



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

Reference

RTR/MTS-TDL103119v131

Keywords

MBT

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:

<http://www.etsi.org/standards-search>

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 at www.etsi.org/deliver.

Users of the present document should be aware that the document may be subject to revision or change of status.

Information on the current status of this and other ETSI documents is available at

<https://portal.etsi.org/TB/ETSIDeliverableStatus.aspx>

If you find errors in the present document, please send your comment to one of the following services:

<https://portal.etsi.org/People/CommitteeSupportStaff.aspx>

If you find a security vulnerability in the present document, please report it through our
Coordinated Vulnerability Disclosure Program:

<https://www.etsi.org/standards/coordinated-vulnerability-disclosure>

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 2022.

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.....	9
3.1 Terms.....	9
3.2 Symbols.....	9
3.3 Abbreviations	9
4 Basic Principles	9
4.1 Introduction	9
4.2 Implementation Scope	10
4.3 Document Structure.....	11
5 Graphical Representation Editor	11
5.1 Scope and Requirements	11
5.2 Architecture and Technology Foundation	11
5.2.1 Graphical Editor.....	11
5.2.2 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	29
6 Using TDL with TOP.....	30
6.1 Usage Scenarios	30
6.2 Defining Structured Test Objectives	31
6.2.0 Overview	31
6.2.1 Domain part of TDL-TO.....	31
6.2.2 Data definitions.....	32
6.2.3 Configuration.....	32
6.2.4 Test purpose behaviour	33
6.3 Transforming Test Objectives into Test Descriptions	34
6.3.1 Overview	34
6.3.2 Data.....	34
6.3.3 Configurations	36
6.3.4 Behaviour.....	37
6.3.5 Transformation Conventions and Assumptions.....	38
6.4 Defining Test Descriptions.....	40
6.4.1 Overview	40
6.4.2 Data and Configuration.....	40
6.4.3 Test Behaviour and Time.....	41
6.5 Transforming Test Descriptions into TTCN-3 Test Cases	42
6.5.1 Overview	42
6.5.2 Data.....	42
6.5.3 Configuration.....	43
6.5.4 Behaviour.....	44
7 UML Profile Editor	46

7.1	Scope and Requirements	46
7.2	Architecture and Technology Foundation	46
7.3	Implemented Facilities	46
7.3.1	Applying the Profile.....	46
7.3.2	Hints for the Transformation of UP4TDL Models into TDL Models.....	47
7.3.3	Editing Models with the Model Explorer	48
7.3.4	Editing TDL-specific Properties with the TDL Property View	48
7.3.5	Editing Models with TDL-specific Diagrams.....	48
8	Using TDL with External Data Type Specifications.....	53
8.1	Generalized Process	53
8.1.1	Process Overview	53
8.1.2	Example Instantiation	55
8.2	Using TDL with OpenAPI™ Specifications	56
8.2.1	Overview	56
8.2.2	Examples	57
8.3	Using TDL with ASN.1 Specifications	58
8.3.1	Overview	58
8.3.2	Examples	60
9	TDL Runtime / Execution	62
9.1	Java: Code generator	62
9.1.1	Architecture	62
9.1.2	Test Runtime Interface (TRI) for Java.....	64
9.1.2.1	Overview	64
9.1.2.2	TRI: SystemAdapter.....	66
9.1.3	Mappings	67
9.1.4	Executable Code	68
Annex A:	Technical Realisation of the Reference Implementation.....	70
History		71

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 Web server (<https://ipr.etsi.org/>).

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™** and **LTE™** 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.

Foreword

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

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.

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 part 1 of the multi-part deliverable [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. It also provides usage instructions for end users.

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 and tool-support for the mapping TDL elements to TTCN-3 code;
- implementation and tool-support for 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 are not necessary for the application of the present document but they assist the user with regard to a particular subject area.

[i.1] Eclipse Foundation™: Eclipse IDE Website (last visited 20.12.2021).

NOTE: Available at <https://eclipse.org>.

[i.2] Eclipse Foundation™: Eclipse Xtext™ Website (last visited 20.12.2021).

NOTE: Available at <https://eclipse.org/Xtext/index.html>.

