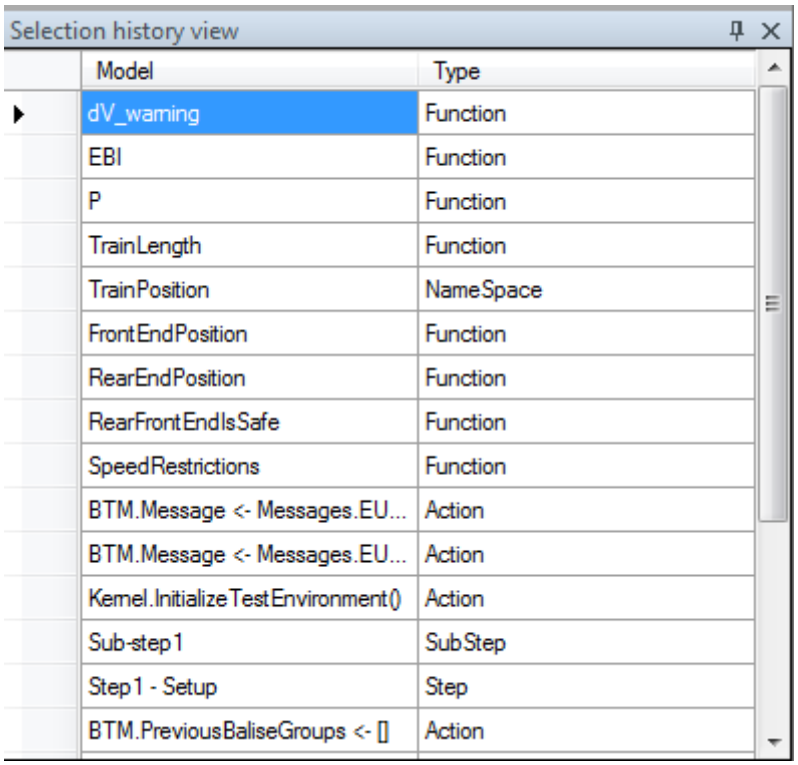

Figure 91: Selection history view

The way to navigate to one of the listed elements is by double-clicking on its name or on its base type.

5.5 Shortcuts view

EFSW provides a way to quickly access some model elements without searching them in the model hierarchical tree. The shortcuts are opened after opening a data dictionary and they are located on the middle left side of EFSW main window in the shortcuts tab; Figure 92 depicts the shortcut view window. If it is closed, it can be re-opened by selecting View/Show tools/Show shortcuts view in the menu or by using "Crtl+E".

To add an element of the model to the shortcuts use the **Drag&Drop** feature; shortcuts can then be assembled in different folders of the shortcut view. The shortcuts can be created, deleted and renamed using the contextual menu actions, accessed by right-clicking on the desired element.

To select the model element corresponding to a shortcut, just double-click on it.

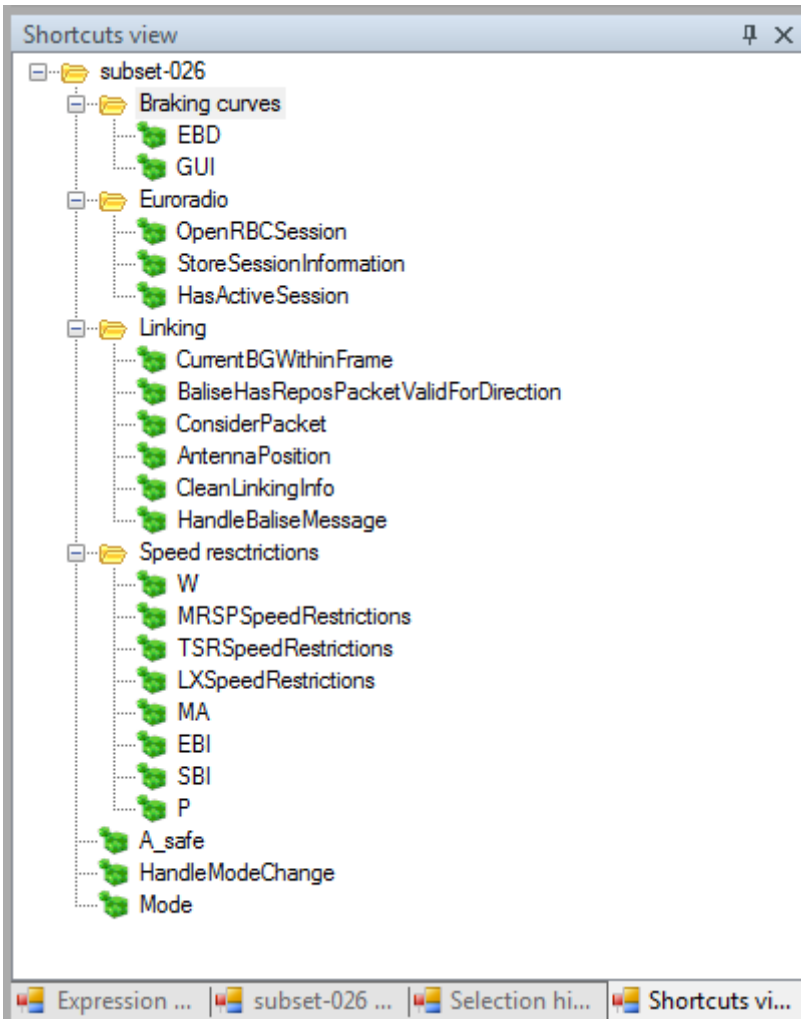

Figure 92: Shortcuts view

5.6 Tools related to the data dictionary view

Figure 93 shows all the actions related with the EFSM which are described on the following sub-sections.

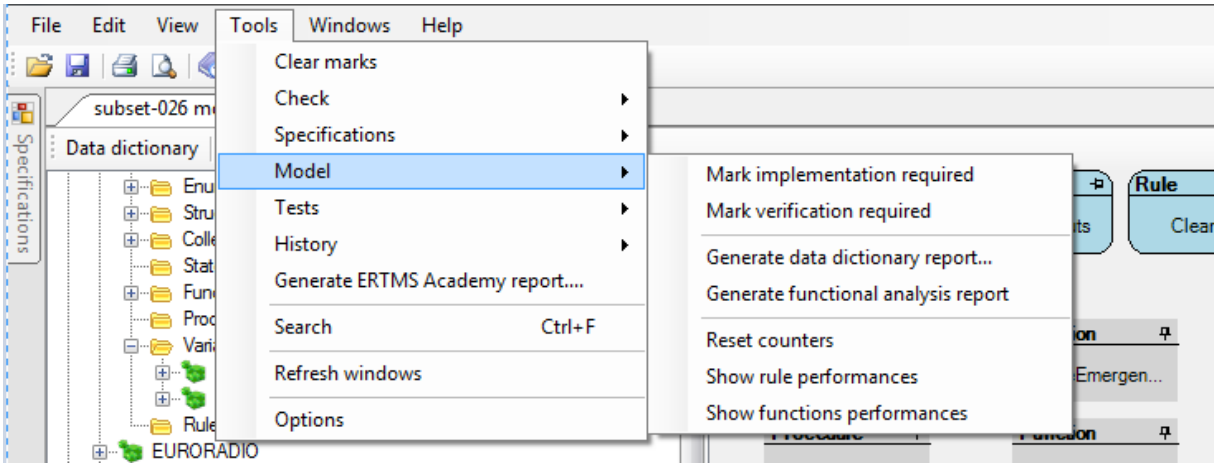

Figure 93: Tools related with the ERTMSFormalSpecs Model

5.6.1 Search for elements requiring an implementation

The Mark implementation required action puts an info message on all the model elements that have not been marked as implemented. Section 3.5.2 describes the colours legend associated for the messages. Figure 94 shows the message attached to the model element.

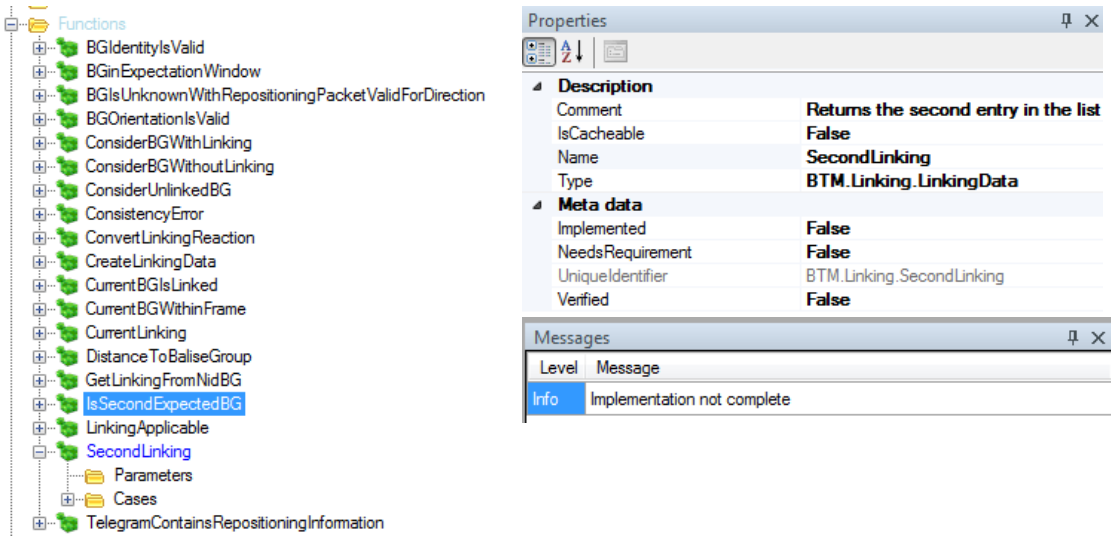

Figure 94: Representation of elements requiring implementation.

5.6.2 Search for elements requiring a verification

The Mark verification required action displays all the model elements whose implementation has not yet been verified. Elements are marked following the same rule as the one presented in Section 5.6.1 and following the colour legend of Section 3.5.2. See Figure 95.

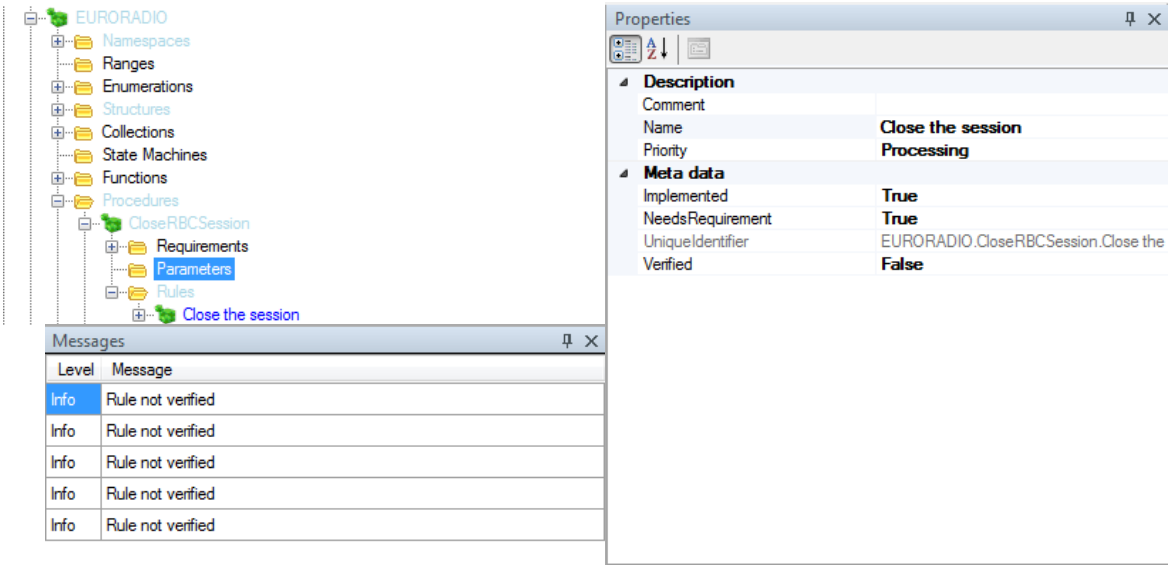

Figure 95: Representation of the elements requiring verification.

5.6.3 Show rule performance

The performance of the rules on EFSW can be measured taking into account the following traces:

- ***ExecutionTime*:** the total time used to execute the rule.
- ***ExecutionCount*:** amount of times the rule has been executed.
- ***Average*:** relationship between the ExecutionTime and the ExecutionCount.

RuleName	ExecutionTime	ExecutionCount	Average
Kernel.Handle Outputs	264	4	66
Kernel.HandleOutputs.Send JRU messages	233	4	58
JRU.JRU.HandleMessagesOut.SendMessage	202	4	50
Kernel.InitializeTestEnvironment.Train data	94	1	94
Kernel.TrainData.Initialize.TrainData	94	1	94
Kernel.TrainData.InitializeTrainData.Train data	94	14	6
JRU.JRU.InitializeMessage.Send DMI Symbol Status message	62	4	15
JRU.JRU.HandleMessagesOut.UpdateVariable	31	84	0
DMI.InputInformation.CleanUp	16	320	0
EURORADIO.PositionReport.Handle second RBC/RBC handover position report	16	0	0
JRU.JRU.InitializeMessage.Send Magnetic Shoe Brake Status message	16	1	16
DMI.OUT.MRSP.Output when mandatory	16	0	0
DMI.DMIStruct.UpdateOUTVariables.Update local time	15	4	3
Kernel.HandleOutputs.Update the DMI	15	4	3
JRU.JRU.InitializeMessage.Send Additional Brake Status message	15	1	15
Kernel.InitializeTestEnvironment.Train position	15	5	3
JRU.JRU.InitializeMessage.Send Additional Data message	15	4	3
DMI.OUT.RBCContactInformation.Display.Updates the request status	0	0	0
DMI.OUT.TrainRunningNumber.TrackMode.Updates the status of the request according to the mode	0	0	0
DMI.IN.ErtmsEtcsLevelEntryRequest.InitiateRequest.InitiateRequest	0	0	0
DMI.IN.DriverIdEntryRequest.InitiateRequest.InitiateRequest	0	1	0

Figure 96: Representation of the different rules performance

5.6.4 Show functions performance

The performance of the functions on EFSW can be measured taking into account the following traces:

- ***ExecutionTime*:** the total time needed to execute a function.
- ***ExecutionCount*:** amount of times a rule has been executed.
- ***Average*:** relationship between the ExecutionTime and the ExecutionCount.

FunctionName	ExecutionTime	ExecutionCount	Average
JRU.CreateHeader	62	34	1
JRU.CreateDMISymbolStatusChangeMessage	62	9	6
Kernel.DateAndTime.Now	61	242	0
JRU.DriverActed	32	4	8
JRU.CreateAdditionalDataMessage	31	8	3
JRU.CreateMagneticShoeBrakeStatusJruMessage	16	1	16
JRU.LevelChangedToNtc	16	5	3
DMI.OUT.MRSPMandatory	16	8	2
JRU.SendAdditionalDataMessage	16	4	4
JRU.SendNewMessage	16	4	4
JRU.DMISymbolsConvertors.LE04Activated	16	9	1
EURORADIO.OrderToTerminateHandingOverRBCIsReceived	16	4	4
JRU.LevelChanged	16	13	1
DMI.MasterDMI	16	1225	0
JRU.DMISymbolsConvertors.LE11Activated	15	9	1
JRU.CreateAditionalBrakeStatusMessage	15	1	15
JRU.SendTrainDataMessage	15	4	3
JRU.TrainDataHaveChanged	15	6	2
Kernel.DateAndTime.LocalTime	15	4	3
JRU.SendDMIStatusMessage	15	5	3
Kernel.AcceptInformation.ModeRules.AcceptSRDistanceInformationFromLoop	0	0	0
Kernel.AcceptInformation.ModeRules.AcceptDefaultGradientForTSR	0	0	0

Figure 97: Representation of function performance

5.6.5 Reset counters

The Reset counters action sets all the counters related with the rules performance and functions performance. See Show rule performance (Section 5.6.3) and Show functions performance (Section 5.6.4) for further details.

5.7 Model validations

5.7.1 Check for dead model

The model validation engine can detect functions or procedures which are never used by the model. These are called dead functions or dead procedures. Detecting such procedures and functions can be performed using the contextual menu Tools/Check/Check for dead model or by using "*Ctrl+D*".

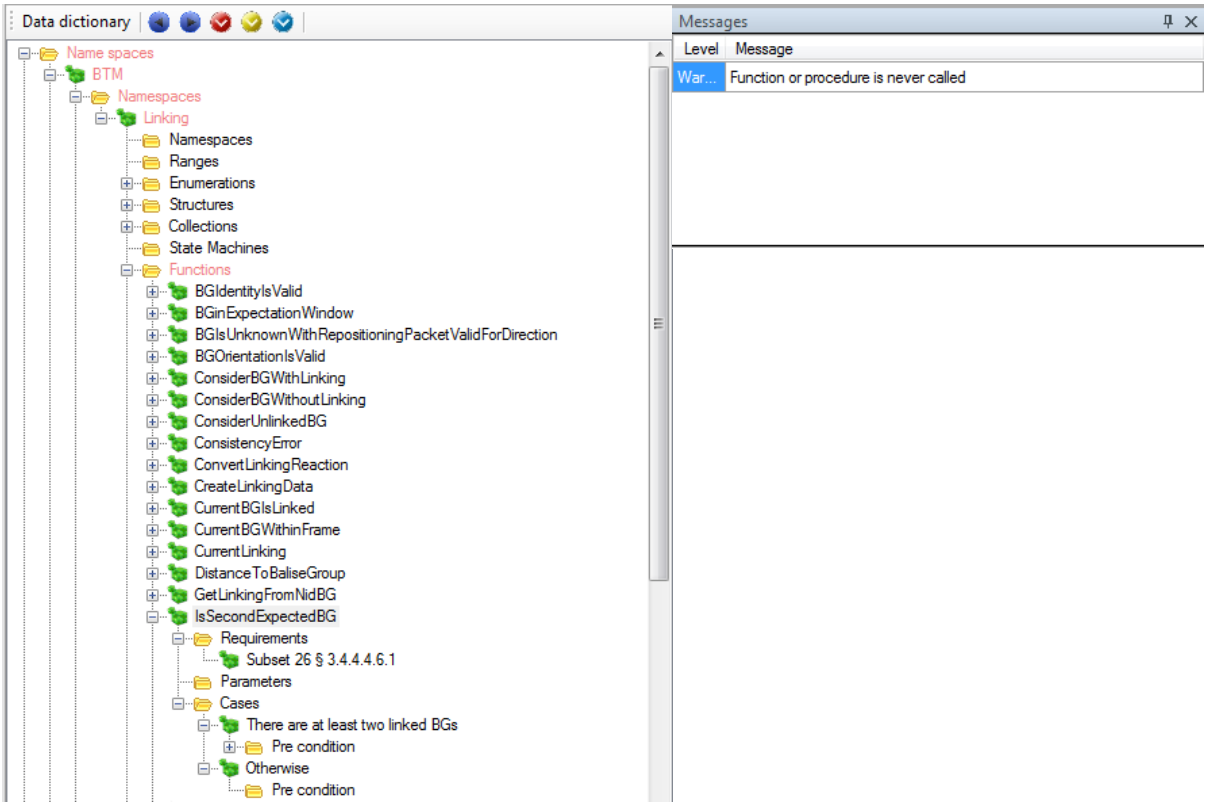

Figure 98: Validation done on dead functions or procedures

5.7.2 Check model

ERTMSFormalSpecs allows performing static tests on the model. During this process, the tool performs syntactical and semantical verifications on the model. The model validation engine can be activated by

selecting the contextual menu Tools/Check/Check model or by pressing "*Ctrl+R*". The following table describes in detail the verifications performed by EFSW:

Model	Description	Severity
Action	The variable which is modified by the action must exist in the system	Error
Action	The variable type must match the expression type	Error
All model elements	If the implementation flag of the model element is set to implemented, all its children must have the same value on this marker.	Warning
Assignation	When assigning a value to a variable of type range, the value must be within the range's minimum and maximum values	Error
Case	The expression type of the case must be the same as the function	Error
Collections	All the collections must define a maximal size.	Error
Enumeration	Enumeration value names must be unique	Error
Enumeration	The same numerical value cannot be assigned to two different values of an enumeration.	Error
Enumeration	The enumeration identifier must be valid	Error
Expectation	The variable which is checked by the expectation must exist in the system	Error
Expectation	The variable which is checked by an expectation must match the expression type. The resulting value must be a Boolean.	Error
Expression	IN operator should be used instead of == between expressions of type StateMachine. This is used to verify that expressions of the form `CurrentState == S1` is not used, since this does not take sub states of S1 into consideration and would result to false if `CurrentState` is `S1.subS1`	Warning
Expression	Expressions must be syntactically correct	Error
Expression	Expressions must be type correct	Error
Expression	Designators used in expressions must refer to a valid model element	Error
Expression	Designators used in expressions must have a valid type	Error
Expression	Collections cannot be compared with EMPTY, compare with [] instead.	Error
Expression	Abstract types cannot be instantiated.	Error

<table>
<tr><td>Field of a structure</td><td>The mode of a field of a structure must be at as restrictive or more than the mode of its enclosing field according to the following table
<table>
<tr><th>Enclosing field</th><th colspan="5">Field</th></tr>
<tr><th></th><th>Constant</th><th>Incoming</th><th>In/Out</th><th>Internal</th><th>Outgoing</th></tr>
<tr><td>Constant</td><td>✓</td><td></td><td></td><td></td><td></td></tr>
<tr><td>Incoming</td><td>✓</td><td>✓</td><td></td><td>✓</td><td></td></tr>
<tr><td>In/Out</td><td>✓</td><td>✓</td><td>✓</td><td>✓</td><td>✓</td></tr>
<tr><td>Internal</td><td>✓</td><td></td><td></td><td>✓</td><td></td></tr>
<tr><td>Outgoing</td><td>✓</td><td></td><td></td><td>✓</td><td>✓</td></tr>
</table></td><td>Warning</td></tr>
<tr><td>Frame</td><td>The Cycle time duration does not resolve to a type that is compatible with Time</td><td>Error</td></tr>
<tr><td>Function</td><td>This element should be documented.</td><td>Info</td></tr>
<tr><td>Function</td><td>All the functions must have a return type.</td><td>Error</td></tr>
<tr><td>Paragraphs</td><td>Paragraph state does not correspond to implementation status. This can occur in several cases
The paragraph has been modelled but is not applicable
The paragraph model state is N/A or is Not Implementable but the paragraph is applicable.</td><td>Warning</td></tr>
<tr><td>Paragraph</td><td>All paragraphs must be linked to, at least, one Scope.</td><td>Warning</td></tr>
<tr><td>Paragraphs</td><td>Paragraphs must have the same Scope as all their children.</td><td>Warning</td></tr>
<tr><td>Paragraphs</td><td>Two paragraphs cannot have the same identifier</td><td>Error</td></tr>
<tr><td>PreCondition</td><td>Operator == should not be used for state comparison</td><td>Warning</td></tr>
<tr><td>PreCondition</td><td>Operator != should not be used for state comparison</td><td>Warning</td></tr>
<tr><td>PreCondition</td><td>The variable on which the pre-condition is computed must exist in the system</td><td>Error</td></tr>
<tr><td>PreCondition</td><td>The variable type must match the Operand type, according to the operator semantics. The resulting value must be a Boolean.</td><td>Error</td></tr>
<tr><td>Procedure</td><td>This element should be documented.</td><td>Info</td></tr>
<tr><td>Range</td><td>Special values value cannot be duplicated</td><td>Error</td></tr>
<tr><td>Ranges</td><td>Special values names cannot be duplicated</td><td>Error</td></tr>
<tr><td>Ranges</td><td>Integer precision : Invalid min value for integer range : must be an integer</td><td>Error</td></tr>
<tr><td>Ranges</td><td>Integer precision : Invalid max value for integer range : must be an integer</td><td>Error</td></tr>
<tr><td>Ranges</td><td>Double precision : Invalid min value for integer range : must have a decimal part</td><td>Error</td></tr>
<tr><td>Ranges</td><td>Double precision : Invalid max value for integer range : must have a decimal part</td><td>Error</td></tr>
<tr><td>Ranges</td><td>Cannot parse min value for range</td><td>Error</td></tr>
<tr><td>Ranges</td><td>Cannot parse max value for range</td><td>Error</td></tr>
<tr><td>Ranges</td><td>Default value's precision does not correspond to the type's precision.</td><td>Error</td></tr>
</table>

Requirement link	The link to a requirement must refer to a valid requirement	Error

Requirement related	A model element related to a requirement (type, variable, procedure, function, rule) should refer to at least one requirement. This is only true if the flag **NeedsRequirement** is set to true. Either set the flag **NeedsRequirement** to false or provide a link to a requirement.	Info
Requirement related	When the implementation of a requirement is completed, all model elements which implement that requirement must also be marked as “implementation completed”.	Warning
Rule	This element should be documented.	Info
Rule	An incoming variable cannot be modified by the EFS.	Error
Rule	An outgoing variable cannot be read by the EFS.	Error
Rule	If one of the pre-condition is in the form *Variable == ‘Request.Response’* there must be an action which sets that variable to *‘Request.Disabled’*. This ensures that all requests for which a response was received are disabled.	Error
Special values	The precision of a Special value does not correspond to the type’s precision.	Error
State machine	This element should be documented	Info
State machine	The initial state of a state machine is not empty and corresponds to a state of that state machine	Error
State machine	The name of states in a state machine are valid (e.g. do not contain space)	Error
Statement	Assignation of EMPTY cannot be performed on variables of type collection. Use [] instead.	Error
Structures	The referenced interfaces must point to a valid interface.	Error
Structures	All the elements inherited from interfaces must be implemented.	Error
Structure elements	The type of the elements in a structure must be the same as the type of the default value.	Error
Structure elements	Sub elements of a structure must have unique names	Error
Structure elements	The type of a structure element cannot be abstract (an interface or a collection of interfaces).	Error
Sub-sequence	Sub sequences should hold at least one test case	Warning
Sub-sequence	First test case of a subsequence should hold at least one step. Only for the first step of the first test case on the sub-sequence	Warning
Sub-sequence	First step of the first test case of a subsequence should be used to setup the system, and should hold 'Setup' or 'Initialize' in its name. Only for the first step of the first test case on the sub-sequence	Warning

Subset-076 Steps	Cannot find Balise messages for this step. Only available for translation rules where a key word has been found.	Warning
Subset-076 Steps	Cannot find Euroloop messages for this step. Only available for translation rules where a key word has been found.	Warning
Subset-076 Steps	Cannot find RBC message for this step. Only available for translation rules where a key word has been found.	Warning
Translation	Translations must contain action and/or expectations, or must be linked to a requirement.	Warning
Translation	Source text of the translations rules must be unique	Error
Type	This element should be documented.	Info
Type	Types are uniquely identified	Error
Type	A type should define a valid default value	Error
Typed Element	The declaration of a typed element (variable, structure element, function) defined in the model must have a valid type	Error
Typed Element	Recursive types are not allowed	Error
Variable	The variable semantics cannot be empty. Fill the field comment defined for the variable	Info
Variable	When its type has no comment an Info message is displayed also on the variable.	Info
Variable	The type of a variable must be the same as the type of the default value.	Error
Variable	The type of a variable cannot be abstract (an interface or a collection of interfaces).	Error
Variable	The expression of the default value of a variable must be correct	Error

Table 3: Check verifications on EFS

After the validation is performed, each node of the data dictionary view is displayed according to the colouring rules in Table 2, Section 7.5.2.

Figure 99 shows an example of the result of the Check model action. In this case, there is an error in one of the actions of the procedure HandleBaliseMessage.

