



**Methods for Testing and Specification (MTS);  
The Test Description Language (TDL);  
Reference Implementation**

---

**Reference**

---

RTR/MTS-103119v1.5.1

---

---

**Keywords**

---

language, MBT, testing

---

**ETSI**

---

650 Route des Lucioles  
F-06921 Sophia Antipolis Cedex - FRANCE

---

Tel.: +33 4 92 94 42 00 Fax: +33 4 93 65 47 16

Siret N° 348 623 562 00017 - APE 7112B  
Association à but non lucratif enregistrée à la  
Sous-Préfecture de Grasse (06) N° w061004871

---

**Important notice**

---

The present document can be downloaded from the  
[ETSI Search & Browse Standards](#) application.

The present document may be made available in electronic versions and/or in print. The content of any electronic and/or print versions of the present document shall not be modified without the prior written authorization of ETSI. In case of any existing or perceived difference in contents between such versions and/or in print, the prevailing version of an ETSI deliverable is the one made publicly available in PDF format on [ETSI deliver](#) repository.

Users should be aware that the present document may be revised or have its status changed,  
this information is available in the [Milestones listing](#).

If you find errors in the present document, please send your comments to  
the relevant service listed under [Committee Support Staff](#).

If you find a security vulnerability in the present document, please report it through our  
[Coordinated Vulnerability Disclosure \(CVD\)](#) program.

---

**Notice of disclaimer & limitation of liability**

---

The information provided in the present deliverable is directed solely to professionals who have the appropriate degree of experience to understand and interpret its content in accordance with generally accepted engineering or other professional standard and applicable regulations.

No recommendation as to products and services or vendors is made or should be implied.

No representation or warranty is made that this deliverable is technically accurate or sufficient or conforms to any law and/or governmental rule and/or regulation and further, no representation or warranty is made of merchantability or fitness for any particular purpose or against infringement of intellectual property rights.

In no event shall ETSI be held liable for loss of profits or any other incidental or consequential damages.

Any software contained in this deliverable is provided "AS IS" with no warranties, express or implied, including but not limited to, the warranties of merchantability, fitness for a particular purpose and non-infringement of intellectual property rights and ETSI shall not be held liable in any event for any damages whatsoever (including, without limitation, damages for loss of profits, business interruption, loss of information, or any other pecuniary loss) arising out of or related to the use of or inability to use the software.

---

**Copyright Notification**

---

No part may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm except as authorized by written permission of ETSI.

The content of the PDF version shall not be modified without the written authorization of ETSI.

The copyright and the foregoing restriction extend to reproduction in all media.

© ETSI 2025.  
All rights reserved.

# Contents

Intellectual Property Rights .....	5
Foreword.....	5
Modal verbs terminology.....	5
1     Scope .....	6
2     References .....	6
2.1     Normative references .....	6
2.2     Informative references.....	6
3     Definition of terms, symbols and abbreviations.....	8
3.1     Terms.....	8
3.2     Symbols.....	8
3.3     Abbreviations .....	8
4     Basic Principles .....	9
4.1     Introduction .....	9
4.2     Implementation Scope .....	9
4.3     Document Structure.....	10
5     TDL Toolset .....	10
5.1     Graphical Representation Editor .....	10
5.1.1     Scope and Requirements.....	10
5.1.2     Graphical Editor Architecture.....	11
5.2     Structured Test Objective Representation .....	12
5.3     Implemented Facilities .....	13
5.3.1     Creating Models.....	13
5.3.2     Viewing and Editing Models .....	17
5.3.3     Exporting Structured Test Objectives .....	25
5.3.4     Validating Models .....	27
5.4     Usage Instructions .....	27
5.4.1     Development Environment .....	27
5.4.2     End-user Instructions .....	28
6     Using TDL with External Data Type Specifications.....	29
6.1     Generalized Process .....	29
6.1.1     Process Overview .....	29
6.1.2     Example Instantiation .....	31
7     TDL Runtime/Execution .....	32
7.1     Java™: Code generator .....	32
7.1.1     Architecture .....	32
7.1.2     Test Runtime Interface (TRI).....	33
7.1.2.1     Overview .....	33
7.1.2.2     Interface ProviderModule .....	33
7.1.3     Mappings .....	34
7.1.4     Communication Control Flow .....	35
7.1.5     Executable Code .....	36
8     Web-based Editors and Tools.....	39
8.1     Overview .....	39
8.2     Architecture .....	39
8.3     Evaluation and Recommendations .....	40
8.3.1     Overview .....	40
8.3.2     Custom Application .....	40
8.3.3     Web-based IDE Extension.....	41
8.3.4     Recommendation .....	42
<b>Annex A:     Technical Realisation of the Reference Implementation.....</b>	<b>43</b>

<b>Annex B: UML Profile Editor.....</b>	<b>44</b>
B.1 Scope and Requirements .....	44
B.2 Architecture and Technology Foundation .....	44
B.3 Implemented Facilities .....	44
B.3.1 Applying the Profile .....	44
B.3.2 Hints for the Transformation of UP4TDL Models into TDL Models .....	45
B.3.3 Editing Models with the Model Explorer .....	46
B.3.4 Editing TDL-specific Properties with the TDL Property View .....	46
B.3.5 Editing Models with TDL-specific Diagrams .....	47
History .....	51

---

# Intellectual Property Rights

## Essential patents

IPRs essential or potentially essential to normative deliverables may have been declared to ETSI. The declarations pertaining to these essential IPRs, if any, are publicly available for **ETSI members and non-members**, and can be found in ETSI SR 000 314: *"Intellectual Property Rights (IPRs); Essential, or potentially Essential, IPRs notified to ETSI in respect of ETSI standards"*, which is available from the ETSI Secretariat. Latest updates are available on the [ETSI IPR online database](#).

Pursuant to the ETSI Directives including the ETSI IPR Policy, no investigation regarding the essentiality of IPRs, including IPR searches, has been carried out by ETSI. No guarantee can be given as to the existence of other IPRs not referenced in ETSI SR 000 314 (or the updates on the ETSI Web server) which are, or may be, or may become, essential to the present document.

## Trademarks

The present document may include trademarks and/or tradenames which are asserted and/or registered by their owners. ETSI claims no ownership of these except for any which are indicated as being the property of ETSI, and conveys no right to use or reproduce any trademark and/or tradename. Mention of those trademarks in the present document does not constitute an endorsement by ETSI of products, services or organizations associated with those trademarks.

**DECT™**, **PLUGTESTS™**, **UMTS™** and the ETSI logo are trademarks of ETSI registered for the benefit of its Members. **3GPP™**, **LTE™** and **5G™** logo are trademarks of ETSI registered for the benefit of its Members and of the 3GPP Organizational Partners. **oneM2M™** logo is a trademark of ETSI registered for the benefit of its Members and of the oneM2M Partners. **GSM®** and the GSM logo are trademarks registered and owned by the GSM Association.

NOTE: Eclipse™, Xtext™, Sirius™, EMF™, Papyrus™, GMF™, Epsilon™, EVL™ are the trade names of a product supplied by the Eclipse® Foundation. OMG®, XMI™, UML™, OCL™, MOF™ are the trade names of a product supplied by Object Management Group®. This information is given for the convenience of users of the present document and does not constitute an endorsement by ETSI of the product named.

---

# Foreword

This Technical Report (TR) has been produced by ETSI Technical Committee Methods for Testing and Specification (MTS).

The present document is complementary to the multi-part deliverable covering the Test Description Language (TDL). Full details of the entire series can be found in ETSI ES 203 119-1 [i.13].

---

# Modal verbs terminology

In the present document "**should**", "**should not**", "**may**", "**need not**", "**will**", "**will not**", "**can**" and "**cannot**" are to be interpreted as described in clause 3.2 of the [ETSI Drafting Rules](#) (Verbal forms for the expression of provisions).

"**must**" and "**must not**" are **NOT** allowed in ETSI deliverables except when used in direct citation.

---

# 1 Scope

The present document summarizes technical aspects related to the implementation of TDL within the TDL Open source Project (TOP). It describes the implementation details needed for the further development and integration of the tools.

NOTE: For end-user information on the TOP tool implementation refer to the TOP tool online documentation <https://labs.etsi.org/rep/top/ide/-/wikis/UserScenarios>.

The following tools and components are covered in the present document:

- implementation of the TDL meta-model;
- editor for the graphical representation format of TDL;
- editor for the textual representation format of TDL;
- multiple other types of TDL model editors;
- facilities for checking the semantic validity of models according to the constraints specified in the TDL meta-model;
- implementation of the importing of data definitions from OpenAPI™ and ASN.1 specifications;
- implementation and tool-support for execution of TDL models;
- implementation of the UML profile for TDL; and
- editor supporting the creation and manipulation of UML models applying the UML profile for TDL.

NOTE: The implementation of the UML profile for TDL and the corresponding editor descriptions are not aligned with the referenced versions of the TDL specification parts, but are related to an earlier release of the TDL specification parts.

---

## 2 References

### 2.1 Normative references

Normative references are not applicable in the present document.

### 2.2 Informative references

References are either specific (identified by date of publication and/or edition number or version number) or non-specific. For specific references, only the cited version applies. For non-specific references, the latest version of the referenced document (including any amendments) applies.

NOTE: While any hyperlinks included in this clause were valid at the time of publication, ETSI cannot guarantee their long-term validity.

The following referenced documents may be useful in implementing an ETSI deliverable or add to the reader's understanding, but are not required for conformance to the present document.

- [i.1] Eclipse Foundation™: [Eclipse IDE Website](#) (last visited 12.06.2024).
- [i.2] Eclipse Foundation™: [Eclipse Xtext™ Website](#) (last visited 12.06.2024).
- [i.3] Eclipse Foundation™: [Eclipse Sirius™ Website](#) (last visited 12.06.2024).
- [i.4] Eclipse Foundation™: [Eclipse Modeling Framework \(EMF™\) Website](#) (last visited 12.06.2024).

- [i.5] Eclipse Foundation™: [Eclipse Papyrus™ Modeling Environment Website](#) (last visited 12.06.2024).
  - [i.6] Void.
  - [i.7] Eclipse Foundation™: [Graphical Modeling Framework \(GMF™\) Website](#) (last visited 12.06.2024).
  - [i.8] "[Object Constraint Language™ \(OMG® OCL™\), Version 2.4](#)", formal/2014-02-03.
  - [i.9] Eclipse Foundation™: [Eclipse OCL™ \(Object Constraint Language\) Website](#) (last visited 12.06.2024).
  - [i.10] Plutext Pty Ltd: [Docx4j Website](#) (last visited 12.06.2024).
  - [i.11] OMG®: "[XML™ Metadata Interchange \(XMI®\) Specification](#)", Version 2.4.2, formal/2014-04-04.
  - [i.12] Eclipse Foundation™: [Epsilon™ Validation Language \(EVL™\) Website](#) (last visited 12.06.2024).
  - [i.13] ETSI ES 203 119-1: "Methods for Testing and Specification (MTS); The Test Description Language (TDL); Part 1: Abstract Syntax and Associated Semantics".
  - [i.14] ETSI ES 203 119-2: "Methods for Testing and Specification (MTS); The Test Description Language (TDL); Part 2: Graphical Syntax".
  - [i.15] ETSI ES 203 119-3: "Methods for Testing and Specification (MTS); The Test Description Language (TDL); Part 3: Exchange Format".
  - [i.16] ETSI ES 203 119-4: "Methods for Testing and Specification (MTS); The Test Description Language (TDL); Part 4: Structured Test Objective Specification (Extension)".
  - [i.17] ETSI ES 203 119-5: "Methods for Testing and Specification (MTS); The Test Description Language (TDL); Part 5: UML Profile for TDL".
  - [i.18] ETSI ES 203 119-6: "Methods for Testing and Specification (MTS); The Test Description Language (TDL); Part 6: Mapping to TTCN-3".
  - [i.19] Void.
  - [i.20] ETSI ES 203 119-8: "Methods for Testing and Specification (MTS); The Test Description Language (TDL); Part 8: Textual Syntax".
  - [i.21] Void.
  - [i.22] The Apache® Software Foundation: [Apache POI™ Website](#) (last visited 12.06.2024).
  - [i.23] ETSI: [The TDL Website](#) (last visited 12.06.2024).
  - [i.24] ETSI: [The TDL Open Source Project Website](#) (last visited 12.06.2024).
  - [i.25] ETSI TS 136 321: "LTE; Evolved Universal Terrestrial Radio Access (E-UTRA); Medium Access Control (MAC) protocol specification (3GPP TS 36.321)".
  - [i.26] Void.
  - [i.27] Void.
  - [i.28] [Javadoc documentation generator for Java™](#).
- NOTE: Java™ is the trade name of a programming language developed by Oracle Corporation. This information is given for the convenience of users of the present document and does not constitute an endorsement by ETSI of the programming language named. Equivalent programming languages may be used if they can be shown to lead to the same results."
- [i.29] [JUnit testing framework](#).
  - [i.30] [Guice dependency injection framework](#).

- [i.31] [OpenAPI™ Specification, Version 3.1.1.](#)
- [i.32] ETSI EG 203 647 (V1.1.1): "Methods for Testing and Specification (MTS); Methodology for RESTful APIs specifications and testing".
- [i.33] Void.
- [i.34] Recommendation ITU-T X.680 (02/2021): "Information technology - Abstract Syntax Notation One (ASN.1): Specification of basic notation".
- [i.35] Void.
- [i.36] Recommendation ITU-T X.681 (02/2021): "Information technology - Abstract Syntax Notation One (ASN.1): Information object specification".
- [i.37] Void.
- [i.38] Void.
- [i.39] ETSI ES 203 119-9: "Methods for Testing and Specification (MTS); The Test Description Language (TDL); Part 9: Test Runtime Interfaces".

---

## 3 Definition of terms, symbols and abbreviations

### 3.1 Terms

For the purposes of the present document, the following terms apply:

**abstract syntax:** graph structure representing a TDL specification in an independent form of any particular encoding

**concrete syntax:** particular representation of a TDL specification, encoded in a textual, graphical, tabular or any other format suitable for the users of this language

**meta-model:** modelling elements representing the abstract syntax of a language

**System Under Test (SUT):** role of a component within a test configuration whose behaviour is validated when executing a test description

**TDL model:** instance of the TDL meta-model

**TDL specification:** representation of a TDL model given in a concrete syntax

### 3.2 Symbols

Void.

### 3.3 Abbreviations

For the purposes of the present document, the following abbreviations apply:

API	Application Programming Interface
ASN.1	Abstract Syntax Notation One
EBNF	Extended Backus-Naur Form
EMF	Eclipse Modelling Framework
EVL	Epsilon Validation Language
GMF	Graphical Modelling Framework
MOF	Meta-Object Facility
OCL	Object Constraint Language
OMG	Object Management Group®
SUT	System Under Test



TDL	Test Description Language
TOP	TDL Open Source Project
UML	Unified Modelling Language
URI	Unified Resource Identifier
XMI	eXtensible Markup Language Metadata Interchange

## 4 Basic Principles

### 4.1 Introduction

To accelerate the adoption of TDL, an implementation of TDL is provided within TOP in order to lower the barrier to entry for both users and tool vendors in getting started with using TDL. The implementation comprises graphical and textual editors, as well as validation facilities, transformation functionalities, and other tools. In addition, the UML profile for TDL and supporting editing facilities are implemented in order to enable application of TDL in UML-based working environments and model-based testing approaches.

### 4.2 Implementation Scope

The implementation scope includes a graphical editor according to ETSI ES 203 119-2 [i.14] based on the Eclipse platform [i.12] and related technologies, covering essential constructs of TDL. For creating and manipulating models, textual editor for ETSI ES 203 119-8 [i.20] is implemented based on the Eclipse platform and related technologies. The applicability of general-purpose model editing facilities provided by the Eclipse platform and related technologies is discussed as well.

For tools that need to import and export TDL models according to ETSI ES 203 119-3 [i.15], corresponding facilities are implemented based on the Eclipse platform and related technologies. These facilities can be used to transform textual representations based on ETSI ES 203 119-8 [i.20] and ETSI ES 203 119-1 [i.13] into XMI [i.11] serializations according to ETSI ES 203 119-3 [i.15] and can be integrated in custom tooling that builds on the Eclipse platform.

An implementation of ETSI ES 203 119-4 [i.16] includes a dedicated textual editor for structured test objectives, which can be integrated in the textual editor for TDL. The implementation also includes facilities for exporting structured test objectives to Word™ documents using customisable tabular templates.

An implementation of the UML profile for TDL includes a specification of the TDL UML profile abstract syntax according to the mapping from the TDL meta-model to TDL stereotypes and UML meta-classes in ETSI ES 203 119-5 [i.17]. It is integrated with the open-source UML modelling environment Eclipse Papyrus [i.5] as an open TDL UML profile implementation.

An implementation of ETSI ES 203 119-6 [i.18] includes a partial prototypical implementation of the TDL to TTCN-3 mapping based on the Eclipse platform.

Additional functionalities supporting the importing of data definitions from OpenAPI™ [i.31] and ASN.1 [i.34] and [i.36] specifications are also provided as a prototype.

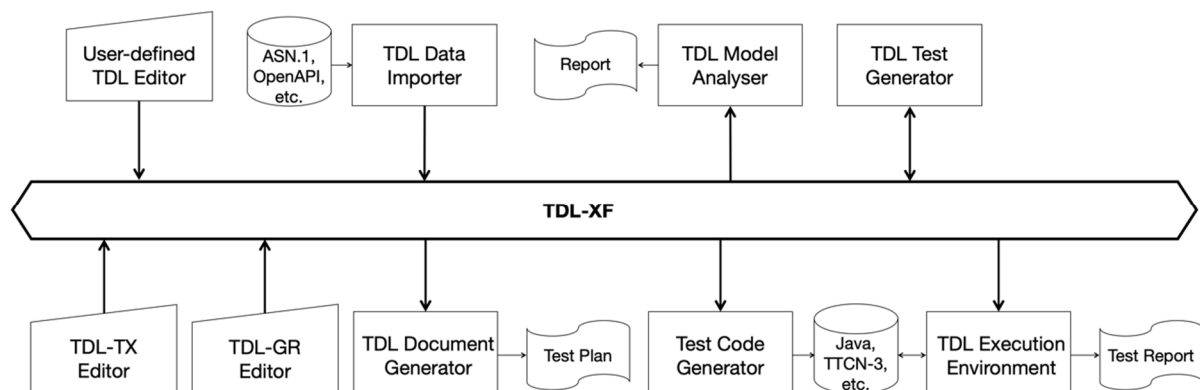


Figure 4.2-1: TDL tool infrastructure

A schematic overview of the implementation is shown in Figure 4.2-1. The TDL exchange format specified in ETSI ES 203 119-3 [i.15] serves as a bridge between the different tool components. Textual editors enable the creation and manipulation of TDL models. Data importers enable the integration and use of existing data specifications in TDL. The graphical editor is used to edit and visualize TDL models as diagrams. Documentation generation, in particular for structured test objectives, can be plugged in to produce Word documents for presenting parts of a TDL model in a format suitable for standardization documents. Test code generation, e.g. for TTCN-3 can be plugged in to produce executable TTCN-3 code or TTCN-3 skeletons to be refined afterwards.