- [i.3] Eclipse Foundation™: Eclipse Sirius™ Website (last visited 20.12.2021).
NOTE: Available at <http://www.eclipse.org/sirius/index.html>.
- [i.4] Eclipse Foundation™: Eclipse Modeling Framework (EMF™) Website (last visited 20.12.2021).
NOTE: Available at <http://www.eclipse.org/modeling/emf/>.
- [i.5] Eclipse Foundation™: Eclipse Papyrus™ Modeling Environment Website (last visited 20.12.2021).
NOTE: Available at <https://www.eclipse.org/papyrus/>.
- [i.6] Eclipse Foundation™: UML™ Profiles Repository Website (last visited 20.12.2021).
NOTE: Available at <https://projects.eclipse.org/projects/modeling.upr>.
- [i.7] Eclipse Foundation™: Graphical Modeling Framework (GMF™) Website (last visited 20.12.2021).
NOTE: Available at <http://www.eclipse.org/modeling/gmp/>.
- [i.8] "Object Constraint Language™ (OMG® OCL™), Version 2.4", formal/2014-02-03.
NOTE: Available at <http://www.omg.org/spec/OCL/2.4/>.
- [i.9] Eclipse Foundation™: Eclipse OCL™ Website (last visited 20.12.2021).
NOTE: Available at <https://projects.eclipse.org/projects/modeling.mdt.ocl>.
- [i.10] Plutext Pty Ltd: Docx4j Website (last visited 20.12.2021).
NOTE: Available at <http://www.docx4java.org/trac/docx4j>.
- [i.11] "OMG® XML™ Metadata Interchange (XMI™) Specification", Version 2.4.2, formal/2014-04-04.
NOTE: Available at <http://www.omg.org/spec/MOF/2.4.2/>.
- [i.12] Eclipse Foundation™: Epsilon™ Validation Language (EVL™) Website (last visited 20.12.2021).
NOTE: Available at <http://www.eclipse.org/epsilon/doc/evl/>.
- [i.13] ETSI ES 203 119-1 (V1.6.1): "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 (V1.5.1): "Methods for Testing and Specification (MTS); The Test Description Language (TDL); Part 2: Graphical Syntax".
- [i.15] ETSI ES 203 119-3 (V1.5.1): "Methods for Testing and Specification (MTS); The Test Description Language (TDL); Part 3: Exchange Format".
- [i.16] ETSI ES 203 119-4 (V1.5.1): "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 (V1.1.1): "Methods for Testing and Specification (MTS); The Test Description Language (TDL); Part 5: UML Profile for TDL".
- [i.18] ETSI ES 203 119-6 (V1.3.1): "Methods for Testing and Specification (MTS); The Test Description Language (TDL); Part 6: Mapping to TTCN-3".
- [i.19] ETSI ES 203 119-7 (V1.3.1): "Methods for Testing and Specification (MTS); The Test Description Language (TDL); Part 7: Extended Test Configurations".
- [i.20] ETSI ES 203 119-8 (V1.1.1): "Methods for Testing and Specification (MTS); The Test Description Language (TDL); Part 8: Textual Syntax".

- [i.21] ETSI EG 203 130 (V1.1.1): "Methods for Testing and Specification (MTS); Model-Based Testing (MBT); Methodology for standardized test specification development".
- [i.22] The Apache Software Foundation: Apache POI Website (last visited 20.12.2021).
- NOTE: Available at <https://poi.apache.org>.
- [i.23] ETSI: The TDL Website (last visited 20.12.2021).
- NOTE: Available at <https://tdl.etsi.org>.
- [i.24] ETSI: The TDL Open Source Project Website (last visited 20.12.2021).
- NOTE: Available at <https://tdl.etsi.org/index.php/open-source>.
- [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] ETSI TS 103 029: "IMS Network Testing (INT); IMS & EPC Interoperability test descriptions (3GPP Release 10)".
- [i.27] ETSI TS 129 214 (V15.6.0): "Universal Mobile Telecommunications System (UMTS); LTE; Policy and charging control over Rx reference point (3GPP TS 29.214 version 15.6.0 Release 15)".
- [i.28] Javadoc documentation generator for Java™.
- NOTE: Available at <https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html>.
- [i.29] Junit testing framework.
- NOTE: Available at <https://junit.org>.
- [i.30] Guice dependency injection framework.
- NOTE: Available at <https://github.com/google/guice>.
- [i.31] OpenAPI™ Specification, Version 3.0.3.
- NOTE: Available at <https://swagger.io/specification/>.
- [i.32] ETSI EG 203 647 (V1.1.1): "Methods for Testing and Specification (MTS); Methodology for RESTful APIs specifications and testing".
- [i.33] IETF draft-bhutton-json-schema-00: "JSON Schema: A Media Type for Describing JSON Documents", December 8, 2020.
- NOTE: Available at <https://datatracker.ietf.org/doc/html/draft-bhutton-json-schema-00>.
- [i.34] Recommendation ITU-T X.680: "Information technology - Abstract Syntax Notation One (ASN.1): Specification of basic notation". 02/2021.
- [i.35] ETSI TS 103 666-1 (V15.0.0): "Smart Secure Platform (SSP); Part 1: General characteristics (Release 15)".
- [i.36] Recommendation ITU-T X.681: "Information technology - Abstract Syntax Notation One (ASN.1): Information object specification". 02/2021.
- [i.37] ETSI TS 103 597-3 (V1.1.1): "Methods for Testing and Specification (MTS); Test Specification for MQTT; Part 3: Performance Tests".
- [i.38] ISO 8601: "Date and time format".

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
MBT	Model-Based Testing
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 editors for ETSI ES 203 119-8 [i.20] and ETSI ES 203 119-1 [i.13], annex B are 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™ and ASN.1 specifications are also provided as a prototype.