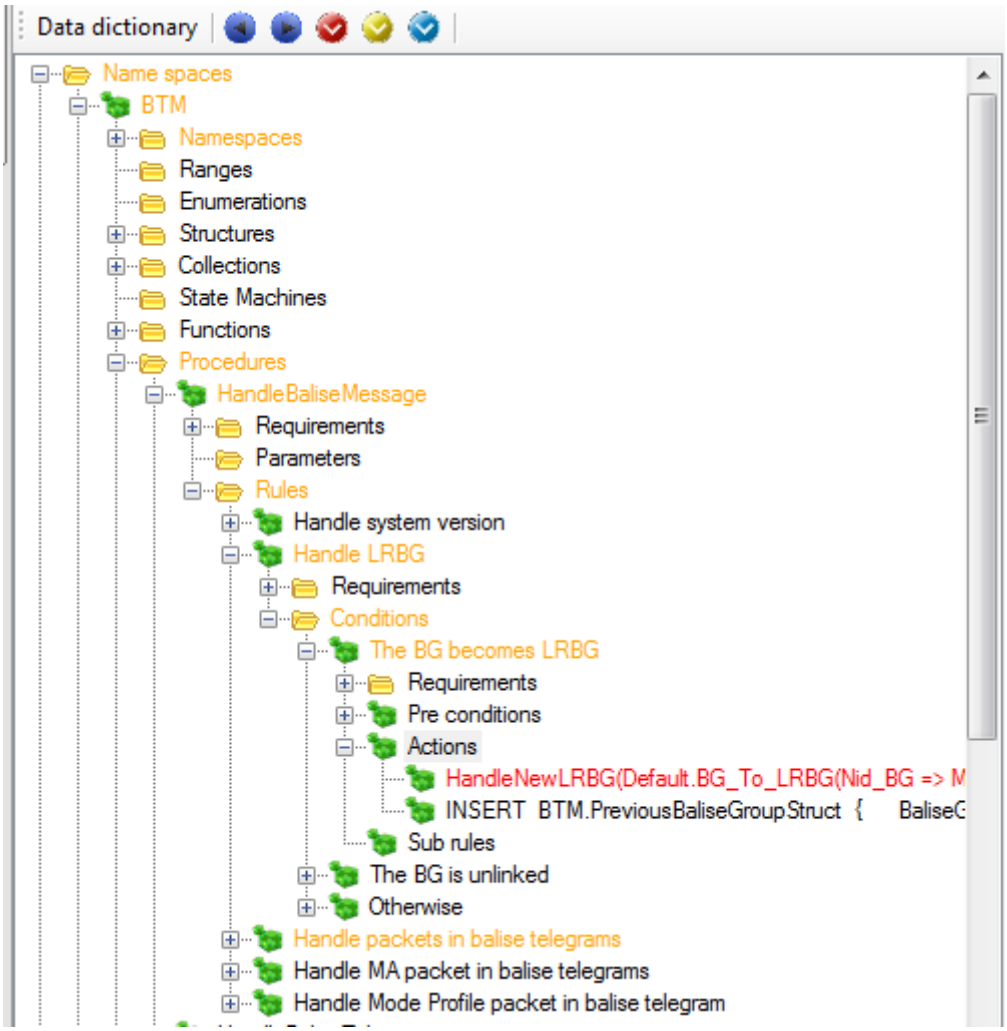

Figure 99: Example of model warnings/errors

5.8 Expressions

Expressions are used by EFSW to evaluate values for pre-conditions, actions (these two cases are described on Section 5.3.2.5) and expectations (see Section 6.3). They follow the BNF grammar described in EFSW Technical Guide (see [1]).

6 ERTMSFormalSpecs Test browser and execution environment

EFSW provides an environment to define and run several kinds of tests:

- **UNISIG Subset-076 test sequences**
- **Additional functional tests targeted at EFS model coverage**
- **User-defined tests**

Tests are used to ensure that the model corresponds to the expected behavior. In some sense, tests provide an alternate model of the system. They set the system in a specific situation and check the system behavior. Tests which are defined on EFSW follow the structure presented on Subset-076: they are grouped in frames, each frame is split in one or more sub-sequence, each sub-sequence is divided in test cases and each test case is further split in steps. Last, and this does not correspond to Subset-076 structure, steps are split in sub-steps. This allows seamless translation of a Subset-076 step which requires several atomic actions to be performed.

6.1 Opening the test browser

The test browser allows editing tests and executing them against the model. The test browser is automatically opened when launching EFSW and in case it is closed by the user it can be re-opened using the View\Show tests view menu as depicted by Figure 100.

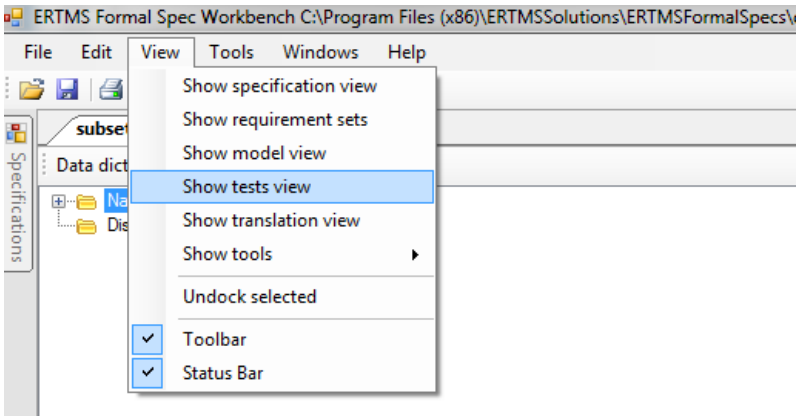

Figure 100: Opening the test view

6.2 Overview

The test browser is composed by several parts. See Figure 101.

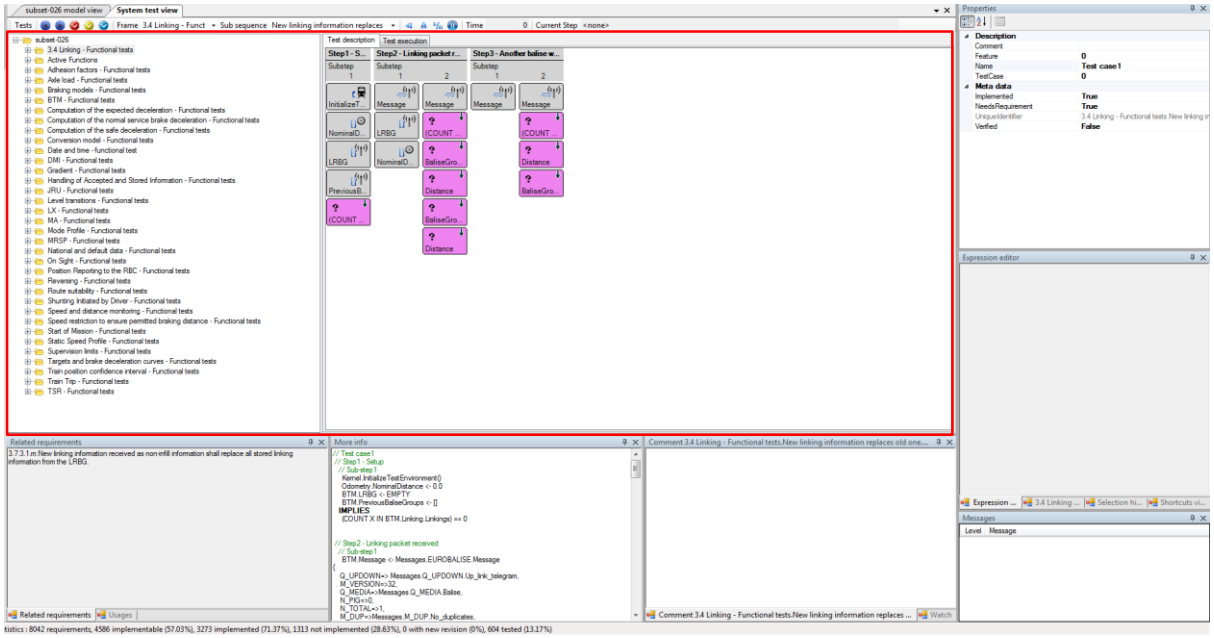

Figure 101: General overview of the system test view

The left tree view represents the tests structure and allows selecting a test element. As stated before, EFS Tests are structured according to the structure of Subset-076 tests (frames, sub sequences, test cases, steps and sub-steps). This is shown in Figure 102.

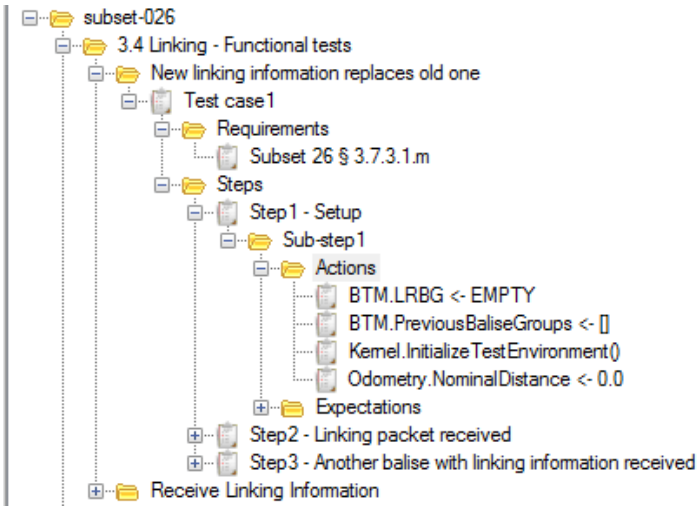

Figure 102: Test tree view

The **property**, **expression editor**, **history**, **selection history view**, **shortcuts** and **description** windows have been described previously and keep the same functions and characteristics as described in Section 5.2.

6.3 Test structure

EFSW graphically displays the test structure using the time lines. Figure 103 shows the two kinds of time lines: **test description** and **test execution**.

- **Test description:** graphical description of the test that should be executed in terms of actions and expectations.
- **Test execution:** the result of the execution of a test.

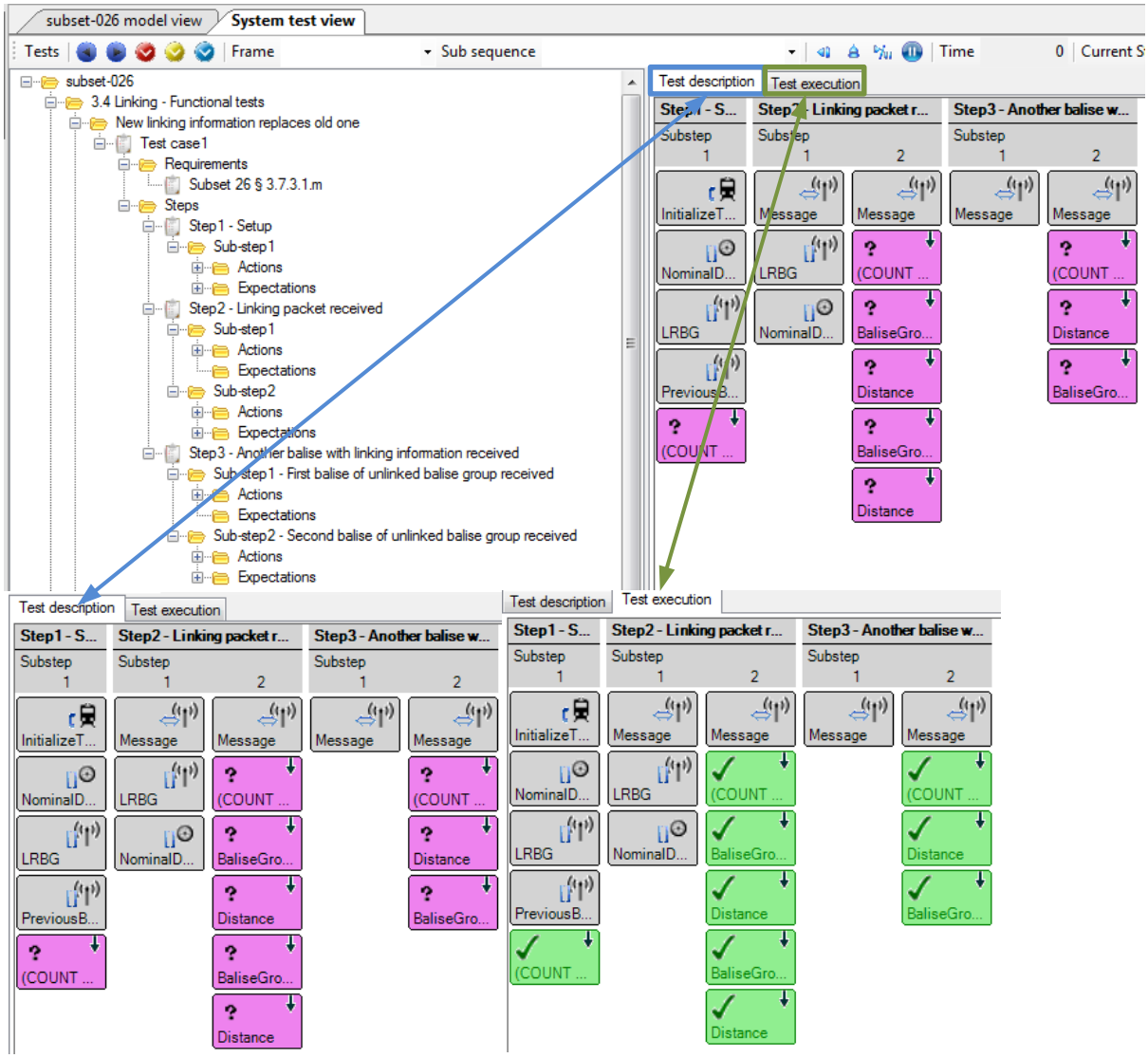

Figure 103: General overview of the EFST main window: test description and execution tabs

In both cases, time lines show the following elements:

- **Steps:** used to group sub-steps. Usually, the first step of the first test case of a sub sequence is called *Step 1 – Initialize test environment,* and is used to initialize the system (typically using the procedure call *InitializeTestEnvironment()*). Steps are depicted by the topmost box in the time line.
- **Sub-steps:** used to group the test actions and expectations. Sub-steps are depicted by the grey boxes just below the step box.
- **Actions:** modifications performed on the state of the system. Actions are depicted by grey boxes with rounded corners, and are placed below the sub-step boxes.
- **Expectations:** conditions that need to be satisfied by the system after the actions have been applied, for the test to be considered successful. These conditions can be either punctual or last for a given duration in time. They can also block progression of the scenario until they are satisfied, or simply need be satisfied once in the given timespan. Expectations are depicted by a non-grey box with rounded corners, below the sub-step box and any actions present in the sub-step. Their colour depends on the situation (see below).

For further details about the test elements and the test environment see Section 6.4.

Figure 104 displays a typical test description. **Actions** are represented by grey boxes with rounded corners, whereas **expectations**, in this case, are represented by purple boxes with rounded corners and a question mark (**?**).

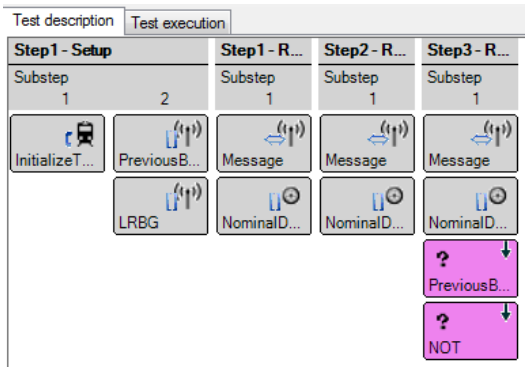

Figure 104: Time line representation on EFST

The result of the same test after execution is depicted in Figure 105: **actions** are still grey boxes with rounded corners, but **successful expectations** are now green with a ✓ sign.

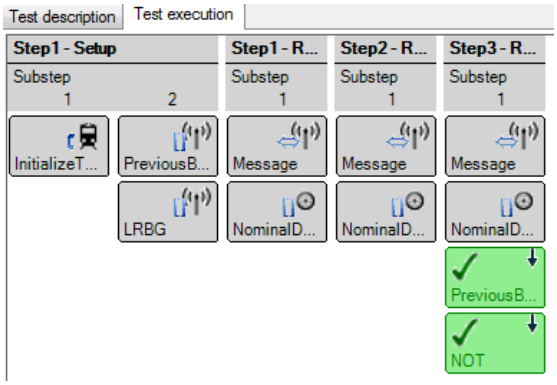

Figure 105: Test execution view

If an **expectation** was not successfully satisfied, it is displayed as a red box with a red cross (✖).

6.4 Test description

The test description tab presents a graphical display used to describe and edit a test. The following sub-sections describe the operations available for tests. These actions are performed directly on the test hierarchical tree by right-clicking on the desired element and selecting the corresponding menu entry.

6.4.1 Add a test frame

Test frames are the highest component on the hierarchical tree after the root. Test frames are used to group different sub-sequences, a test frame contains at least one sub-sequence.

To add a new test frame, select the subset where the test frame should be added, right click to display contextual menu and select Add frame. See Figure 106.

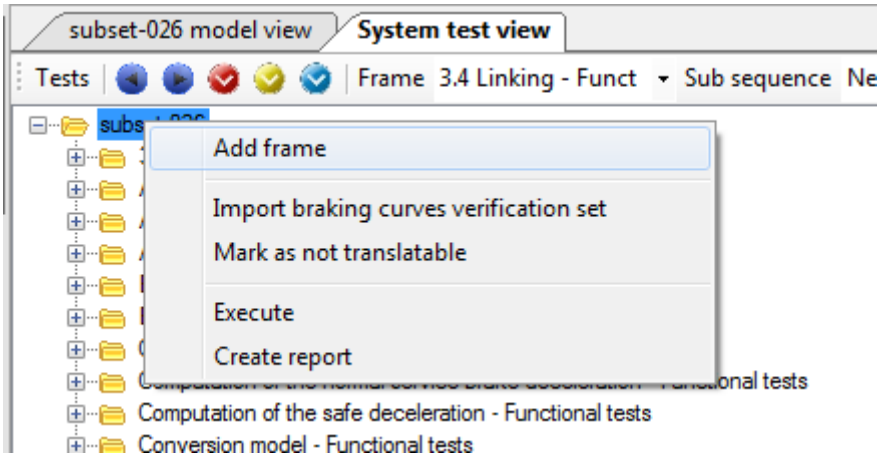

Figure 106: Add a new test frame

The specific properties related with a test frame are:

- **Cycle duration:** the time required to compute a cycle on EFSW (see Section 5.3.2.5).

6.4.2 Add a sub-sequence

Sub-sequences are used to sequence several test cases within a test. Sub-sequences are independent of each other; hence, they should always begin with a complete system setup. Right-clicking on a test frame and selecting **Add sub-sequence** in the contextual menu adds a new sub-sequence to the selected test frame.

Specific properties for all the sub sequences are

- **Completed:** when this property is set to true, it indicates that the current sub-sequence has been fully implemented. This flag is used by the continuous integration process (execution of all tests) to determine when a regression in the model occurs: only completed sub sequences are checked by the continuous integration process.

Sub sequences can be either created manually or imported from Subset-076 (see Section 6.6.3). When the sub-sequence is imported from a Subset-076 database, the Subset-076 Description part of the properties presents the information imported from Subset-076 test sequences.

- **D_LRBG:** the distance to the last relevant balise group.
- **Level:** the information related to the current ERTMS/ETCS level of application.
- **Mode:** the information related to the current EVC mode.
- **NID_LRBG:** the identifier of the last relevant balise group.
- **Q_DIRLRBG:** the orientation of the train relative to that of the LRBG.
- **Q_DIRTRAIN:** the direction of train movement relative to the LRBG orientation.
- **Q_DLRBG:** the side of the LRBG the estimated front end of the train is.
- **RBC_ID:** RBC unique identifier.
- **RBC_Phone:** telephone number of the RBC**.**

6.4.3 Add new test case

The test cases consist of a sequence of steps to follow during the execution of the test. Right-clicking on a sub-sequence and selecting **Add test case** in the contextual menu adds a new test case to the selected sub-sequence.

Requirements can be linked to test cases by dragging that requirement and dropping it on the related test case. To remove a requirement from a test case, select the requirement and right-click on it, select **Delete** in the contextual menu.

Properties specific to a test case are

- **Feature:** linked to Subset-076.
- **Test case:** linked to Subset-076. The feature and test case identify the test case in Subset-076.

6.4.4 Add new steps, sub-steps, actions and expectation

Steps, sub-steps, actions and expectations definitions can be found on the beginning of this section (Section 6.3).To add a new step, right-click on the empty space of the test description tab. Figure 107, illustrates how to add a new step on the current test:

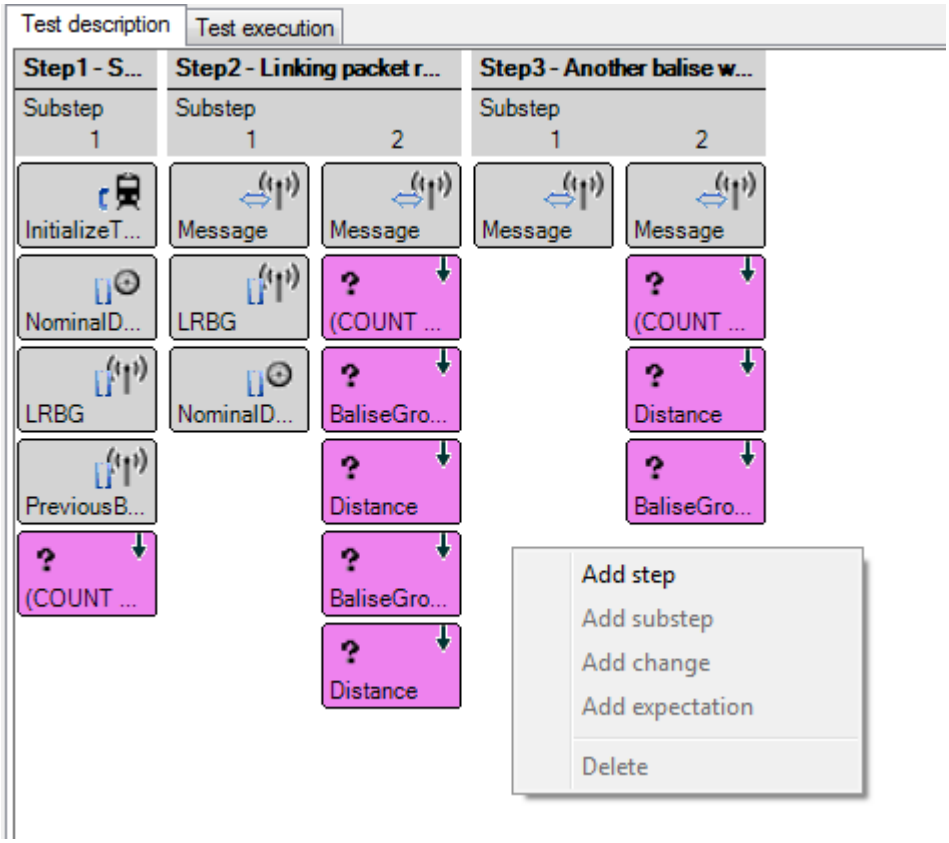

Figure 107: Adding a new step using the contextual menu

Specific properties of steps are:

- **Completed:** when set to True, indicates that the current step has been fully implemented**.**

Additionally, steps may also contain information related to Subset-076. These properties are displayed under the tag **Subset76**. They are:

- **Distance:** the distance at which the step occurs.
- **Input Output:** the mode of the variable.
- **Interface:** the interface to which must be applied.
- **Order:** the order of the sub-step on the test sub-sequence.
- **Test Level In:** the EVC level at the beginning of the test step. For further information about the levels see Section 2.6 of [2].
- **Test Level Out:** the EVC level after executing the selected step.
- **Test Mode In:** the EVC mode at the beginning of the step. For further details about the Modes see Section 4.4 of [2].
- **Test Mode Out:** the mode after the execution of the test step.
- **Translated:** if the current step has been already translated using the tool described on Section 8.3.
- **Need Translation:** if the selected sub-step needs to be translated as described in Section 8.3.
- **User Comment:** this comment comes directly from Subset-076 database, and cannot be modified using EFS.

To add a new sub-step to an existing step, right-click on the step, and select the **Add sub-step** in the contextual menu.

Properties specific to a sub-step are

- **Skip engine:** indicates that the engine of EFS (i.e. the model rules) should not be activated after executing the actions in this step. This flag is used to automatically translate steps in Subset-076 where, for instance, a new expectation is added in the test, but the system should not run.

To add an action or an expectation in a sub-step, select the sub-step then right-click to select the element to add in the contextual menu.

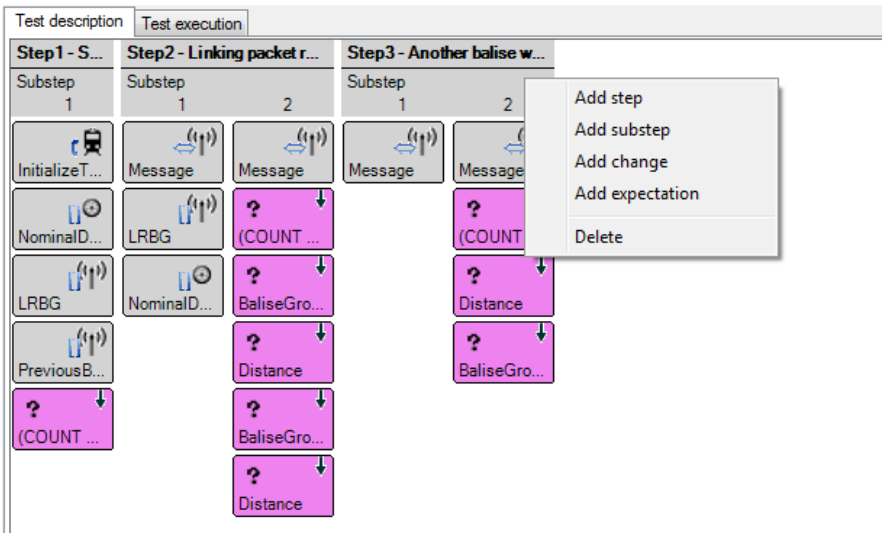

Figure 108: Adding a new test element on an already existing item

Figure 109 shows the properties related with an action:

- **Expression:** the operations to be performed when the action is executed. This expression can also be edited in the Expression window.

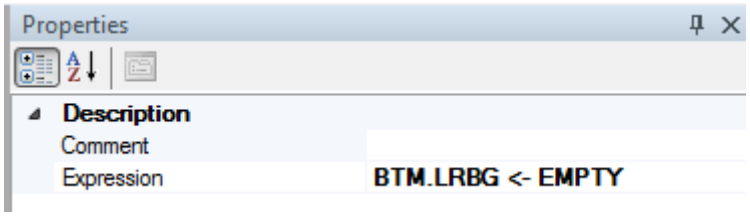

Figure 109: Action properties

Expectations hold the following information:

- **Blocking:** prevents the activation of the following sub-step as long as this expectation is not reached.
- **Condition:** if the condition is not satisfied, the expectation is not taken into account during the verification phase and the expectation is marked as reached in the test execution time line.
- **Cycle phase:** the phase of the system's processing cycle at which the expectation is verified. See Section 5.3.2.5 for more information about processing cycles.
- **Expression:** the element or expression to be checked in order to determine whether the expectation has been satisfied or has failed.
- **Kind:** whether the evaluation of the expectation is punctual or continuously checked:
  - **Instantaneous:** the expectation should be satisfied at least once before the time specified, otherwise, it is considered as Failed.
  - **Continuous:** the expectation should always be satisfied during the duration specified, otherwise it is considered as Failed.
- **DeadLine:** the duration associated to the expectation. The semantics of this duration depends on the expectation kind (see above).

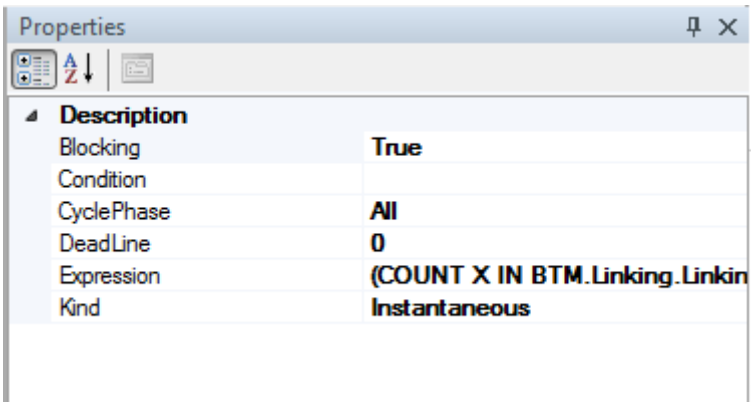

Figure 110: Expectation properties

6.4.5 Remove steps, sub-steps, actions and expectation

One can delete elements from the test using the contextual menu. Select the corresponding element, right click to open the contextual menu, and select **Delete**, as shown in Figure 111.