The implementation is published as part of the TOP [i.24] on the TDL [i.23].

## 4.3 Document Structure

The present document contains three main technical clauses focusing on relevant technical details. The Graphical Representation editor implementing ETSI ES 203 119-2 [i.14], as well as related facilities implementing ETSI ES 203 119-1 [i.13], ETSI ES 203 119-3 [i.15] and ETSI ES 203 119-4 [i.16] are described in clause 5. Implementation details for using TDL with external data type specifications are covered in clause 6 of the present document. The implementation of an execution environment for the testing of RESTful API services with TDL (e.g. as described in ETSI EG 203 647 [i.32]) is outlined in clause 7. The findings of a feasibility study of a web-based version of the TOP tool implementation is described in clause 8.

An UML Profile Editor implementing ETSI ES 203 119-5 [i.17] is described in annex B.

NOTE: The UML Profile Editor for TDL complies to an earlier release of the TDL specification parts.

---

# 5 TDL Toolset

## 5.1 Graphical Representation Editor

### 5.1.1 Scope and Requirements

TDL graphical editor implementation has two major requirements. The main objective is to provide means to visualize TDL models according to the graphical notation. The second objective is to facilitate layout of diagrams in a way that is suitable for documentation. For the second purpose, it is essential to provide graphical editing capabilities. Although often provided by modelling frameworks, the ability to graphically edit the underlying models (that is, to create new elements and set their properties) is not considered essential for this implementation.

Eclipse provides several graphical modelling tools to help build editors. Sirius [i.3] was chosen for its declarative approach that provides separation between meta-model mappings and implementations of graphical elements. With the existence of predefined common graphical elements, such as containers and connectors, the effort of implementing a graphical editor with a custom syntax in Sirius is only spent on the parts that diverge from those common elements.

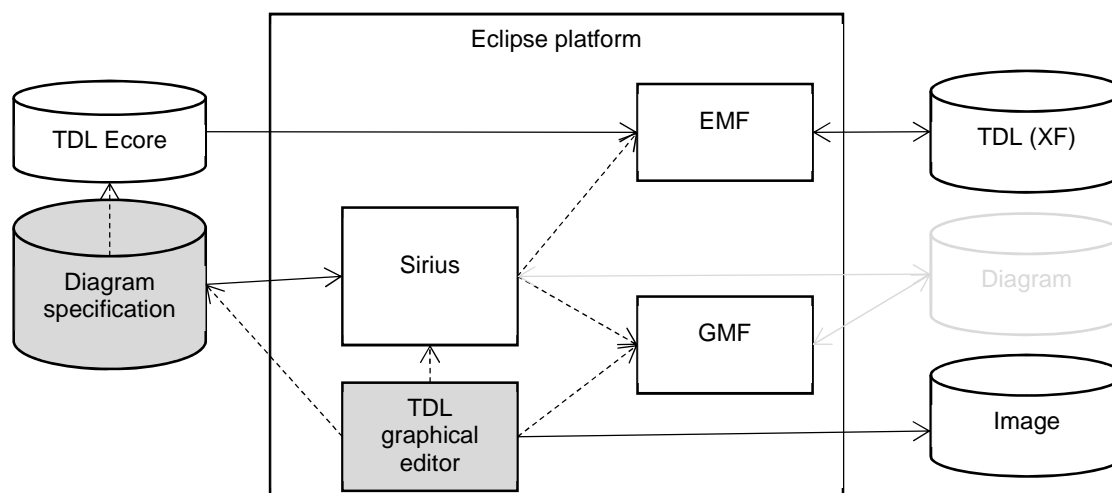
Another area that requires a custom implementation is the layout of graphical elements. This covers both the absolute placement of nodes on the diagram as well as the size and internal contents of each node. Due to the rather hierarchical nature of the TDL graphical syntax, several additional base graphical elements are introduced. Some peculiar limitations of Sirius have also been identified prior to the implementation, which also need appropriate workarounds. The goal of implementing a diagram layout is to automate diagram creation to the extent that the sizes and contents of graphical elements are adjusted by layout algorithms while the absolute placement of diagram elements is solved by using built in layout implementations. This will guarantee that only minimal user interaction with the diagram editor is needed for achieving the desired layouts.

Diagram export for documentation purposes is provided by the framework. The implementation can provide complimentary export to the Word® document format.

Due to the peculiarities and intended use of structured test objectives, it was determined that instead of graphical shapes that can be exported as images, the graphical representation is realized as tables exported directly in a Word document according to user-defined templates. These tables can then be manipulated further as necessary to fit in within an existing document.

## 5.1.2 Graphical Editor Architecture

The TDL graphical editor is built on top of the Eclipse platform to benefit from its wide range of modelling tools. The main Eclipse projects that are used as basis for this implementation are shown in Figure 5.1.2-1. Sirius is a technology that allows declarative creation of graphical editors that work with EMF models. It uses GMF [i.7] to create visual diagram elements and link those to model objects. Model management and serialization is done by EMF [i.4].



NOTE: Components with grey background are part of the implementation that is covered by the present document.

**Figure 5.1.2-1: Dependencies and data flows of the TDL graphical editor**

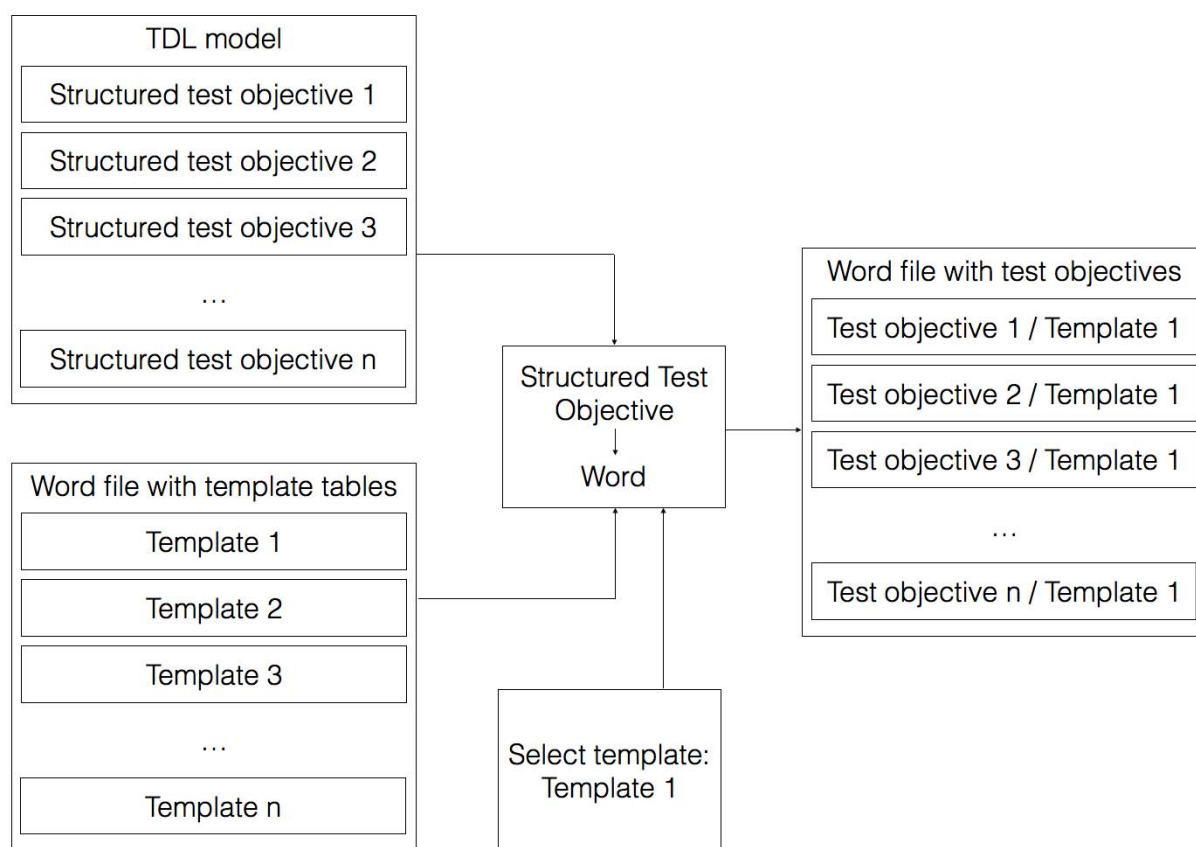
Every EMF model is based on a meta-model that is defined in terms of meta-modelling system named Ecore. The TDL meta-model in UML format was converted to an Ecore meta-model (TDL Ecore) using the Papyrus UML and EMF facilities. Furthermore, Java™ code for the TDL meta-model was generated based on the TDL meta-model.

Sirius creates diagram editors by interpreting diagram specification files. These files contain TDL meta-model references in the form of Java or OCL [i.8] queries. OCL support is provided by the Eclipse OCL project [i.9], Java queries are references to classes that are part of the TDL graphical editor and editor source code. Diagram specifications also contain definitions of Sirius specific styles that are applied to model objects when rendering them on diagrams. Since the TDL graphical editor requires customized shapes, it has dependencies on both the Sirius API and the Eclipse GMF. Several extensions to GMF classes have been implemented in Sirius in order to configure shapes according to the customized styles. GMF facilities are then used to export the diagrams as images.

Some of the labels in the graphical shapes, in particular labels related to data specification and data use have a complex structure. For their realization, facilities provided by Xtext [i.2] are used to serialize model fragments related to data use as text according to an annotated EBNF grammar derived from the formal label specifications in ETSI ES 203 119-2 [i.14].

## 5.2 Structured Test Objective Representation

Structured test objectives are exported as tables in a Word document according to user-defined templates. The export relies on facilities provided by Xtext as well as the Apache POI library [i.22] (previously the Docx4j library [i.10] was used) providing API for manipulating Word documents. The exporting facilities take a Word document containing one or more templates in the form of tables with placeholders and a TDL model containing one or more structured test objectives as input. The user has to provide the name of the desired template as an additional input. For a given TDL specification, the selected template is used to generate a tabular representation for every structured test objective. The placeholders in the template are replaced by the content serialized from the corresponding TDL element according to Xtext mappings in a similar manner as the labels for the TDL graphical editor. Existing packaging structures within the TDL specification are used to organize the generated tabular representations with corresponding headings. The generation process is sketched in Figure 5.2-1. The generated tables in the new Word document can be further manipulated or merged into an existing document containing additional information. Additional templates may be defined by the users to suit their specific needs.



**Figure 5.2-1: Structured test objective generation process**

## 5.3 Implemented Facilities

### 5.3.1 Creating Models

#### Overview

Model instances are the primary artefacts for TDL. They carry the semantic information. In a modelling environment there are various means for creating, viewing, and manipulating model instances of a particular meta-model. Comprehensive modelling environments typically provide generic facilities that enable working with model instances of arbitrary meta-models, provided the meta-model is known. Generic facilities provide sufficient capabilities for performing basic tasks on model instances. However, due to their generic nature, they are often cumbersome to work with, lack support for certain features that are not expressed in the meta-model directly (unless customized), and do not provide domain-specific features, such as syntactical customization beyond basic adaptations.

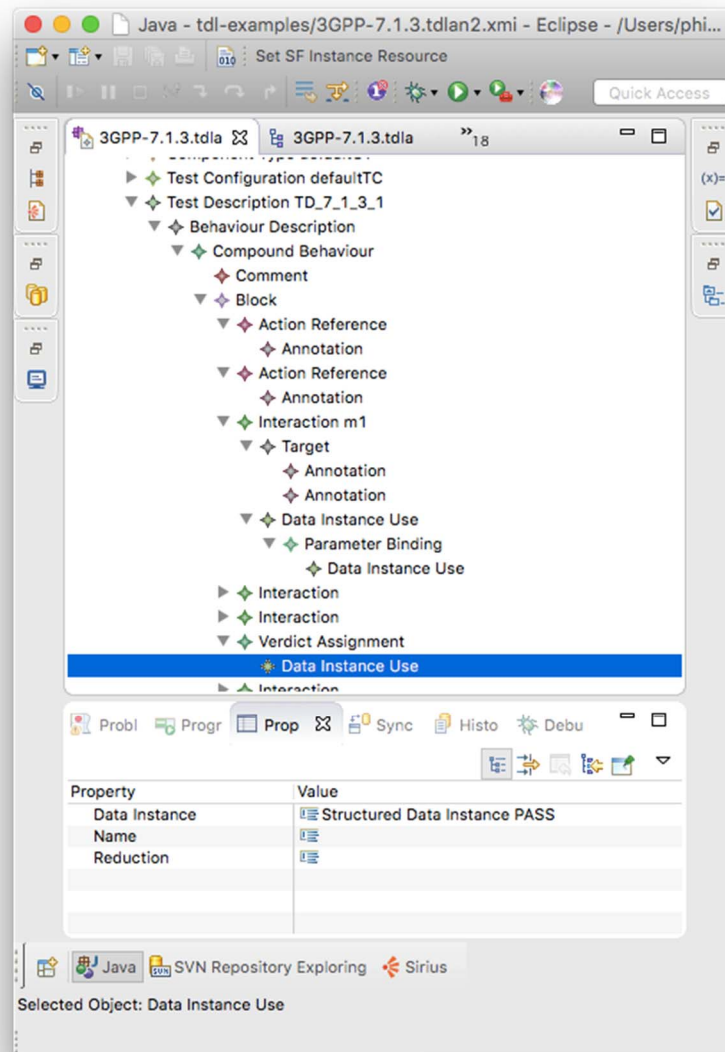
Custom syntax implementations address some of the shortcomings of generic model editors. Such implementations enable the specification of a customized representation of a model instance in a format that is tailored to a specific group of users. There may be multiple custom syntax implementations mapped to the same meta-model, serving different stakeholders or even different purposes for the same stakeholder. Custom syntax implementations may cover only a subset of the meta-model, restricting the access to certain features that are not relevant for specific stakeholders or purposes. Modelling environments provide platforms for the realization of custom syntax implementations. Custom syntax implementations may rely on secondary artefacts that store the concrete representation of the TDL model instance.

TDL model instances may be produced automatically by tools. The exchange format for TDL enables the interoperability of tools producing model instances and tools for manipulating model instances.

#### Generic Model Editors

The EMF provides facilities for generating basic tree editors for a given meta-model, which can then be customized to an extent while still remaining within the tree editor paradigm. In addition, the EMF also provides generic reflective model editors which provide quick access to model instances of any meta-model. An example of such an editor for TDL is shown in Figure 5.3.1-1. The example includes a tree-based editor for manipulating the overall structure of a model on top and a detailed property view for manipulating individual properties on the bottom.

Extensions to the EMF platform, such as MoDisco, include additional generic facilities such as the MoDisco Model Browser which provides faceted browsing and editing of model instances. Faceted browsing provides filtering by type, as well as deep navigation across references. In addition, MoDisco also includes tabular views on different parts of the model for a quick overview across multiple dimensions. An example for a TDL model is illustrated in Figure 5.3.1-1. The example includes a faceted browser on the top for navigating and manipulating the overall structure of a model, as well as individual properties of model elements. On the left side of the faceted browser, model elements can be filtered by type. Below the faceted browser, a tabular editor provides more compact representation of multiple model elements at the same level in a model tree, such as the behaviour elements of a block. The property view on the bottom part of the example still allows the manipulation of properties of selected model elements.