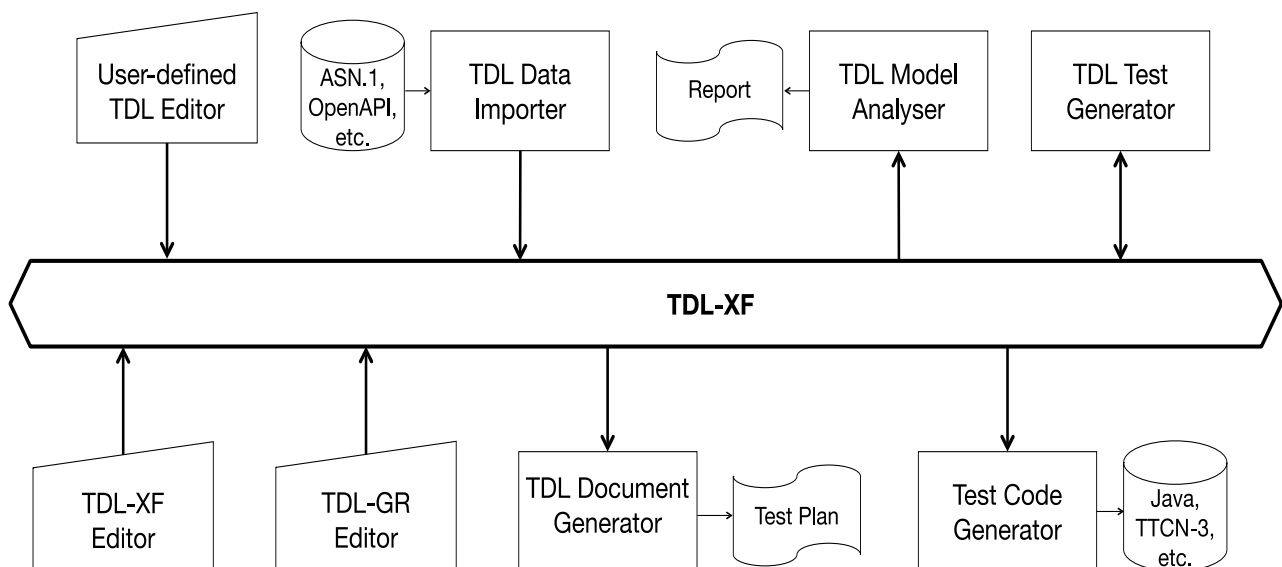


Figure 4.2-1: TDL tool infrastructure

An 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. Illustrative examples and guidelines for the use of TDL to address common use cases with the help of the TOP are described in clause 6. The UML Profile Editor implementing ETSI ES 203 119-5 [i.17] is described in clause 7. The use of TDL with external data specifications is discussed in clause 8. The implementation of an execution environment for the testing of RESTful API services with TDL is outlined in clause 9.

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

5 Graphical Representation Editor

5.1 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.

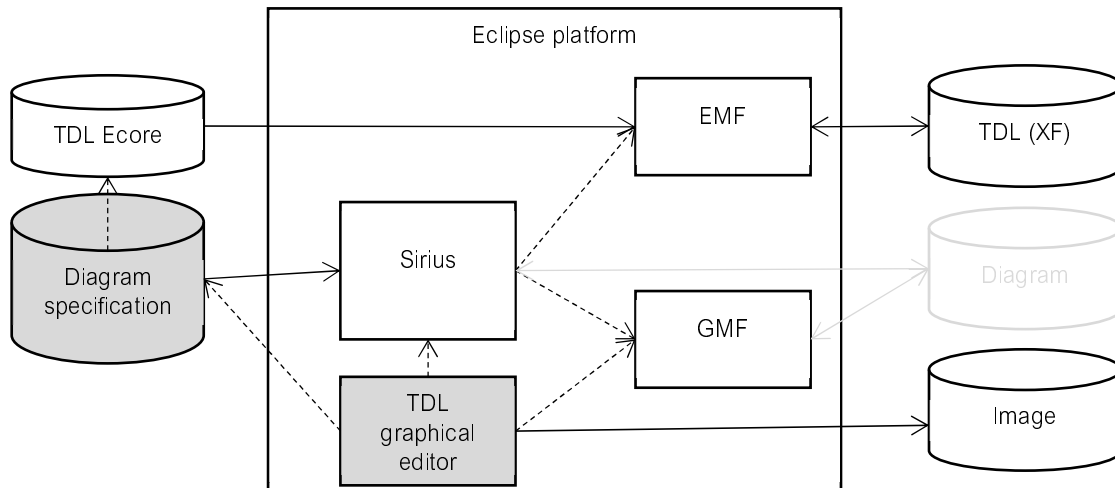
Diagram export for documentation purposes is provided by the framework. The implementation can provide complimentary export to the Word document format.

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 are 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.

5.2 Architecture and Technology Foundation

5.2.1 Graphical Editor

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 5.2.1-1. 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 5.2.1-1: 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].

5.2.2 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 5.2.2-1. 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.