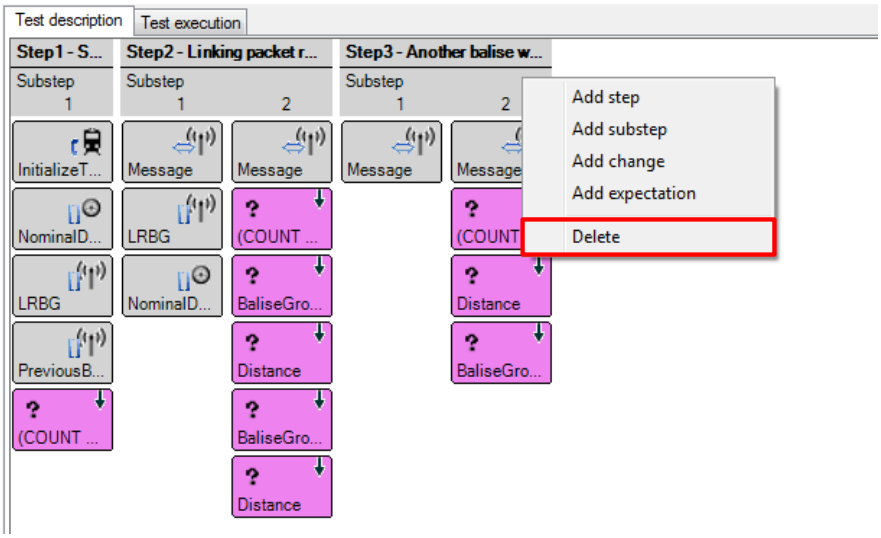

Figure 111: Delete option of the contextual menu

6.4.6 Remove test frames or sub-sequences

Deleting test frames or sub-sequence cannot be performed on the description time line. This is performed using the hierarchical tree: right-clicking on the selected test frame or sub-sequence to remove and select **Delete** in the contextual menu. Note that test cases, steps, sub-steps, actions or expectations can also be deleted using the hierarchical tree.

6.5 Test execution

As stated previously, the test browser is used to execute tests to ensure that the model behaviour matches the specifications and to prevent regression. This is expressed using expectations which are

verified, according to the triggers (actions) that the test defines. Test execution and results are displayed using time lines.

To execute a sub-sequence step by step, first select the test frame, as shown in Figure 112.

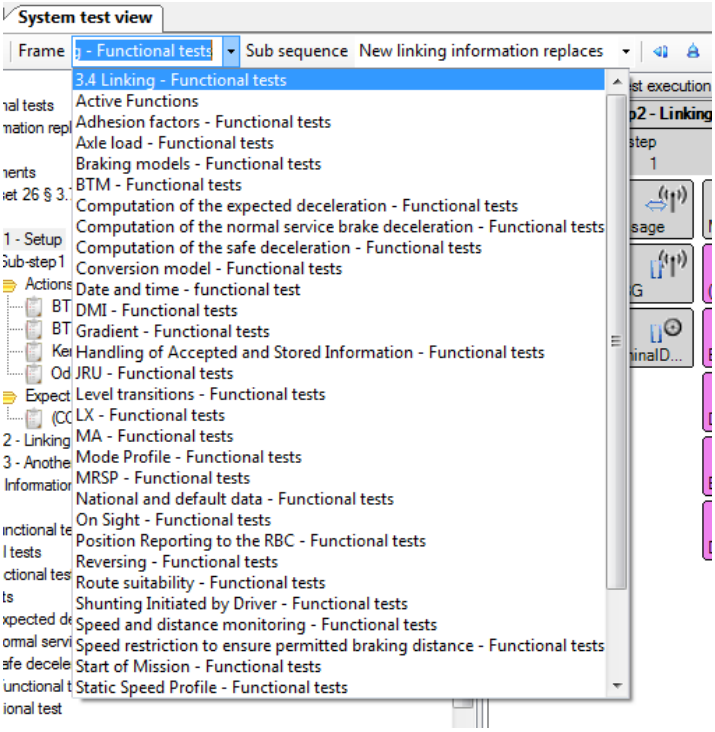

Figure 112: Test frame selection

Once the frame has been selected, select the sub-sequence, as shown in Figure 113.

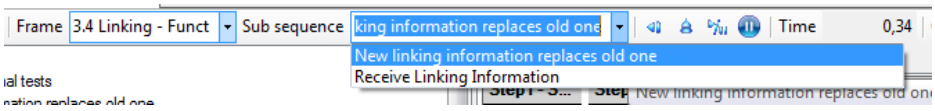

Figure 113: Test frame selection

Finally, to animate the system use the animation buttons depicted on the Figure 114.

- **Step once ( ):** execute the next step of the current test. Each time this button is pushed a new sub-step is displayed on the execution time line, and all windows are updated.
- **Step back ( ):** navigate one step back on the test execution and update all windows.
- **Restart ( ):** empty the time line and restart the current test.
- **Pause ( ):** suspend model execution when an external visualizer interacts with EFS, such as the ERTMS Solutions' DMI or the ERTMS Solutions' TrackMap display.

Figure 114: Buttons controlling test actions

6.5.1 Watch window

During the execution of a test, the values of a model element can be seen on the watch window. See Figure 115.

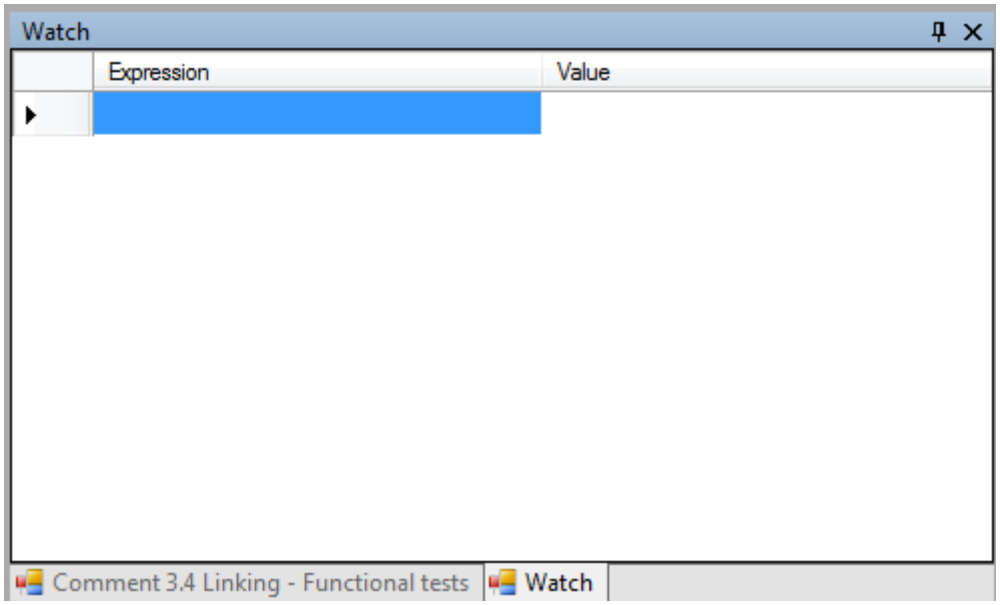

Figure 115: Watch window

Double-clicking on the blue highlighted rectangle displays the expression editor. Edit the expression to select the component of the model or the expression to be evaluated and close the window.

One can also **Drag&Drop** a variable from the hierarchical tree displayed in the model view to add it in the watch window.

6.5.2 Test actions and expectations

Several colours are used to display results of a test execution, as shown in Figure 116.

- **Variable updates due to actions** defined in the test are displayed in grey. Left clicking navigates to the corresponding action in the test browser. Double click displays the explanation view, which describes all the computations performed by the engine to alter the variable's value.
- **Unfulfilled expectations** are displayed in violet. Left clicking navigates to the corresponding expectation in the test browser.
- **Fulfilled expectations** are displayed in green. As for unfulfilled expectations, left clicking navigates to the corresponding expectation in the test browser. Double click displays the explanation view, which describes all the computations performed by the engine to compute the expectation's value.
- **Failed expectations** are displayed in red. Again, left clicking navigates to the corresponding expectation in the test browser. Double click displays the explanation view, which describes all the computations which lead to this expectation failure.

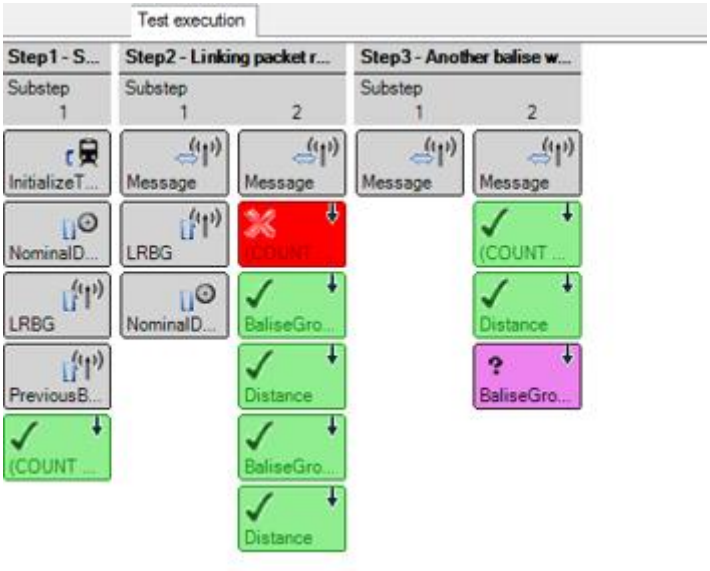

Figure 116: Timeline components

6.5.3 Activated rules and variable updates

The execution time line can also display the rules that have been activated, and the variable updates performed during the model execution. By default, to limit their number, this information is filtered out, but the filter can be configured thanks to the Configure filter action available in the timeline's contextual menu, as depicted below.

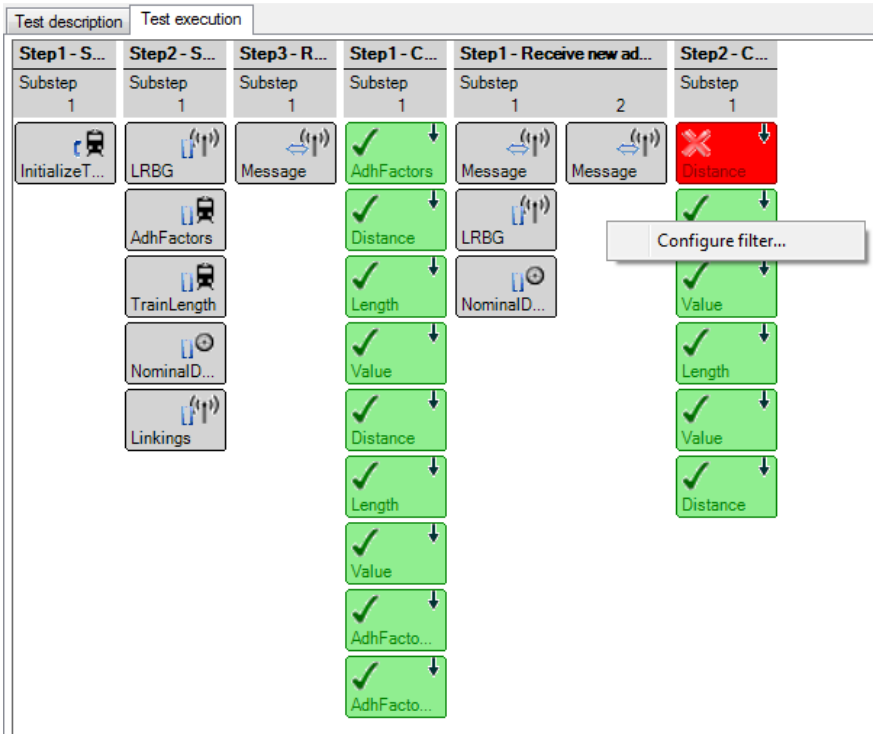

Figure 117: Contextual menu to configure the time line filter

This opens the filtering dialog, as shown in Figure 118 which provides the following options

- **Show/hide rule activations:** display or hide the rules of the model which were activated during the execution cycle on the execution time line (e.g. the rules whose preconditions evaluated to true). Activated rules are displayed as blue boxes with rounded corners.
- **Show/hide variable updates:** display or hide the variable updates during the execution cycle for the selected variables. They are represented as pink/salmon boxes with rounded corners.
- **Show/hide expectations:** display or hide the expectation boxes in the execution time line (see Section 6.3).
- **Namespace and Variables filtering:** only the selected namespaces or variables are displayed in the execution time line.
- **Regular expression filtering:** This is an alternate mean to identify the events that shall be displayed in the time line. When defined, elements whose message is matched by the regular expression are also displayed in the time line.

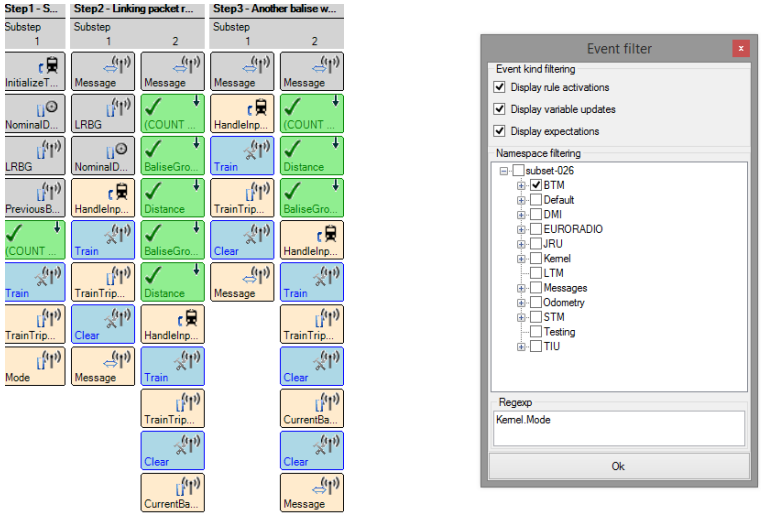

Figure 118: Filter options and corresponding displayed events

For instance, the filter displayed in Figure 118 shows the filters to select

- Events that occurred in namespace BTM
- Events that match the regular expression "Kernel.Mode"

And the corresponding selected events.

Table 4 summarises the icons used in the time-line and their meaning.

Variables and expressions	
	The affected variable's mode is IN
	The affected variable's mode is OUT
	The affected variable's mode is IN OUT
	The affected variable's mode is INTERNAL
	Procedure call
Namespaces	
	Related to namespace BTM or EURORADIO
	Related to namespace JRU
	Related to namespace Odometry
	Related to namespace DMI
	Related to namespace Kernel
Expectations	
	Continuous expectation
	Instantaneous expectation
	Failed expectation
	Successful expectation
	Expectation whose state has not yet been determined.
Rule activation	
	Rule activation marker

Table 4: Icons used on the execution time line

6.5.4 Expectation failure

Figure 119 shows an example of a failed expectation in the execution of a test. When an expectation fails, the corresponding node of the hierarchical tree in the test view is coloured using the scheme presented in 3.5.2 : a failed expectation corresponds to an error.

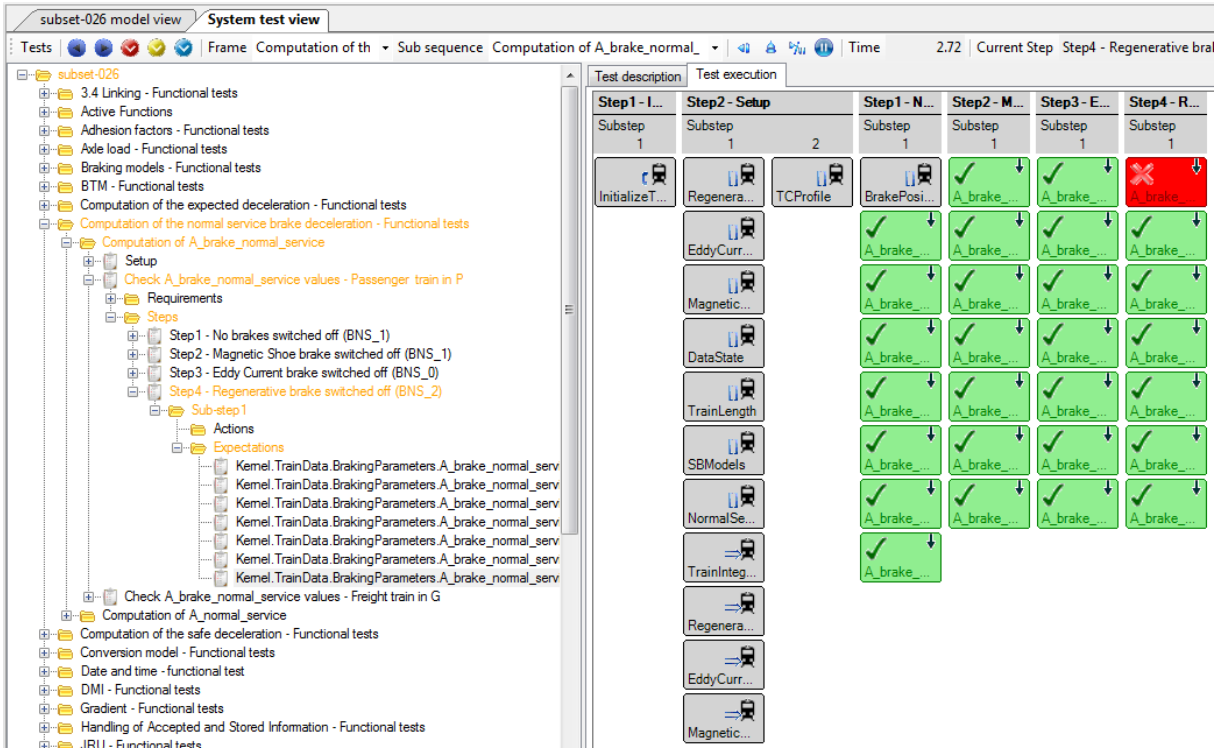

Figure 119: Failed expectation

The relevant message is displayed in the Messages window, as displayed in Figure 120.

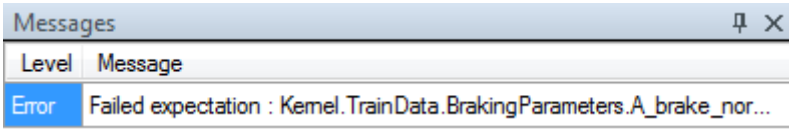

Figure 120: Error message related with the failed expectation

6.5.5 Explain view

During the execution of a test, double-clicking on any element (except steps and sub-steps) in the execution time line shows its explanation window.

The explain view is divided in two sections:

- **Actions tree:** all the actions taken.
- **Onboard and trackside messages:** the messages sent from/to the train are explained.

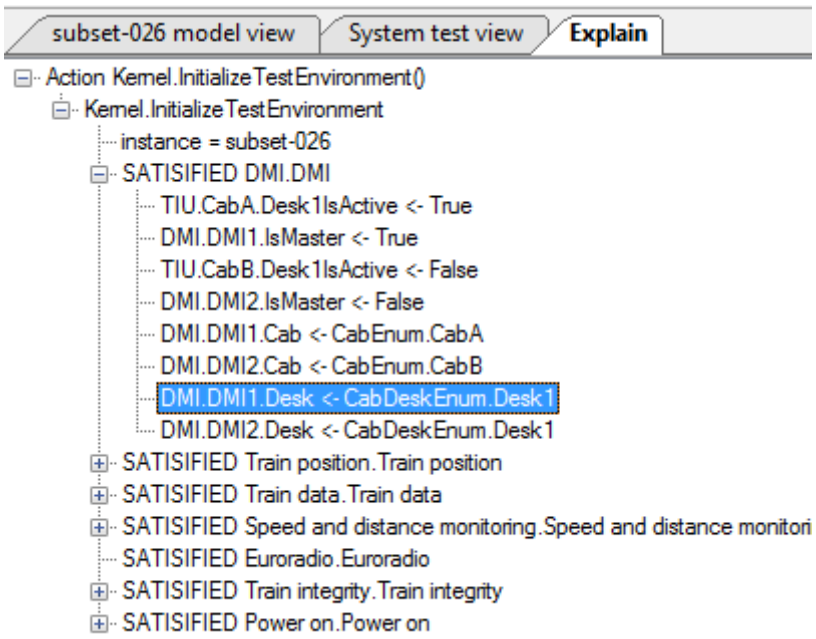

Figure 121: Action tree explain view

Figure 121 shows the explain view actions tree of the procedure *initializeTestEnvinronment()*. The explain view provides the status of all the rules related with this procedure. These statuses are:

- **SATISFIED:** the conditions of this rule have been satisfied and the rule is applied.
- **NOT APPLICABLE:** means the conditions of this rule have not been satisfied and the rule is not applied.

The tree also shows updates to the variables, with the new value specified. Double clicking on a tree node selects the corresponding model element.

The **Onboard and trackside messages** are used to provide a clear image of the exchanged messages between the onboard and the trackside. Figure 122 depicts an example of a message sent from the trackside, via balise telegram, to the onboard.

```
NID_C=>140,
NID_BG=>9618,
Q_LINK=>Messages.Q_LINK.Unlinked,
Sequence1 =>
[
   Messages.EUROBALISE.SubStructure1
   {
      TRACK_TO_TRAIN=>Messages.PACKET.TRACK_TO_TRAIN.Message
      {
         LINKING =>Messages.PACKET.TRACK_TO_TRAIN.LINKING.Message
         {
            NID_PACKET=>5,
            Q_DIR=>Messages.Q_DIR.Nominal,
            L_PACKET=>400,
            Q_SCALE=>Messages.Q_SCALE._1_m_scaleC,
            D_LINK=>1500,
            Q_NEWCOUNTRY=>Messages.Q_NEWCOUNTRY.Same_country___railway_administration__no_NID_C_follows,
            NID_BG=>9620,
Q_LINKORIENTATION=>Messages.Q_LINKORIENTATION.The_balise_group_is_seen_by_the_train_in_nominal_direction,
Q_LINKREACTION=>Messages.Q_LINKREACTION.Apply_service_brake,
Q_LOCACC=>1,
N_ITER=>0
         }
   }
```

Figure 122: Onboard and trackside messages view

6.5.6 Execute several steps at once

Test execution on EFSW is not limited to the execution of a single step each time. EFSW allows several steps to be executed at the same time.

6.5.6.1 Executing test steps until a certain expectation is satisfied

Select the desired target step on the hierarchical tree and right-click on it. On the contextual menu select the Run until expectation reached entry. This executes all the steps in the sub-sequence until the expectations of the selected step are satisfied. Steps after the selected step are not executed. See Figure 123.

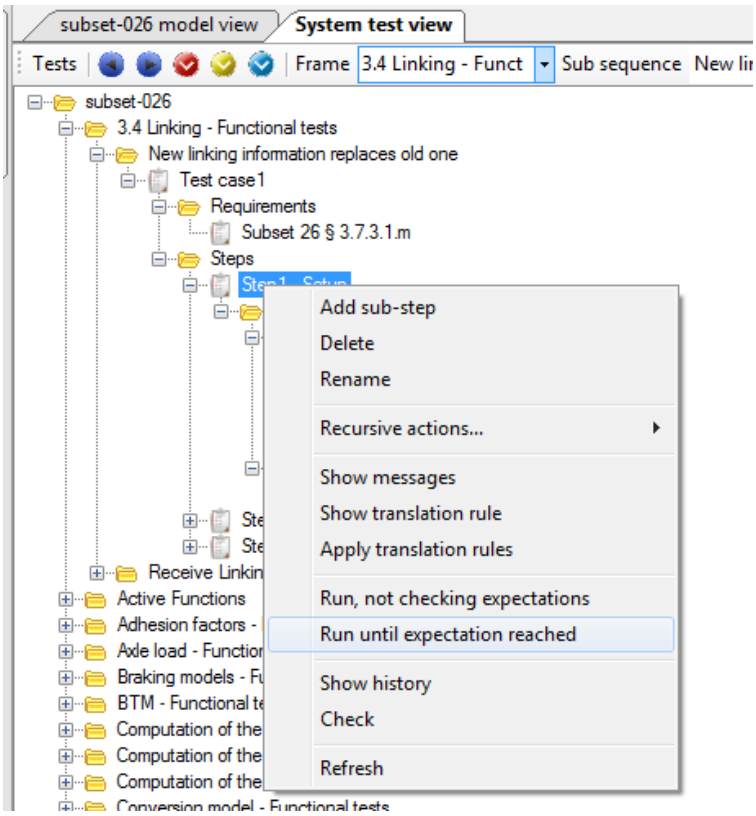

Figure 123: Execute a sub-sequence until expectation reached

This way of executing test steps is compatible with executing a test step by step described in Section 6.5.

6.5.6.2 Execute steps without checking its expectations are satisfied

The entry “Run, not checking expectations” executes the test as described in 6.5.6.1, the execution stops before the step expectation are checked.

6.5.6.3 Executing a sub-sequence

To execute a complete sub-sequence, right-click on the selected test subsequence in the hierarchical tree view and select the **Execute** contextual menu entry, as displayed in Figure 124. This executes all steps of the sub-sequence, even if an expectation is failed during the execution. Results are displayed in the execution time line.

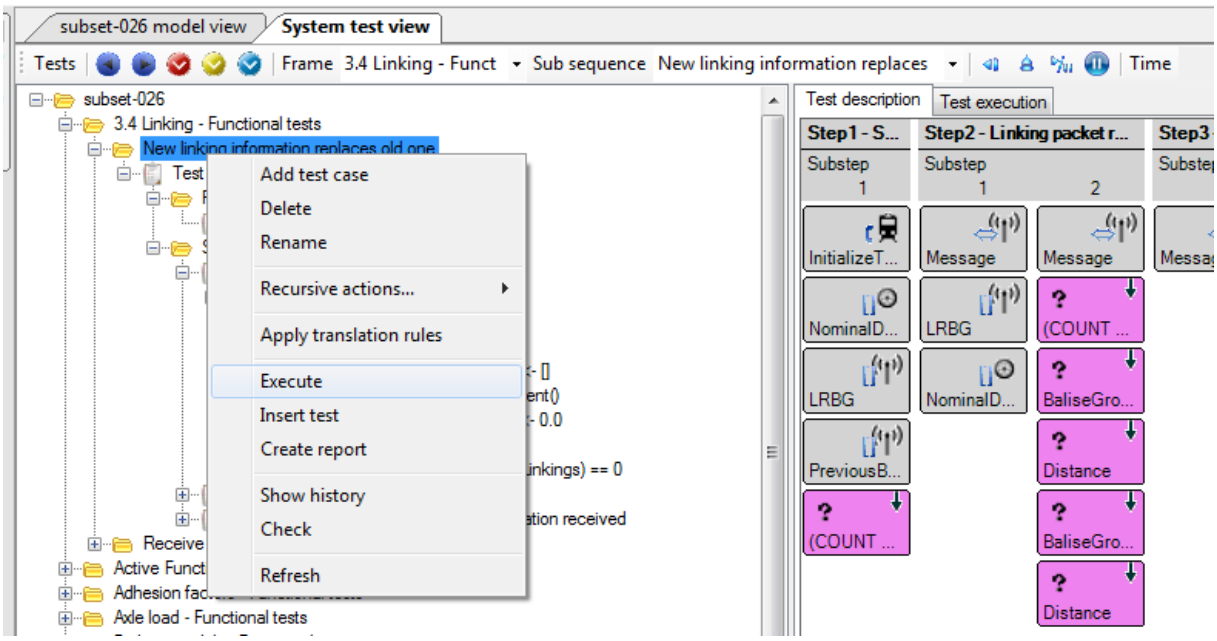

Figure 124: Executing a sub-sequence by the contextual menu

6.5.6.4 Execute a test frame

Tests can also be executed at the frame level, using the corresponding **Execute** entry in the frame's contextual menu, as shown in Figure 125.