**Figure 5.3.1-1: Example of reflective model editor**

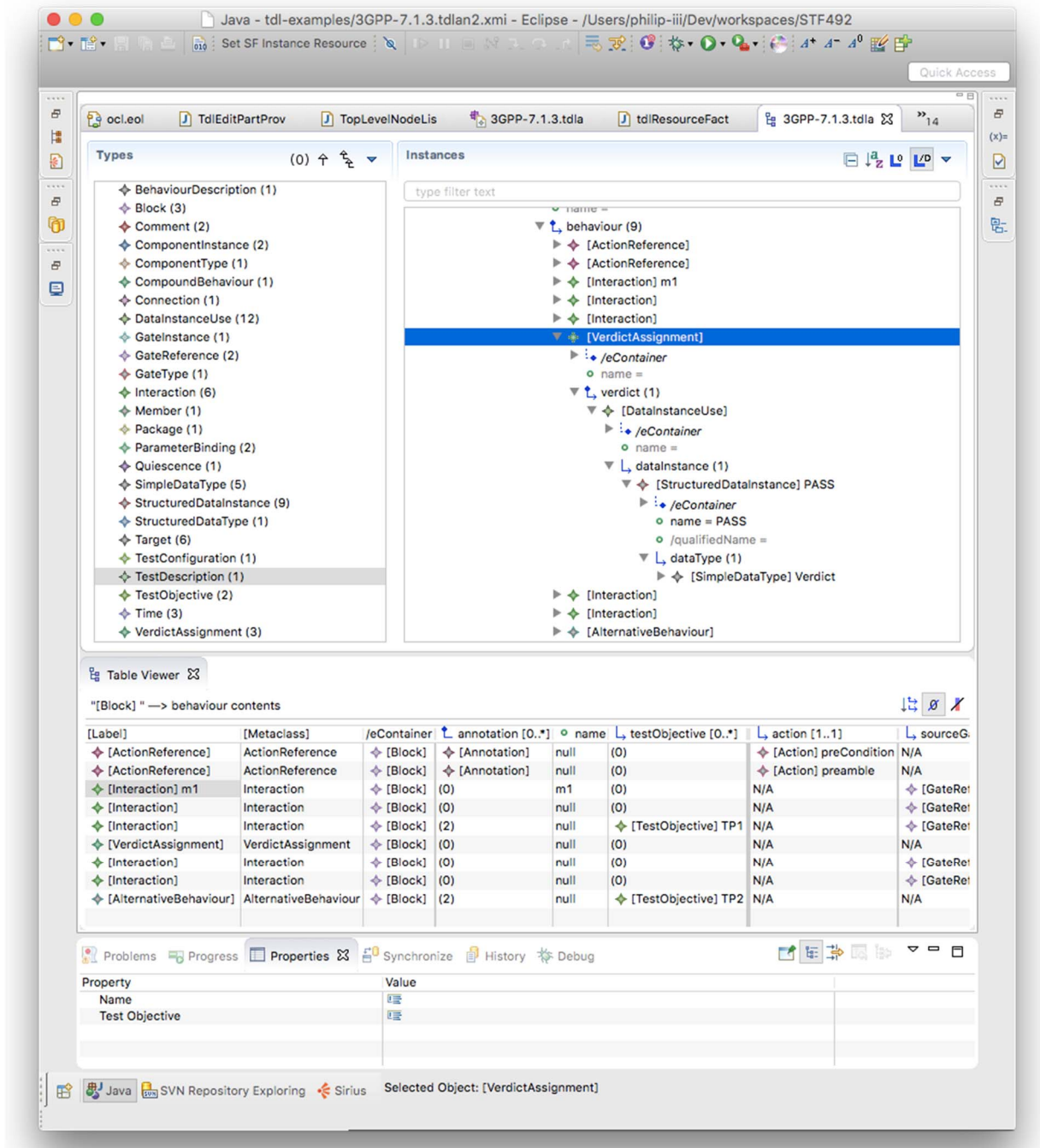
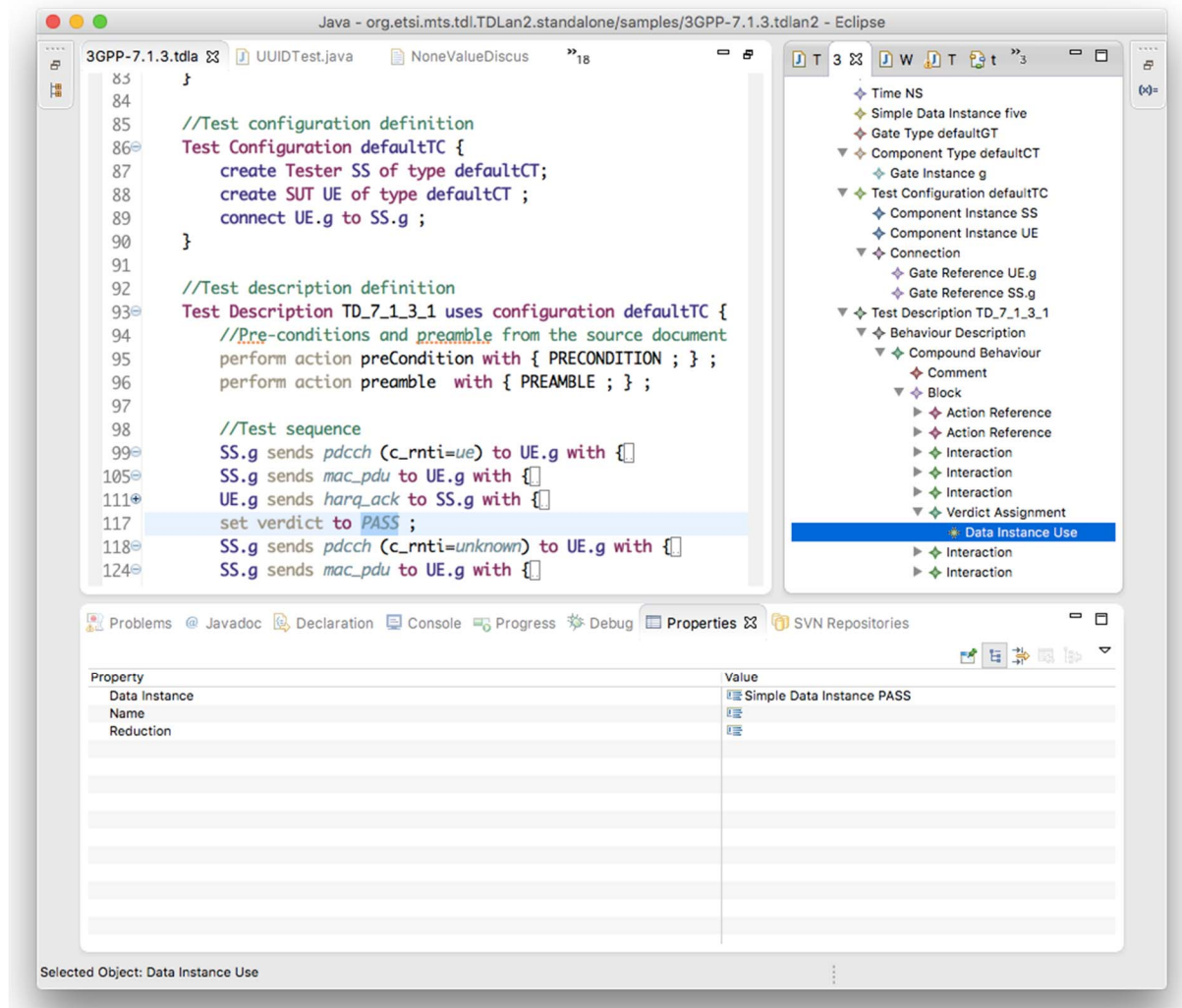


Figure 5.3.1-2: Example of MoDisco faceted model browser

## Textual Editor

Xtext [i.1] provides facilities for the automatic generation of a default textual syntax. It serves as the base for further refinements resulting in customized syntax definitions. Due to it being automatically generated, it is very similar in structure to the meta-model. As a consequence, it is also rather cumbersome to write actual test descriptions in the default syntax notation.

The TOP includes a customized textual syntax that implements the syntax from annex B of ETSI ES 203 119-1 [i.13] (up to version 1.7.1) and from ETSI ES 203 119-8 [i.20]. Apart from the grammar specification, it also includes further customizations in the scoping and linking facilities for handling references, imports, and other peculiarities, as well as enhanced semantic syntax highlighting which provides customisable styles for identifiers based on their type and usage. An example of the customized editor is shown in Figure 5.3.1-3. It features a textual representation of a test description as well as linked tree-based editor showing the same model instance in the tree-based paradigm. Current version of the grammar specification and the additional customizations can be found in annex A of the present document as part of the 'org.etsi.mts.tdl.TDLan2\*' projects for the syntax from annex B of ETSI ES 203 119-1 [i.13] (up to version 1.7.1) and as part of the 'org.etsi.mts.tdl.TDLtx\*' projects for the syntax from ETSI ES 203 119-8 [i.20].

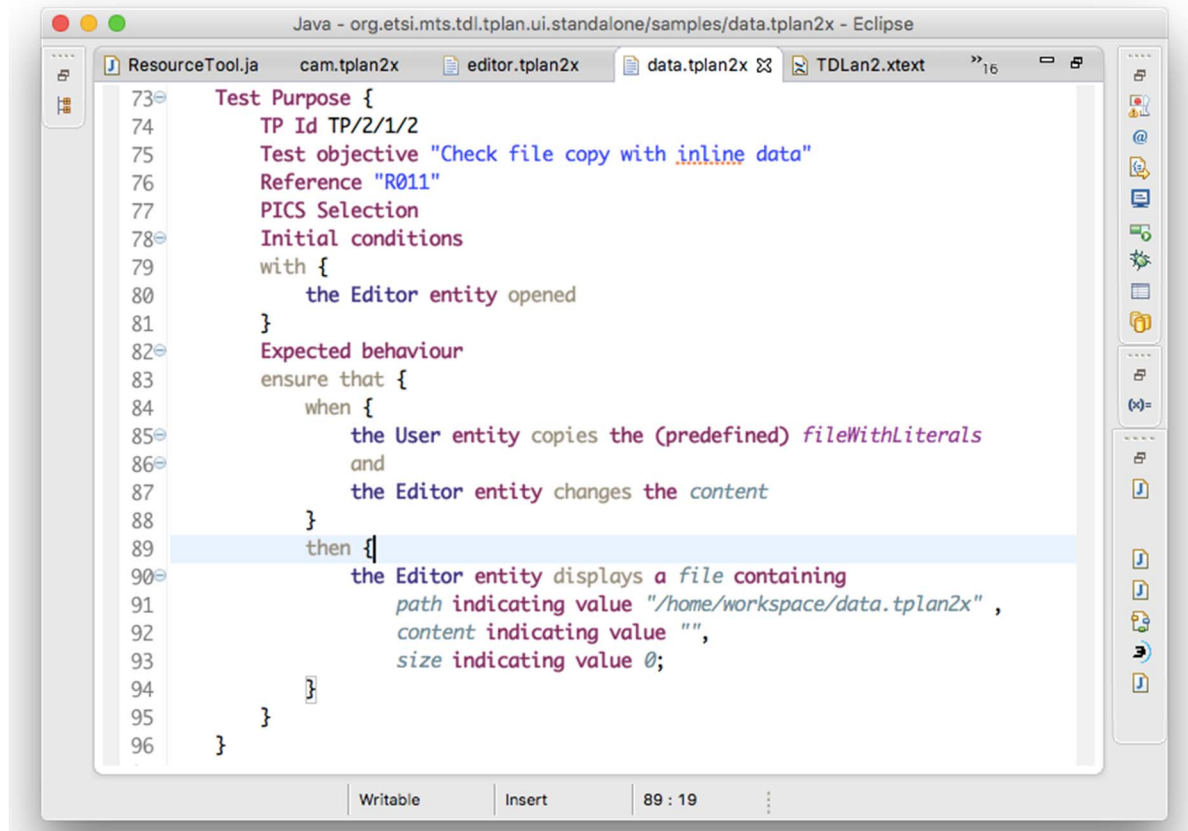


**Figure 5.3.1-3: Example of customized textual editor for TDL**

Similar to the editor for TDL, the TOP also includes a customized textual syntax that is tailored for the specification of structured test objectives. It implements the syntax from annex C of ETSI ES 203 119-4 [i.16] (up to version 1.5.1) and from clause 8 of ETSI ES 203 119-4 [i.16]. It also includes further customizations in the scoping and linking facilities, as well as enhanced semantic syntax highlighting, in a similar manner as the editor for TDL. An example of the customized editor is shown in Figure 5.3.1-4. It features a textual representation of a structured test objective. Current version of the grammar specification and the additional customizations can be found in annex A of the present document as part of the 'org.etsi.mts.tdl.TPLan2\*' projects for the syntax from annex C of ETSI ES 203 119-4 [i.16] (up to version 1.5.1) and as part of the 'org.etsi.mts.tdl.TDLtx\*' projects for the syntax from clause 8 of ETSI ES 203 119-4 [i.16].

Associated tooling provides means for the transformation between different syntax notations and model representations. Model instances in one notation can be transformed automatically into XMI representations and/or other textual or graphical syntax representations. This tooling integrates the APIs from different platforms for task specific automation. A current version of this tooling and detailed technical information can be found in annex A as part of the 'org.etsi.mts.tdl.tools.\*' and 'org.etsi.mts.tdl.rt.\*' projects.





**Figure 5.3.1-4: Example of customized textual editor for structured test objectives**

## Import and Export

The TDL implementation relies largely on the import and export facilities provided by the EMF. By default, the EMF does not activate the GUID support for XMI which is prescribed in ETSI ES 203 119-3 [i.15]. The TDL meta-model implementation needs to be adapted to activate the GUID support for model elements. The necessary adaptation involves selecting the correct resource type (XMI) in the generator model and activating the GUID support by overriding the corresponding method in the TDL resource implementation. Additionally, an implementation of the operations defined for the elements in ETSI ES 203 119-1 [i.13] and ETSI ES 203 119-4 [i.16] is necessary. This implementation is realized by means of embedded OCL expressions within the meta-model implementation. The relevant modifications can be found in the 'org.etsi.mts.tdl.model' project within annex A.

## 5.3.2 Viewing and Editing Models

### Principles of building model diagrams

The GMF framework that the TDL graphical editor is built upon follows the Model-View-Controller architecture. The model is an instance of TDL meta-model. The view is comprised of the shapes displayed on the diagram. The controller takes care of creating the shapes based on model objects and their associations, cross-references, and containments. In GMF, controllers are called 'editparts'.

The major part of the TDL graphical editor implementation consists of defining the corresponding 'editparts'. In the case of Sirius, these are not implemented directly but rather defined in terms of mappings. A mapping is a relation between a certain model object and a shape. Sirius interprets each mapping and uses the appropriate 'editpart' as a controller providing the mapping configuration data.

Mappings can be defined as nodes, edges, or containers (and some additional items specific to sequence diagrams). Each mapping includes a reference to the meta-class of the model object that it applies to, as well as the query that is used to lookup objects from the model based on the current context object. Similar to models and diagrams, mappings are also hierarchical. Edge mappings also define the queries that determine the corresponding shapes its endpoints connect to.

## Sirius diagrams

Sirius provides several diagram kinds that can be configured by providing diagram-specific model-object mappings. For TDL, the generic diagram and the sequence diagram are of particular interest.

Generic diagrams contain nodes and connections between the nodes with no specific constraints on their layout. Composite nodes containing other nodes are also supported, but only a few limited layout options are available for inner node placement: free-form and table (lines of text).

Sequence diagrams contain vertical parallel lines known as lifelines. Lifelines have headers with labels. Nodes and connectors between the lifelines - the fragments - are laid out as a horizontal stack. Nodes may cover any number of lifelines, connectors may only be drawn between two lifelines. Composite nodes containing sub-fragments (called combined fragments) are also supported.

Sirius editors are defined in configuration files known as viewpoint specifications. The TDL viewpoint specification defines a single viewpoint that contains two diagram descriptions named "TDL Behaviour" and "Generic TDL".

TDL Behaviour is a sequence diagram description. The root object of such diagrams is an instance of 'TestDescription'. The diagram description also defines the visual order of elements both horizontally and vertically. The vertical ordering contains behaviours recursively included in the 'TestDescription' as they occur semantically. The horizontal ordering contains 'GateReference's that are defined in the 'TestConfiguration' associated with the diagram's 'TestDescription' instance.

Generic TDL is a generic diagram description. The root object of such diagrams is an instance of 'Package'. There is no predefined order of objects defined for this diagram kind.

## Sirius diagram customization

The Sirius diagram specification model does not provide enough flexibility in terms of configuring all possible layouts required by the TDL graphical syntax. The diagrams are rendered by interpreting predefined configuration elements that do not have any extension mechanisms built in. Thus, some simple and composite figures need to be customized at a lower level.

The Sirius diagram rendering is built on top of the GMF runtime. Thus, it is possible to customize Sirius diagrams by means of extension points provided by GMF. The 'org.eclipse.gmf.runtime.diagram.ui.editpartProviders' extension point allows the replacement of default Sirius 'editparts' with customized 'editparts' dynamically, depending on which model object is being rendered, and depending on which diagram it is being rendered on. Classes defined in the extensions use mapping identifiers from the diagram specification to decide whether and which custom 'editparts' should be provided for the rendering of a diagram. All other mappings will rely on the default 'editparts' provided by the Sirius implementation.

## Implemented EditParts

All of the 'editpart' implementations are located in the 'org.etsi.mts.tdl.graphical.sirius.part' package.

The 'MultipartContainerCompartmentEditPart' extends GMF's 'ListCompartmentEditPart'. This class adds grid layout that allows contained shapes to fill the available area within the container. It also removes all borders from contained shapes in order to get rid of shadows and places horizontal lines between the contained shapes instead. Lastly, it removes the ability of being dragged and selected from the contained shapes in order to facilitate moving the whole compartment shape as one. The mapping that uses this 'editpart' has to be a container.

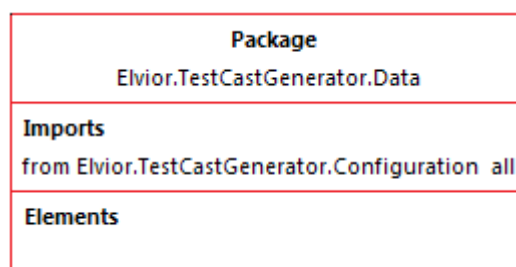


Figure 5.3.2-1: Example of 'MultipartContainerCompartmentEditPart'

The 'NodeListWithHeaderEditPart' extends the 'AbstractDiagramListEditPart' from the Sirius API. It is intended to be used within a 'MultipartContainerCompartmentEditPart' and provides functionality that allows the container to control its drag and selection handling. It removes all line borders from the contained shapes and replaces the borders with margins. The mapping that uses this 'editpart' has to be a container with list presentation style. The first label of the shape is the label of that container's style. The children of that mapping have to be nodes with square style.

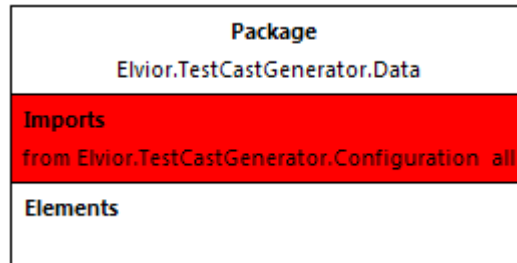


Figure 5.3.2-2: Example of 'NodeListWithHeaderEditPart'

The 'TopLevelNodeListWithHeaderEditPart' extends the 'NodeListWithHeaderEditPart' and adds the ability to be included directly on the diagram or inside a container with free-form presentation style. It also fixes a bug in the 'AbstractDiagramElementContainerEditPart.reInitFigure()' method.

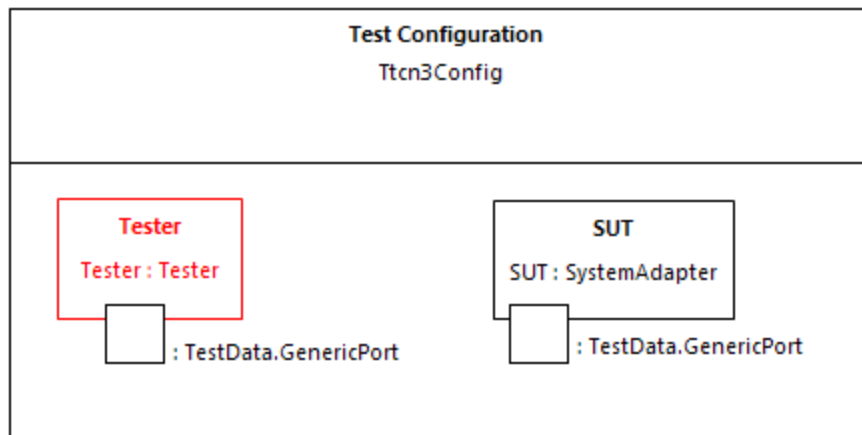


Figure 5.3.2-3: Example of 'NodeListWithHeaderEditPart'

The 'EditPartConfiguration' is used to specify additional style and layout properties supported by some custom 'editparts'. It is used, for example, to draw double border for specified edit parts using a 'TwoLineMarginBorder'.



Figure 5.3.2-4: Example of 'TopLevelImageNodeListWithHeaderEditPart'

The 'NodeContainerEditPart' extends the 'AbstractDiagramContainerEditPart' provided by the Sirius API. The default container is modified by disabling standalone selection and dragging and delegating those functions to the parent. All borders are removed from the shape. It is intended to be used as a child of 'MultipartContainerCompartmentEditPart'.