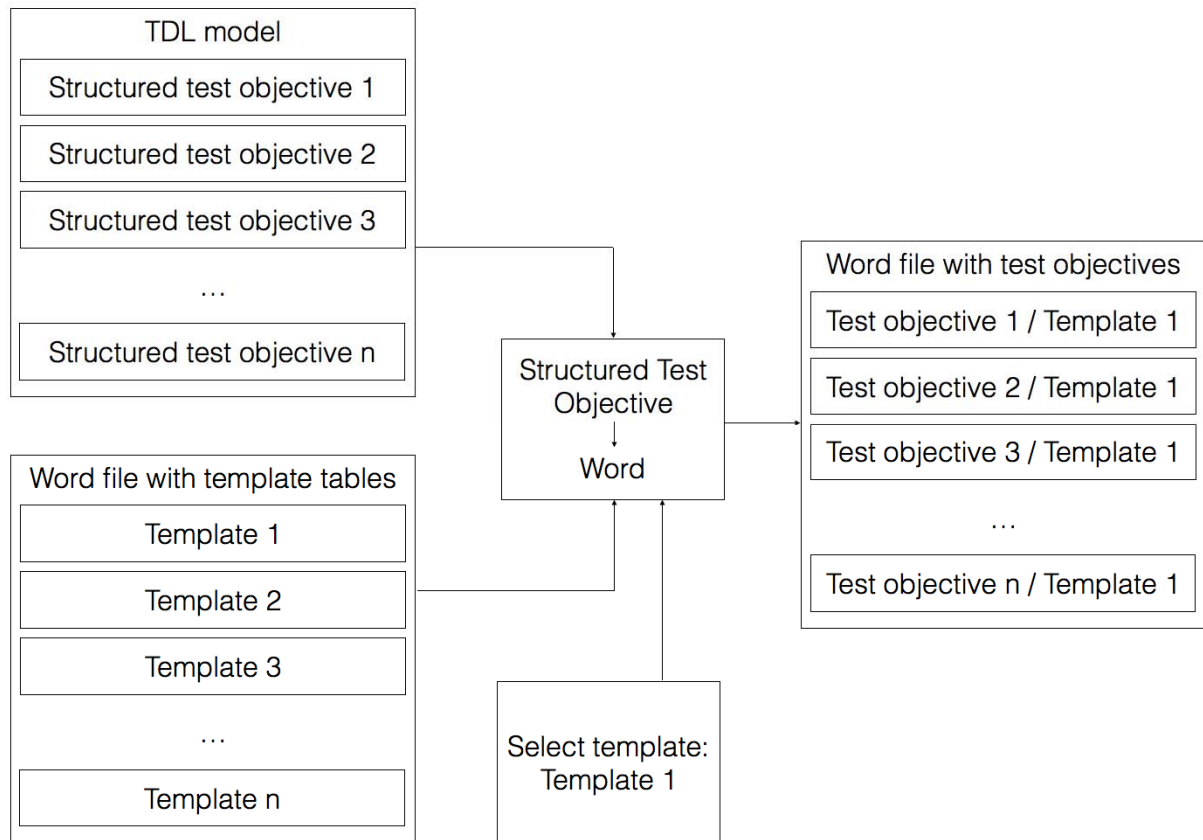


Figure 5.2.2-1: Structured test objective generation process

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.