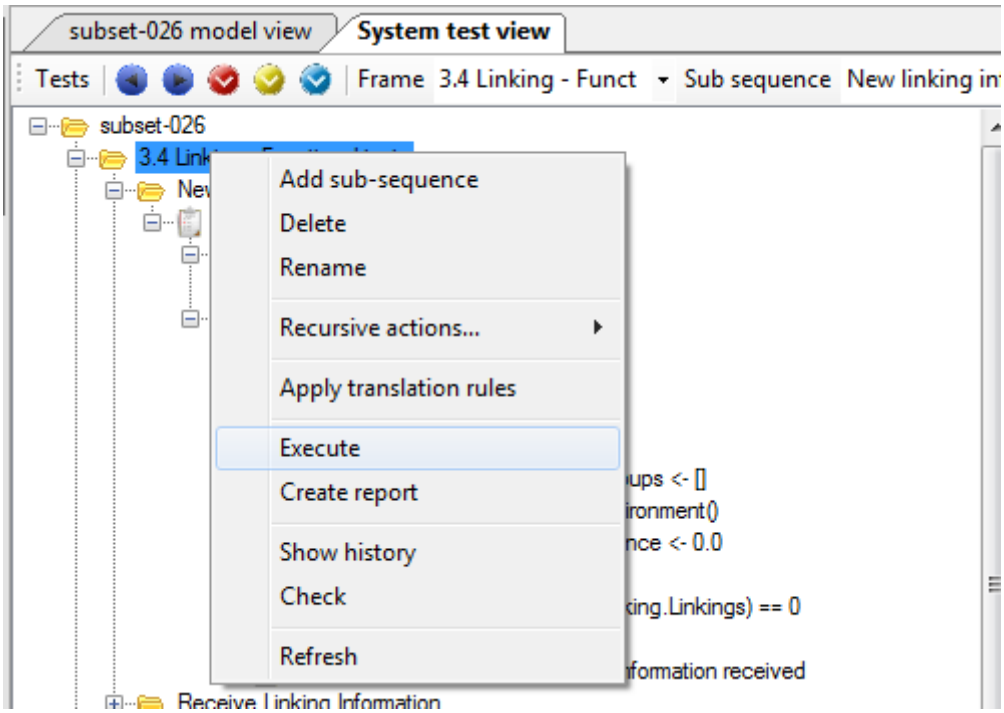

Figure 125: Executing a test frame

Selecting this option executes all sub-sequences defined in the frame, and does not produce a graphical result on the execution time line, since a time line can only present the result for a single sub-sequence. However, after execution completes, EFSW provides a summary of the test execution: how many sub-sequences were executed, how many of them were successful and the amount of failures.

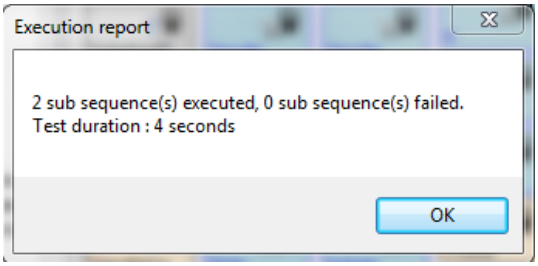

Figure 126: Test frame execution result

6.5.6.5 Execute all the tests related to a dictionary

EFSW allows the execution of all the test frames present in a dictionary. Select the desired dictionary in the test window, right-click and select **Execute** in the contextual menu. See Figure 127.

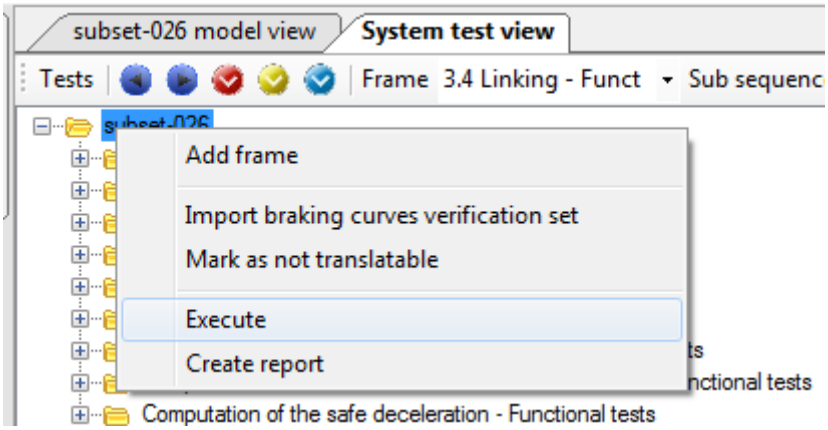

Figure 127: Subset test sequences execution

The result of executing all the tests linked to a subset is the same as the result described in Section 6.5.6.4.

6.6 Test tools

Figure 128 shows the actions which can be executed on the tests structure.

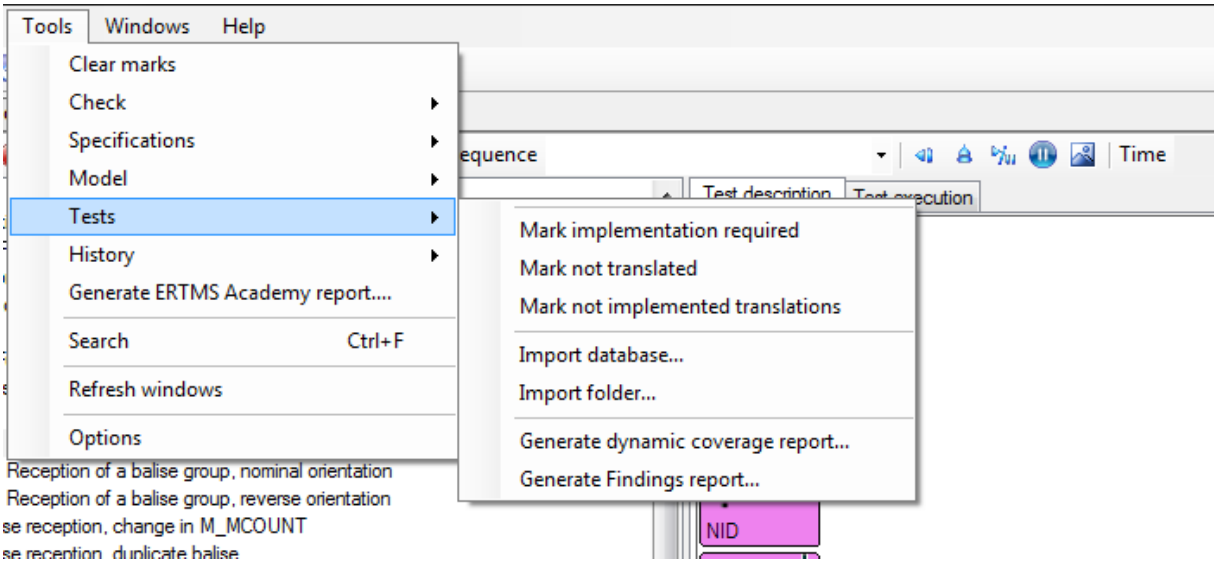

Figure 128: Actions which can be performed on the tests.

6.6.1 Search for test elements requiring an implementation

The Mark implementation required action displays all the test elements that have not yet been marked as implemented.

6.6.2 Search for test elements not translated

The Mark not translated action displays all the test elements that have not yet been translated (see Section 7 about translations).

6.6.3 Importing database/folder

6.6.3.1 Import data base:

Import database allows the user to import a ERA Subset-076 data base file and to incorporate it in a new test frame. The name given to the new test frame is by default subset-076.

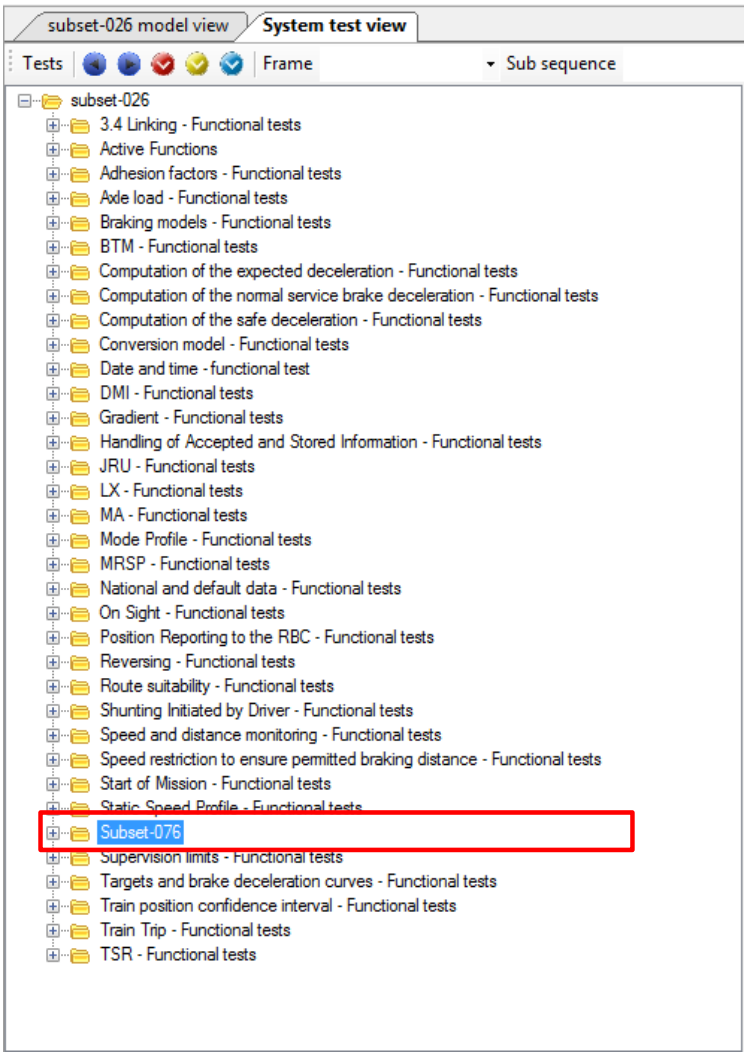

Figure 129: Import database file.

6.6.3.2 Import folder

When more than one database file need be imported, the system allows the user to select the folder where these files are stored. EFSW imports all the files at once and creates a single test frame with as many sub-sequences as data base files in the selected folder.

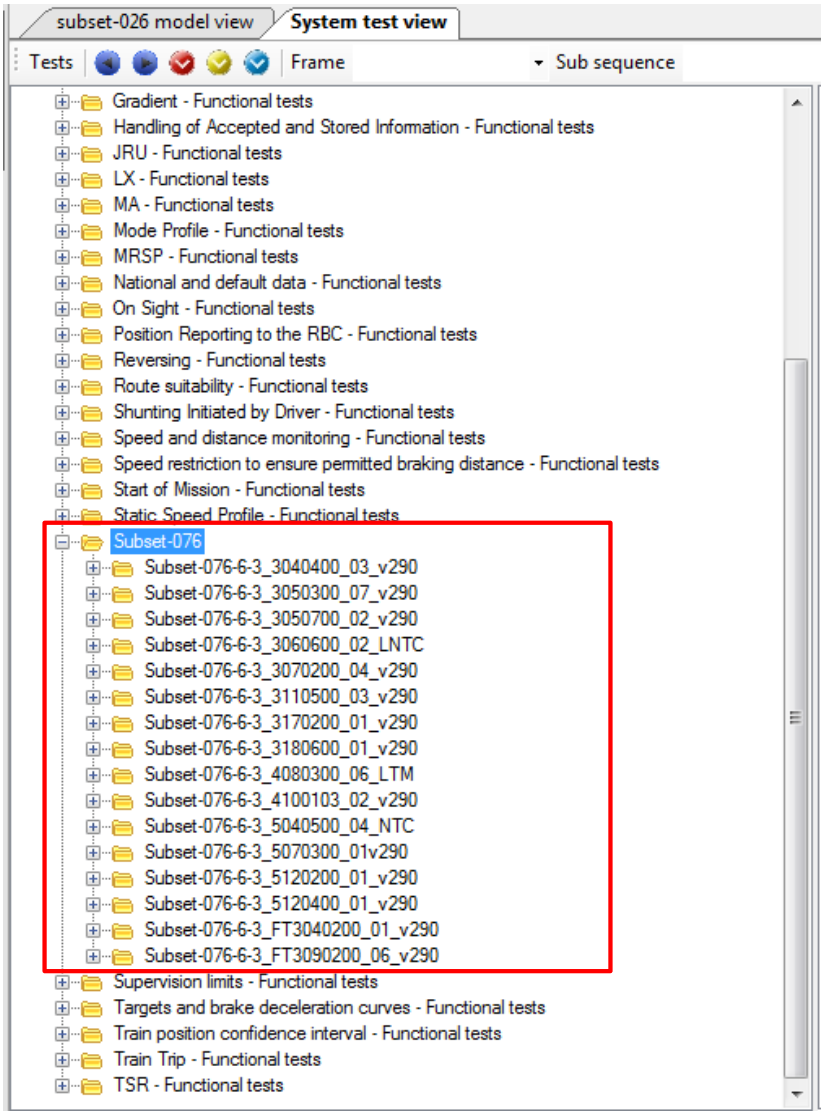

Figure 130: Import folder result

7 Model updates

EFS offers the possibility to create an update for a data dictionary in a second data dictionary. This is useful to compare different modelling approaches or to test a potential model update that will not necessarily be accepted. The update can modify existing model elements or add new ones, and as long as it is active these changes will permanently be in effect. This section describes the creation of an update data dictionary and how to update a model in one.

1.1 Creating a new update

To create a new update, select File\New\Update. If more than one dictionary is currently open, a popup window will appear asking the user to choose which dictionary will be updated. Note: it is possible to create an update for a data dictionary that is already updating another.

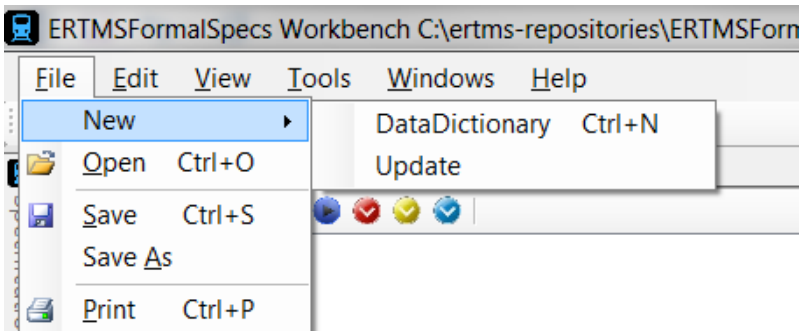

Figure 131: Create a new update

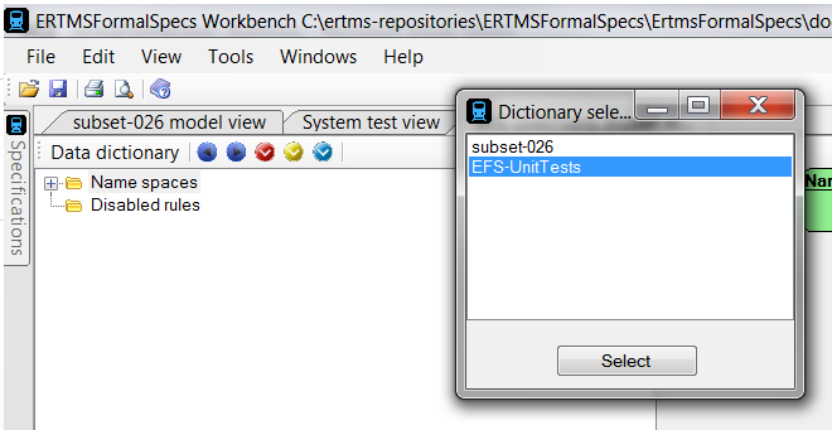

Figure 132: Dictionary selection if more than one dictionary is open

When the update is created, its model view will be opened.

To open an existing update, select File\Open and browse for the update file as with a normal data dictionary.

1.2 Making modifications

In an update data dictionary, the user can add model elements normally and modify existing model elements. To modify an element, right-click on the element in the base data dictionary model view to open its contextual menu. Under that menu, select “Add…\Update”.

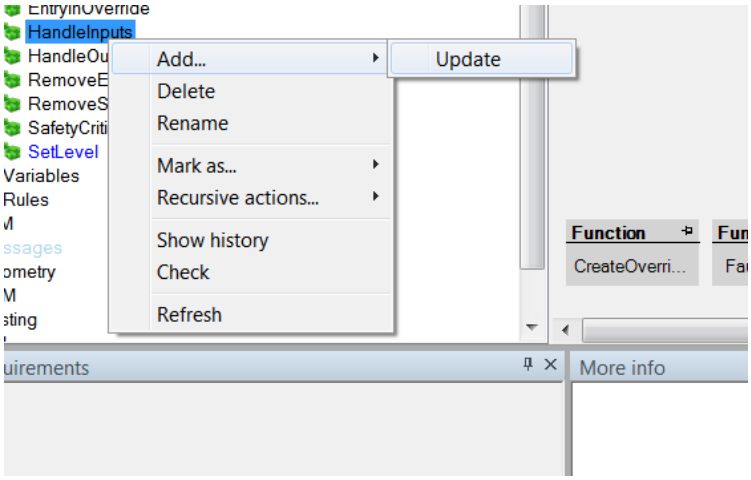

Figure 133: Creating an update for a procedure

The new element will be selected automatically by EFS. Please note that this is the only way to update a model element; if a new model element with the same identifier is created in the update dictionary, EFS will indicate this as an error.

Messages

Level	Message
Error	Expression Kernel.Mode may have several interpretations Kernel.Mode(DataDictionary.Variables.Variable), Kern...
Error	Cannot assign a value to Kernel.Mode
Error	Cannot determine type of target Kernel.Mode

Figure 134: The errors thrown by EFS if a model element is incorrectly updated

If an update for the model element already exists in the updating dictionary, EFS will simply select it in the update dictionary when the user tries to create an update for it.

The new model element can be modified in any way the user desires, exactly as a regular model element. All modifications will be applied immediately.

8 Translations

EFS can import Subset-076 tests, as presented in Section 6.6.3. However, these tests are expressed as a text and cannot be directly applied on the EFS model. EFSW defines a way to automatically translate Subset-076 textual tests into a sequence of actions and expectations using a translation dictionary, and provides a tool to fill the translation dictionary. This is the scope of this section.

8.1 Opening the translation view

The translation view is opened using the tool menu entry View/Show translation view, as shown in Figure 135.

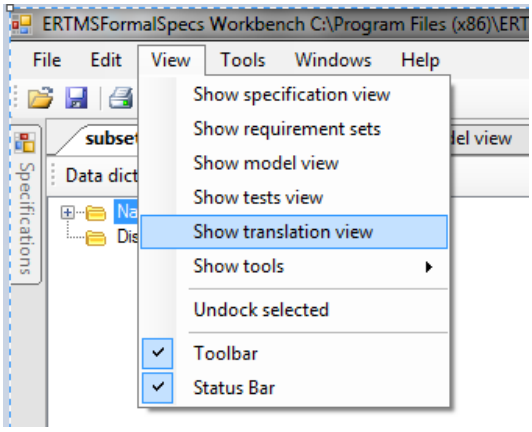

Figure 135: Opening the translation rules view

As usual, if several EFS files are opened – which is usually the case when considering Subset-076 tests – a dialog box is provided to select the EFS file from which the translation view should be opened. See Figure 136.

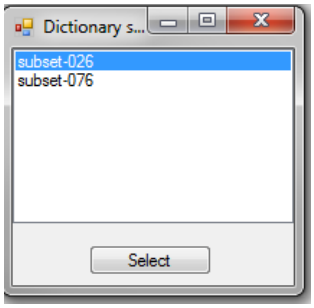

Figure 136: Selection for the translation rules when two EFS files are open

8.2 Overview

The translation browser presents the available translations. It proposes, as shown in Figure 137, hierarchical view of the translations, grouped using folders. Each translation contains two parts

- **The set of source texts:** when one of these texts is found in the Subset-076 test description, the translation is applied.
- **Sub-steps:** actions and expectations.

In addition, some translations are linked to requirements, for traceability purposes.

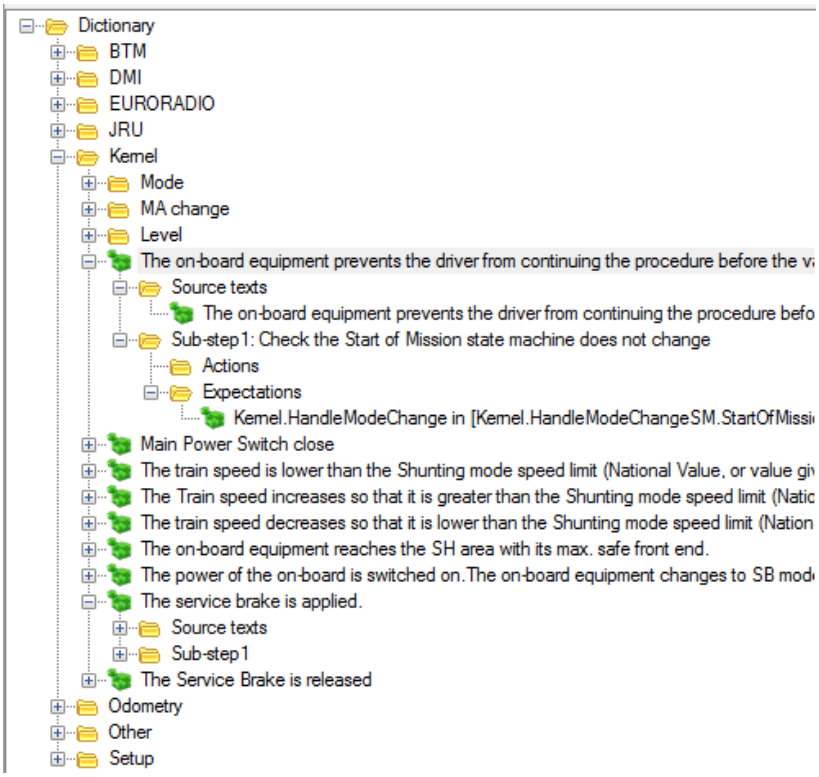

Figure 137: Translation view

Each translation can be edited as presented in section 6.4, using the description time line, as shown in Figure 138.

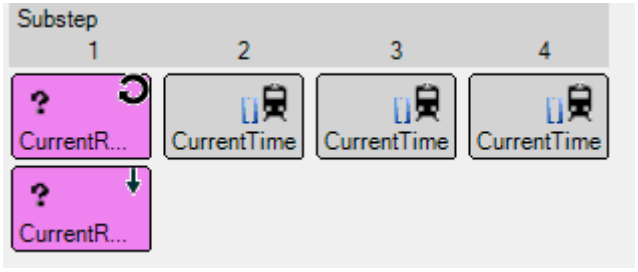

Figure 138: Translation description representation

8.3 Translating

To apply translation rules, select the node which needs to be translated in the test browser, and select Apply translation rules in the contextual menu, as shown below.

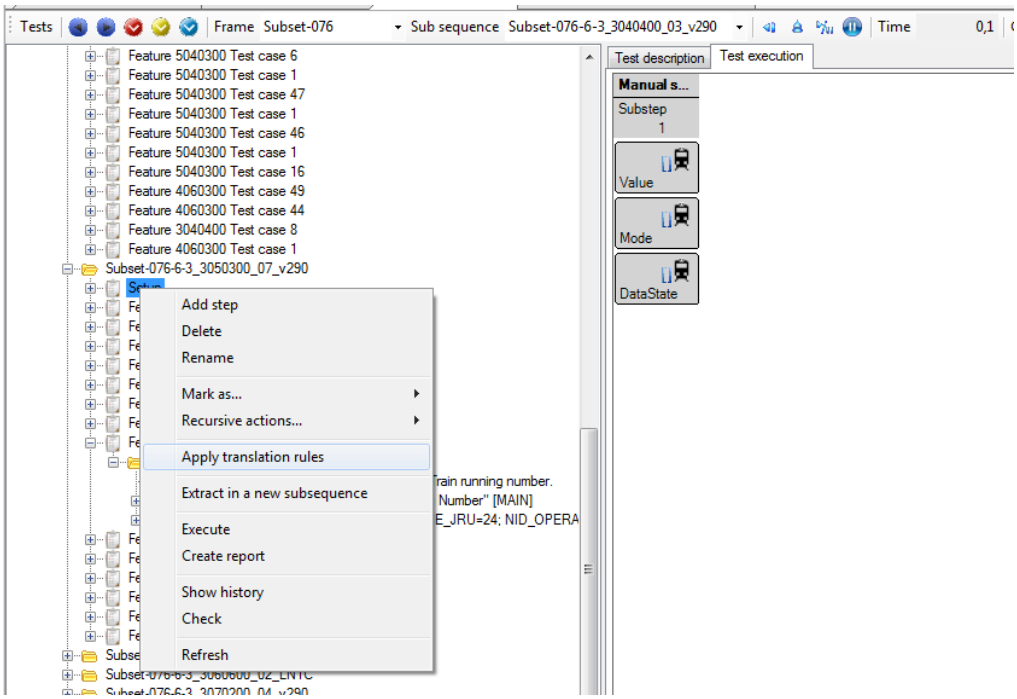

Figure 139: Apply translation rules

Translations are performed according to the following process. For each step that must be translated – a step with the **TranslationRequired** flag set to true, as defined in Section 6.4.4 – EFS searches for the translation that matches the step text and (possibly) comment. The translation's sub steps are then added to the step.

When translating a sub-sequence using the translation dictionary, EFS performs the following operation, as depicted in Figure 140:

1. For each step in the sequence, EFS finds the matching translation. The translation matches when it contains a source test for which
    a. The source text name corresponds to the step description
    b. If the source text holds comments, one of these comments matches the step comment.
    c. If two translations match the step, the more specific translation is used, that is, the one which matches both **description** and **comment**.
2. The translation's sub-steps are copied in the step
3. The action and expectation are adapted to that specific step by replacing the template variables with the values described in the step, according to following table.

Sub sequence related		
%D_LRBG	DLRBG	Number
%Level	Level	Enumeration Default.Level
%Mode	Mode	Enumeration Default.Mode
%NID_LRBG	NID_LRBG	Number
%Q_DIRLRBG	Q_DIRLRBG	Number
%Q_DIRTRAIN	Q_DIRTRAIN	Number
%Q_DLRBG	Q_DLRBG	Number
%RBC_ID	RBC_ID	Number
%RBCPhone	RBC phone number	String
Step		
%Step_Distance	Distance at which the step takes place	Number
%Step_LevelIN	Level before the step occurs	Enumeration Default.Level
%Step_LevelOUT	Level after the step occurs	Enumeration Default.Level
%Step_ModeIN	Mode before the step occurs	Enumeration Default.Mode
%Step_ModeOUT	Mode after the step occurs	Enumeration Default.Mode

Table 5: Replacements for template variables

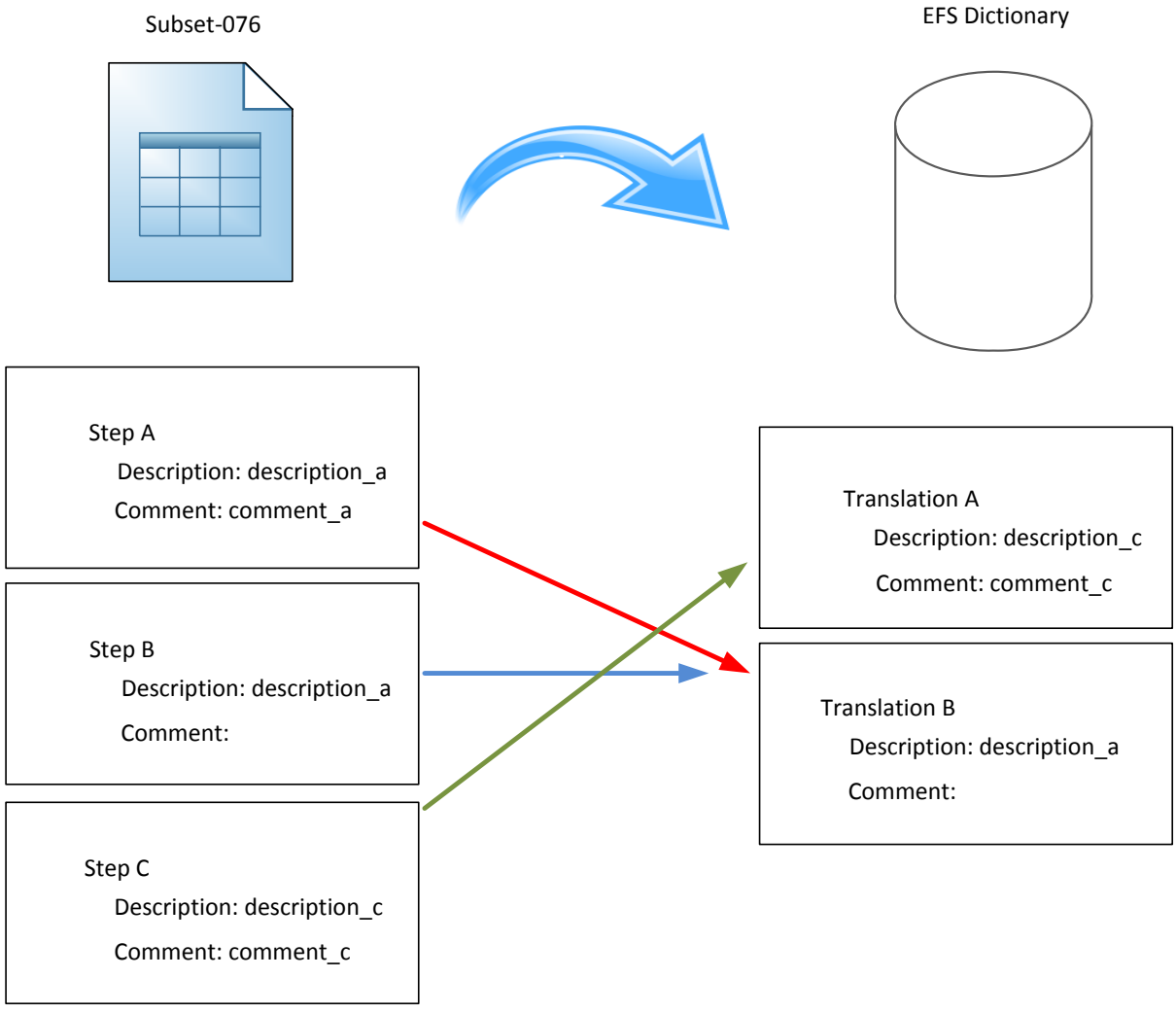

Figure 140: Translation process

In addition, if the distance of the current step is different from the distance of the previous step, the translation process automatically adds a new action in the step which follows the template

```
OdometryInterface.UpdateDistance ( %Step_Distance )
```

8.4 Adding translations in the translation dictionary

A new translation can be added in the translation dictionary either by using the contextual menu in the hierarchical view or by dragging a step from the test window to a folder of the translation view (this creates a new translation, fully configured to match the selected step).