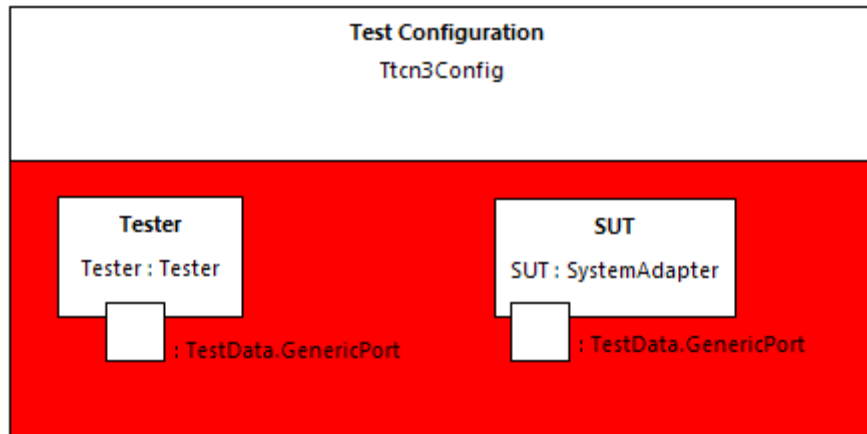


Figure 5.3.2-5: Example of 'NodeContainerEditPart'

The 'InteractionUseConfiguringEditPart' extends the 'AbstractNotSelectableShapeNodeEditPart' provided by the Sirius API. The class modifies the default interaction use shape by setting custom layout to it. The custom layout stretches the container's children to fill the available vertical space and leaves sufficient margin to the top for the label of the container. If the interaction use mapping has image style then the image background is made opaque.

This class is mapped to (an abstract) sub-mapping of interaction use. That mapping does not need to have a style as it will not be visible. The first label of the interaction use is the label of the container. The rest of the labels are sub-nodes with square styles.

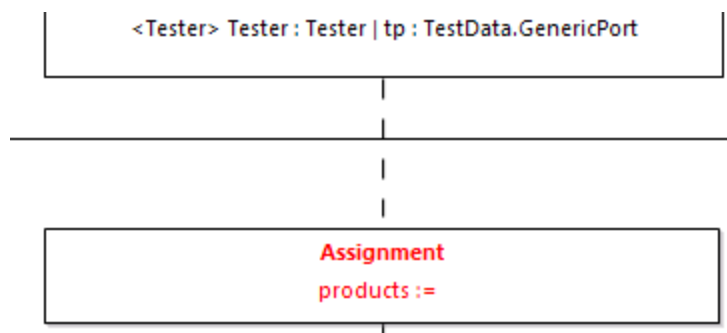


Figure 5.3.2-6: Example of 'InteractionUseConfiguringEditPart'

The 'MultiPartLabelEditPart' extends the 'TopLevelNodeListWithHeaderEditPart' and adds the ability to place labels horizontally in a row. This allows mappings that define different fonts for different parts of labels.

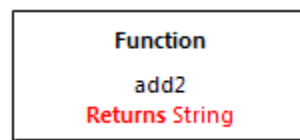


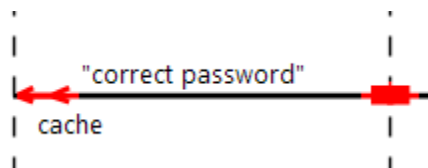
Figure 5.3.2-7: Example of 'MultiPartLabelEditPart'

The 'CombinedFragmentLabelEditPart' extends the 'MultiPartLabelEditPart' to inherit support for mixed font labels. It overrides the default layout behaviour via a 'LayoutListener' from the 'Draw2d' API and places the shape always to the upper right corner of a combined fragment block.



Figure 5.3.2-8: Example of 'CombinedFragmentLabelEditPart'

The 'InteractionDecoratorProvider' is contributed via the 'org.eclipse.gmf.runtime.diagram.ui.decoratorProviders' extension point in order to draw special rotatable shapes at the ends of connectors. This class is configured to work specifically with 'Interaction's.



**Figure 5.3.2-9: Example of 'InteractionDecoratorProvider'**

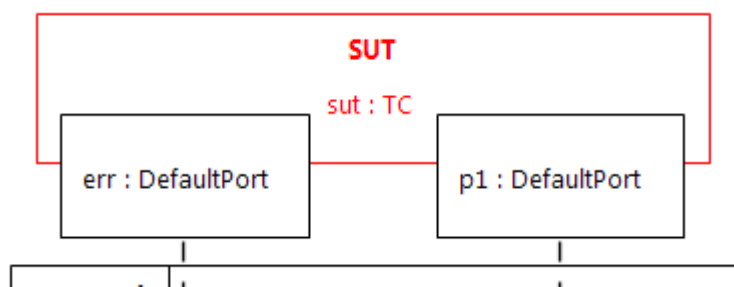
#### Implemented layouts

All layout implementations are located in the 'org.etsi.mts.tdl.graphical.sirius.layout' package.

The 'SequenceDiagramFreeformLayoutProvider' overrides the default placement of elements on the diagram layer. It also fixes the layout of shapes modified by the 'InteractionUseConfiguringEditPart' that would otherwise be cropped to the default size and would not trigger the layout of contents on container resize. It is contributed via the 'org.eclipse.sirius.diagram.ui.layoutProvider' extension point and its use is triggered by the arrange command.



**Figure 5.3.2-10: Example of custom figure placement: node with attachment**



**Figure 5.3.2-11: Example of custom figure placement: under-lapping container**

The layout customizations are implemented via the diagram 'arrange' mechanism, which is normally triggered only when the user invokes the 'arrange' command. Additional triggers are implemented in order to facilitate the automatic diagram creation upon user creating and updating the model. The 'RefreshExtensionProvider' is contributed via the 'org.eclipse.sirius.refreshExtensionProvider' extension point. It invokes the 'arrange' command when the model is modified and subsequently reloaded into the diagram editor. The 'LayoutEditPolicyProvider' is contributed via 'org.eclipse.gmf.runtime.diagram.ui.editpolicyProviders' extension point and it invokes the arrange command when a 'GateReference' or 'ComponentInstance' shape is moved by the user in order to keep the under-lapping shape properly aligned.

## Editor-specific meta-model

The Sirius sequence diagram configuration sets implicit requirements on the structure of the meta-model that is used in the mapping definitions. The TDL meta-model does not comply with these requirements in all cases. For example, the mappings of combined fragments tend to fail at runtime when the begin and end occurrence objects (as understood by Sirius) are the same. Since TDL does not define occurrences at all, some adaptation is needed to provide these occurrence objects. Sirius and the underlying framework require that model objects used in diagrams are defined by a meta-model. Extending the TDL meta-model with pure fabrications, just to facilitate graphical editor implementations, would be a bad practice. Therefore, a separate domain-agnostic meta-model was created for this purpose.

The meta-model named 'tdlviewer' is defined in the 'extension.ecore' file and is registered as dynamic. This means that the meta-model may be used reflectively without any code generation (which is a standard practice with meta-model implementations in EMF). The 'tdlviewer' contains a single meta-class 'End' with a single attribute 'begin'. The 'begin' holds a reference to the model object which this instance of 'End' is paired with. The object itself is used as the begin occurrence in the mappings. The creation of virtual end objects is implemented in the 'org.etsi.mts.tdl.graphical.extensions.BehaviourProvider' class.

## Label serialization

Some of the labels in ETSI ES 203 119-2 [i.14] are particularly complex, especially the labels related to 'DataUse'. Mappings for such labels in the diagrams are realized by means of Xtext. A partial annotated EBNF grammar defines the relevant mappings. The serialization facilities of Xtext are invoked in the corresponding context in order to obtain the textual representation of the object of interest (such as a 'DataInstanceUse') only. The implementation of the label serialization is provided in the 'org.etsi.mts.tdl.graphical.labels.data\*' projects. The label serialization is integrated into the viewpoint by means of the 'org.etsi.mts.tdl.graphical.extensions.DataUseLabelProvider' class which is registered with the viewpoint specification.

## Configured mappings

A summary of the mappings is provided in Tables 5.3.2-1 and 5.3.2-2. The details of the diagram mapping definitions can be found in the Sirius viewpoint-specification file 'org.etsi.mts.tdl.graphical.viewpoint/description/TDL.odesign' within the 'org.etsi.mts.tdl.graphical.viewpoint' project in annex A.

Table 5.3.2-1: Mappings in the behaviour diagram specification

Meta-class	Mapping (<kind>: <identifier>)	Editpart (if not default)
GateReference	Instance Role: gateReference	
GateReference	Execution: lifelineExecution	
GateReference	End Of Life: lifelineEnd	
TimeConstraint	Node: timeConstraint	
TimeLabel	Node: timeLabel	
Target	Basic Message: interaction	
	Relation Based Edge: timeConstraintAttachment	
	Relation Based Edge: timeLabelAttachment	
CompoundBehaviour ParallelBehaviour AlternativeBehaviour UnboundedLoopBehaviour BoundedLoopBehaviour ConditionalBehaviour PeriodicBehaviour DefaultBehaviour InterruptBehaviour	Combined Fragment: combinedBehaviour	
BoundedLoopBehaviour	Container: boundedLoopBehaviour Node: boundedLoop.keyword boundedLoop.iteration	CombinedFragmentLabelEditPart
PeriodicBehaviour	Container: periodicBehaviour Node: periodicBehaviour.keyword Node: periodicBehaviour.iteration	CombinedFragmentLabelEditPart
Block	Operand: block	
Break Stop	Interaction Use: globalAction	
Assertion	Interaction Use: assertion Node: assertion.config Node: assertion.condition Node: assertion.otherwise	InteractionUseConfiguringEditPart
VerdictAssignment	Interaction Use: verdictAssignment Node: verdictAssignment.config	InteractionUseConfiguringEditPart
TimerStart TimerStop TimeOut	Interaction Use: timerOperation Node: timerOperation.config	InteractionUseConfiguringEditPart
Assignment	Interaction Use: assignment Node: assignment.config Node: assignment.assignment	InteractionUseConfiguringEditPart
ActionReference	Interaction Use: actionReference Node: actionReference.config Node: actionReference.action Node: actionReference.actualParameter	InteractionUseConfiguringEditPart
InlineAction	Interaction Use: inlineAction Node: inlineAction.config Node: inlineAction.Body	InteractionUseConfiguringEditPart
TestDescriptionReference	Interaction Use: testDescriptionReference Node: testDescriptionReference.config Node: testDescriptionReference.testDescription Node: testDescriptionReference.actualParameter Node: testDescriptionReference.componentBindings	InteractionUseConfiguringEditPart
Wait Quiescence	Interaction Use: timeOperation Node: timeOperation.config Node: timeOperation.period	InteractionUseConfiguringEditPart
ComponentInstance	Container: componentInstance Node: componentInstance.name	TopLevelNodeListWithHeaderEditPart

**Table 5.3.2-2: Mappings in the package diagram specification**

Meta-class	Mapping (<kind>: <identifier>)	Editpart (if not default)
Comment	Node: comment	
	Relation Based Edge: commentedElement	
	Relation Based Edge: simpleDataInstance_dataType	
	Relation Based Edge: structuredDataInstance_dataType	
Connection	Element Based Edge: testConfiguration.connection	
DataElementMapping	Element Based Edge: dataElementMapping.mapping	
	Relation Based Edge: dataElementMapping.association	
AnnotationType	Container: annotationType Node: annotationType.name	TopLevelNodeListWithHeaderEditPart
SimpleDataType	Container: simpleDataType Node: simpleDataType.name	TopLevelNodeListWithHeaderEditPart
Time	Container: time Node: time.name	TopLevelNodeListWithHeaderEditPart
SimpleDataInstance	Container: simpleDataInstance Node: simpleDataInstance.name	TopLevelNodeListWithHeaderEditPart
Package	Container: package Container: package.name Node: name Container: package.imports Node: Import Container: package.packagedElements Node: packagedElement	MultipartContainerCompartmentEditPart  NodeListWithHeaderEditPart
Action	Container: action Container: action.name Node: name Container: action.parameter Node: Parameter Container: action.body Node: Body	MultipartContainerCompartmentEditPart NodeListWithHeaderEditPart  NodeListWithHeaderEditPart  NodeListWithHeaderEditPart
ComponentType	Container: componentType Bordered: gateInstance Container: componentType.name Node: name Container: componentType.timers Node: componentType.timer Container: componentType.variables Node: componentType.variable	MultipartContainerCompartmentEditPart  NodeListWithHeaderEditPart
TestConfiguration	Container: testConfiguration Container: testConfiguration.name Node: name Container: testConfiguration.configuration Container: testConfiguration.componentInstance Bordered: testConfiguration.gateReference Node: testConfiguration.componentInstance.name	MultipartContainerCompartmentEditPart NodeListWithHeaderEditPart  NodeContainerEditPart TopLevelNodeListWithHeaderEditPart
TestObjective	Container: testObjective Container: testObjective.name Node: name Container: testObjective.description Node: Description Container: testObjective.objectiveURI Node: URI	MultipartContainerCompartmentEditPart  NodeListWithHeaderEditPart
StructuredDataType	Container: structuredDataType Container: structuredDataType.name Node: name Container: structuredDataType.member Node: member	MultipartContainerCompartmentEditPart NodeListWithHeaderEditPart



Meta-class	Mapping (<kind>: <identifier>)	Editpart (if not default)
StructuredDataInstance	Container: structuredDataInstance Container: structuredDataInstance.name Node: name Container: structuredDataInstance.memberAssignment Node: memberAssignment	MultipartContainerCompartmentEditPart  NodeListWithHeaderEditPart
DataResourceMapping	Container: dataResourceMapping Container: dataResourceMapping.name Node: name Container: dataResourceMapping.resourceURI Node: resourceURI	MultipartContainerCompartmentEditPart NodeListWithHeaderEditPart
DataElementMapping	Container: dataElementMapping Container: dataElementMapping.name Node: name Container: dataElementMapping.parameterMapping Node: parameterMapping	MultipartContainerCompartmentEditPart  NodeListWithHeaderEditPart
TestDescription	Container: testDescription Container: testDescription.name Node: name Container: testDescription.parameter Node: Parameter Container: testDescription.objective Node: Objective Container: testDescription.configuration Node: Configuration Container: testDescription.behaviour Container: BehaviourConfiguration Node: Component	MultipartContainerCompartmentEditPart NodeListWithHeaderEditPart  NodeListWithHeaderEditPart  NodeListWithHeaderEditPart
Function	Container: function Container: function.name Node: name Container: function.returnType Node: function.returnType.keyword Node: function.returnType.type Container: function.parameter Node: Parameter Container: function.body Node: Body	MultipartContainerCompartmentEditPart  MultiPartLabelEditPart  NodeListWithHeaderEditPart

### 5.3.3 Exporting Structured Test Objectives

Structured test objectives are exported as tables in a Word document according to user-defined templates. The implementation expects templates to be placed in tables and feature the following placeholders which are mapped to the corresponding elements for a structured test objective referenced as 'self':

- <TESTOBJECTIVENAMELABEL\_PLACEHOLDER> mapped to 'self.name'
- <DESCRIPTIONLABEL\_PLACEHOLDER> mapped to 'self.description'
- <URIOFOBJECTIVELABEL\_PLACEHOLDER> mapped to 'self.objectiveURI', separated by comma in case of multiple 'objectiveURI's
- <CONFIGURATIONLABEL\_PLACEHOLDER> mapped to 'self.configuration.name'
- <PCSSELECTIONLABEL\_PLACEHOLDER> mapped to 'self.picsReference'
- <INITIALCONDITIONSLABEL\_PLACEHOLDER> mapped to 'self.initialConditions'
- <EXPECTEDBEHAVIOURLABEL\_PLACEHOLDER> mapped to 'self.expectedBehaviour'
- <FINALCONDITIONSLABEL\_PLACEHOLDER> mapped to 'self.finalConditions'

- <EXPECTEDBEHAVIOURLABEL\_WHENPART\_PLACEHOLDER> mapped to 'self.expectedBehaviour.whenClause'
- <EXPECTEDBEHAVIOURLABEL\_THENPART\_PLACEHOLDER> mapped to 'self.expectedBehaviour.thenClause'

Each template table is expected to have a unique identifier in the heading row. The implementation expects the user to select an identifier of a template in order to export the structured test objectives according to the corresponding template. An example of a template based on the syntax specification in ETSI ES 203 119-4 [i.16] is shown on Table 5.3.3-1. Additionally, shading can be used within templates to hide optional parts when their content is empty. Multiple related optional compartments can be marked to be hidden, e.g. the heading 'Final Conditions' and the corresponding compartment, by using the same shading.

Additional placeholders may be defined by users, however, the implementation also needs to add support for them. The mappings are currently implemented at a lower level - in code. Additional filtering may be performed to streamline the output. This may include hiding some keywords and punctuation. The example shown in Table 5.3.3-1 is exported from the model used in annex C of ETSI ES 203 119-4 [i.16] (up to version 1.5.1). A filter has been applied to hide the 'entity' keywords in the output. Finally, 'EventTemplateOccurrence's may be optionally replaced by the corresponding 'EventOccurrenceSpecification' from the referenced 'EventTemplateSpecification' while applying replacements for overridden 'Argument's and 'EntityReference's. The details of the export of structured test objectives to Word tables can be found in the 'org.etsi.mts.tdl.to.docx\*' projects in annex A. The example template as well as additional templates are included in the 'templates.docx' document.

**Table 5.3.3-1: Structured test objective template example**

TO_1_TABLE_TEMPLATE	
<b>TP Id</b>	<TESTOBJECTIVENAMELABEL_PLACEHOLDER>
<b>Test Objective</b>	<DESCRIPTIONLABEL_PLACEHOLDER>
<b>Reference</b>	<URIOFOBJECTIVELABEL_PLACEHOLDER>
<b>PICS Selection</b>	<PICSSELECTIONLABEL_PLACEHOLDER>
<b>Initial Conditions</b>	
<INITIALCONDITIONSLABEL_PLACEHOLDER>	
<b>Expected Behaviour</b>	
<EXPECTEDBEHAVIOURLABEL_PLACEHOLDER>	
<b>Final Conditions</b>	
<FINALCONDITIONSLABEL_PLACEHOLDER>	

**Table 5.3.3-2: Exported structured test objective according to the template in Table 5.3.3-1**

<b>TP Id</b>	TP_7_1_3_1_1
<b>Test Objective</b>	
<b>Reference</b>	ETSI TS 136 321 [i.25], clause 5.3.1
<b>PICS Selection</b>	
<b>Initial Conditions</b>	
with { the UE in the "E-UTRA RRC_CONNECTED state"	
<b>Expected Behaviour</b>	
ensure that { when { the UE receives a "downlink assignment on the PDCCH for the UE's C-RNTI" and the UE receives a "data in the associated subframe" and the UE performs a HARQ operation } then { the UE sends a "HARQ feedback on the HARQ process" } }	
<b>Final Conditions</b>	

## 5.3.4 Validating Models

### Overview

Means for defining and validating constraints on models are an integral part of modelling environments. Model constraints are used to impose semantic restrictions on top of the abstract syntax provided by the meta-model. There are different approaches for the specification, integration, and validation of such constraints. OCL is the de facto standard for the specification and realization of constraints on object-oriented meta-models. OCL expressions can be integrated into the meta-model by means of annotations, which can be used for automated validation of model instances, provided adequate tool support is available. An alternative approach is the specification constraints as an add-on which can then be applied to the model instances.