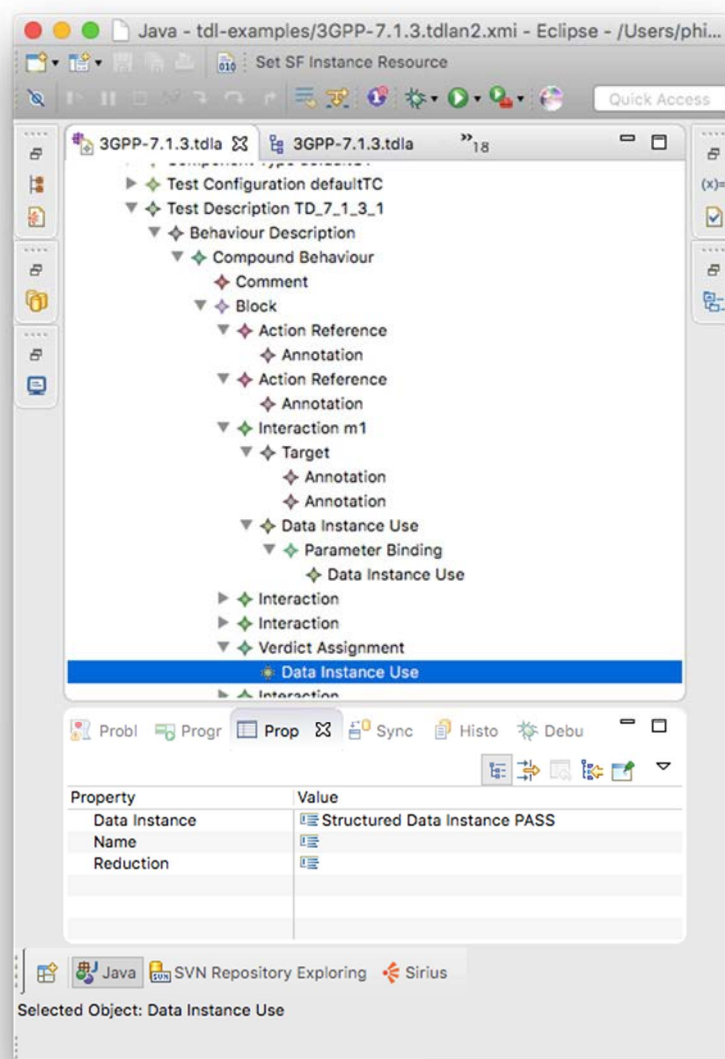


Figure 5.3.1-1: Example of reflective model editor

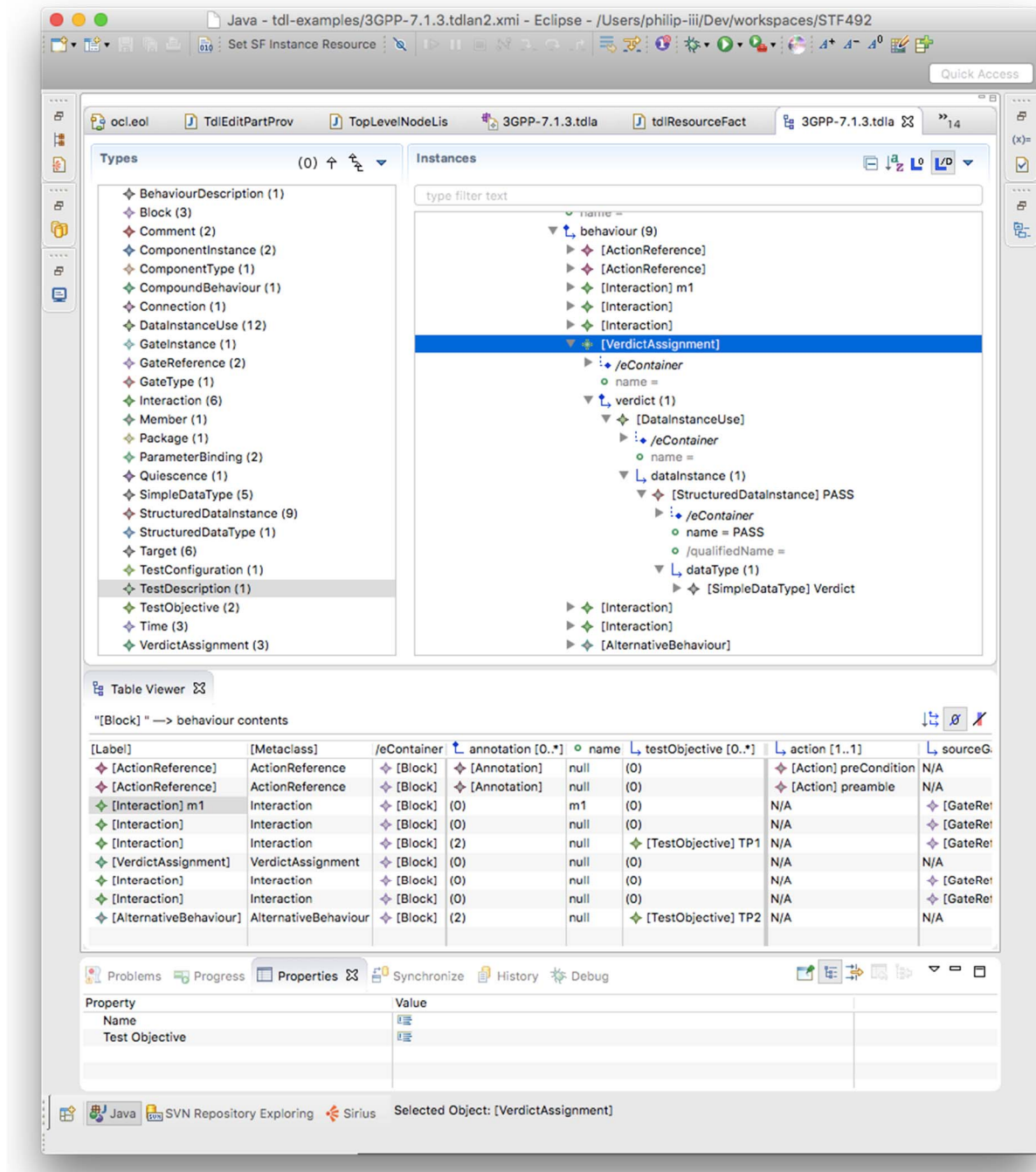


Figure 5.3.1-2: Example of MoDisco faceted model browser

Textual Editor

Xtext [i.2] 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 screenshot displays the Eclipse IDE interface. The main editor window shows a Java file named `3GPP-7.1.3.tdla` with the following content:

```

83     }
84
85     //Test configuration definition
86     Test Configuration defaultTC {
87         create Tester SS of type defaultCT;
88         create SUT UE of type defaultCT ;
89         connect UE.g to SS.g ;
90     }
91
92     //Test description definition
93     Test Description TD_7_1_3_1 uses configuration defaultTC {
94         //Pre-conditions and preamble from the source document
95         perform action precondition with { PRECONDITION ; } ;
96         perform action preamble with { PREAMBLE ; } ;
97
98         //Test sequence
99         SS.g sends pdccch (c_rnti=ue) to UE.g with {}
100        SS.g sends mac_pdu to UE.g with {}
101        UE.g sends harq_ack to SS.g with {}
102        set verdict to PASS ;
103        SS.g sends pdccch (c_rnti=unknown) to UE.g with {}
104        SS.g sends mac_pdu to UE.g with {}

```

The right-hand side of the IDE shows the Project Explorer, displaying the project structure. The selected object is `Data Instance Use`, which is highlighted in blue. The structure includes:

- Time NS
 - Simple Data Instance five
 - Gate Type defaultGT
 - Component Type defaultCT
 - Gate Instance g
 - Test Configuration defaultTC
 - Component Instance SS
 - Component Instance UE
 - Connection
 - Gate Reference UE.g
 - Gate Reference SS.g
 - Test Description TD_7_1_3_1
 - Behaviour Description
 - Compound Behaviour
 - Comment
 - Block
 - Action Reference
 - Action Reference
 - Interaction
 - Interaction
 - Interaction
 - Verdict Assignment
 - Data Instance Use** (Selected)
 - Interaction
 - Interaction

The bottom of the IDE shows the Properties view, which is currently empty. The status bar at the bottom indicates the selected object is `Data Instance Use`.