8.5 Navigation

EFSW provides a way to easily navigate to the translation that would be used to translate a step, using the **Show Translation Rule** in the step contextual menu, as shown in Figure 141.

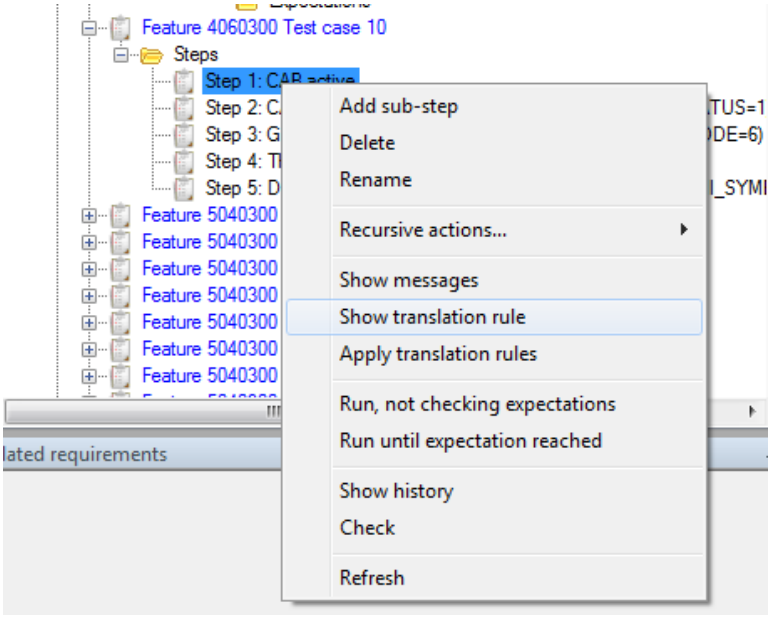

Figure 141: Show translation rule contextual menu

8.6 Show messages

Along with the test textual description, Subset-076 also provides the messages exchanged between the onboard and trackside systems. These messages are imported with the test description from the

Subset-076 database and can be visualised using Show messages in the contextual menu of a step, as in Figure 142.

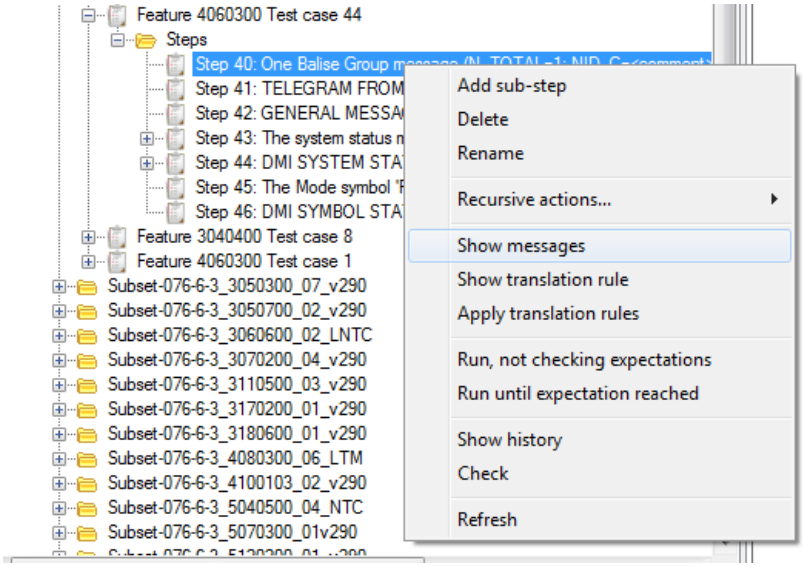

Figure 142: Show messages

This opens a structured view of the message, as presented in Figure 143 .

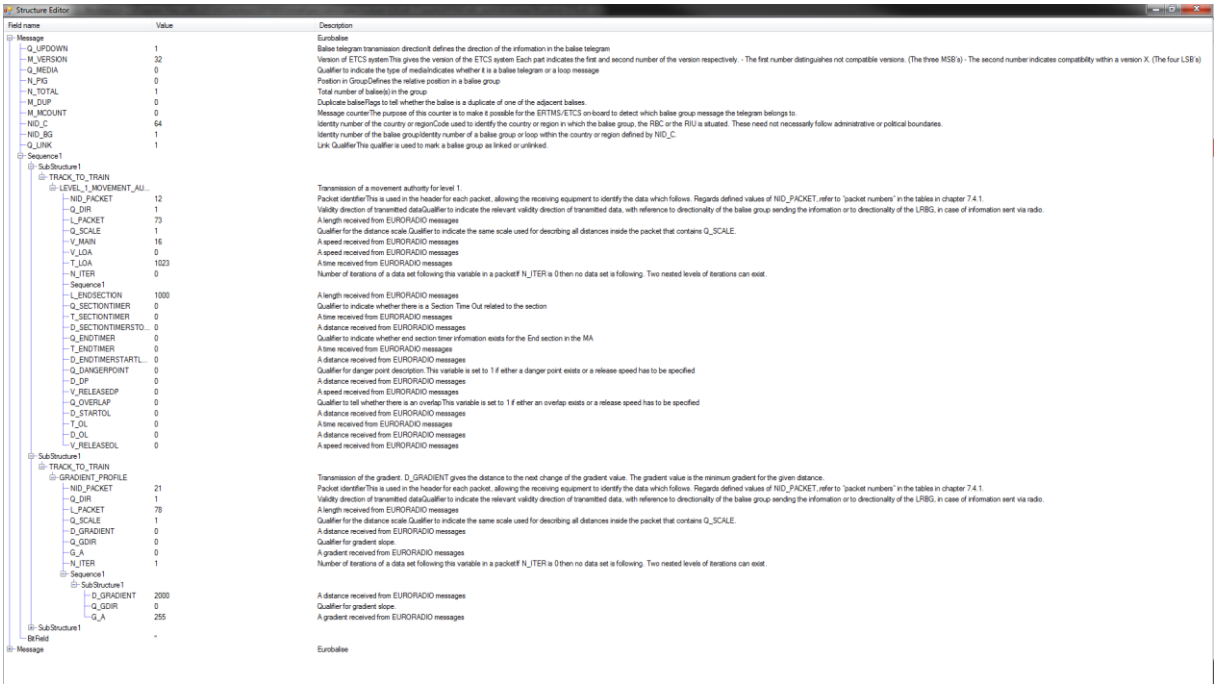

Figure 143: General overview of the message show window

8.7 Adding requirements to the translation view browser

Requirements can be linked to the translation rules present in the dictionary browser simply by dragging and dropping the selected requirement on the translation in the translation dictionary.

9 History

EFSW allows accessing the underlying Git[8] repository to provide historical data on requirements, model and tests. These features are only available when the model belongs to a git repository. To access the history information for the trainborne model, clone the repository located at

https://github.com/openETCS/ERTMSFormalSpecs

Moreover, git should be installed on the machine and accessible in the $PATH.

9.1 History and model comparison

Access to the model history and comparison features is available in the Tools/History as presented in Figure 144.

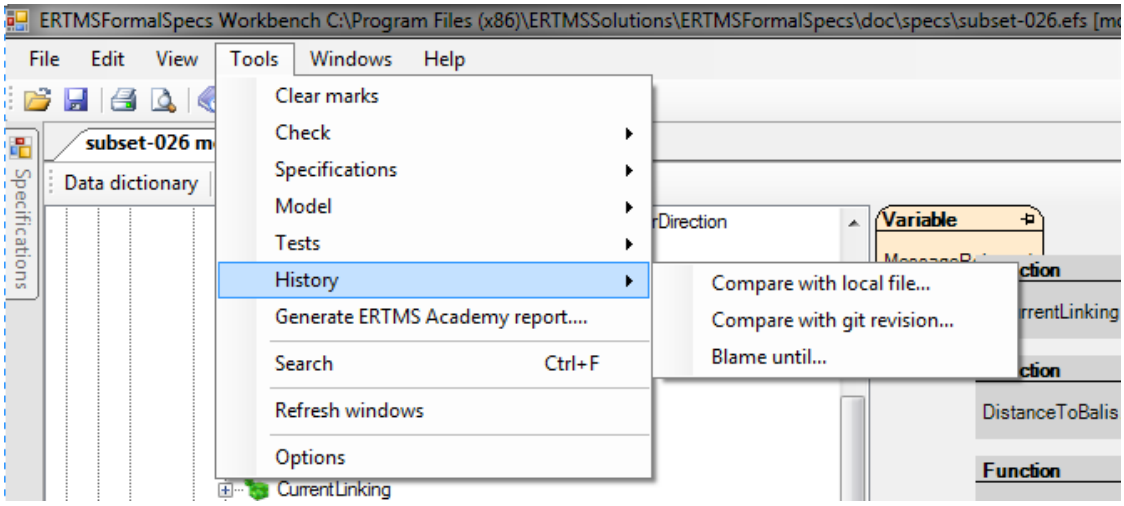

Figure 144: Possible history actions

9.1.1 Compare with local file

Two .efs files can be compared in a structured way, using **Compare with local file…** The comparison is done between the .efs file currently loaded and the one selected in the file browser. Differences between those two files are marked as Info messages (see Section 3.5 as displayed in Figure 145.

The message has the following structure:

CHANGED <fieldname> FROM text1 TO text2,

which indicates that the field <fieldname> has the value text2 in the currently open file, and has value text1 in the selected file.

[8] More information about Git can be found at http://git-scm.com/.

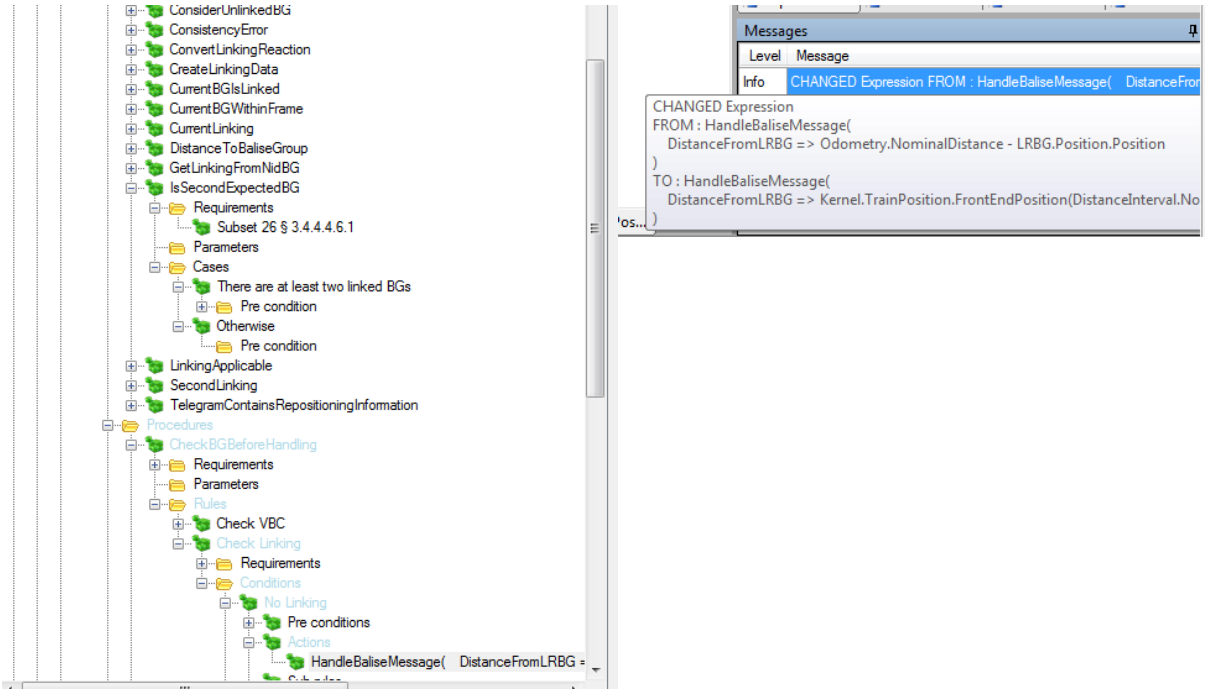

Figure 145: Result of comparing two definitions of Subset-026

9.1.2 Compare with git revision

The comparison described in Section 9.1.1 can be performed with a previous version of the current model. When executing **Compare with git repository…** EFSW opens the revision selector which displays the commit date, author and commit message of all commits available in the repository, as shown in Figure 146. Selecting the version to compare to the current version is done by double clicking on it.

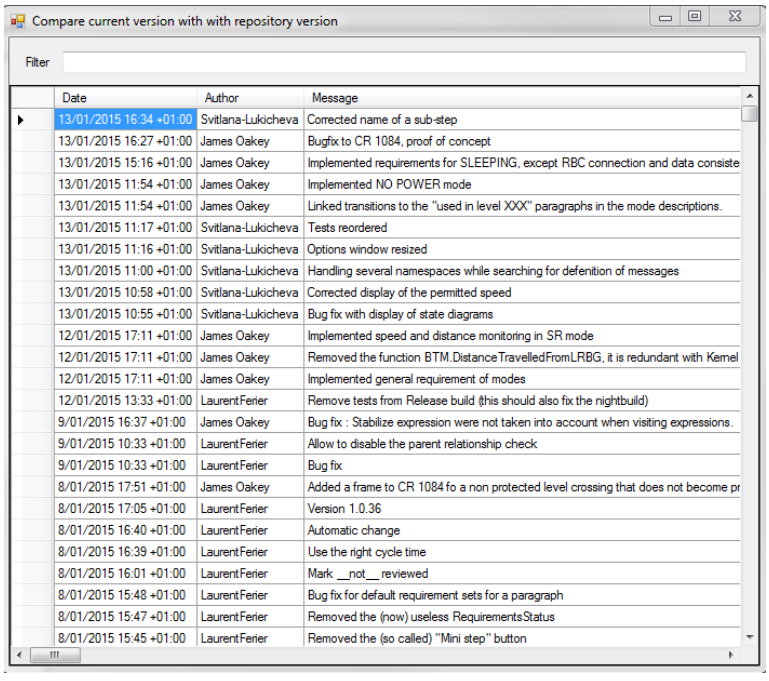

Figure 146: Remote GIT version to be selected for comparison

After comparison, elements that have been modified are highlighted in blue (see Section 9.1.1).

9.1.3 Blame until

The tool **Blame until…** builds a database of the changes between the current version and a version selected from the Git repository, using the selector dialog presented in Figure 147. Double click on an entry to select it.

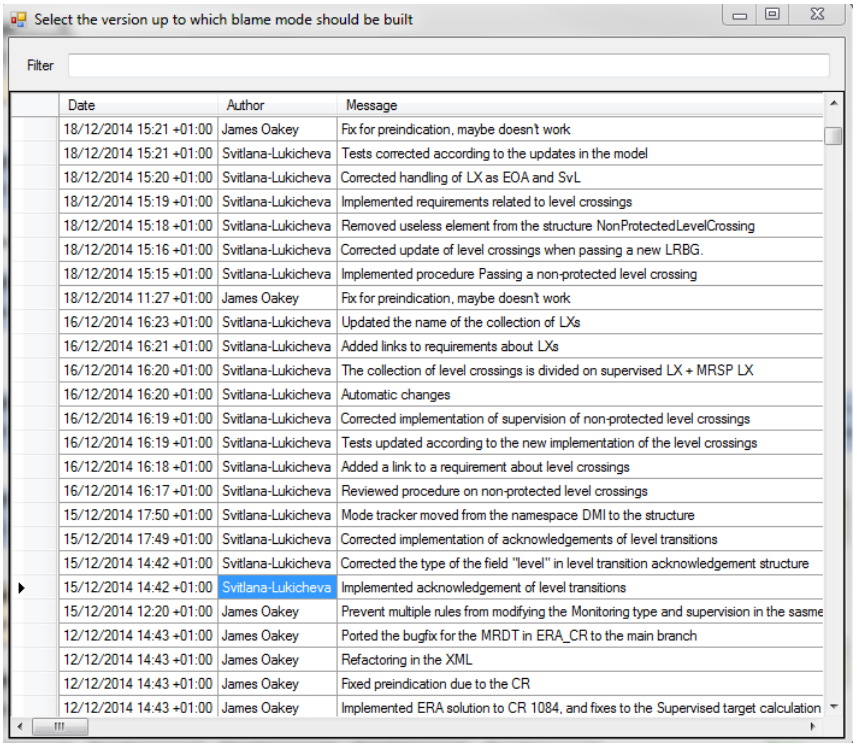

Figure 147: Selection of the blame until filter

During the process, EFSW displays the dialog presented inFigure 148.

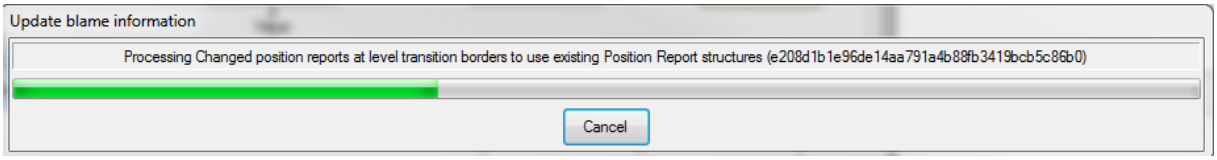

Figure 148: Processing the information for the blame until option

Data gathered during this process are used to fill the History window presented in Figure 149. This window is automatically updated when selecting an element (requirement, model object, test …) in EFSW.

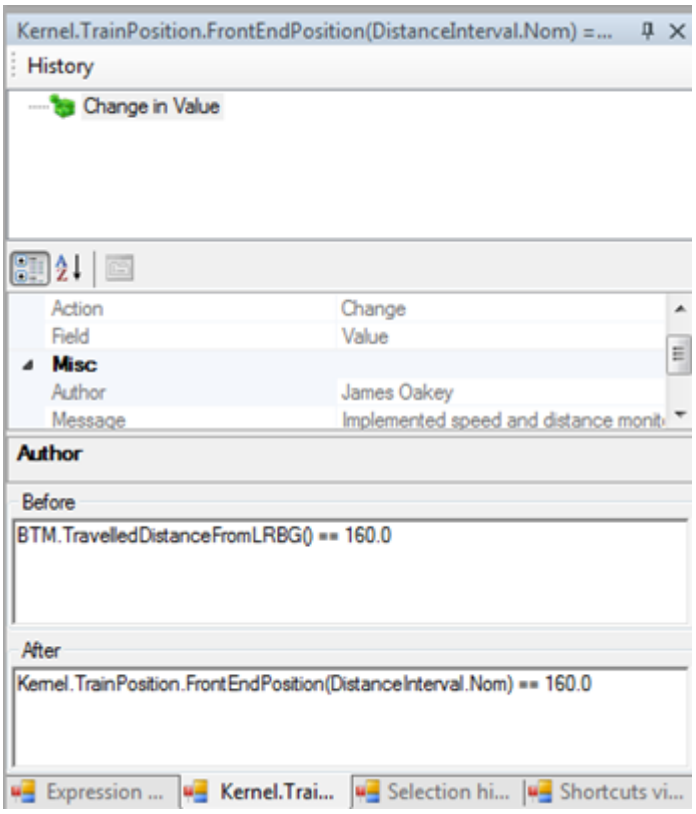

Figure 149: History window

This window holds the following information

- **History:** traces all changes in the selected element.
- **Properties** of the change.
- **Action:** the operation performed, which can be either Add (a new element has been added), Remove (an element has been removed) or Change (a modification has been performed in the element).
- **Field:** part of the element that was altered.
- **Author:** the name of the author of the commit.
- **Message**: the message the author provided during the commit.
- **Date:** the date of the commit.
- **Before:** in case of a change or a deletion, the value of the field before the action was performed.
- **After:** in case of a change or an addition, the value of the field before the action was performed.

10 ERTMSFormalSpecs reports

10.1 Specification coverage report

10.1.1 Purpose

The purpose of the specification coverage report is to provide a summary of the implementation of the requirements based on the modelled elements, along with implementation statistics.

The **specification coverage** part of the report provides the status of all the requirements specified in the requirement document. This status can be one of the following:

- **Implemented:** modelling of the requirement is complete. This ensures that all model elements linked to this requirement are also marked as implemented.
- **Not implementable:** the paragraph does not need to be implemented because it is not a requirement.
- **Not implemented:** modelling of the requirement is not complete.
- **N/A:** the implementation status of this paragraph is not known. In this case, implementation is not performed.

The **requirement coverage report** provides the list of all the requirements of the specification along with the model elements that implement them.

The **model coverage report** provides the list of requirements associated with each model element.

10.1.2 Structure

The specification coverage report can be composed of four following chapters:

- **Specification coverage**. This chapter holds two sections:
  - **Statistics**. This section provides a table with the following information:
    - Total number of specification paragraphs.
    - Number of applicable paragraphs. The paragraph is applicable if its scope is "OBU" (on board unit) or "OBU and Track" and if its type is "Requirement".
    - Number and percentage of covered paragraphs. This percentage is computed from the total number of applicable paragraphs. A paragraph is considered as covered if it is marked as "implemented" and all the EFS elements that are related to this paragraph are marked as "implemented".
  - **Specification**. This section provides a table with an entry for each specification paragraph. For each paragraph the table provides its scope ("OBU", "OBU and Track" or "Track"), type and implementation status.
- **Covered requirements**. This chapter provides the list of requirements covered by the model and can be composed by the two following sections:
  - **Statistics**. This section provides a table with the following information:
    - Number of applicable paragraphs.
    - Number and percentage of covered requirements.

- **Covered requirements**. This section provides a table with an entry for each covered requirement. For each covered requirement, the table provides the list of the model elements that implement it with the associated comment.

- **Non-covered requirements**. This chapter provides the list of requirements that are not yet covered by the model and can be composed of two following sections:
  - **Statistics**. This section provides a table with the following information:
    - Number of applicable paragraphs.
    - Number and percentage of non-covered requirements.
  - **Non-covered requirements**. This section provides the list with the non-covered requirements.
- **Model coverage**. This chapter provides the list of implemented model elements and can be composed of the following sections:
  - **Statistics**. This section provides a table with the following information:
    - Number of implemented model elements.
    - Number and percentage of modelled paragraphs.
  - **Implemented rules**. This section provides a table containing an entry for each implemented rule. For each implemented rule the table provides the (list of) the paragraph(s) it implements.
  - **Implemented types**. This section provides a table containing an entry for each implemented type. For each implemented type the table provides the (list of) the paragraph(s) it implements.
  - **Implemented variables**. This section provides a table containing an entry for each implemented variable. For each implemented variable the table provides the (list of) the paragraph(s) it implements.

10.1.3 Launch the specification reporting

The specification coverage report creation window is accessible via Tools/Specifications/Generate coverage report... (See Figure 150).

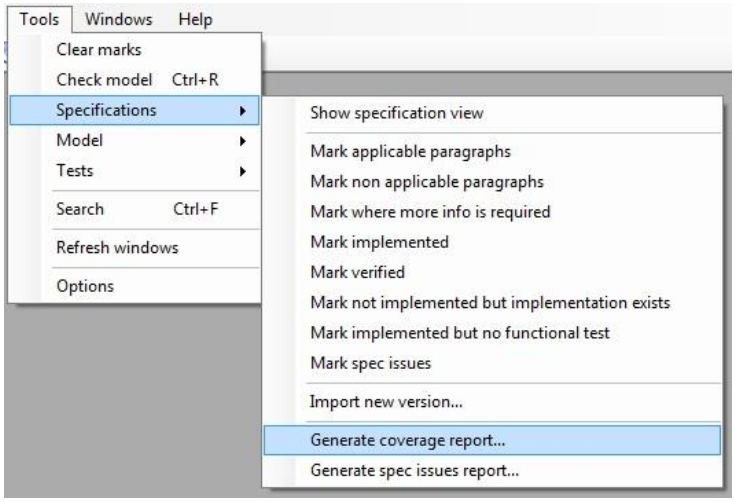

Figure 150: Launch the specification coverage report

This opens the dialog which allows selecting the report options, as depicted below.

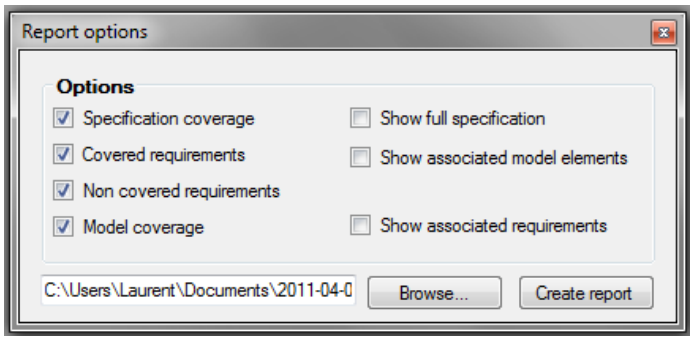

Figure 151: Specification coverage report options

The figure above shows the window used to select the different report options. The check boxes in the left column allow creating the corresponding report chapters with their "Statistics" section. The check boxes of the right column add additional information when checked, as described below:

- **Show full specification** check box is available only when the "Specification coverage" check box is selected. This option adds the "Specification" section to "Specification coverage" chapter.
- **Show associated model elements** check box is available only when the "Covered requirements" check box is selected. This option adds the information about the elements implementing the covered requirements in "Covered requirements" section of "Covered requirements" chapter.
- **Show associated requirements** check box is available only when the "Model coverage" check box is selected. This option adds the list of requirements modelled by each model element in the sections "Implemented rules", "Implemented types" and "Implemented variables" of "Model coverage" chapter.

The "Browse" button allows selecting the folder and the name of the generated report.

10.2 Generate spec issue report

The purpose of this report is to summarise the specification issues encountered during analysis. One can generate this report using the menu item Tools/Specifications/Generate spec issues report… .

10.2.1 Structure

The report is divided into different chapters.

- **More information needed**: indicates the requirements for which the description is not precise enough and requires more information. These issues are composed by:
  - **Description**: the requirement text, as written in the specification.
  - **Comment**: a comment made by the developer.
- **Specification issues report**: provides the requirements which pose problems, and cannot be modelled as such. Each issue is composed of
  - **Description**: the requirement text, as written in the specification.
  - **Comment**: a comment made by the developer.
- **Design choices**: Provides the list of new requirements requires to model the system. They are composed of
  - **Description**: the new requirement text
  - **Comment**: an optional comment.

10.2.2 How to create a specification issue

An element of the specifications appears on the spec issues report when its **SpecIssue** flag is set to true. Section 4.2.2 describes the properties of the specifications.