A constraint specification typically consists of a context indicating the meta-class to which the constraint applies, and an invariant indicating the conditions that will hold true in the given context for valid models. For example, the requirement *"a 'NamedElement' shall have the 'name' property set and the 'name' shall not be an empty String"* is specified in OCL as follows:

```
context NamedElement
  inv: not self.name.isUndefined() and self.name.size() > 0
```

where 'self' refers to the instance of the 'NamedElement' meta-class.

### Integrated Approach

The integrated approach involves the definition of semantic constraints within the meta-model itself by means of annotations. Modelling environments can then generate integrated validation facilities based on the annotations. The validation facilities can be invoked automatically so that immediate feedback can be provided to the users when they work with models. The main benefit of an integrated approach is that the constraints become an embedded part of the meta-model. However, there are also certain limitations associated with the integrated approach. Modifications to constraints would require changing the meta-model and related generated resources. Tool support for constraints included as embedded annotations is very inconsistent. Immediate feedback while helpful, can sometimes get in the way. In case a model is refined over multiple steps before it becomes valid, checking constraints at any point before that would be superfluous.

### Add-on Approach

In contrast to the integrated approach, the add-on approach relies on semantics constraints defined separately from the meta-model. Such constraints can be checked on demand as required by the specific usage scenario. In addition, the evaluation of such constraints can also be conducted in a more flexible manner, where only subsets of constraints are checked as necessary at a given point in time, thus limiting the amount of superfluous violations for models which are known to be incomplete at that point in time. Add-on constraints can also be modified, maintained, and extended independently from the meta-model. Certain technologies, such as the Epsilon Validation Language (EVL) [i.12] also extend the capabilities of OCL by providing means to specify guards on constraints determining conditions under which the evaluation of a constraint is to be skipped.

The constraints for TDL are realized according to the add-on approach within the 'org.etsi.mts.tdl.constraints' project. The project contains the constraint realization in the 'tdl.evl' file as well as supporting resources for common and extended functionalities. A standalone launcher is implemented to enable the checking of constraints independent of other tooling. It can also be used as a foundation for integrated solutions.

## 5.4 Usage Instructions

### 5.4.1 Development Environment

The latest information on setting up a development environment for the TDL toolset implementation can be found at the link: [Setting up a local development environment](#).

## 5.4.2 End-user Instructions

### Installation

End-user TOP tools installation instructions are available on the website [Installation](#).

### Creating TDL models

Once the TOP tools are installed, the following steps allow TDL models to be created with the graphical editor:

- 1) Make sure an explorer view is open in Eclipse (Project Explorer or Model Explorer, for example).
- 2) Select 'New -> Project...' from the 'File' menu or the right-click contextual menu in the explorer view.
- 3) In the 'New Project' wizard, select new TDL Project.
- 4) Enter a name for the project and press 'Finish'.
- 5) In the explorer view, expand the newly created project, expand the 'model.tdl', right-click on the 'Package Model' and select 'New Representation -> new Generic TDL'.
- 6) Enter a name for the new diagram and press 'OK'.
- 7) A new 'Generic TDL' diagram is created where the predefined types are already shown.
- 8) Start creating new elements by using the palette.

The editing of models with tree editors is described in clause 5.3.1. For creating models with the textual editors, end users need to create a new file with the file extensions '.tdlan2' for TDL models according to annex B of ETSI ES 203 119-1 [i.13] (up to version 1.7.1), '.tdltx' or '.tdltxi' for TDL models according to ETSI ES 203 119-8 [i.20] (brace- or indentation-based, respectively), or with the file extension '.tplan2' for TDL models containing structured test objectives specified according to annex C of ETSI ES 203 119-4 [i.16] (up to version 1.5.1). All files need to be located within projects in Eclipse. The newly created files are already associated with the respective editor so that the users can benefit from the enhanced editing capabilities such as syntax checking, syntax highlighting, auto-completion, etc.

### Validating Models

Open the TDL model (with file extensions '.tdl', '.tdlan2', '.tdltx', '.tdltxi', or '.tplan2') with any of the available editors (reflective, faceted, or textual) and press the 'Validate TDL model' button. Any constraint violations will be shown in a popup dialog.

### Translating Models

Open the XMI or textual representation of a TDL model (with file extensions '.tdl', '.tdltx', '.tdltxi', '.tdlan2', or '.tplan2') and press the 'Translate TDL model' button. A popup dialog will ask about the desired target representation format. The translated representation of the TDL model into the target representation will be named the same way as the original model (with an additional extension '.tdl', '.tdltx', '.tdltxi', '.tdlan2', or '.tplan2') and placed in the same location.

### Working with Diagrams

In Sirius and, therefore, in the TDL graphical editor, diagrams are called representations. A representation is always related to one model element that is the root of the representation. There are two representation kinds in the TDL viewer. The 'Generic TDL' representation takes an instance of 'Package' as its root and represents the contents of that 'Package' laid out as a graph. The 'TDL Behaviour' representation displays the behaviour of a 'TestDescription' instance laid out as a sequence diagram.

In order to create a new diagram, open the Create Representation wizard on a project, choose the appropriate representation kind and on the last page, select the root element matching the chosen representation kind. Created representation is automatically opened in an editor and the representation also becomes visible in the explorer view (under the node 'representations.aird').

The diagram editor may be used to adjust the layout of the shapes, although the implementation takes care of most of the layout tasks.

## Exporting Diagrams

Diagrams may be exported to image files. Use the context menu of representation nodes in the explorer view or directly in the diagram canvas. Note that although it is not necessary to have the diagrams open while editing models, the diagram editors need to be opened before exporting the diagrams in order to refresh the visual elements with the semantic model.

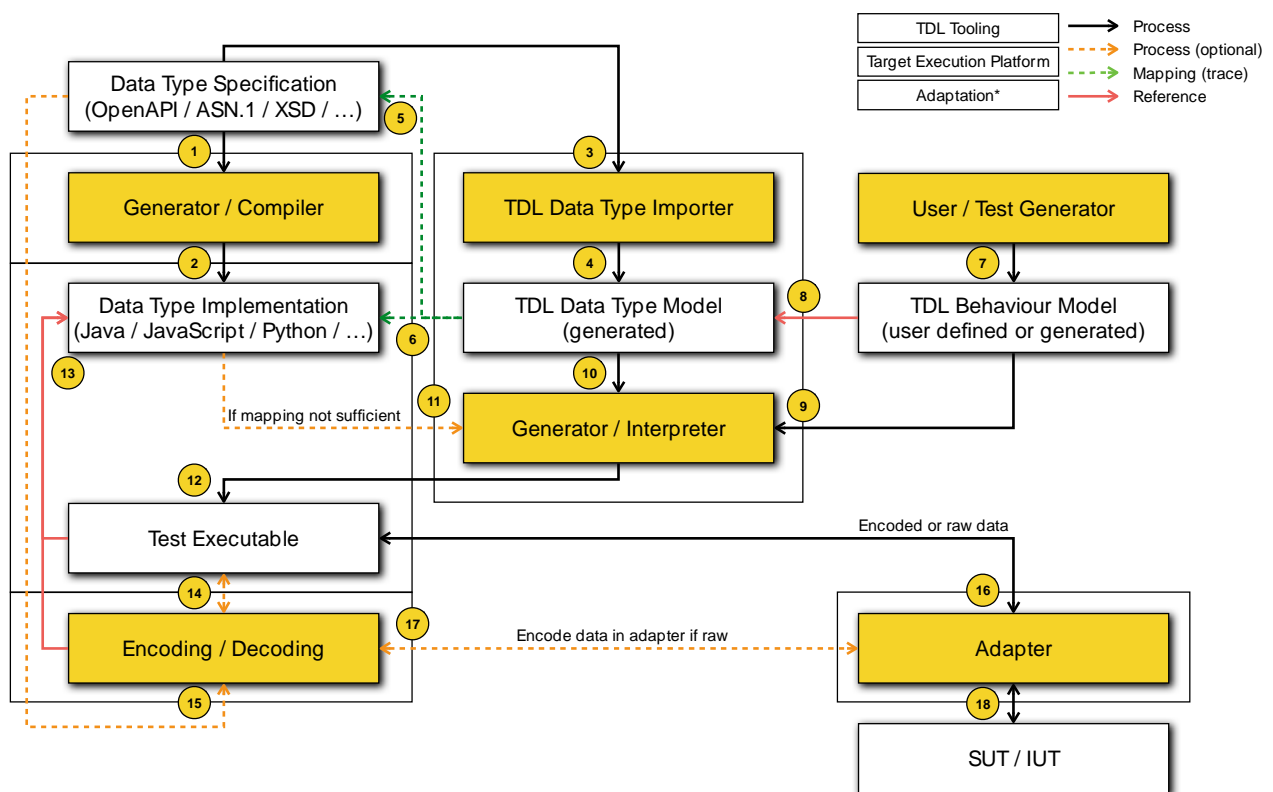
## Exporting Structured Test Objectives

Open the TDL model (with file extensions '.tdl' or '.tplan2') with any of the available editors (reflective, faceted, or textual) and press the 'Generate Document' button or select the 'TDL -> Generate Document' from the menu. The generated Word document will be named the same way as the model (with an additional extension '.docx') and placed in the same location.

# 6 Using TDL with External Data Type Specifications

## 6.1 Generalized Process

### 6.1.1 Process Overview



**Figure 6.1.1-1: Overall process for importing existing external data type specifications in TDL**

Formalised data type specifications are sometimes provided as part of the base specification for the systems to be tested. These can be in the form of protocol definitions, e.g. including data type definitions in ASN.1, or entire interface specifications, e.g. as informative or normative OpenAPI™ specifications. Requiring TDL users to redefine the data types present in such specifications can be very time-consuming and error-prone, especially when the base specifications continue to evolve and the test specifications need to be continuously aligned. Additionally, data specifications in TDL are inherently abstract and need to be mapped to concrete data implementations in the target execution platform. The existing formalised data type specifications are also used for the system implementation, where code generators and compilers provide data type implementations based on the provided specifications. Ideally, the tests would make use of the generated data type implementations as well.

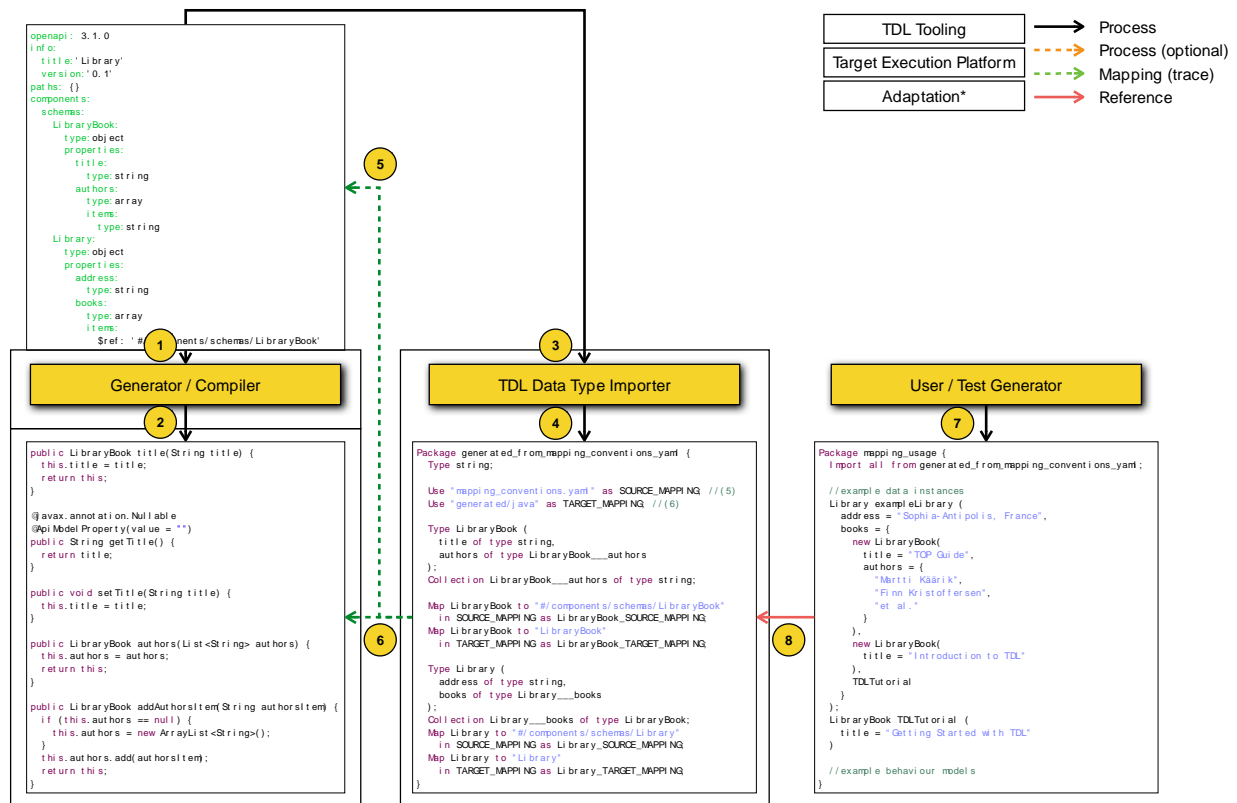
In order to make TDL aware of the existing formalised data type specifications, they need to be "imported" in TDL in the sense that there needs to be a TDL data type model which contains all the relevant information from external data type specifications, including mappings to the source from which the data types are derived, for traceability and other purposes. Users or test generation tools can then produce TDL behaviour models and TDL data instances using the TDL data types derived from the external data type specifications. As there may be differences between the capabilities of the external data type specification language and TDL, there may also be different ways of deriving the TDL data types. A set of guidelines can be helpful to ensure consistent derivation and mapping. For example, some languages support nested anonymous type definitions, whereas TDL only supports "flat" data type definitions. In such cases, the guidelines indicate how the nested type definitions can be flattened by extracting them and following certain naming conventions, e.g. based on the name of the containing data definitions. Corresponding tool support is essential for larger data type specifications. Following an overview of the overall process and an example instantiation, specific guidelines for OpenAPI™, Yang, and ASN.1 are provided in the subsequent clauses.

The overall process for the importing and use of data types from OpenAPI™, Yang, ASN.1, and other specifications is outlined in Figure 6.1.1-1. The following aspects need to be considered:

- Data type specifications are used as input (1) for generators or compilers producing (2) data type implementations in the target execution platform, such as Java™, JavaScript™, or Python™.
- The data type specifications are also used as input (3) for the TDL data type importer, which generates (4) a TDL data type model.
- The data type model includes mappings to the data type specification (5) for traceability and to the data type implementation (6) for operationalisation.
- TDL behaviour models (7), which are either defined manually or generated automatically, import and use the TDL data type model (8) generated from the data type specification.
- The TDL behaviour models and the TDL data type model are then processed by a generator or an interpreter (9 and 10) to produce a test executable for the target platform (12). The generator or interpreter may also need to make use of the data type implementation in some cases (11).
- The test executable uses (13) the data type implementation and interfaces (16) with the adapter to communicate with the SUT/IUT (18).
- For the communication with the SUT/IUT, the data usually needs to be encoded and decoded. Depending on the circumstances, the test executable may interface with the encoding and decoding functionality directly (14) or the encoding and decoding may be handled by the adapter (17).
- The encoding and decoding functionality generally relies on the data type implementation (13), but may also need to make use of the original data type specification (15) if the data type implementation does not include all the necessary information.

The outlined process is simplified and generalized. In practice, there may be different stages in the TDL behaviour model specification, including the definition of structured test objectives, the definition of totally ordered test descriptions, as well as the refinement of the totally ordered test descriptions into locally ordered test descriptions. Depending on the context, some of the stages may be required or omitted. The overall process remains the same as only the level of detail in the TDL behaviour models is affected in the different stages. While structured test objective specifications may not necessarily need to be concerned with details of the target execution platform, including the data type implementations, the test executable, and the adaptation layer, the mapping information for the target platform can be already provided by the TDL data type importer from the start for reference, or be added later.

## 6.1.2 Example Instantiation



**Figure 6.1.2-1: Example instantiation of the overall process from Figure 6.1.1-1**

A concrete example for the outlined process is illustrated in Figure 6.1.2-1. It includes snippets from the following artefacts:

- Given a data type specification in OpenAPI™ (1), the corresponding tooling can be used to generate data type implementations in Java (2), JavaScript, and other target languages.
- Based on the data type specification, the TDL data model (4) including the mappings to the data type specification (5) and the data type implementation in Java (6) are generated.
- A user then specifies the TDL data instances and behaviour models (7) using the generated TDL data model (8).
- With the help of the above artefacts, a test executable can be assembled either by means of code generation or by means of interpretation. Corresponding encoding and decoding functionalities may be provided by third-party components or rely on the generated data type implementation.

While this example illustration is built around an OpenAPI™ specification with Java as the target platform, the same process can be applied to other target platforms or other kinds of data type specifications, such as Yang or ASN.1 specifications. The TOP provides basic capabilities for importing data type specifications from OpenAPI™ Yang and ASN.1, which can be optionally installed in addition to the core meta-model implementation and the different editors.

Further details on the implemented support for external data types in TOP can be found in the [online user documentation](#).