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 B of ETSI ES 203 119-4 [i.16] from clause [ref] 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 B of ETSI ES 203 119-4 [i.16] and as part of the 'org.etsi.mts.tdl.TDLtx*' projects for the syntax from clause [ref] of ETSI ES 203 119-4 [i.16].

ETSI

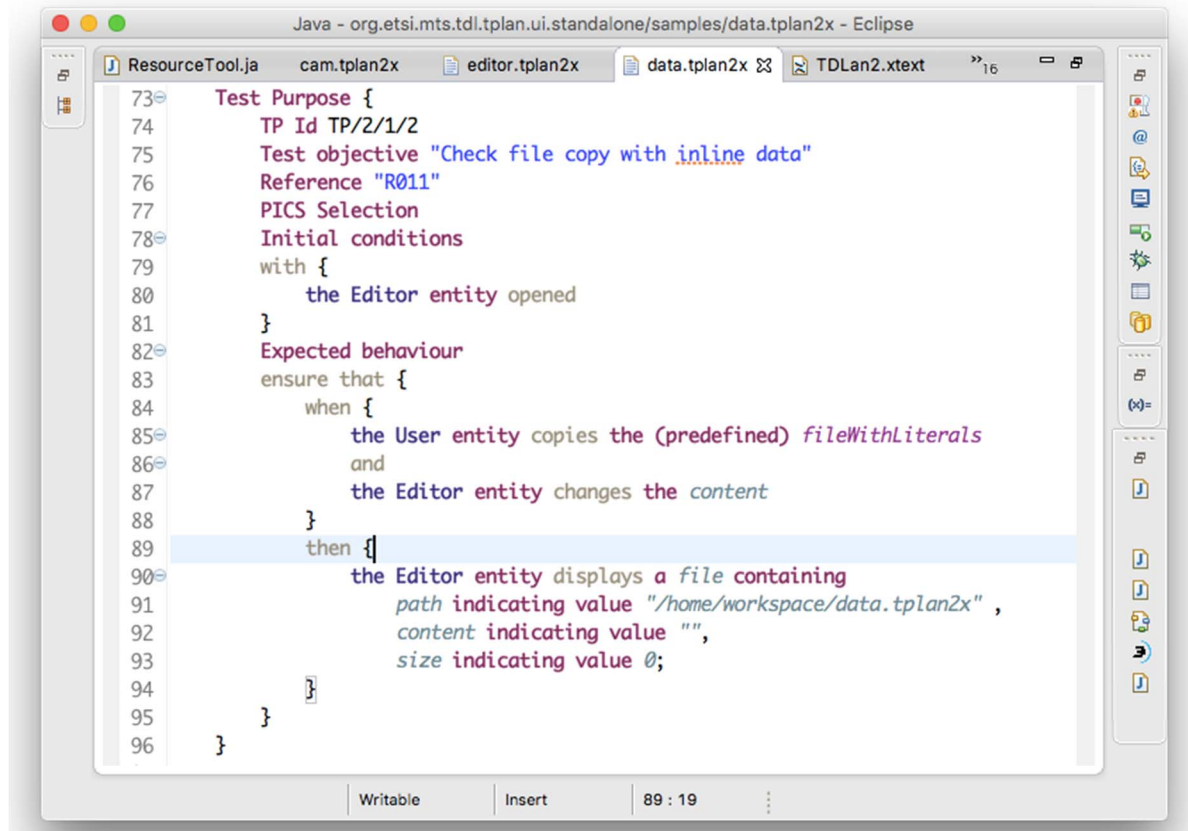


Figure 5.3.1-4: Example of customized textual editor for structured test objectives

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.

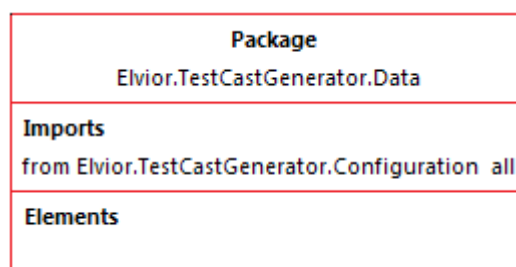


Figure 5.3.2-1: Example of 'MultipartContainerCompartmentEditPart'

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.

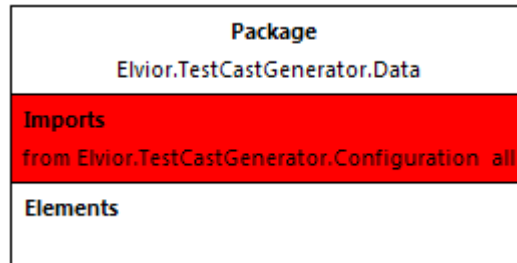


Figure 5.3.2-2: Example of 'NodeListWithHeaderEditPart'

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.