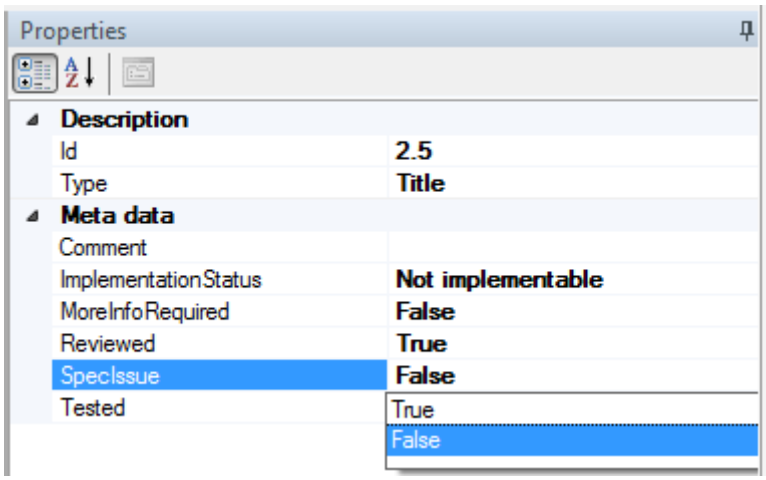

Figure 152: Setting to true the SpecIssue flag

This action should be performed to all the requirements which contain any kind of specification issue.

10.2.3 Launch the report

To generate the Specs issues report, go to Tools/specifications/Generate spec issues report. After clicking on the option, a menu indicating the different options contained on the report appears.

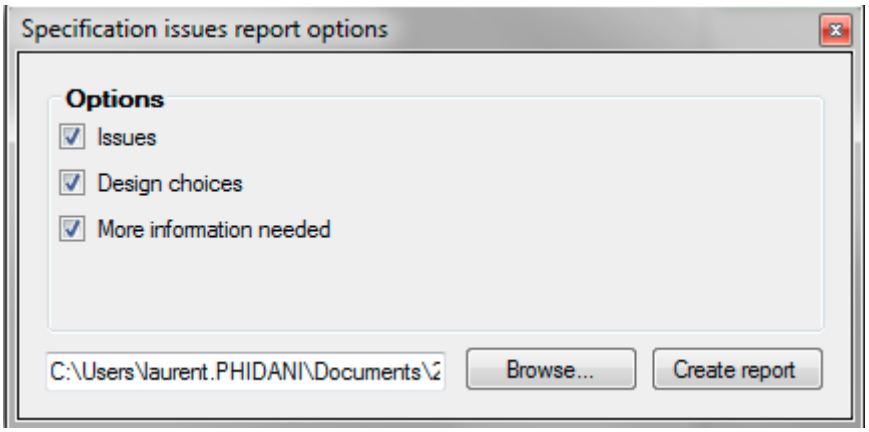

Figure 153: Contents of the specification issues report

Select the place to store the resulting report and then create the report. Figure 154 shows an example of the specification issues report.

More information needed

3.5.5.1.b is not precise enough

3.5.5.1.b	
Description	If an error condition requiring the termination of the communication session is detected on-board (e.g., not compatible system versions between on-board and trackside).
Comment	[stan@ertmssolutions.com] Any other errors requiring this?

3.6.6.9.b is not precise enough

3.6.6.9.b	
Description	it is told not to do so, OR
Comment	How can it be told not to do so?

3.13.9.3.5.6 is not precise enough

3.13.9.3.5.6	
Description	In case the calculation of the GUI curve is enabled, for display purpose only, the P speed related to SBD shall be calculated for the estimated train front end as follows: V_P_EOA (d_estfront) = min { V_SBD (d_estfront + Vest * (T_driver + T_bs1)), V_GUI_EOA (d_estfront) } V_P_EOA (d_estfront) = 0 if d_estfront + Vest * (T_driver + T_bs1) >= d_EOA
Comment	V_GUI_EOA is not defined.

Figure 154: Extract of the specification issues report

10.3 Generate data dictionary report

10.3.1 Purpose

The purpose of the data dictionary report is to provide information about the model. The report can be created on two different levels of details:

- **Default level:** the report provides only the list of implemented model elements together with their associated requirements.
- **Detailed level:** the report provides all the available details for each implemented model element.

10.3.2 Structure

The report is divided in several chapters each one corresponding to one of the data dictionary namespaces. Depending on the user's choice, each chapter can contain information about its

- **Ranges**
- **Enumerations**
- **Structures**
- **Collections**
- **Functions**
- **Procedures**
- **Variables**
- **Rules**

For each element described above, the data dictionary report provides a comment describing its utility. The report indicates the implementation and verification status of each model element.

10.3.3 Launch the model report

The data dictionary report creation window is accessible via Tools/Model/Generate data dictionary report... (See Figure 155).

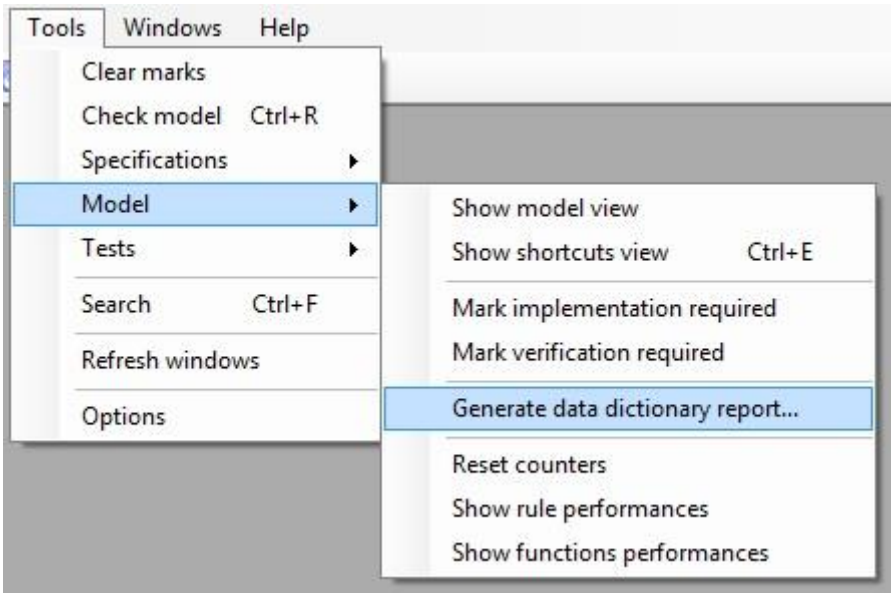

Figure 155: Launch the data dictionary report

This opens the dialog which allows selection of the report options, as depicted below.

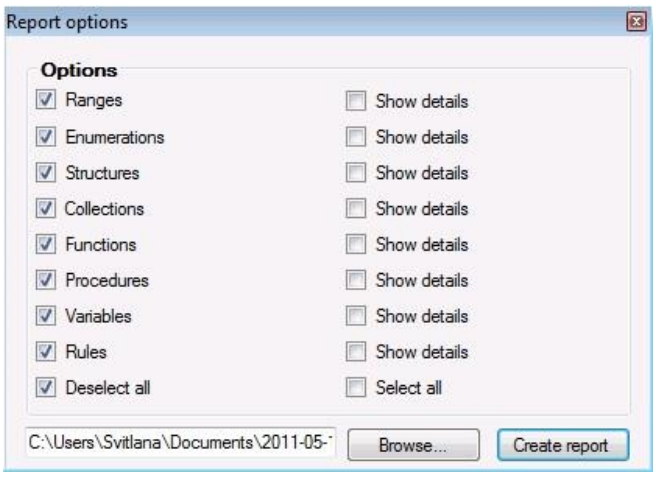

Figure 156: Data dictionary report options

The different check boxes filter the type of elements to be included in the report, and specify whether it has to describe the details of these elements.

10.4 Generate functional analysis report

10.4.1 Purpose

The functional analysis report describes each function and procedure's relationship to their namespace. It clearly indicates the functions that are exposed by a namespace and the locations where they are used.

10.4.2 Structure

The functional analysis report is divided in several chapters, one per namespace, and presents each namespace's **Exposed functions/procedures**. An exposed function or exposed procedure is defined in a namespace and used in another. This report provides the relationship between namespaces.

Each entry presents the following information, as shown in Figure 157.

- **Function** or **procedure name.**
- **Function** or **procedure parameters,** identified by a name and a type.
- **Function return value.**
- **Requirements** related to the function or procedure.
- **Known usages:** provides the list of packages which are using the function or procedure.

Function MaxSpeedFunction

MaxSpeedFunction	
This function provides the maximum speed	
Parameters	
Name	**Type**
Distance	Double
Return value	
Default.BaseTypes.Speed	
Related requirements	
No requirements related to this element	

Known usages

Usage
Kernel.TrackDescription.AxleLoad
Kernel.SpeedAndDistanceMonitoring.DecelerationCurves.GUI
Kernel.LX
Kernel.MRSP
Kernel.TrackDescription.PermittedBrakingDistance
Kernel.TrackDescription.StaticSpeedProfile
Kernel.TSR

Figure 157: Extract of a functional analysis report.

10.4.3 Launch the functional analysis report

The Generate functional analysis report can be accessed by Tools/Model/Generate functional analysis report.

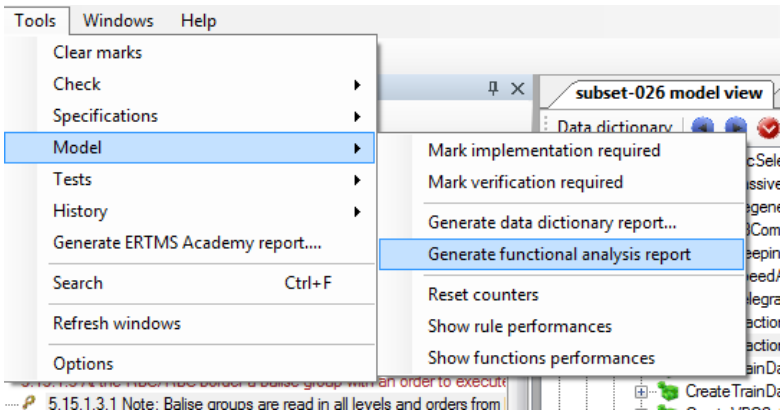

Figure 158: Launching the functional analysis report tool

10.5 Dynamic tests coverage report

10.5.1 Purpose

The purpose of the dynamic tests coverage report is to provide dynamic coverage of the model by some or all of the tests. The report can be created for the complete functional tests set of the model

or for a certain element of the tests tree. In that case the report is created for that element and all the elements beneath it in the test hierarchical tree. For example, a report can concern

- **The whole test tree,** containing information of all its levels.
- **All the available frames.**
- **All the frames** and **all the sub sequences.**
- A particular **sub sequence** with **all it sub cases.**
- A particular **test case.**

10.5.2 Structure

For each selected level, the report provides the percentage of activated rules of the EFS model and (if selected) the list of activated rules and/or the list of rules that weren't activated.

10.5.3 Launch the test coverage reporting

The specification coverage report creation window is accessible via Tools/Tests/Generate dynamic coverage report... (Figure 159).

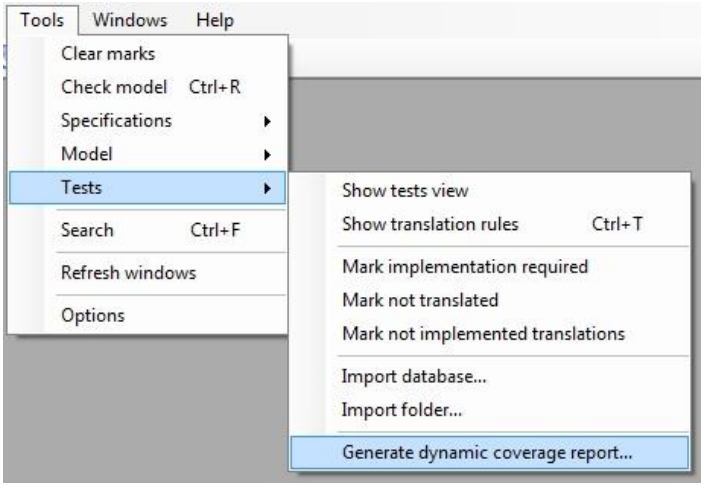

Figure 159: Launch the dynamic test coverage report

The option of a **partial** report creation for a selected item of the tests tree is accessible via the "Create report" option from its contextual menu.

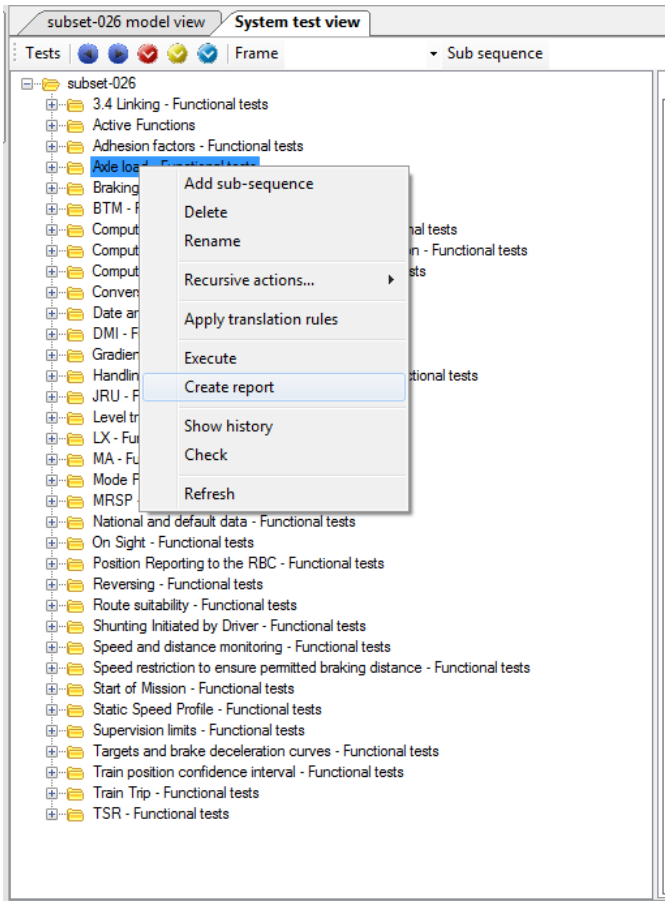

Figure 160: Report creation for a specific element on the test hierarchical tree

This action opens the dialog box displayed below.

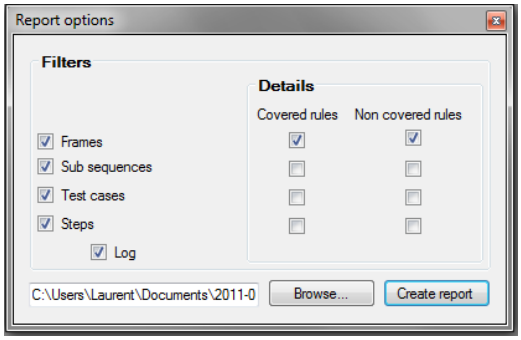

Figure 161: Dynamic tests coverage report options

Figure 161 shows the dialog which offers the different report options. The check boxes in "Filters" group box allow selecting different levels of the report. The corresponding check boxes in the "Details" group box include the list of covered and/or not covered rules to the corresponding level. "Log" check box allows enables the log information for the different steps.

The "Browse" button allows the user to select the name and the folder of the generated report.

10.6 Generate findings report

10.6.1 Purpose

The findings report contains the findings detected while modelling Subset-076 tests:

- **Comments**: possible improvements to the subset-076 specification.
- **Questions**: elements of the Subset-076 which its explanation and description is not clear enough.
- **Bugs**: problem detected on the test specification.

10.6.2 Structure

The report is divided in two different chapters. The first one contains the currently open findings and on the second one the findings which have been addressed.

10.6.3 Launch the findings report

To activate the findings report, open Tools/Tests/Generate Findings Report… Figure 162 illustrates how to access to the findings report

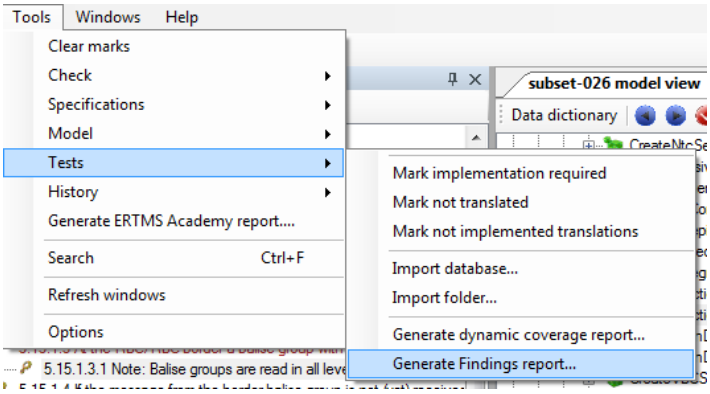

Figure 162: Activating the findings report

Then, select a place to save it and click on create report. Figure 163 illustrates the contextual menu related with this procedure.

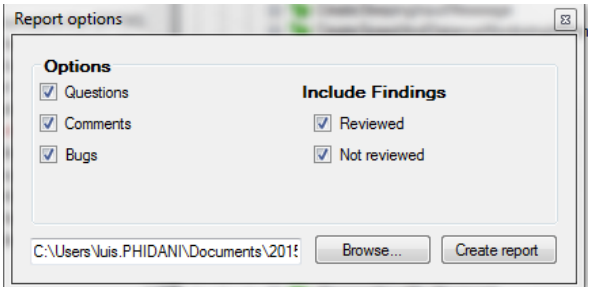

Figure 163: Selecting the contents of the findings report

The file contents are described in Section 10.6.1. The findings described on the report can be classified according their revision status.

10.7 Generate ERTMS academy report

10.7.1 Purpose

The ERTMS academy report describes the evolution, in terms of number of implemented requirements and test, of a student during a given period of time.

10.7.2 Structure

The report is composed by a single chapter which contains all the information related to the student's progress. It holds a list of additions and deletions in the model. Each entry on the list contains:

- **Author:** student responsible of the addition or deletion.
- **Comment:** explicative message written before committing and pushing the modifications on GIT.
- **Statistics:** brief of the elements modified, deleted or added on the model.

Figure 164 shows a typical entry for this list.

Added on 17/02/2014 15:20:28 +01:00	
Author	luis(luis@ertmssolutions.com)
Comment	Test Coment from las Pull Request
Statistics	JRU.efs_ns 2 addition(s), 2 deletion(s) JRU - Functional tests.efs_tst 91 addition(s), 0 deletion(s) 2 file(s) changed, 93 addition(s), 2 deletion(s)

Figure 164: ERTMS Academy report extract.

10.7.3 Launch the ERTMS academy report

To activate the ERTMS academy report open Tools/Generates ERTMS Academy report… Figure 165 proves the procedure to generate an ERTMS Academy report.

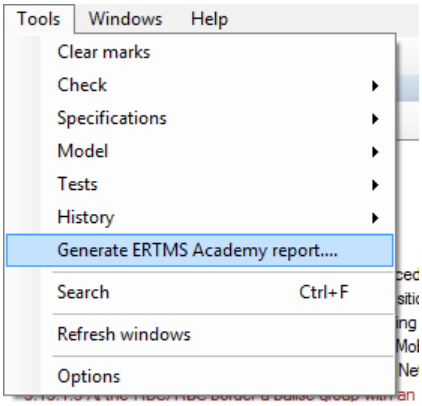

Figure 165: Activating the ERTMS Academy report

Then, provide the name of the student and the point from the last ERTMS Academy report done. See Figure 166.

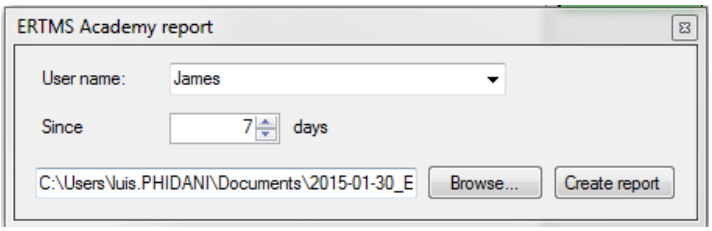

Figure 166: Configuration for creating the ERTMS Academy report

11 ERTMSFormalSpecs general tools

11.1 Clear marks

This tool is located on Tools/Clear marks, as Figure 167 shows.

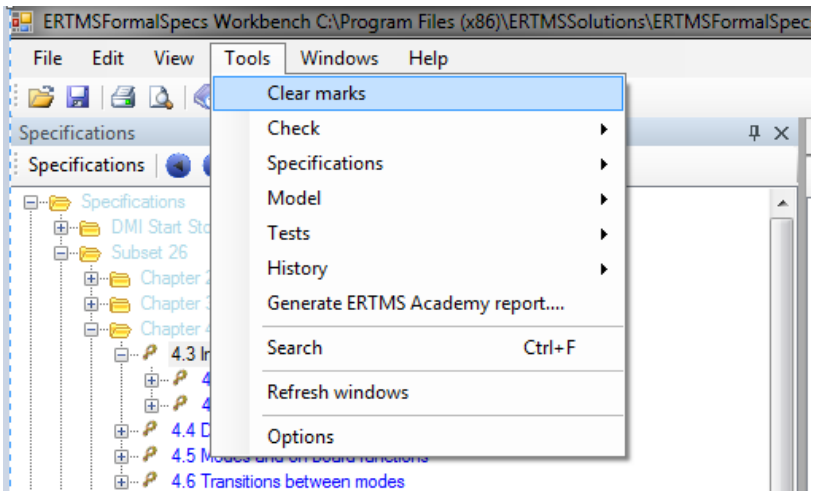

Figure 167: Clear marks option location

It lets the user to unmark all marked elements in the system (remove all messages), see Section 3.5.

11.2 Search

EFSW provides a way to search elements in the entire model (requirements, model, tests …). This feature is located on Tools/Search or can be activated using the Crtl+F, which displays the following search dialog (Figure 168).

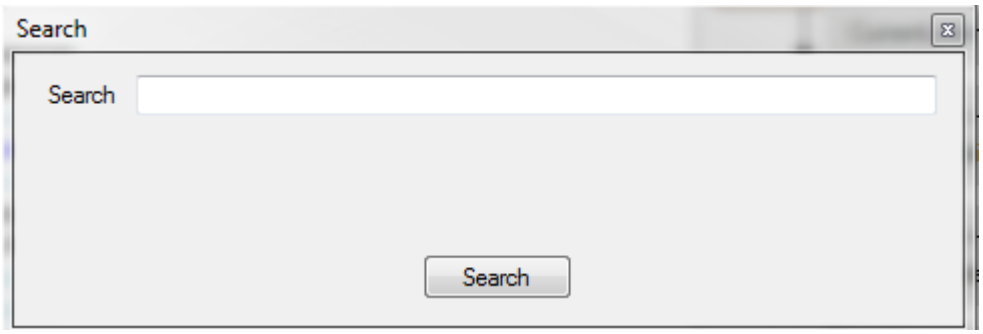

Figure 168: Contextual menu for searching

The elements related with the search criteria are highlighted in blue as shown in Figure 169.

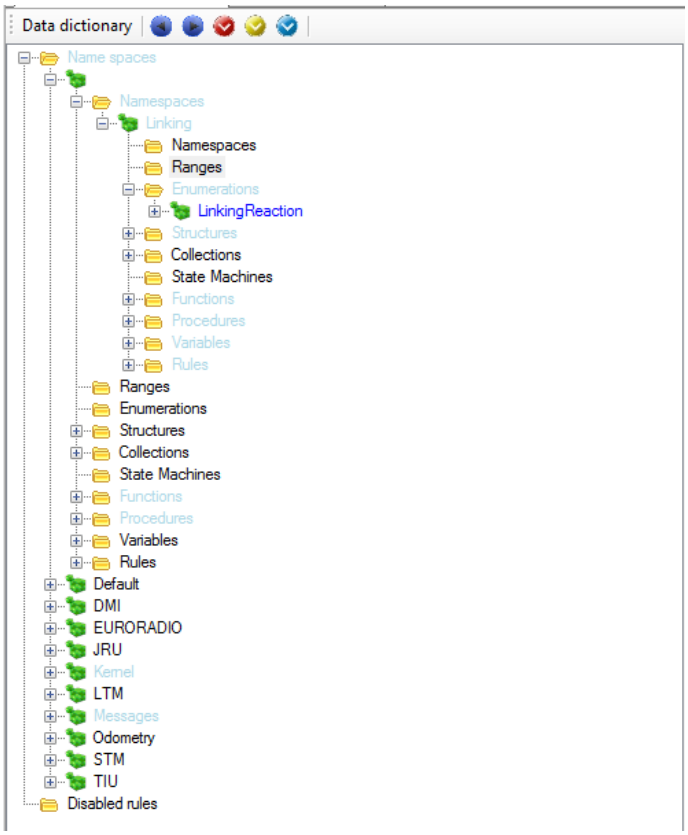

Figure 169: Results of the search feature

11.3 Refresh windows

To refresh the window's contents, use the Tools/Refresh windows. This re-computes and re-draws all windows.

11.4 Options

EFSW behaviour can be configured using the Option dialog box. To open the Options dialog, click on Tools/Options. This dialog allows editing the following options

- **Display all variables in structure editor**: whether the structure editor should display sub-variables whose value is EMPTY.
- **Enclosing messages:** the messages related to the enclosing elements should be displayed when selecting a model element.
- **Requirements as a list:** when set to true, the window only displays the requirement identifier, and otherwise, the window displays the requirement identifier along with the requirement text.
- **Lock opened files:** when set to true, EFS locks the files opened during a session. This forbids external applications, such as GIT or a text editor to access those files, and hence work on an inconsistent set of sources.

12 Shortcuts

The following table summarises the shortcuts available in EFSW.

Shortcut	Meaning	Location
Ctrl+N	New file	File
Ctrl+O	Open a file	File
Ctrl+S	Save modifications	File
Ctrl+P	Print	File
Ctrl+Z	Undo	Edit
Ctrl+Y	Redo	Edit
Ctrl+X	Cut	Edit
Ctrl+C	Copy	Edit
Ctrl+V	Paste	Edit
Ctrl+A	Select all	Edit
Ctrl+E	Show shortcuts view	View
Ctrl+R	Check the model	Tools
Ctrl+D	Check for dead model	Tools
Ctrl+F	Search	Tools
Ctrl+F1	contents	Help

Table 6: Quick access controls

12.1 Auto completion

The expression editor provides a feature to auto-complete the expression. It becomes active when typing "Ctrl+space". It is applicable for all the elements present on the model, and provides the name of the related element. If there are several elements enclosed, the auto completion feature offers a list of possible names. Figure 170 illustrates an example of the auto-completion feature.

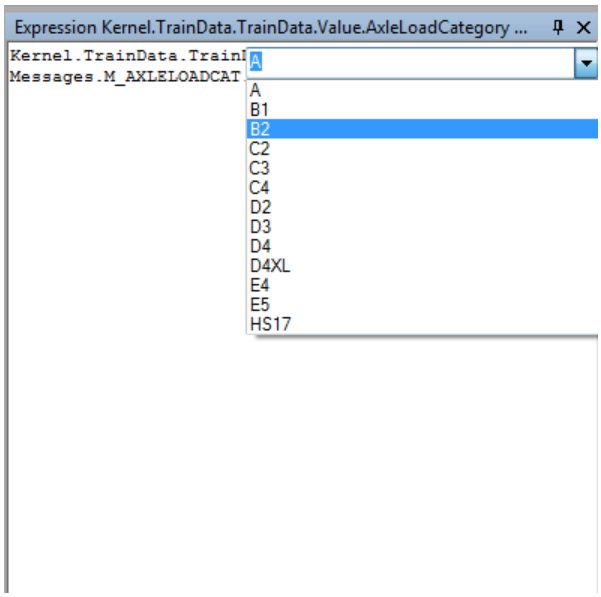

Figure 170: Auto completion feature suggestion list

12.2 Quick navigation to a model element

EFSW allows easy navigation from element usage to definition, using "Ctrl+click" as shown in Figure 171. This is only available in the *Expression editor* and *More info* views.