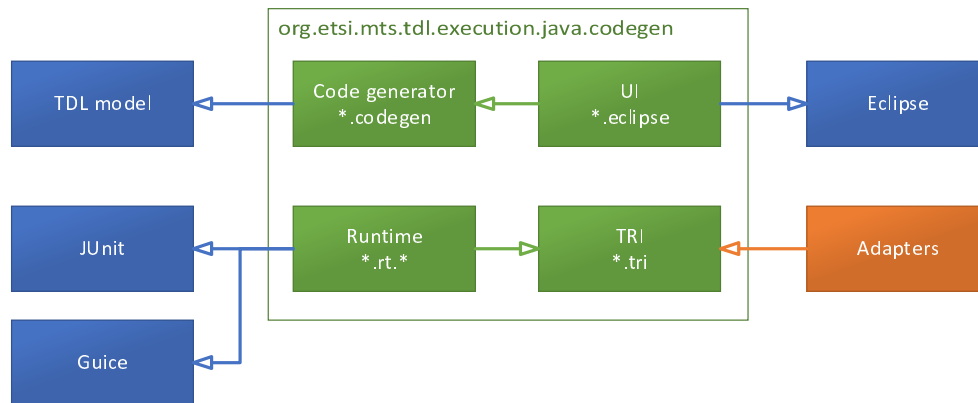
## 7 TDL Runtime/Execution

### 7.1 Java™: Code generator

#### 7.1.1 Architecture

A code generator converts TDL test descriptions to Java code and provides a runtime environment as well as a Test Runtime Interface (TRI) for users to implement the adaptation to real test environment and SUT. As an alternative to interpreter, code generation removes the dependencies to TDL meta-model and simplifies deployment of the executable tests.

The source code for the code generator is available in TOP [i.24] in 'org.etsi.mts.tdl.execution.java.codegen' project. The project includes code documentation in Javadoc [i.28] format and instructions for setup and use.



**Figure 7.1.1-1: Project structure and dependencies of Java code generator**

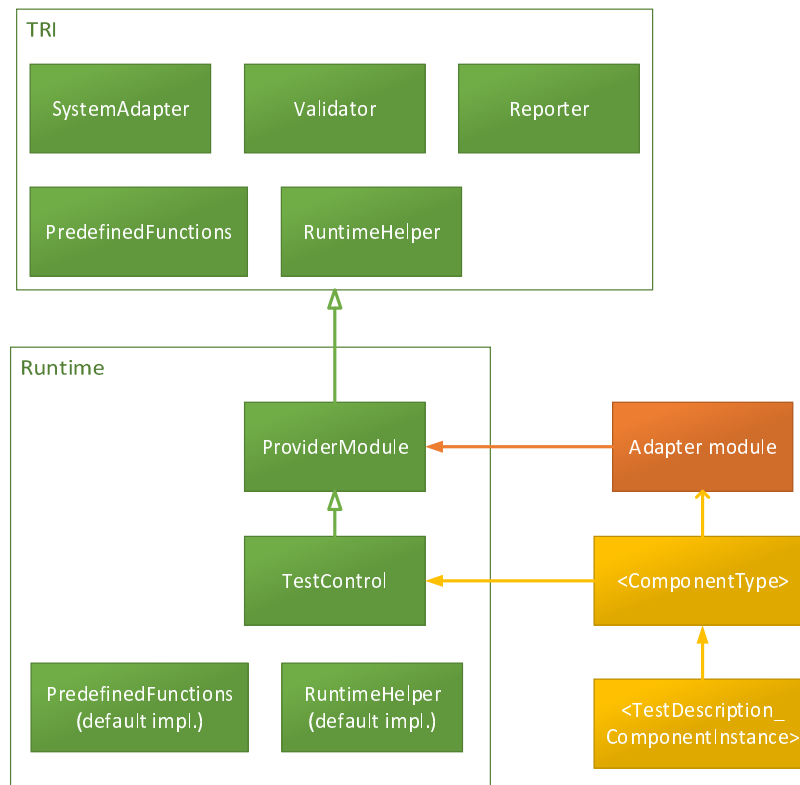
In addition to code generator, the 'org.etsi.mts.tdl.execution.java.codegen' project provides various User Interface (UI) components for triggering and configuring the code generation. Runtime code has dependencies to JUnit [i.29] for its test reporting and assertion functions as well as Guice [i.30] for resolving platform adapter implementations. The interfaces that should be realized and provided by end users are collectively called Test Runtime Interface (TRI).

In Guice parlance, the component that provides (Guice term for associating a class to a type) interface realizations is called a module. The name of the module that provides the TRI interface bindings (the 'adapter module') is configured in generator settings. It may provide implementations for following interfaces (listed in 'ProviderModule' class):

- 'SystemAdapter': a required component that manages interactions between runtime and SUT;
- 'Validator': a required component that provides data matching functionality;
- 'Reporter': an optional component that implements test logging;
- 'PredefinedFunctions': optional customized implementation of TDL predefined functions; and
- 'RuntimeHelper': optional customized implementation of various environment specific functions.

Default implementations for 'PredefinedFunctions' and 'RuntimeHelper' are provided by the runtime.





**Figure 7.1.1-2: Structure of runtime classes of Java code generator**

The core component of the execution engine is 'TestControl', which is the base for all generated tester components. It provides access to instances of TRI components and contains helper functions to handle complex execution logic (such as alternatives) and asynchronous nature of interactions and time operations.

For each TDL 'ComponentType', a Java class is generated that extends the 'TestControl'. It adds fields for variables and timers and invokes the adapter module. A sub-class of the component type class is generated for each tester 'ComponentInstance' participating in a 'TestDescription'. The component classes include time labels and provide the test execution code that can be invoked by the JUnit framework.

## 7.1.2 Test Runtime Interface (TRI)

### 7.1.2.1 Overview

TDL runtime interfaces are specified in [i.39].

### 7.1.2.2 Interface ProviderModule

implements com.google.inject.Module

Example of Guice injector module that is used by execution engine to configure the environment specific adapters.

Note that this interface should only be used as an example to provide all relevant implementations and it should not be implemented directly, as extending it breaks the annotation declarations.

Methods:

```

public org.etsi.mts.tdl.execution.java.tri.PredefinedFunctions
providePredefinedFunctions(org.etsi.mts.tdl.execution.java.tri.RuntimeHelper helper)

public org.etsi.mts.tdl.execution.java.tri.Reporter provideReporter()

public org.etsi.mts.tdl.execution.java.tri.RuntimeHelper
provideRuntimeHelper()

public org.etsi.mts.tdl.execution.java.tri.SystemAdapter

```

```
provideSystemAdapter()
```

```
public org.etsi.mts.tdl.execution.java.tri.Validator provideValidator()
```

### 7.1.3 Mappings

Most mappable TDL model elements are mapped to appropriate Java elements for test execution. In TDL, mapping specifications consist of two levels: resource mapping and element mapping. Resource mappings should refer to either Java package or class (using qualified names). Element mappings should refer to either Java class, field or method.

As the mappings can refer to different kinds of Java elements, then a predefined annotation should be added to both resource and element mappings in one of following combinations:

- resource mapping with 'JavaPackage' annotation and element mapping with 'JavaClass' annotation; or
- resource mapping with 'JavaClass' annotation and element mapping with either 'JavaField', 'JavaStaticField', 'JavaMethod' or 'JavaStaticMethod' annotation.

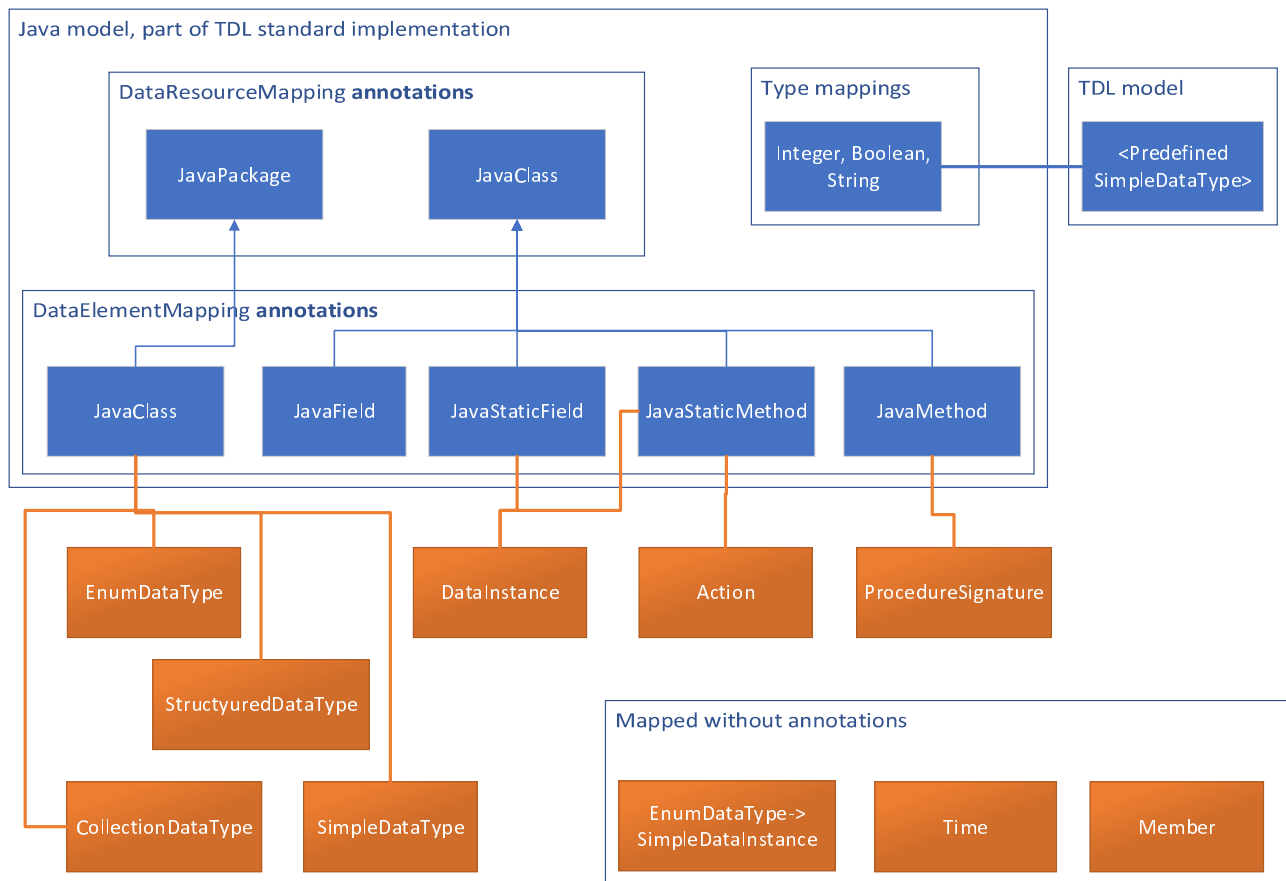
Those annotations are provided by the Java model that is part of the TDL standard implementation.

Following mappable TDL elements do not require any annotations as the mapped Java element is clear:

- 'SimpleDataInstance's in 'EnumDataType' are mapped to Java enum literals;
- 'Time' types are mapped to Java longs; and
- 'Members are mapped to Java fields.

The TDL Java model provides mappings for predefined TDL 'SimpleDataType's. TDL predefined functions are mapped to appropriately named functions with function parameter types replaced with corresponding Java types.

In addition to predefined 'Time' instance 'Second', the code generator also supports 'Time' instances named 'MilliSecond' and 'NanoSecond' using name-based mapping and assuming that appropriate type mappings (to Java long) are also provided.



**Figure 7.1.3-1: Required annotations for Java mappings**

When an element mapping refers to a non-static Java field or method then 'RuntimeHelper' TRI interface is called to get an instance object to use with the field or method specification. The class from corresponding resource mapping is used as the argument for that call.

## 7.1.4 Communication Control Flow

'SystemAdapter' implements the communication mechanisms between test execution and SUT. The interface includes methods to support both message- and procedure-based interactions. For procedure calls, the interface defines separate methods depending on whether the tester is the caller or the callee. The encoding and decoding of data is generally done by the 'SystemAdapter'.

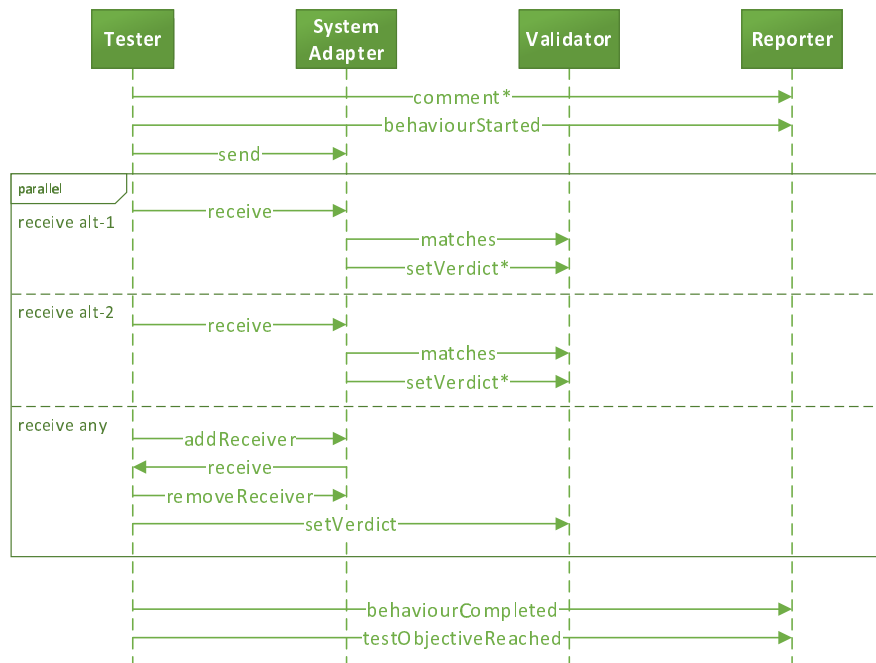
A 'SystemAdapter' implementation is assumed to be able to handle multiple concurrent calls to 'receive' method. The implementation of the 'receive' method should block until a message is received that corresponds to the data type that was provided or the call is interrupted by the caller. This means that incoming packets should support repeated decoding attempts.

If no 'receive' calls are active when a packet arrives then the 'SystemAdapter' notifies all registered 'Receiver's and pass undecoded data to them. This also happens when none of the waiting 'receive' calls correspond to received data (that is, decoding with expected type does not succeed). The registered 'Receiver's are generally used to detect discrepancies between tester and SUT behaviour.

'ignoreUntil' is a special case of receive method, which ignores and discards (that is, does not pass to asynchronous 'Receiver's) incoming data until one arrives that matches (in terms of both type and values) the expected data. A 'Validator' instance may be used for matching.

The 'call' method blocks until a reply is received (or the call is interrupted by the caller) and it returns either the return value or an exception. It is up to the caller to determine, the semantics of the returned value. The 'receiveCall' method works similarly to the 'receive' method.

The following diagram describes an example scenario of sending a message and receiving two alternative responses and a default handling with asynchronous 'Receiver'.



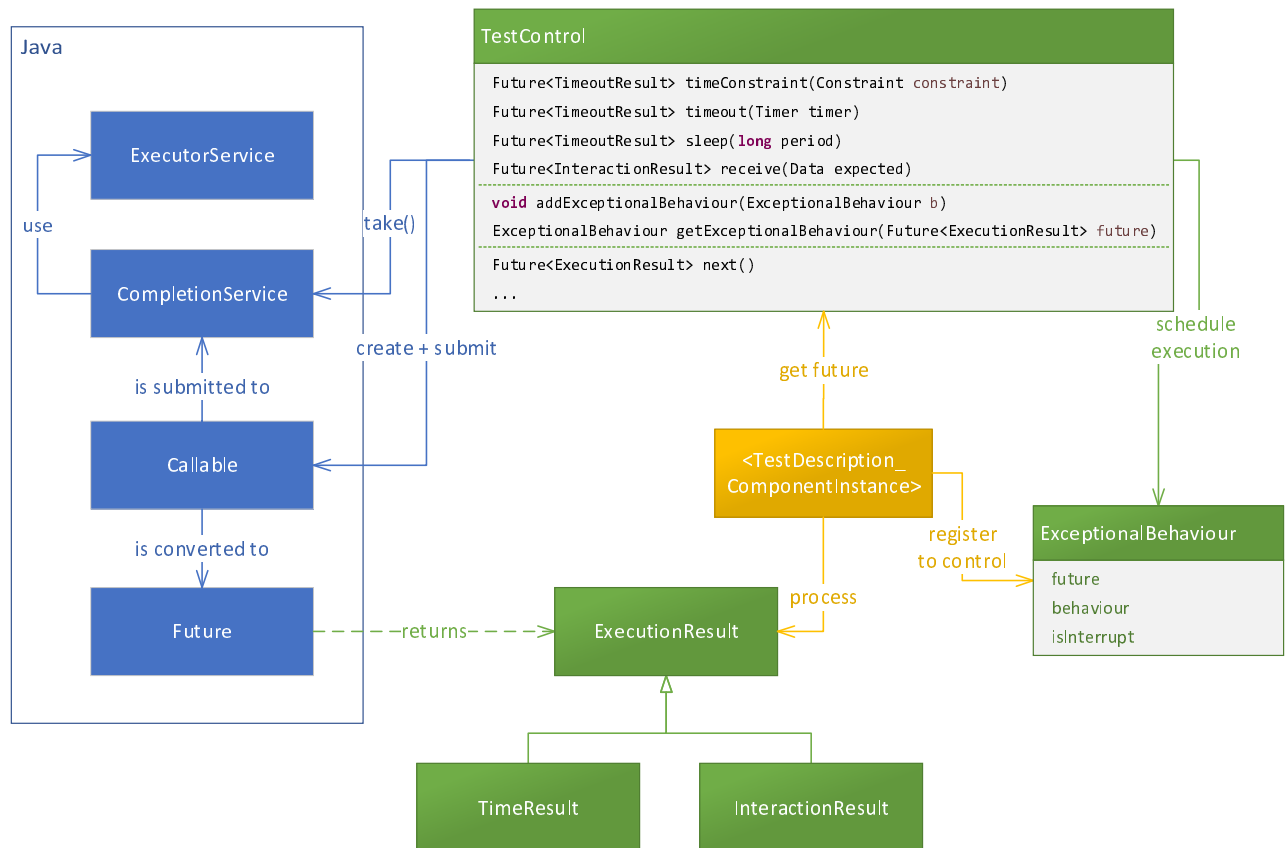
**Figure 7.1.4-1: Example of method calls involving the SystemAdapter**

To avoid excessive adaptation, the execution engine supports calling procedures directly (bypassing the system adapter) if the 'ProcedureSignature's are mapped to Java methods (see clause 7.1.3). This code generation feature is configurable in settings.

## 7.1.5 Executable Code

Most TDL behaviours have obvious counterparts in Java. For those elements and the ones that are explicitly mapped, the Java code generation is little more than language translation.