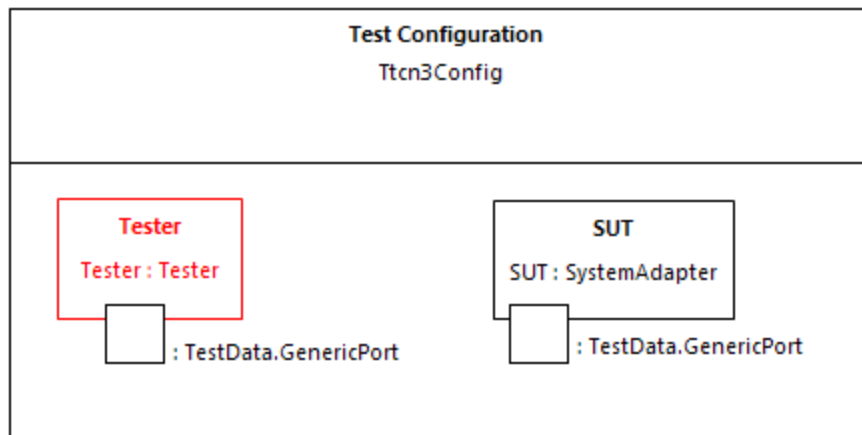


Figure 5.3.2-3: Example of 'NodeListWithHeaderEditPart'

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'.

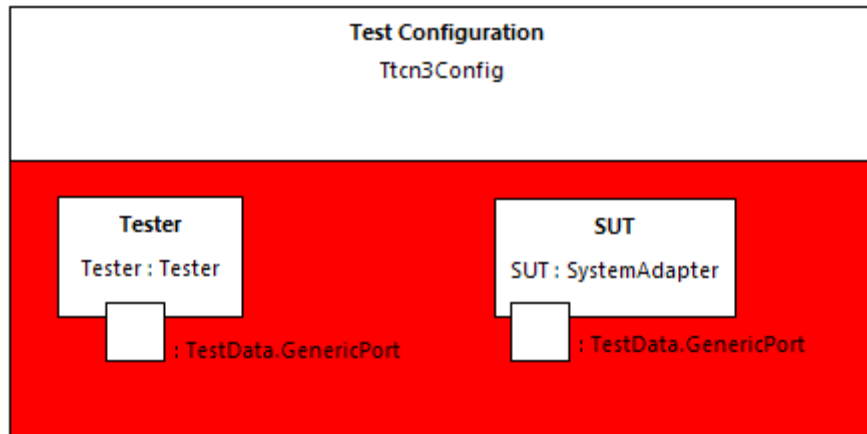


Figure 5.3.2-5: Example of 'NodeContainerEditPart'

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.

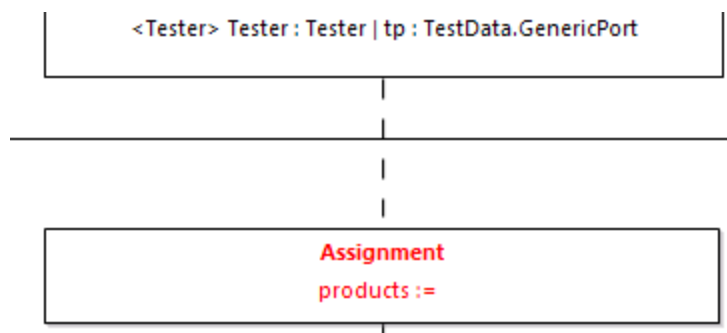


Figure 5.3.2-6: Example of 'InteractionUseConfiguringEditPart'

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.

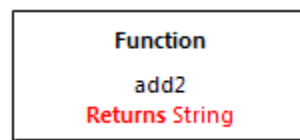


Figure 5.3.2-7: Example of 'MultiPartLabelEditPart'

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.

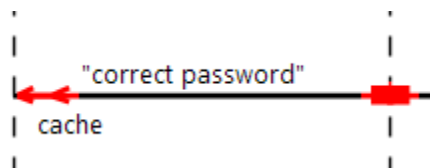


Figure 5.3.2-9: Example of 'InteractionDecoratorProvider'