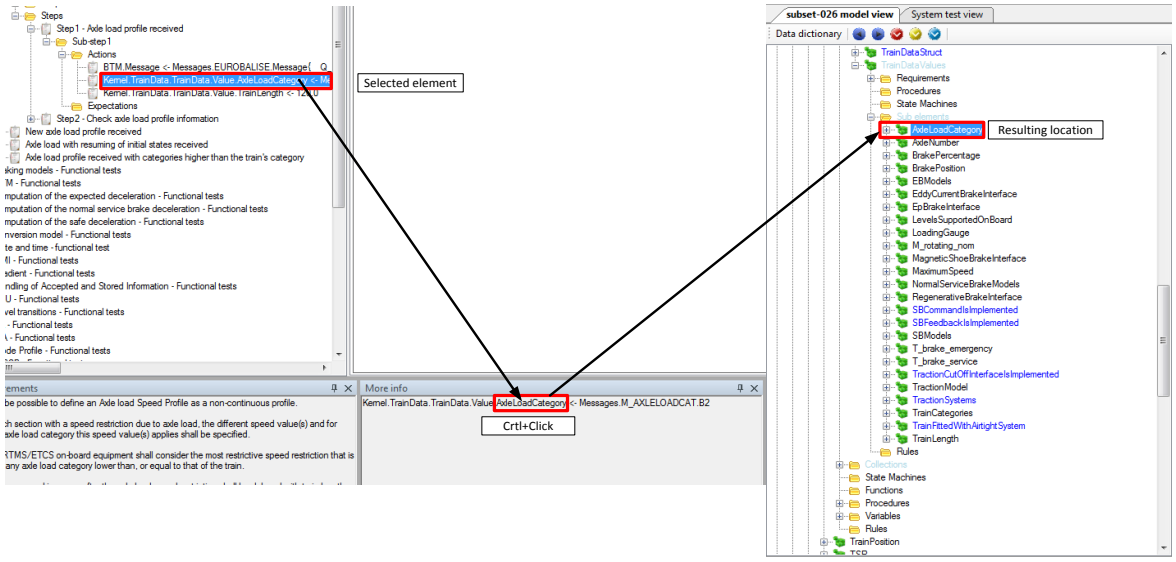

Figure 171: Representation of the Crtl+Click shortcut

Moreover, right clicking on an element in the *More info* window or in the *Expression editor* displays that element's short description, as presented in Figure 172.

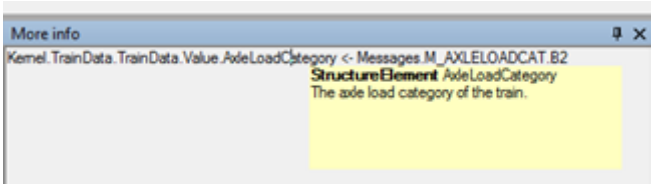

Figure 172: Quick access to the selected element related information

12.3 Undock and dock selected

The model view tab and the system test view tab can be undocked from the EFSW main window. To undock any of these tabs go to *view/undock selected*. Once, one these two windows has been undocked it can be re-docked again by *ctrl+click+drag*. This operation cannot be performed on the side panels; so the specifications cannot be undocked.

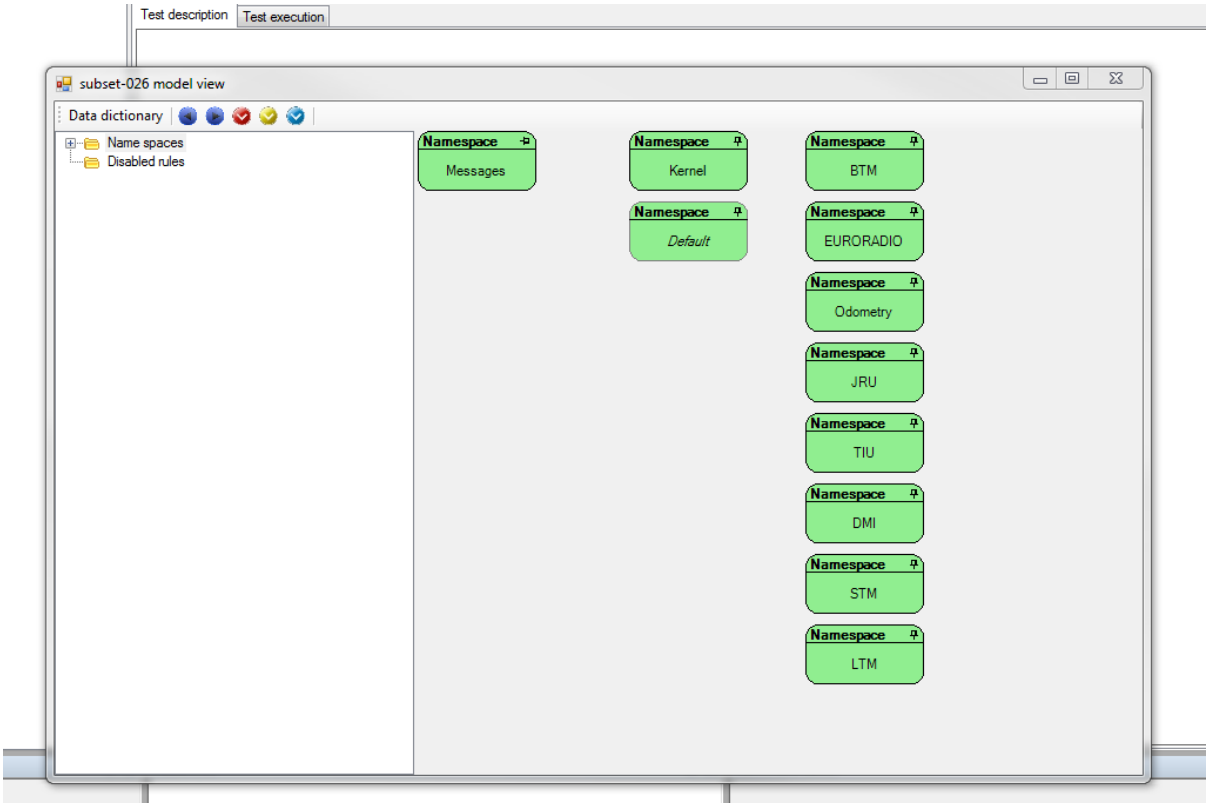

Figure 173: Model view after being undocked

13 EFS Model

The EFS Model is defined by the class diagram provided as annex to this document. This section describes the data structure used by ERTMSFormalSpecs. This structure is persisted in a set of XML files.

13.1 Element Dictionary

The dictionary is the main element of the exported model. This dictionary is composed of the following elements:

- **The *specifications* used to model the system, presented in Section 13.2**
- **The *requirement sets* allowing to classify the requirements, presented in Section 13.3**
- **The ERTMS *namespaces* allowing to group the model elements (e.g. Kernel, DMI, TIU, …), presented in Section 0**
- **The *tests* allowing to verify the model, presented in Section 13.5**
- **The *test translations* allowing to automatically translate tests from Subset-076, presented in Section 13.6**
- **The *shortcuts* providing an easy access to the specific model elements, presented in Section 13.9**

Each element of the dictionary is identified by the following information:

Field	Description
Guid	**the unique generated identifier of the element**
Name	**the name which identifies the element in the dictionary**

13.2 Specifications

The specification related to the model holds the following information:

Field	Description
Chapters	**a list of chapters**
Version	**the version of the baseline**

Each chapter holds a sequence of paragraphs, which hold the following information:

Field	Description
Comment	**a comment related to the paragraph**
ReqRefs	**references to requirements related to that paragraph**
Message	**a message describing the content of variables described in the paragraph, if any**
Subs	**list of sub-paragraphs**
TypeSpecs	**specification of special or reserved values described by the paragraph, if any**
Reviewed	**indicates if the paragraph has been reviewed by the requirement analyst**
InfoRequired	**indicates if the implementation of this paragraph requires some additional information to be completed**
SpecIssue	**indicates if there is an issue in the specification (incoherence, paragraph incomplete, …)**

Status	**the implementation status of the paragraph (N/A, NotImplementable, Implemented, NewRevisionAvailable)**
BI	**the version when the paragraph has been modified last**
Id	**the id of the paragraph, corresponding to the id from the Subset-026**
Name	**the name of the paragraph, needed for the backward compatibility**
Optional	**needed for the backward compatibility**
Scope	**needed for the backward compatibility**
Type	**type of the paragraph (TITLE, DEFINITION, REQUIREMENT, NOTE, DELETED, PROBLEM or TABLE_HEADER)**
Version	**provides the version number of the paragraph. For instance, a version 3.4.0 of Subset-026 can hold a set of paragraph identified as version 3.0.0 if those have not been modified in subsequent releases**
Tested	**indicates if the paragraph is tested**
Onboard	**needed for the backward compatibility**
Trackside	**needed for the backward compatibility**
Rollingstock	**needed for the backward compatibility**
FunctionalBlock	**obsolete. This has been replaced by Requirement sets**
FunctionalBlockName	**obsolete. This has been replaced by Requirement sets**
ParagraphRevision	**holds the text and version of a new revision of the paragraph, which has not yet been integrated**
RequirementSets	**holds the reference of the requirement set in which the paragraph belongs**

13.3 Requirement sets

The requirement sets allow to classify the requirements in logical sets. For example, the requirements can be classified according to their scope (OBU, trackside or rolling stock) or according to their functional block (Procedure Start of Mission, Levels and transitions, System data etc.). The requirement sets hold the following information:

Field	Description
X	**the X coordinate of the corresponding graphical element**
Y	**the Y coordinate of the corresponding graphical element**
Width	**the width of the corresponding graphical element**
Height	**the height of the corresponding graphical element**
RecursiveSelection	**indicates that, when a paragraph is related to this requirement set, all its sub paragraphs are also related to it.**
RequirementsStatus	**obsolete**
Default	**indicates that, when a new paragraph is created, it is automatically related to this requirement set**
Pinned	**provides the status of the pin in the graphical view. To be used in a later version of EFS**
Dependancies	**indicates on which this requirement set is dependant, e.g. the requirement sets that need be completed before trying to model and test this one.**
SubSets	**requirement sets are hierarchical. This provides the sub requirement set of this one.**

13.4 Namespaces

The model is split into several namespaces which can hold sub-namespaces, ranges, enumerations, structures, collections, state machines, functions, procedures, variables and rules, which shall be further described below. Namespaces are identified by a name and contain a comment and the elements related to the graphical information described in Paragraph 13.4.1.

13.4.1 Information common to all model elements

The elements in the model reference requirements for which they are created. They hold the following information:

Field	Description
ReqRefs	**references to requirements related to that model element**
Comment	**comments associated to that model element**
Implemented	**the implementation status**
Verified	**the verification status**
NeedsRequirement	**indicates if this element needs to be attached to a requirement**
X	**the X coordinate of the corresponding graphical element**
Y	**the Y coordinate of the corresponding graphical element**
Width	**the width of the corresponding graphical element**
Height	**the height of the corresponding graphical element**
Hidden	**indicates that the model element is hidden in the graphical view, either completely, or only its relations with other elements, depending on the graphical view used.**
Pinned	**provides the status of the pin in the graphical view. To be used in a later version of EFS**

13.4.2 Types

The following section presents types that can be specified inside a namespace. All the types hold the following information:

Field	Description
Default	**the default value to use when variables of this type are instantiated**

13.4.2.1 Ranges

Ranges allow to specify that a value is integral and has a minimum and a maximum value. They hold the following information:

Field	Description
MinValue	**the minimum value**
MaxValue	**the maximum value**
Precision	**the precision (integer or floating point)**
SpecialValues	**the special values associated to a range; each special value provides a meaningful name to a specific value of the range**

13.4.2.2 Enumerations

An enumeration allows to define the possible (literal) values a variable can take. Enumerations hold the following information:

Field	Description
EnumValues	**the values associated to this enumeration; each one of these values is identified by a name, a value and the information indicating whether the arithmetic operations are forbidden for that value**
SubEnums	**enumerations that share the enclosing enumeration type, but whose range is shorter than the enclosing's**

13.4.2.3 Structures

Structures allow to structure variables using a C-like struct mechanism. They hold the following information:

Field	Description
StructureElements	**elements composing this structure**
Rules	**lists the rules attached to this structure, which will be applied to all instances of the latter; the rules are described in the Section 13.4.6**
StructureProcedures	**lists the procedures related to this structure, which will be executed on each instance of the latter; the procedures are described in the Section 13.4.4**
StructureStateMachines	**lists the state machines defined for this structure; the state machines are described in the Section 13.4.2.5**

The structure elements hold the following information:

Field	Description
TypeName	**the name of the type of this structure element; it can be either a range, an enumeration, a structure, a collection or a state machine**
Mode	**the mode of the structure element, the same as the mode of variables described in the Section 13.4.3**

13.4.2.4 Collections

A collection allows to define a variable which can hold a set of values, all these values must be of the collection's type. Collections hold the following information:

Field	Description
TypeName	**the name of the type of the elements stored in the collection**
MaxSize	**the maximum size of the collection**

13.4.2.5 State machines

State machines allow to define the possible states in which a variable of that type can be. A state machine holds the following information:

Field	Description
States	**the initial state of the state machine**
InitialState	**the set of states defined for this state machine**

Rules	**the rules defining the actions to execute while being in that state machine or the transitions to execute between its states; the rules are described in the Section 13.4.6**

Each state can be further decomposed into another state machine. A state holds the following information:

Field	Description
StateMachines	**the set of sub-state machines**
EnterActions	**the actions to execute while entering that state**
LeaveActions	**the actions to execute while leaving that state**

13.4.3 Variables

The system state is represented in the model using the variables, which can be of any type defined in Section 13.4.2. A variable holds the following information:

Field	Description
DefaultValue	**the default value to use when instantiating this variable; it overrides the type default value**
Type	**the type of the variable**
Mode	**the mode of the variable which can be either** • **Incoming: input variable, provides information to the enclosing system** • **Outgoing: output variable, provides information to the external world** • **In Out: the variable can be modified by the external world and provides information to the enclosing system** • **Internal: internal variable** • **Constant: variable that can be assigned once, then only read**
SubVariables	**the set of sub-variables (in case the variable's type is struct)**

13.4.4 Procedures

Procedures allow to define a specific process, which can be split in several Kernel activations. A procedure holds the following information:

Field	Description
Rules	**lists the rules to be applied when invoking the procedure**
Parameters	**lists the formal parameters of this procedure; during invocation, the actual values of the parameters are associated to each formal parameter**

A procedure parameter is a simple association between a name and a type. It holds the following information

Field	Description
Type	**the type of the parameter**

13.4.5 Functions

Functions are used to compute a value, according to the function's parameters. Each function holds the following information:

Field	Description
Type	**the return type of the function**
Cacheable	**obsolete. Function caching is now performed automatically by EFS.**
Parameters	**lists the formal parameters of this function; during invocation, the actual values of the parameters are associated to each formal parameter**
Cases	**define the values the function can take, depending on conditions over the parameters**

A function parameter is a simple association between a name and a type. It holds the following information:

Field	Description
Type	**the type of the parameter**

The value of a function is expressed in term of the case. A case of a function associates a set of pre-conditions, based on the parameter's value, and an expression which provides the function value in case of the corresponding pre-conditions are satisfied, as defined in Section 13.4.6.1. A case holds the following information:

Field	Description
PreConditions	**the set of pre-conditions**
Expression	**the expression of the case**

13.4.6 Rules

Rules allow to define the behavior of the system. A rule can hold the following information:

Field	Description
Priority	**defines when the rule should be activated** • **Verification: the rule is applied during the verification phase** • **UpdateInternal: the rule is applied to update internal values, to prepare the processing phase** • **Processing: this phase is the main processing phase** • **UpdateOut: during this phase, the OUT variables are updated** • **CleanUp: during this phase, the values of temporal variables are erased**
Conditions	**lists the mutually exclusive cases for this rule: when executing a rule, at most one of these conditions is executed; when two conditions could be executed, the first one takes precedence**
SubRules	**needed for the backward compatibility**

Each RuleCondition is composed of the following items:

Field	Description
PreConditions	**allow to determine when the rule must be activated (see 13.4.6.1)**
Actions	**define the actions to take when the rule condition is activated (see 13.4.6.2)**
SubRules	**lists the sub-rules of the rule condition**

13.4.6.1 Pre-conditions

Pre-conditions are evaluated to check if a rule must be activated. The pre-condition body holds the conditional expression, which should evaluate to a Boolean, see 14.1.1 for more information about available expressions.

13.4.6.2 Actions

Actions are system state modifications which are applied when the rule's pre-conditions are satisfied. Actions body holds the statement to be performed, see 14.1.9 for more information about available statements.

13.5 Tests

13.5.1 Frames

Tests are decomposed into frames. A frame holds the following information:

Field	Description
Cycle duration	**duration, in ms, of the execution of one cycle**
Comment	**comments associated to that frame**
SubSequences	lists the sub-sequences composing that frame

13.5.2 Sub-sequences

Frames are decomposed into sub-sequences. A sub-sequence holds the following information

Field	Description
TestCases	**lists the test cases composing this sub-sequence**
Level	**as stored in Subset-076 database**
Mode	**as stored in Subset-076 database**
D_LRBG	**as stored in Subset-076 database**
NID_LRBG	**as stored in Subset-076 database**
Q_DIRLRBG	**as stored in Subset-076 database**
Q_DIRTRAIN	**as stored in Subset-076 database**
Q_DLRBG	**as stored in Subset-076 database**
RBCPhone	**as stored in Subset-076 database**
RBC_Id	**as stored in Subset-076 database**
Completed	**indicates if the implementation of this sub-sequence is completed**
Comment	**comments associated to this sub-sequence**

13.5.3 Test case

Test cases hold the following information:

Field	Description
Steps	**lists the steps to be executed for that test case**
Case	**test case number, as stored in Subset-076 database**
Feature	**as stored in Subset-076 database**
ReqRefs	**references to requirements related to that test case**
Comment	**comments associated to this test case**
Implemented	**the implementation status**
Verified	**the verification status**
NeedsRequirement	**indicates if this test case needs to be attached to a requirement**

13.5.4 Test step

A test step consists of a specific step to be taken during a test. It holds the following information:

Field	Description
SubSteps	**lists the sub-steps to be executed for this step**
IO	**as stored in Subset-076 database**
LevelIN	**as stored in Subset-076 database**
LevelOUT	**as stored in Subset-076 database**
ModeIN	**as stored in Subset-076 database**
ModeOUT	**as stored in Subset-076 database**
TranslationRequired	**indicates if this step needs to be translated**
Translated	**indicates if this step is translated**
Distance	**indicates the position of the train for that step, as stored in Subset-076 database**
TCS_Order	**as stored in Subset-076 database**
Description	**as stored in Subset-076 database**
Comment	**comments associated to this step. This is the comment that can be edited in EFS**
UserComment	**as stored in Subset-076 database**
Messages	**messages that are sent to/by the train, if any, as stored in Subset-076 database**

13.5.5 Sub-step

A sub-step holds the following information:

Field	Description
Actions	**the set of variable modifications to be taken at the beginning of this sub-step (see 13.4.6.2)**
Expectations	**the set of conditions that must be satisfied to consider this step as successful (see 13.5.5.1)**
Comment	**comments associated to this sub-step**
SkipEngine	**indicates whether the model has to be animated during this sub-step or its animation is postponed until the next sub-step**

13.5.5.1 Expectation

An expectation describes the expected state of a variable to be verified at least once before the end of a test step. An expectation holds the following information:

Field	Description
Blocking	**indicates that the next sub-step cannot be executed until this expectation has not be satisfied; when this flag is set to true, the test scenario is stopped until the expectation is reached, otherwise, the test scenario continues with this expectation pending**
DeadLine	**the time before which this expectation should be satisfied**
Kind	**indicates if the expectation is** • **instantaneous : it has to be satisfied once** • **or continuous : it has to be satisfied continuously until its deadline has been reached**
CyclePhase	**indicates the cycle phase (corresponding to rules priorities, described in Section 13.4.6) during which the expectation has to be satisfied (default value: all cycle phases)**
Value	**holds the expression which should be evaluated to true**
Condition	**if this condition is present, the expectation is only evaluated when the condition is satisfied**
Comment	**comments associated to this expectation**

13.6 Translations

Translations define pattern matching rules and are used to automatically create actions and expectations for sub-steps imported from Subset-076. The translations can be sorted in different translation folders; the set of the folders and the translations constitutes the translation dictionary.

13.7 Translation dictionary

The translation dictionary contains all the elements related to the translations and holds the following information:

Field	Description
Folders	**the set of folders containing translations and/or another translation folders; the folders are identified by their name**
Translations	**the set of translations**

13.8 Translation

The translations hold the following information:

Field	Description
SourceTexts	**the set of source texts (in English) that are translated by this translation**
SubSteps	**the set of sub-steps corresponding to the translation**
Comments	**the translation is only selected for translating a step when one of the source text matches the step description, and either Comments is empty or one of the comments match the step comment.**

13.9 Shortcuts dictionary

The shortcuts allow to access directly to model elements. The shortcut dictionary holds the following information:

Field	Description
Folders	**the set of folders containing each one a set of shortcuts and/or a set of shortcut folders; the folders are identified by their name**
Shortcuts	**the set of shortcuts; a shortcut is identified by its name**

14 Interpretation model

This section presents the interpretation model of the data dictionary.

14.1.1 Expression evaluation

Expressions follow the following grammar[9][10]

Expression$_0$ ***::=*** ***LET*** *Identifier* ***<-*** *Expression$_0$* ***IN*** *Expression$_0$*

Expression$_i$ ***::=*** *Expression$_{i+1}$* ***{*** *Op$_i$ Expression$_i$* ***}?*** $\forall$ *I in 0..5*

Expression$_6$ ***::=*** *DerefExpression*

| *FunctionCall*

| *ListExpression*

| *UnaryExpression*

Note that you can use **=>** instead of **<-** in the LET expression. The following table presents the operators by level

Operator	*Values*
op$_0$	***OR***
op$_1$	***AND***
op$_2$	***==*** | ***!=*** | ***<=*** | ***>=*** | ***not in*** | ***in***
op$_3$	***+*** | ***-***
op$_4$	******* | ***/***
op$_5$	***^***

14.1.1.1 Examples

This section presents simple examples of expression. The formal semantics is provided in the next section.

Example	Description
A + B	Computes the sum of A and B
A + B * C	Computes the sum of A and B multiplied by C
X IN Coll	If Coll references a variable of type collection, this expression checks that X belongs to the collection

[9] Note that this grammar is subject to further evolutions (for instance, to allow more arithmetic operators).

[10] The notation used is the an extended BNF grammar (see for instance http://en.wikipedia.org/wiki/Extended_Backus-Naur_Form) where :

- **::=** defines a new grammar rule
- ***** stands for 0, 1 or more
- + stands for 1 or more
- braces **{ }** are used to group elements together
- **[]** is used to indicate that one of the enclosed element should be chosen (for instance, [0..9] means either 0, 1, 2, … or 9
- ^ is a negation. For instance, [^0..9] means everything but a number.
- **…** is used for lists, and represents 0, 1 or more elements, separated by the list separator
- As a shortcut, literal are presented in green, non terminals are in `blue`, whereas grammar operators are displayed in **bold**

LET X <- 5 IN X + X + 1	The value of this expression is 11, since the value of X is 5

14.1.1.2 Semantics

The semantics of the LET operator is the following

$$\sigma(\textbf{LET}\ Id\ \textbf{<-}\ e1\ \textbf{IN}\ e2\ ) = \sigma(e2) \text{ where } Id = \sigma(e1)$$

This states that to compute the value of a LET expression, one need to compute the value of e2, using the value of the e1 as value for Id.

The semantics of the operator expression is defined by the following rules

$$\sigma(e1\ Op_i\ e2) = \sigma_{opi}(\sigma(e1), \sigma(e2))$$

This simply states that, to compute the value of an expression, one must first evaluate the left part of the expression, then the right part of the expression and combine those two values using the operator.

The semantics of each operator depends on the type it is applied to.

14.1.1.3 Operators for Integers, Ranges, Enumerations, State Machines and Strings

For variables of type Integer, Range, Enumeration, State Machine and String, the semantics of the operators is provided by the following table

Op	Semantics $\sigma_{op}(x, y)$ =	Constraints
OR	*x = true or y = true*	*x.type* = Boolean and *y.type = Boolean* *Note: the OR operator is lazy : if x is true, then y is not evaluated*
AND	*x = true and y = true*	*x.type* = Boolean and *y.type = Boolean* *Note: the AND operator is lazy : if x is false, then y is not evaluated*
==	*x = y*	*x.type = y.type* *Note: this only applies for integer type, enumeration and string*
	$\lvert x - y \rvert < \varepsilon$	*x.type = y.type* *Note: In this case,* ε = 0.000000001 *Note: this only applies for float types.*
!=	**not** $\sigma_{==}(x,y)$	*x.type = y.type*
IN	N/A	*x.type is collection or state machine*
NOT IN	N/A	*x.type is collection or state machine*
+	*x + y*	*x.type = y.type* and (*x.type* is Integer or *x.type = Range)*
-	*x – y*	*x.type = y.type* and (*x.type* is Integer or *x.type = Range)*
*****	*x * y*	*x.type = y.type* and (*x.type* is Integer or *x.type = Range)*
/	*x / y*	*x.type = y.type* and (*x.type* is Integer or *x.type = Range)* and $y \neq 0$
^	x^y	*x.type* is Integer *and y.type* is Integer

14.1.1.4 Operators for collections

For variables of type Collection, the semantics of the operators is provided by the following table

Op	Semantics $\sigma_{op}(x, y)$ =	Constraints
OR	N/A	**N/A**
AND	N/A	**N/A**
==	*x*.Size = *y*.Size and $\forall i : \sigma_{==}(x[i], y[i])$	***x.type* is Collection and *y.type* is Collection**
!=	**not** $\sigma_{==}(x,y)$	***x.type* is Collection and *y.type* is Collection**
IN	$\exists i : \sigma_{==}(x , y[i])$	***y.type* is Collection**
NOT IN	**not** $\sigma_{in}(x, y)$	***y.type* is Collection**
+	N/A	**N/A**
-	N/A	**N/A**
*****	N/A	**N/A**
/	N/A	**N/A**
^	**N/A**	**N/A**

This table states two lists are equal when they hold the same number of elements and each element of one of them can be found at the same location in the second list. The `in` operator is a simple "belongs to" operation between an element and a list.

14.1.2 Unary expression and Term

A term can either be a designator, function call, a string literal, an integer or a list[11].

UnaryExpression ::= NOT (*Expression*$_0$)
| (*Expression*$_0$)
| *StructureExpression*
| *Term*

Term ::= *Designator*
| *String*
| *Integer*
| *Double*
| *List*

The semantics of expressions with parenthesis is straightforward, the next sections will focus on structure expressions and terms.

14.1.3 Structure expression

Structure expression defines an instance of a structure and allows to set the value or that instance's members. It follows the grammar

---

StructureExpression ::= *Expression*$_0$ { *Designator* => *Expression*$_0$,... }

The first expression of the structure expression must reference the structure (defined in the model) to be instantiated. The expressions enclosed within the braces define the value to be set to specific sub-elements of the instance.

Note that you can use the <- operator instead of =>.

14.1.3.1 Examples

This section presents simple examples of expression. The formal semantics is provided in the next section.

Example	Description
Point { Dir => Direction.Left, Position => 1000.0	This expression evaluates to a point, whose direction is set to Left and whose position is set to 1000.0.

[11] Note that while procedure calls are available in EFS, these are not considered as unary expression (or term), since a procedure call does not return a value.

}	The other values that can be stored in a point are set to their default value

14.1.3.2 Semantics

Op	Semantics $\sigma_{op}(x, y)$ =	Constraints
expr{…}	Evaluates to a new instance of the structure identified by σ(expr). All field values are set to their default value.	*σ(expr) references a structure defined in the model*
Id => expr	**Assigns σ(expr) to the field value referenced by *inst*.Id, where *inst* is the instance currently being built**	*Note : all identifier defined in the structure which do not appear in the list are initialized to their default value*

14.1.4 Designator

A designator is used to identify elements of the model. It is a sequence of letters and digits as presented in the following rule

*Designator ::= Letter { Letter | Digit | '_' }**

Finding the model element referenced by a single designator is searched in the following locations

- The parameters of the enclosing function or procedure (the evaluation context stack)
- The variable bound by an enclosing list operator expression
- The instance in which the designator is evaluated
- One of the enclosing scope in which the expression is declared
- The predefined elements defined in the system
- The default namespace of the EFS system

14.1.4.1 Examples

This section presents simple examples of expression.

Example	Description
Dir	Dir is a valid designator. If the instance is the Point defined in the previous section, it will refer to the Dir member of that Point structure
00_Dir	This is not a valid identifier, since it does not start with a letter

14.1.5 DerefExpression

A DerefExpression is used as a scoping mechanism to identify types, variables, functions, procedures or literals. It follows the rule

---

*DerefExpression ::= Expression { . Expression }**

14.1.5.1 Examples

This section presents simple examples of expression. The formal semantics is provided in the next section.