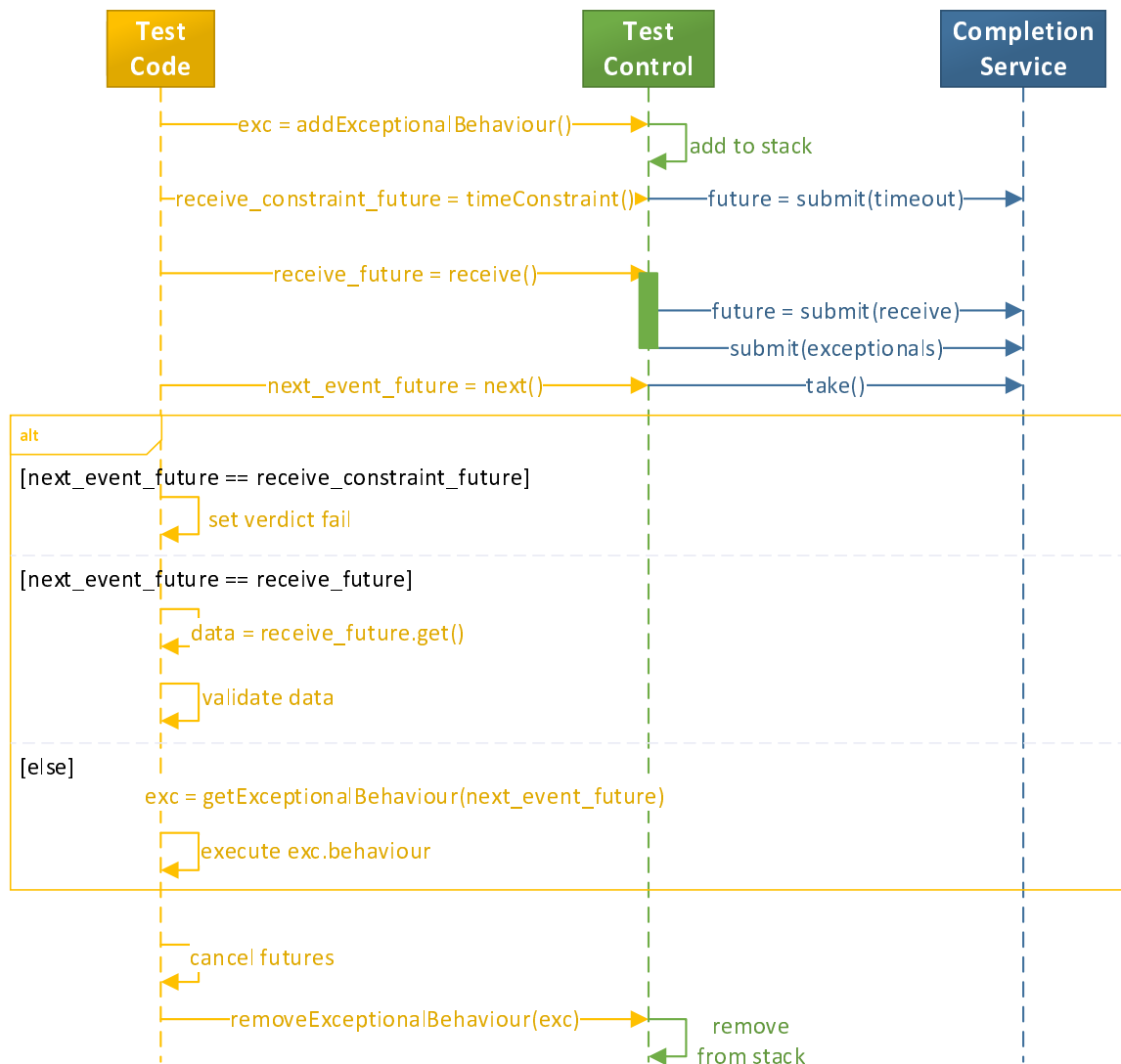
A special control structure is implemented for the handling of asynchronous events (either time- or interaction-related) and execution of out-of-order blocks (exceptional and alternative behaviours).



**Figure 7.1.5-1: Event handling component dependencies in TDL Java runtime**

The 'TestControl' class provides utility methods for creating and scheduling (submitting) the callable for various asynchronous behaviours. It also enables/disables any added/removed 'ExceptionalBehaviour's. Internally, Java's 'CompletionService' is used to take the first completed future and pass it to test code for processing.

Figure 7.1.5-2 describes an example scenario of receiving a message with time constraint while a default behaviour is activated.



**Figure 7.1.5-2: Event handling process in TDL Java runtime**

JUnit framework annotation is used to mark generated test description method as JUnit test case, which allows easy execution with any JUnit tool. JUnit assertions are used to validate data.

'TimeLabel' class is implemented in runtime according to the semantics specified in [i.13]. The time label mechanism is also used to implement TDL 'Timer's. All time units are converted into milli-seconds for internal evaluation.

Special treatment of TDL 'VariableUse's is needed as the TDL assumes that all data is immutable while in Java that is not the case. 'RuntimeHelper' 'clone' method is used to clone structured variable values before they are assigned as arguments to parameterized 'DataUse's. This prevents potential modifications of variables.

Following TDL features are currently not supported by the code generator (as the present document reflects a specific milestone):

- 'ParallelBehaviour' and 'PeriodicBehaviour'
- 'OptionalBehaviour'
- 'TestDescriptionReference' arguments

## 8 Web-based Editors and Tools

### 8.1 Overview

This clause provides guidance on the design and use of web-based editors for creating and managing test descriptions in TDL. It is based on a first exploration of available technologies and potential implementation and integration of necessary components for the realisation of web-based editors for TDL. The editors enable users to create, edit, and delete TDL test descriptions in a user-friendly and efficient manner, without the need to install and set up software. This clause describes the architecture of web-based editors, as well as the recommended best practices for its implementation. It also provides examples of how the editor can be realised and used to create various types of test descriptions and outlines the benefits of using the editor for TDL test description management. This clause is intended for developers, testers, and other stakeholders involved in the creation and management of TDL test descriptions.

The recent trend towards web-based infrastructures and Software-as-a-Service (SaaS) models for deploying Integrated Development Environments (IDEs) offers benefits, such as eliminating the need for complex software installation and configuration, enabling deployment in shared cloud environments, and ensuring platform independence. However, in scenarios where both desktop and web applications need to be supported, it is optimal to choose a technology stack that supports both. While it allows for the distribution of different language services, such as compilers and generators, across multiple platforms, it also presents challenges requiring a careful consideration, e.g. regarding the responsiveness.

This clause explores different means for implementing web-based editors for TDL, including integration into an IDE framework such as Eclipse Theia and a prototype standalone editor. The focus is on investigating the feasibility of using a modular architecture to support various products and evaluating the reusability of components to ensure productive and cost-efficient development. The exploration aims to identify the necessary adjustments and components needed to build both a simpler editor and one that can be integrated into an existing system, providing an ideal solution for different application areas and target groups.

### 8.2 Architecture

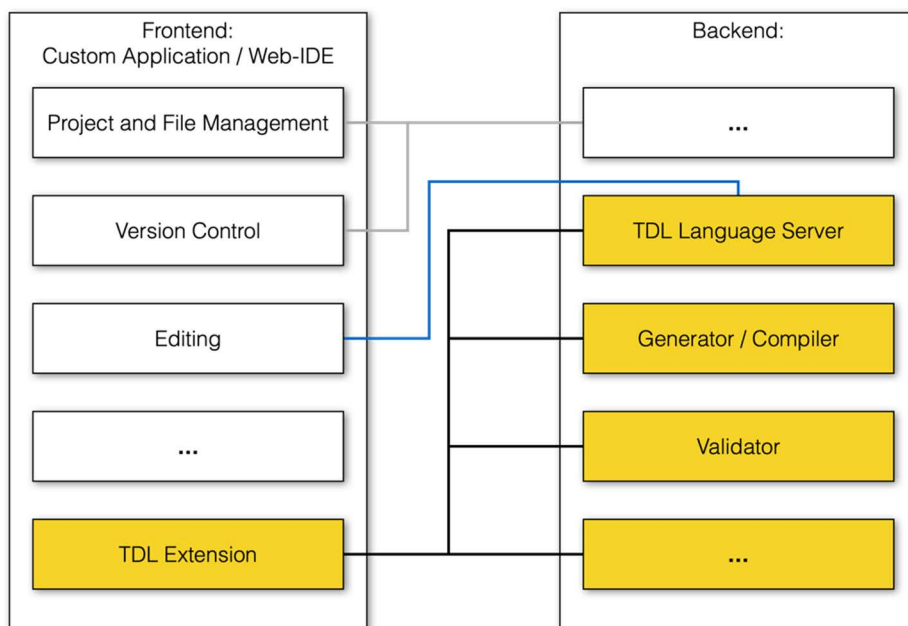
Web-based applications are typically divided into frontend and backend components. The frontend components include the user interface, which in a web-based editor for TDL will at the very least include an editor that provides syntax highlighting and error reporting, ideally also content assist and navigability. The editor would connect to a backend that processes the content of the editor and performs all the necessary tasks, providing feedback. The backend typically comprises a language server as well as generators or other components that can also be exposed through user interface elements in the frontend. A language server for TDL can be derived from the textual editors for TDL by making use of the facilities provided by the underlying framework.

The following components from TOP are the minimum subset for basic functionality including support for the brace-based textual syntax:

- org.etsi.mts.tdl.model: meta-model implementation for TDL
- org.etsi.mts.tdl.common: shared functionalities
- org.etsi.mts.tdl.tx: TDL textual syntax implementation
- org.etsi.mts.tdl.tx.ide: IDE and language server implementation for the TDL textual syntax

In addition, the org.etsi.mts.tdl.txi and org.etsi.mts.tdl.txi.ide components are needed for supporting the indentation-based syntax. Similarly, corresponding components are needed for supporting the legacy syntaxes for TDL.

A generic architecture for a web-based TDL editor is outlined in Figure 8.2-1. TDL support can be added with a frontend extension providing the user interface components as well as integration with the text editor component. In the backend, necessary components from the TOP project could be reused and exposed as services through defined interfaces, e.g. RESTful APIs or similar.



**Figure 8.2-1: Generic architecture for a web-based TDL editor**

## 8.3 Evaluation and Recommendations

### 8.3.1 Overview

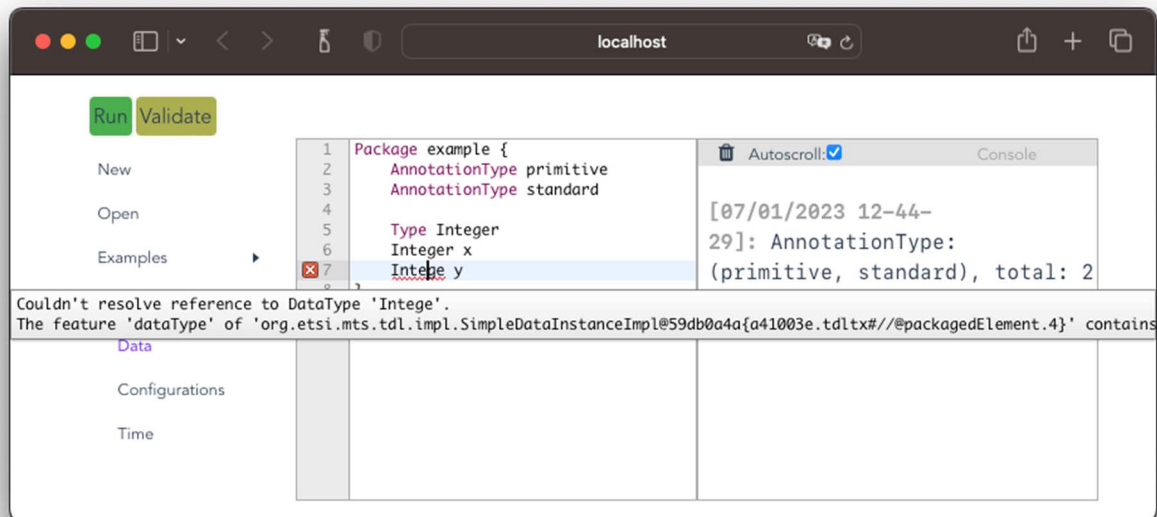
To explore the feasibility for implementing a web-based editor for TDL, both as a standalone component for integration of an arbitrary custom applications and as an extension for integration in existing web-based IDEs, two prototypes were created. The potential advantages and disadvantages of both options are summarised below, based on the experiences with the preparation of the prototypes.

### 8.3.2 Custom Application

In a custom web application, there is full flexibility in the way the editor is used and integrated, however, all functionalities need to be setup and integrated from scratch. While there may be available components for the target platform, they still need to be adapted and integrated for use with the web-based TDL editor. The prototype implementation is based on existing guides on how to get started. It only covers rudimentary functionalities beyond the editing, including an example for integrating a code-generation and execution based on the editor content. The generation example only counts the number of AnnotationType definitions. A console view shows the results from the execution. The backend includes a basic language server and generator component. An example of the resulting application is shown in Figure 8.3.2-1.

The prototypical implementation demonstrated the feasibility of incorporating TDL-specific editors in a custom web application, divided into frontend and backend components. The architecture allows for the addition of more components, as shown with the code generation functionality. Challenges were encountered due to dependencies and modifications needed for language server. Integrating a graphical viewer or editor, as well as other capabilities can be a next step for a custom application.





**Figure 8.3.2-1: Web-based TDL editor in a custom application**

### 8.3.3 Web-based IDE Extension

The host web-based IDE provides the basic workbench capabilities including a basic text editor that can be connected to a language server in order to provide augmented editing capabilities. Rather than starting from scratch, the emf.cloud demonstrator was used as a starting point as it provides many of the needed functionalities and the necessary integrations. The EMF.cloud project aims to adapt the Eclipse Modelling Framework for use in web-based platforms. The demonstrator is built on top of Eclipse Theia and showcases common functionalities for model-based domain-specific language tooling, including textual, graphical, and form-based editors backed by the same underlying model which is exposed by means of a model-server component. Using the demonstrator as a blueprint, adaptations for TDL-specific editors were identified and prototypically implemented. An example is shown in Figure 8.3.3-1.

The prototypical implementation demonstrated the feasibility of incorporating TDL-specific editors in a web-based IDE. However, the integration process was complex and required extensive debugging. The demonstrator code base is moving very fast, with frequent major refactorings and updates to dependencies leading to compatibility issues and unexpected bugs. Despite these challenges, the prototype implementation can serve as a foundation for future activities, such as integrating a graphical editor and other capabilities.

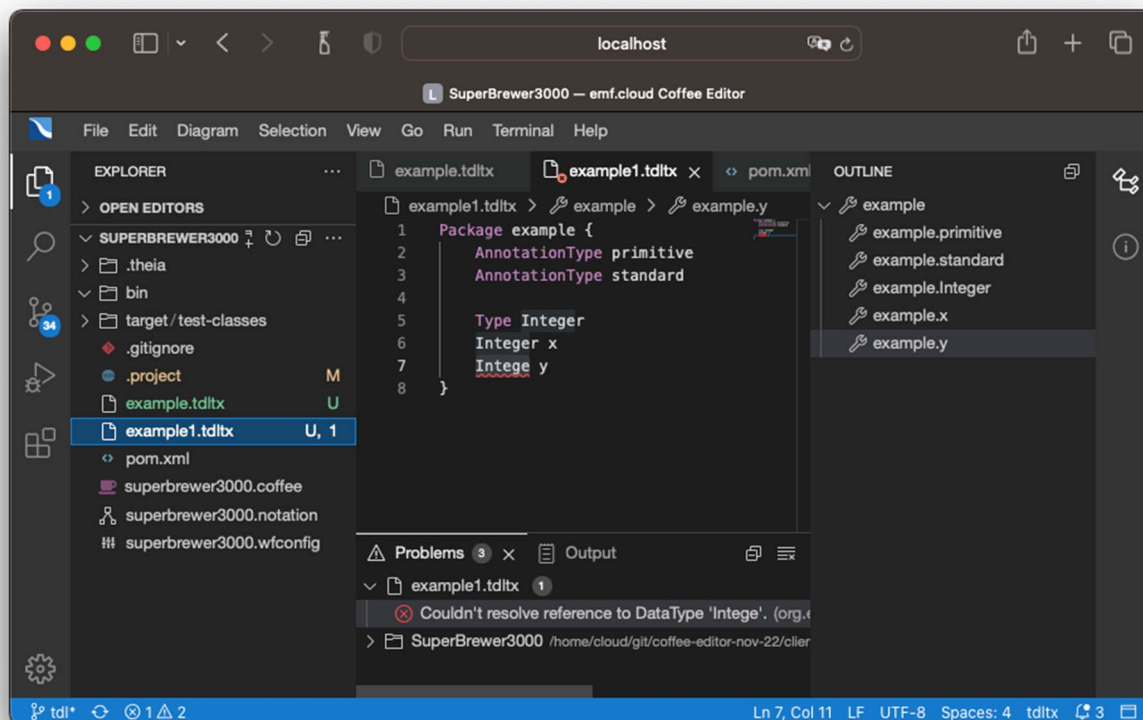


Figure 8.3.3-1: Web-based TDL editor in an online IDE

### 8.3.4 Recommendation

Two web-based editors were prototypically implemented, each with its own set of challenges. Building the editor from scratch appeared more promising for simple tasks, providing better control compared to the complex platform of Eclipse Theia and the emf.cloud demonstrator. It is also better suited for integration in other custom platforms and applications, as well as for demonstration and learning purposes, e.g. on the TDL website. However, for larger programming teams, constructing a full-fledged IDE is better suited. The resulting artifacts can also be used for other IDEs, such as Visual Studio Code.

The web-based editor can be accessed via this link: <https://top.etsi.org/ttfs/page-test/index.html>.

---

## Annex A:

# Technical Realisation of the Reference Implementation

The technical representation of the TDL reference implementation is available as an open source project available on the [TOP website](#). The open source project serves as a possible starting point for implementing and extending tools for TDL as described in the present document. An open source project is well suited for a technical contribution which can, over time, evolve beyond the scope of the present document. Further information regarding the use of the technical representation as well as contributing to it can be found on the [TDL website](#).

---

## Annex B: UML Profile Editor

### B.1 Scope and Requirements

The UML Profile for TDL (UP4TDL) was developed to enable the application of TDL in UML based working environments. UP4TDL introduces TDL-related domain-specific concerns to the UML meta-model by means of stereotypes which extend UML meta-classes with additional properties, relations, or constraints. The implementation of the UP4TDL covers basic functionalities to support the creation and manipulation of UML models applying the UP4TDL profile.

NOTE: The UML profile editor description is not aligned with the latest version of the TDL specification parts, but are related to an earlier release of the TDL specification parts.

---

### B.2 Architecture and Technology Foundation

The UML based editor is also built on top of the Eclipse platform. At a high level, it contains two main components: the UML Profile for TDL (UP4TDL) implementation described in ETSI ES 203 119-1 [i.13], annex C, and the facilities for editing UP4TDL models. The profile is static. This allows the implementation of derived properties. The profile implementation is independent of the editing facilities provided in the context of this reference implementation and can be used by other UML tools. A model-to-model transformation from UP4TDL models to TDL Ecore models allows generating TDL in the XMI format specified in ETSI ES 203 119-3 [i.15].

The TDL profile implementation is located in the 'org.etsi.mts.up4tdl' project, while the validation implementation is located in the 'org.etsi.mts.up4tdl.validation' project. The implementation of the editing facilities can be found in the 'org.etsi.mts.up4tdl.diagram.\*' projects. The 'ElementType' framework is used for manipulating model elements in Papyrus. Specialized 'ElementType's are in located in the 'org.etsi.mts.up4tdl.service.type' project.

---

### B.3 Implemented Facilities

#### B.3.1 Applying the Profile

##### Overview

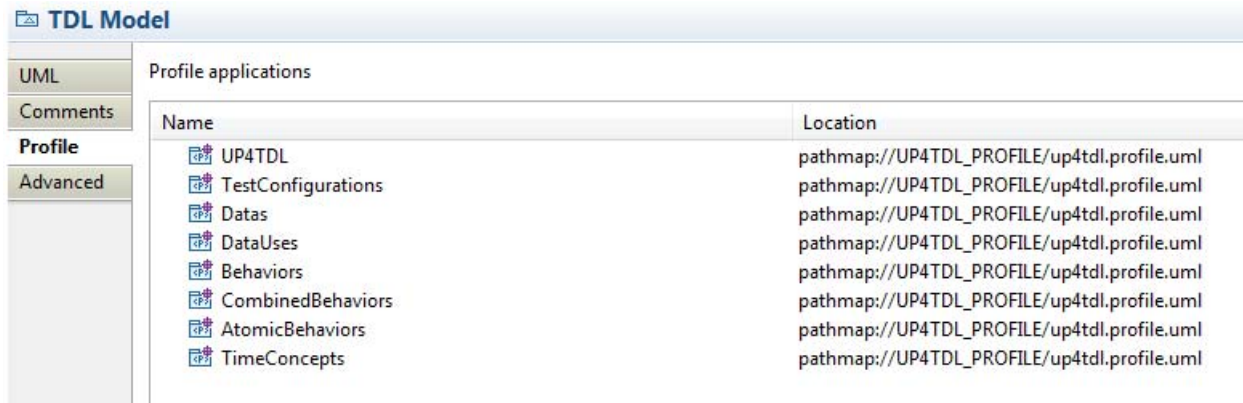
A UML profile allows users to build models with additional constraints and specific properties, while still relying on the UML meta-model. A UP4TDL model is then a UML model with additional constraints and properties tailored towards the domain of TDL.

##### Stereotype

The extension mechanism of a UML profile is based on *stereotypes*. A stereotype of a UML profile always *extends* (directly or indirectly) a UML meta-class. For example the 'ComponentInstance' concept from TDL extends the 'Property' concept of UML and it has the specific property allowing users to define its role ('Tester' or 'SUT').

##### Applying the UP4TDL profile on a UML model

Applying UP4TDL concepts on a UML model implies the application of UP4TDL stereotypes on UML elements. To do this, the UP4TDL profile (or one of its sub-profiles) will be added to the package (or the model) containing the UML element as shown in Figure B.3.1-1.



The screenshot shows the 'TDL Model' window with a sidebar on the left containing 'UML', 'Comments', 'Profile', and 'Advanced'. The 'Profile' tab is selected, displaying a table of 'Profile applications'.

Name	Location
UP4TDL	pathmap://UP4TDL_PROFILE/up4tdl.profile.uml
TestConfigurations	pathmap://UP4TDL_PROFILE/up4tdl.profile.uml
Datas	pathmap://UP4TDL_PROFILE/up4tdl.profile.uml
DataUses	pathmap://UP4TDL_PROFILE/up4tdl.profile.uml
Behaviors	pathmap://UP4TDL_PROFILE/up4tdl.profile.uml
CombinedBehaviors	pathmap://UP4TDL_PROFILE/up4tdl.profile.uml
AtomicBehaviors	pathmap://UP4TDL_PROFILE/up4tdl.profile.uml
TimeConcepts	pathmap://UP4TDL_PROFILE/up4tdl.profile.uml

Figure B.3.1-1: UML profile application

The stereotype applied on the UML model allow the specification of stereotype properties. In Figure B.3.1-2, the stereotype 'ComponentInstance' is applied to a 'UML::Property'. This allows the user to specify the role property, in this case, 'tester'.

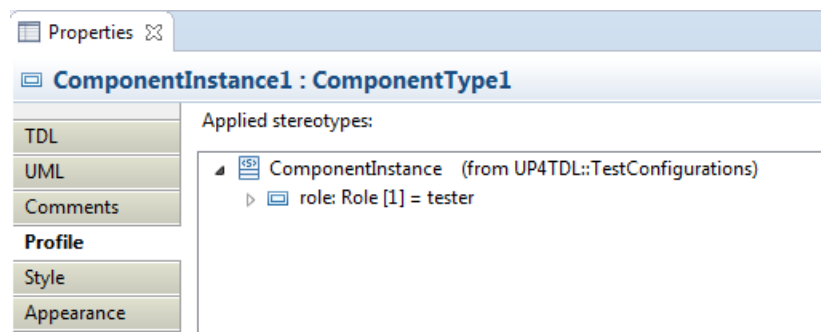


Figure B.3.1-2: Stereotype property specification

## B.3.2 Hints for the Transformation of UP4TDL Models into TDL Models

### Overview

Most translations are straightforward one-to-one mappings between UP4TDL concepts and concepts from TDL meta-model. The exceptions are detailed below.

### ElementImport

In TDL, 'ElementImport' can reference several 'Element's, while in UML, the corresponding concept 'UML::ElementImport' concept (direct mapping without stereotype) can only reference one. So the model-to-model transformation can potentially turn one 'TDL::ElementImport' into several 'UML::ElementImport's.

### SimpleDataInstance and StructuredDataInstance

Both 'SimpleDataInstance' and 'StructuredDataInstance' are mapped to the same concept 'UML::InstanceSpecification'. To determine whether it is a simple or a structured instance, one needs to check the type of the 'UML::InstanceSpecification'. If the 'InstanceSpecification's type is a 'PrimitiveType', then it is a 'SimpleDataInstance', otherwise it is a 'StructuredDataInstance'.

## Property Identification

There are two direct mappings from 'UML::Property' to TDL concepts - for 'TDL::Variable' and 'TDL::Member'. In order to determine which kind of property it is, one needs to check the container. If the property is contained in a 'ComponentInstance', then it corresponds to a 'Variable'. Otherwise, if the property is contained in a 'DataType', then it corresponds to a 'Member'.

## B.3.3 Editing Models with the Model Explorer

As shown in clause B.3.1, UP4TDL elements can be created from UML elements by applying a stereotype on them. Both steps can be performed in a row from the model explorer, using TDL specific 'New TDL Child' creation options. The model elements are sorted in the 'New TDL Child' menu according to the diagram they are supposed to appear in, as shown in Figure B.3.3-1.

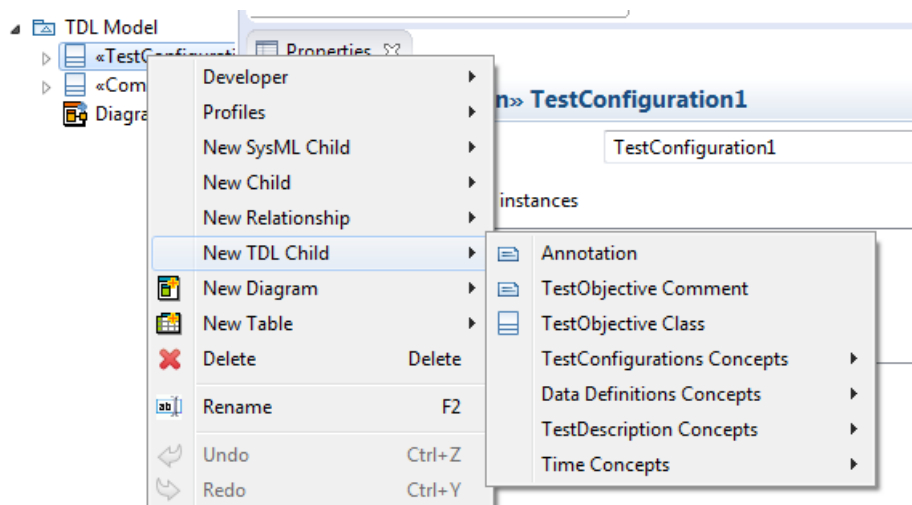


Figure B.3.3-1: Adding TDL-stereotyped elements

## B.3.4 Editing TDL-specific Properties with the TDL Property View

Editing the properties of a UP4TDL model with the standard property view, can be inconvenient for two reasons. On the one hand, some properties from the UML base meta-class are not relevant for the associated TDL 'Element'. On the other hand, some properties of a TDL 'Element' are not properties of the base meta-class. Even when the properties of a TDL 'Element' and the base UML 'Element' match, they might not have the same name. Editing a UP4TDL model would then require expertise in both UML and TDL, as well as knowledge of the UP4TDL profile specifics. There is a 'TDL Tab' for the property view, which makes the task of editing TDL specific properties easier. Figure B.3.4-1 illustrates the property view of a 'ComponentInstance', which contains its 'Name', 'Type', and 'Role' properties.

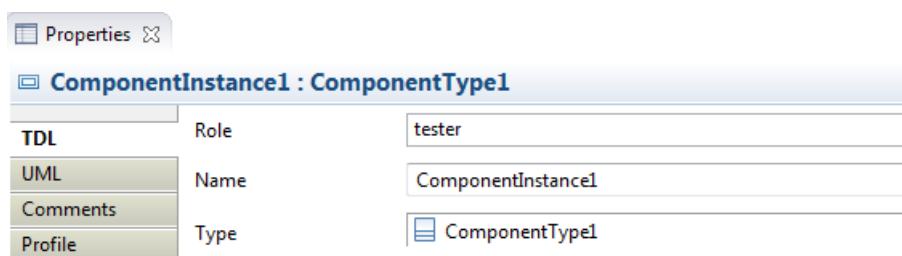


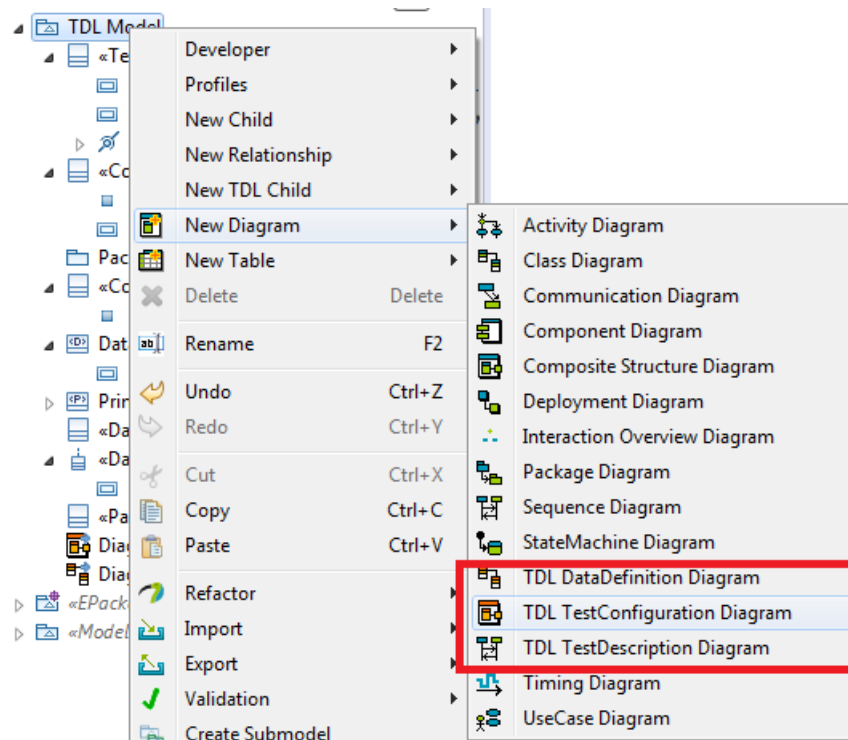
Figure B.3.4-1: Editing TDL-specific properties

## B.3.5 Editing Models with TDL-specific Diagrams

### Overview

Editing a UP4TDL model can be done using the property view and model explorer only. In order to provide a graphical representation of a model being edited, TDL Diagrams specializing UML Diagrams are implemented. There are 3 kinds of TDL Diagrams: TDL DataDefinition Diagram, TDL TestConfiguration Diagram and TDL TestDescription Diagram. There are two main editing facilities for all of these diagrams: the creation of an element using the 'palette' and the 'drag and drop' of an existing element from the model explorer.

The TDL-specific diagrams can be initialized from the model explorer as shown in Figure B.3.5-1.



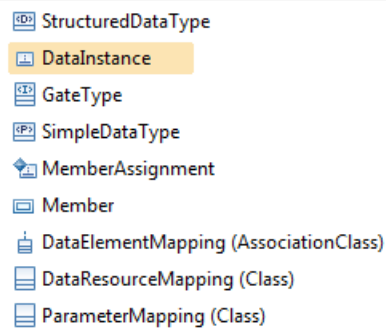
**Figure B.3.5-1: Creating TDL-specific diagrams**

### The TDL DataDefinition Diagram

The DataDefinition Diagram is based on the UML Class Diagram. It is used to represent the following TDL Elements:

- StructuredDataType
- SimpleDataType
- MemberAssignment
- Member
- DataElementMapping
- DataResourceMapping
- ParameterMapping
- DataInstance
- GateType

The palette for the TDL DataDefinition diagram is shown in Figure B.3.5-2.



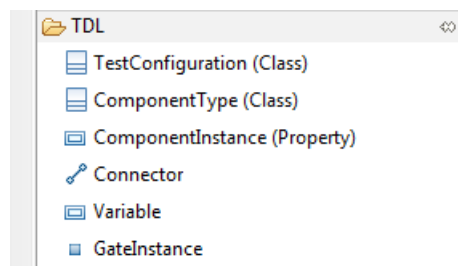
**Figure B.3.5-2: DataDefinition Diagram palette**

### The TDL TestConfiguration Diagram

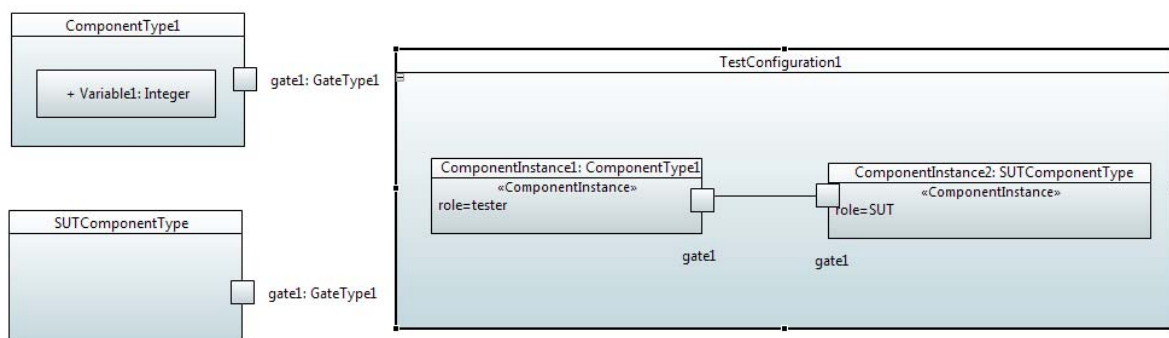
The TestConfiguration Diagram is based on the UML Composite Diagram. It is used to represent the following TDL elements:

- TestConfiguration
- ComponentInstance
- ComponentType
- GateInstance
- Connection
- Variable

The palette for the TDL TestConfiguration Diagram is shown in Figure B.3.5-3. An example of the TDL TestConfiguration Diagram is shown in Figure B.3.5-4.



**Figure B.3.5-3: TestConfiguration Diagram palette**



**Figure B.3.5-4: TestConfiguration Diagram example**



The following specific behaviours have been implemented for the TestConfiguration Diagram:

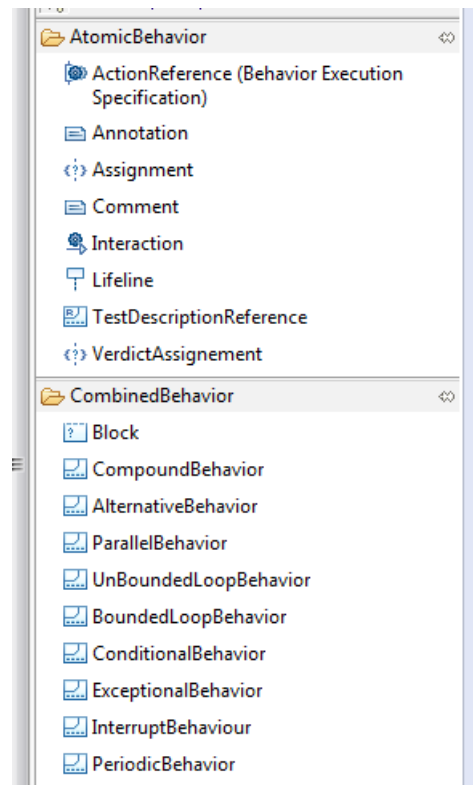
- Dragging a 'ComponentType' to a 'ComponentInstance' specifies the type of the 'ComponentInstance'.
- Dragging a 'GateType' to a 'GateInstance' specifies the type of the 'GateInstance'.
- Dragging a 'GateInstance' from the palette on a 'ComponentInstance' adds it to its 'ComponentType'.

#### Editing a TDL TestDescription Diagram

The TestDescription Diagram is based on the UML Sequence Diagram. It is used to represent the following TDL elements:

- TestDescription
- Annotation
- Comment
- Lifeline
- CombinedBehaviours:
  - Block
  - CompoundBehaviour
  - AlternativeBehaviour
  - ParallelsBehaviour
  - UnboundedLoopBehaviour
  - BoundedLoopBehaviour
  - ConditionalBehaviour
  - ExceptionalBehaviour
  - InterruptBehaviour
  - PeriodicBehaviour
- AtomicBehaviours:
  - ActionReference
  - Assignment
  - Interaction
  - TestDescriptionReference
  - VerdictAssignment

The palette for the TDL TestDescription Diagram is shown in Figure B.3.5-5.



**Figure B.3.5-5: TestDescription Diagram palette**

---

# History

Document history		
V1.1.1	February 2018	Publication
V1.2.1	September 2020	Publication
V1.3.1	March 2023	Publication
V1.4.1	September 2023	Publication
V1.5.1	May 2025	Publication