Example	Description
A.B	This expression references the element B located in A
DMI.ActiveDMI().Present	This expression referenced the structure member **Present** in the structure returned by the function call DMI.ActiveDMI()

14.1.5.2 Semantics

Finding the model element referenced by a designator depends on its position in the expression:

- The first element of a sequence of designators separated by the "." operator is searched for at the following locations
  - The parameters of the enclosing function or procedure (the evaluation context stack)
  - The variable bound by an enclosing list operator expression
  - The instance in which the designator is evaluated
  - One of the enclosing scope in which the expression is declared
  - The predefined elements defined in the system
  - The default namespace of the EFS system
- The next elements of the sequence are search in the instance in which the designator is evaluated

14.1.5.3 Function call

The function call follows the grammar

FunctionCall ::= Designator (Expression, …)
FunctionCall ::= Designator (Designator => Expression, …)

Note that you can use the <- operator instead of =>.

14.1.5.4 Examples

This section presents simple examples of expression. The formal semantics is provided in the next section.

Example	Description
DMI.ActiveDMI()	This function does not take any parameter and returns
Kernel.SpeedAtDistance(distance => 1000.0)	This function takes a single parameter, named distance. In this case, the function is evaluated with a parameter distance set to 1000.0
Kernel.SpeedAtDistance(1000.0)	This is exactly the same as the previous call, but the parameters are not named.
Distance(tsr.Length)	Distance is a predefined type in the Default namespace. This converts the length provided into a distance

14.1.5.5 Semantics

Evaluating the value of a function call is performed the following way:

1. **find the function which corresponds to the designator (as presented in Section 14.1.5) and create a new evaluation context stack;**
2. **for each parameter expression, evaluate the expressions and bind a new variable with the value to the corresponding function parameter in the last entry of the evaluation context stack;**
3. **find the first case preconditions (see Section 13.4.6.1) which evaluates to true to determine which expression should be used to evaluate the function call;**
4. **the function call is evaluated to the value of the corresponding case expression;**
5. **remove the last entry from the evaluation context stack.**

Note that, in point 1, the function can either be explicitly defined in the model, or is a conversion function associated to a range. In that case, it shares the name of the range, and follow the prototype

```
Function Range(x : Integer) : Range      (in case of integer range)
Function Range(x : Float) : Range        (in case of float range)
```

It takes any integer as parameters and returns the corresponding range element.

14.1.6 Literal values

Literal values are either string or integer values. They follow the following grammar.

```
Integer ::= 0 [0..9]*
String ::= ' [^']* '
```

14.1.6.1 Examples

This section presents simple examples of expression. The formal semantics is provided in the next section.

Example	Description
1000	This is the integer value 1000
'Start'	This is a string whose value is **Start**

14.1.6.2 Semantics

The semantics of literal values is the corresponding value.

14.1.7 Lists

List values are expressed by enclosing a list of terms (separated by comma) with brackets.

```
List ::= [ Term, … ]
```

14.1.7.1 Examples

This section presents simple examples of expression. The formal semantics is provided in the next section.

Example	Description
[1, 2, 3]	This describes a list with three integer values : 1, 2 and 3
['Start', 'Stop']	This describes a list with two string values : **Start** and **Stop**

14.1.7.2 Semantics

The semantics of a list is the list value which holds the semantics of the terms.

14.1.8 List Expression

A list expression follows the grammar

```
ListExpession ::=
        THERE_IS Designator IN ListValue
        | FORALL Designator IN ListValue
        | FIRST Designator IN ListValue
        | LAST Designator IN ListValue
        | COUNT Designator IN ListValue
        | REDUCE ListValue USING Designator IN Expression INITIAL_VALUE
Expression
        | SUM ListValue | Condition USING Designator IN Expression
        | MAP ListValue USING Designator IN Expression

ListValue ::=  Expression [ | Condition ]?
```

```
Condition ::=  Expression
```

Where the **ListValue** provides a value of type collection, where elements from the Expression (which also evaluates to a list expression) have been filtered out by the condition (if present).

The **Condition** and **USING expressions** can use the bound variable identifier by Designator which references the list element currently being processed. For instance, the expression

```
THERE_IS X IN List | X > 3
```

will be true if there is at least an element in the list which value is above 3 (see below for the operator semantics).

Moreover, the USING expression can use the bound variable named RESULT which is the partial result actually built when considering the current list element. At the start of the evaluation, RESULT holds

- the INITIAL_VALUE for REDUCE expression,
- 0 for SUM expressions
- an empty list for MAP expressions

For instance, the expression

```
REDUCE List
        USING X IN X.Size + RESULT
        INITIAL_VALUE 0
```

would sum the size attribute stored in elements in the list. This expression can be written more concisely using the following expression

```
SUM List USING X IN X.Size
```

14.1.8.1 Examples

This section presents simple examples of expression. The formal semantics is provided in the next section.

Example	Description
THERE_IS X IN Collection	This is true when the collection is not empty
THERE_IS X IN Collection | X.Size > 1000	This expression is true if there is an element in the collection whose size is greater than 1000

FORALL Y IN Collection | Y.Allocated = True	This expression is true if all the element of the collection have the member Allocated set to **True**
FIRST tsr IN TSRs | tsr.Speed > 50.0	Provides the first TSR in the collection whose speed is greater than 50.0
COUNT lx in LXs	Provides the number of lx structures in the LXs collection
SUM LXs USING lx IN 1	This is a bit weird, but it provides the same value as the preceding example.

REDUCE TSRs
  USING tsr
  IN trs.Length + RESULT
  INITIAL_VALUE 0
	This sums all the members Length of the tsrs stored in the collection TSRs

REDUCE TSRs | tsr.Enabled
  USING tsr
  IN trs.Length + RESULT
  INITIAL_VALUE 0
	This sums the length of all elements where the enabled flag is set to true.

SUM TSRs
  USING tsr
  IN tsr.Length
	The provides the same result as the preceding example

MAP TSRs
  USING tsr
  IN tsr.Length
	Provides a new collection holding the tsrs length values

MAP List
  | X.Enabled
  USING X
	Provides a new collection containing only the elements for which the Enabled attribute is true.

MAP TSRs
  USING tsr
  IN ReduceLength(tsr, 50)
	Applies the procedure **ReduceLength** to all element of the collection TSRs and provides the corresponding collection as a result.

14.1.8.2 Semantics

The following table summarizes the list operators and their associated semantics. There operators can only be applied under the following conditions

- **ListValue.Type is Collection.**
- **Condition expression, when defined, evaluate to a Boolean**

Moreover, INITIAL_VALUE and USING expressions must evaluate to the same type for the REDUCE statement.

ListOp	σ_{ListOp} (*List, cond*) =
THERE_IS	$\exists$ i : σ_{cond}(*List*[i])
FORALL	$\forall$ i : σ_{cond}(*List*[i])
FIRST	*List*[i] where $\forall$ j<i : σ_{cond}(*List*[j]) = *False* and σ_{cond}(*List*[i])
LAST	*List*[i] where $\forall$ j>i : σ_{cond}(*List*[j]) = *False* and σ_{cond}(*List*[i])
COUNT	Counts the number of elements in the list
REDUCE	Reduces the collection to a single value, by applying the expression defined in the USING clause on each element of the collection, where X is the value of the current collection element and RESULT the current reduced result (when only part of the list has been processed). The initial value clause is used to initiate the value of RESULT. If a condition is expressed, the source list is first limited to the element matching the condition before applying the USING clause. For instance, the following expression
SUM	Perform a sum over the elements of a list, by applying the USING expression to compute the value of a single element. For instance, the sum of all elements of a list is written
MAP	Creates a new list from the list referenced by the ListValue, by applying the expression in the USING clause on each element on this list. For instance, the expression The MAP expression can also be used to limit a collection to the elements which match a given criteria. For instance, the following expression

14.1.9 Statements

Statements change the state of the system. They follow the grammar

---

Statement ::=

Expression$_0$ <- Expression$_1$

Expression$_0$ (Expression , …)

CollectionStatement

StatementList ::= { Statement ; }+

The first form of statement is a simple variable assignment, whereas the second one is a procedure call.[12] A statement list consists of a list of statement to be executed in sequence.

14.1.9.1 Variable assignation

Assigning a new value for a variable changes the state of the system. After this statement has been executed, the new value of the variable identified by $\sigma(expression_0)$ is $\sigma(expression_1)$.

14.1.9.2 Procedure calls

Executing a procedure call is performed the following way:

1. **find the procedure which corresponds to the $expression_0$ (as presented in Section 14.1.3.2) and create a new evaluation context stack;**
2. **for each parameter expression, evaluate the expressions and bind a new variable with the value to the corresponding procedure parameter in the last entry of the evaluation context stack;**
3. **find the first case preconditions (see Section 13.4.6.1) which evaluates to true to determine which statement should be used to execute the procedure call;**
4. **execute the corresponding statement list;**
5. **remove the last entry of the evaluation context stack.**

14.1.9.3 Collection statements

There are several statements used to manipulate collections

```
CollectionStatement ::=
        | REPLACE Expression IN ListValue BY Expression
        | INSERT Expression IN ListValue WHEN FULL REPLACE Expression
        | REMOVE [FIRST|LAST|ALL] Expression IN ListValue
        | APPLY Statement ON ListValue

ListValue ::=  Expression [ | Condition ]?

Condition ::=  Expression
```

When evaluating the expression, the special iterator variable X can be used to reference the current element in the collection

[12] Note that function calls cannot appear as a statement since in that case, their return value would be lost.

14.1.9.4 REPLACE statement

REPLACE statement replaces the elements that satisfy the condition referenced by Expression in the list referenced by ListValue by the value of the second Expression. For instance, the expression

```
REPLACE X.NID = 4 IN Message.Packets BY HandleThePacket(X)
```

will replace all the elements in the list Message.Packets whose NID is 4 by the value returned by the function HandleThePacket.

14.1.9.5 INSERT statement

INSERT statement inserts a new element referenced by Expression in the list referenced by ListValue. If needed, WHEN FULL REPLACE can be used to specify which existing element has to be replaced by the new element when the list is full. For instance, the expression

```
INSERT CreateNewBaliseGroup()  IN PreviousBaliseGroups
    WHEN FULL REPLACE FarthestBG()
```

will insert an element created by the function CreateNewBaliseGroup in the list PreviousBaliseGroups and if the list is full, it will replace the element given by the function FarthestBG by the new element.

14.1.9.6 REMOVE statement

REMOVE statement removes the first, last or all element(s) satisfying the condition referenced by Expression from the list referenced by ListValue. For instance, the expression

```
REMOVE ALL StoppingConditionSatisfied ( aLX => X ) IN SupervisedLevelCrossings
```

will remove all elements for which the function StoppingConditionSatisfied returned true from the list SupervisedLevelCrossings.

14.1.9.7 APPLY statement

APPLY statement applies a procedure on each element of the list referenced by the ListValue. For instance, the expression

```
APPLY StoreInfo(X) ON Message.Packets | X.NID = 4
```

will call the procedure StoreInfo for each Packet of the variable Message whose NID is 4.

14.1.10 Instantiate a variable

When a variable is instantiated, a default value is provided to that variable. If the default value is defined at the variable declaration, the corresponding expression is evaluated and assigned as the variable's value. If no default expression is available, the default expression provided for the variable's type is used instead.

14.1.11 Predefined functions

This section presents the predefined functions available in the model.

14.1.11.1 AddIncrement

This function responds to the following prototype

AddIncrement (f, Increment): f

It is a higher order function which adds an increment to an existing function, as presented in Subset-026 Paragraph 3.13.9.2.2.

14.1.11.2 AddToDate function

This function responds to the following prototype

AddToDate (StartDate, Increment): date

It adds a number of milliseconds to a date and returns the resulting date (described by the structure *Default.DateAndTime*).

14.1.11.3 Allocate function

This function responds to the following prototype

Allocate (Collection<T>): T

It finds an empty entry in the collection and returns the corresponding entry.

14.1.11.4 Available function

This function responds to the following prototype

Available (Collection<T>): Boolean

It returns true when there is an empty entry in the collection provided as parameter, and false otherwise.

14.1.11.5 Before

This function responds to the following prototype

```
Before (ExpectedFirst, ExpectedSecond, Collection): Boolean
```

It indicates if the expected first entry is before the expected second entry in a collection.

14.1.11.6 CheckNumber function

This function responds to the following prototype

```
CheckNumber (string): Boolean
```

It indicates if the string received as parameter respects the right format (i.e. it is a sequence of numbers followed by a sequence of “F”).

14.1.11.7 DecelerationProfile

This function responds to the following prototype

```
DecelerationProfile (SpeedRestriction, DecelerationFactor): f
```

It creates a new quadratic function which computes the deceleration profile according to a given speed restriction and a deceleration factor function, as presented on the Figure 38 of the Subset-026.

14.1.11.8 Discontinuities function

This function responds to the following prototype

```
Discontinuities(MRSP): Collection<Target>
```

This function computes the discontinuities in the step function representing the MRSP provided as parameter and returns them as a list of structures of type Target. A target is a structure with the following fields

- **Speed: the associated speed (that is, f(x))**
- **Location: the x value where the function is discontinuous**
- **Length: the length of the current step**

This structure is described as *Kernel.SpeedAndDistanceMonitoring.TargetSupervision.Target*. The function MRSP must have the following prototype:

```
MRSP (distance): speed
```

14.1.11.9 DistanceForSpeed

This function responds to the following prototype

```
DistanceForSpeed (f, speed): distance
```

This function computes the distance where the function f will reach the speed provided as parameter.

14.1.11.10 DoubleToInteger function

This function responds to the following prototype

```
DoubleToInteger (Double): Integer
```

Converts a double value received as parameter to an integer value (by rounding it to the nearest integral value).

14.1.11.11 FullDecelerationForTarget function

This function responds to the following prototype

```
FullDecelerationForTarget (Target, DecelerationFactor): f
```

This function computes a deceleration curve defined by *f(distance) = speed*, which crosses the target provided as parameter and finally reaches 0 speed.

14.1.11.12 IntersectAt

This function responds to the following prototype

```
IntersectAt (f1, f2): double
```

This function computes the intersection of a step function (f1) with a curve (f2). The functions f1 and f2 must have the following prototypes:

```
f1 (distance): speed
f2 (speed):    distance
```

This function is used for example to compute the braking to target Indication supervision limit, which is not used for estimated speeds higher than V_MRSP (see Paragraph 3.13.10.4.11 in Subset-026). It allows to determine the location of the Indication supervision limit valid for V_MSRP.

14.1.11.13 Max

This function responds to the following prototype

```
Max (T, T): T
```

It provides the maximum value between two of them.

14.1.11.14 Min

This function responds to the following prototype

```
Min (T, T): T
```

It provides the minimum value between two of them.

14.1.11.15 MinSurface

This function responds to the following prototype

```
MinSurface (f1, f2): f
```

It is a higher order function which takes two functions as parameters and provides the function which minimizes both of them.

14.1.11.16 Not function

This function responds to the following prototype

```
NOT (Boolean): Boolean
```

It computes the negation of the Boolean value provided as parameter

14.1.11.17 Override

This function responds to the following prototype

---

*Override (*Default*, Override):* f

It is a higher order function which overrides the values of a default function by the values of the override (partial) function and returns the corresponding overridden function.

For instance, consider the constant function

f1(x) = 1 for all x

and the partial function

f2(x) = 3 when x > 2 and x < 3

then, the function

Override(f1, f2) = 3 when x > 2 and x < 3

= 1 otherwise

14.1.11.18 RoundToMultiple

This function responds to the following prototype

---

*RoundToMultiple (*double, double*):* double

It rounds the first parameter down to the next lower multiple of the second parameter. This function is needed to implement the requirement 3.11.11.6 of the Subset-026.

For instance,

RoundToMultiple (21.4, 5.0) = 20.0

RoundToMultiple (28.0, 5.0) = 25.0

14.1.11.19 Targets

This function responds to the following prototype

Targets (f): Collection<Target>

This function computes the list of targets of the step function provided as parameter. A target is a structure with the following fields

- **Speed: the associated speed (that is, f(x))**
- **Location: the x value where the function is discontinuous**
- **Length: the length of the current step**

This structure is described as *Kernel.SpeedAndDistanceMonitoring.TargetSupervision.Target*.

The resulting type is a collection of such type. The resulting value combines all the targets for the functions provided as parameter.

14.2 Rule evaluation

Rules are used in EFS to change the state of the system, when specific conditions have been encountered. Processing the system rules is performed in two phases:

1. The system evaluates the set of rules to be activated by considering each rule preconditions (see Section 14.2.1).
2. The state of the system is altered by applying all activated rule's statements. There is no guaranteed rule activation order, hence, two distinct rules are inconsistent when they can be activated at the same time and alter the same model element with different values.

Rule activation has been split into several phases to handle a complete execution cycle, as depicted by the Figure 174.

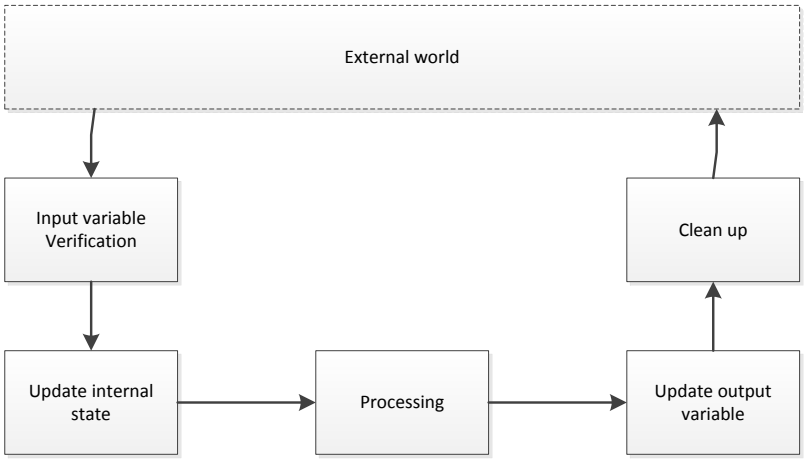

Figure 174 - Processing cycle

The phase "Input variable Verification" is used to ensure the consistency of the input variables and set the system in the right state according to these verifications.

The next phase updates the system state according to the input variables. When this phase is completed, processing can then be performed to handle the core business logic. The system response is performed in the "Update output variable" part of the process, before cleaning up all intermediate results (in the "Clean up" phase). Control is then given back to the enclosing system and the EFS interpreter is ready to process a new cycle.

14.2.1 Rules to be evaluated

Rules are defined at several places in the model. When activating the interpretation machine, the following rules are evaluated to determine the actions to apply.

1. All rules declared in a namespace. Rules are evaluated in that namespace's context.
2. For all variables of type structure: all rules declared in that structure. These rules are evaluated using the corresponding variable as context.
3. For all variables of type state machine: all rules defined in the current state and its enclosing state machines. These rules are evaluated using the corresponding variable as context.
4. If a procedure is called, the rules declared in the procedure are evaluated in sequence.

14.2.2 Evaluating the actions to apply

Evaluating a rule consists of determining the first rule condition which is activated.

A rule condition is activated when all its preconditions evaluate to true.

1. When a rule condition is activated, all its actions are selected to be applied, in a second step. The model element on which the expressions have been computed is kept, to be used in the action application phase.
2. All the sub-rules of the rule condition are evaluated to be activated.

14.2.3 Applying the actions

When the selection phase completes, the statements corresponding to the selected actions are executed to update the model state. Remember that, since there is no guaranteed rule activation order, two distinct rules are inconsistent when they can be activated at the same time and alter the same model element with different values.

14.3 Test execution

14.3.1 Frame

Executing the frame consists of executing all sub-sequences located in that frame. The frame holds an expression which evaluates to the business logic cycle time.

A frame is successful when all the sub-sequences located in that frame are also successful.

14.3.2 Sub-sequence

Executing a sub-sequence consists of executing all test cases located in that sub-sequence.

A sub-sequence is successful when all the test cases located in that sub-sequence are also successful.

14.3.3 Test case

Executing a test case consists of executing all steps located in that test case.

A test case is successful when all the steps located in that test case are also successful.

14.3.4 Step and sub-step

A **step** is composed of several **sub-steps**, each sub-step is composed of

- a sequence of actions. Applying the sub-step on the system consists of applying the action's statement in sequence to modify the system state.
- a sequence of expectation. A sub-step is considered completed when all its blocking expectations have been satisfied or are failed.

Executing a **step** consists of executing its sub-steps in sequence: a sub-step in the sequence can be executed as soon as its previous sub-step is completed.

Interpreting a **sub-step** is performed as follows

1. Apply all actions on the model and add all sub-step expectation to the list of encountered expectations, along with their deadline.
2. If the flag **skip engine** is **not set**, activate the EFS Rule interpretation machine.
3. If the sub-step is completed, continue execution to the next one. A sub-step is considered as completed when
    a. It is marked as skip engine
    b. Or there is no more blocking expectation with an **Unknown** state.

The following table summarizes the computation of an expectation state, according to its Kind, deadline, expression value and previous state.

Kind	Deadline	Value	Previous State	State
Instantaneous	Reached	True	N/A	Success
Instantaneous	Reached	False	N/A	Failed
Instantaneous	Not reached	True	N/A	Success
Instantaneous	Not reached	False	N/A	Unknown
Instantaneous	Not reached	N/A	Success	Success
Continuous	Reached	True	Unknown	Success
Continuous	Reached	False	Unknown	Failed
Continuous	Not reached	True	Unknown	Unknown
Continuous	Not reached	False	Unknown	Failed
Continuous	Not reached	N/A	Failed	Failed

It can be interpreted as following

1. An **instantaneous** expectations is considered Successful when it evaluates to True before its deadline. It is considered as Failed when it never evaluated to true before the deadline is reached.
   As soon as this expectation is evaluated to Successful, it retains that state.
2. A **continuous** expectation is considered Successful when it continuously evaluates to True before its deadline is reached. If the expectation evaluates only once to False before reaching the deadline, it is considered as Failed.

15 Frequently Asked Questions

15.1 Usages view and navigation

The usages window displays the locations where the corresponding element is used, grouped using two different categories:

- Model: other places of the model where the selected element is used.
- Test: displays the test where the selected element of EFSM is used.

To navigate to one of those locations, double click on the left icon.

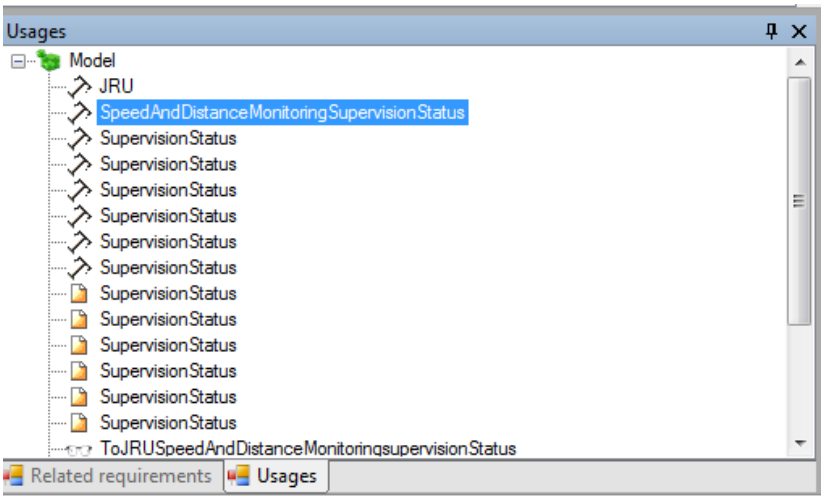

Figure 175: List displaying other locations where an element of the model he is used

15.2 ERTMSFormalSpecs Workbench windows

Sub-windows of EFSW can be moved. To change the location of a sub-window, select it; click and keep clicked on its frame and drag to the desired location. Figure 176 depicts an example to re-allocate a component of the EFSW main window. In this case the selected window is the messages window. Figure 177 depicts the new location of the messages window on EFSW.

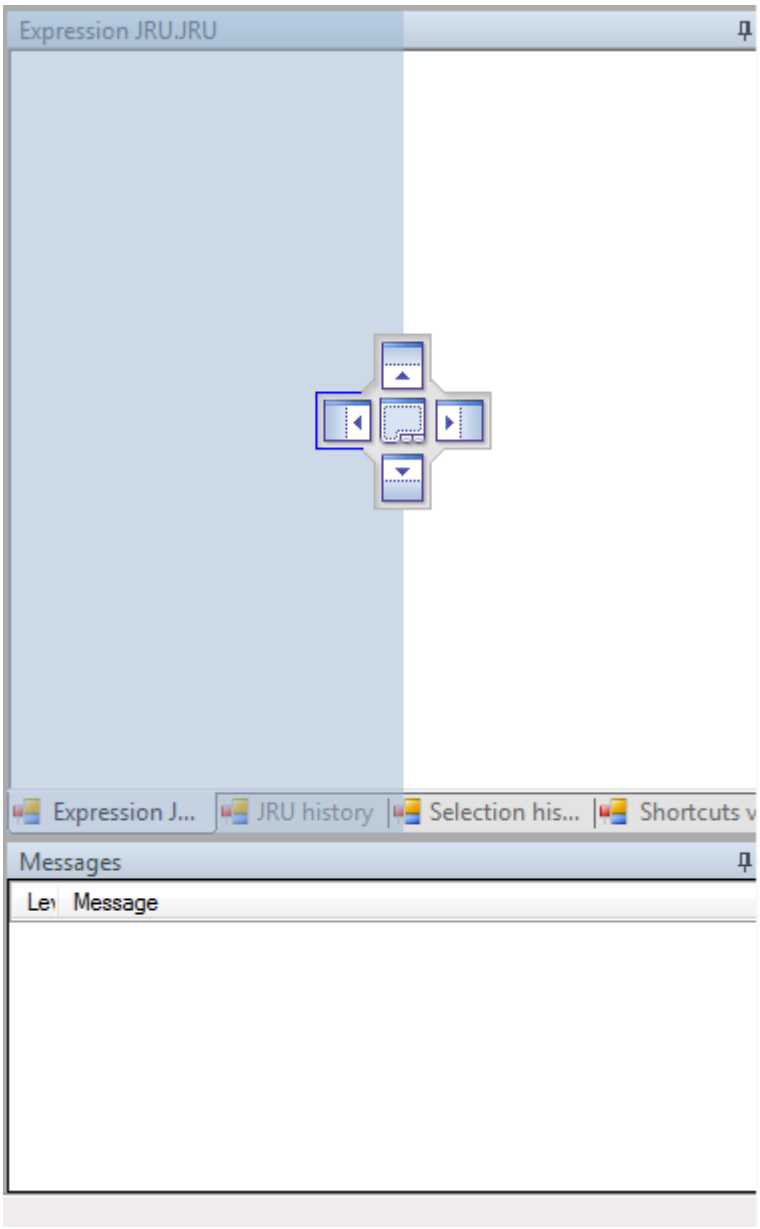

Figure 176: Re-allocation the messages window

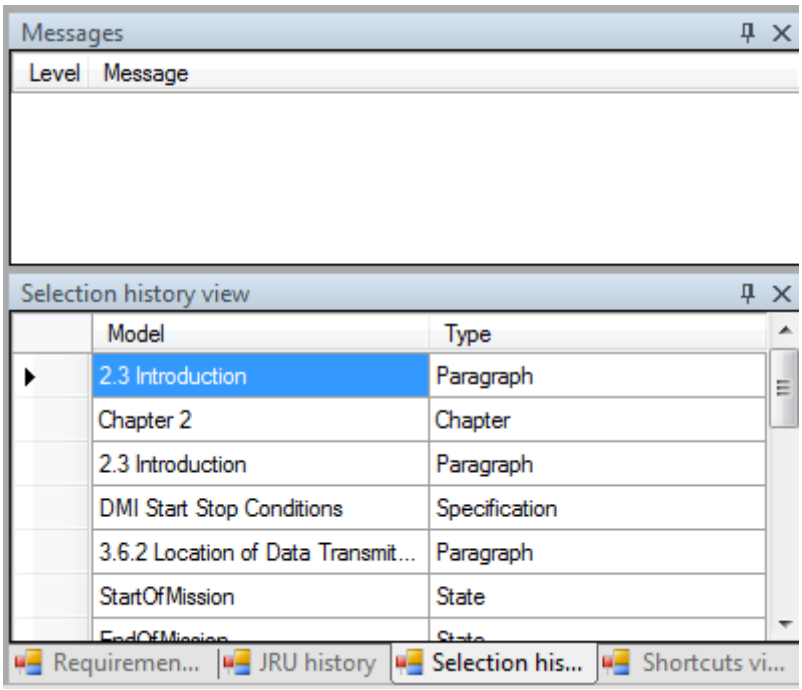

Figure 177: Messages window re-allocation result

16 Table of figures:

17 Index of tables

18 Table of equations