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

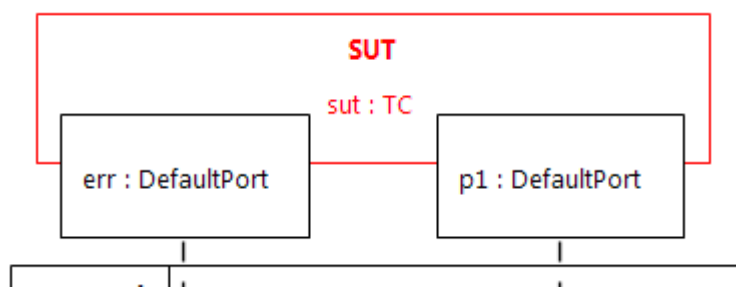


Figure 5.3.2-11: Example of custom figure placement: under-lapping container

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	MultipartContainerCompartmentEditPart
	Container: package.name	
	Node: name	NodeListWithHeaderEditPart
	Container: package.imports	
	Node: Import	
	Container: package.packagedElements Node: packagedElement	
Action	Container: action	MultipartContainerCompartmentEditPart
	Container: action.name	
	Node: name	NodeListWithHeaderEditPart
	Container: action.parameter	
	Node: Parameter	NodeListWithHeaderEditPart
	Container: action.body Node: Body	
ComponentType	Container: componentType	MultipartContainerCompartmentEditPart
	Bordered: gateInstance	
	Container: componentType.name	NodeListWithHeaderEditPart
	Node: name	
	Container: componentType.timers	
	Node: componentType.timer Container: componentType.variables Node: componentType.variable	
TestConfiguration	Container: testConfiguration	MultipartContainerCompartmentEditPart
	Container: testConfiguration.name	
	Node: name	NodeContainerEditPart TopLevelNodeListWithHeaderEditPart
	Container: testConfiguration.configuration	
	Container: testConfiguration.componentInstance	
	Bordered: testConfiguration.gateReference Node: testConfiguration.componentInstance.name	
TestObjective	Container: testObjective	MultipartContainerCompartmentEditPart
	Container: testObjective.name	
	Node: name	NodeListWithHeaderEditPart
	Container: testObjective.description	
	Node: Description	
	Container: testObjective.objectiveURI Node: URI	
StructuredDataType	Container: structuredDataType	MultipartContainerCompartmentEditPart
	Container: structuredDataType.name	
	Node: name	NodeListWithHeaderEditPart
	Container: structuredDataType.member Node: member	

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 shade 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 B of ETSI ES 203 119-4 [i.16]. 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	
TP Id	<TESTOBJECTIVENAMELABEL_PLACEHOLDER>
Test Objective	<DESCRIPTIONLABEL_PLACEHOLDER>
Reference	<URIOBJECTIVELABEL_PLACEHOLDER>
PICS Selection	<PICSSELECTIONLABEL_PLACEHOLDER>
Initial Conditions	
<INITIALCONDITIONSLABEL_PLACEHOLDER>	
Expected Behaviour	
<EXPECTEDBEHAVIOURLABEL_PLACEHOLDER>	
Final Conditions	
<FINALCONDITIONSLABEL_PLACEHOLDER>	

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

TP Id	TP_7_1_3_1_1
Test Objective	
Reference	ETSI TS 136 321 [i.25], clause 5.3.1
PICS Selection	
Initial Conditions	
with { the UE in the "E-UTRA RRC_CONNECTED state" }	
Expected Behaviour	
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" } }	
Final Conditions	

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.ocIsUndefined() 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

TDL graphical editor is built on top of and developed using the Eclipse platform. The Eclipse version in use at the time of writing the present document is "2019-12".

Steps to set up the development environment:

- 1) Download and deploy the Eclipse Modeling Tools package.

- 2) Install additional components using the Eclipse Marketplace:
 - a) Sirius; and
 - b) Xtext.
- 3) Import the following plugin projects:
 - a) org.etsi.mts.tdl.graphical.viewpoint;
 - b) org.etsi.mts.tdl.model;
 - c) org.etsi.mts.tdl.graphical.labels.data; and
 - d) org.etsi.mts.tdl.graphical.labels.data.ui.
- 4) Generate resources if necessary:
 - a) Run the GenerateData.mwe workflows in the org.etsi.mts.tdl.graphical.labels.data project if necessary.

Additional steps for setting up the development environment for the TDL textual editors, constraint implementation, and export of structured test objectives into Word documents include:

- 5) Install additional components using Eclipse Marketplace:
 - a) Epsilon for validation facilities.
- 6) Import additional plugin projects:
 - a) org.etsi.mts.tdl.TDLan2 - for editing TDL according to annex B of ETSI ES 203 119-1 [i.13];
 - b) org.etsi.mts.tdl.TDLan2.ui;
 - c) org.etsi.mts.tdl.TPLan2 - for editing structured test objectives according to annex B of ETSI ES 203 119-4 [i.16];
 - d) org.etsi.mts.tdl.TPLan2.ui;
 - e) org.etsi.mts.tdl.TDLtx - for editing TDL according to ETSI ES 203 119-8 [i.20] with the default brace-based syntax;
 - f) org.etsi.mts.tdl.TDLtx.ui;
 - g) org.etsi.mts.tdl.TDLtxi - for editing TDL according to ETSI ES 203 119-8 [i.20] with the extended indentation-based syntax;
 - h) org.etsi.mts.tdl.TDLtxi.ui;
 - i) org.etsi.mts.tdl.tools.rt - for common tools for translating model instances in different representations;
 - j) org.etsi.mts.tdl.tools.rt.ui;
 - k) org.etsi.mts.tdl.constraints - for OCL constraint implementation;
 - l) org.etsi.mts.tdl.constraints.ui;
 - m) org.etsi.mts.tdl.tools.to.docx.poi - for exporting structured test objectives to Word documents (legacy implementation can be found in org.etsi.mts.tdl.tools.to.docx);
 - n) org.etsi.mts.tdl.tools.to.docx.poi.ui.
- 7) Generate resources if necessary:
 - a) Run the GenerateTDLan2.mwe and/or GenerateTPLan2.mwe, GenerateTDLtx.mwe, GenerateTDLtxi.mwe workflows in the corresponding projects if necessary.

5.4.2 End-user Instructions

Getting Started

The implementation is available from the TOP website. Up-to-date installation and usage instructions are available on the website. The fundamental steps can be summarized as follows:

- 1) Download and deploy the Eclipse Modeling Tools package.
- 2) Install additional components using the Eclipse Marketplace:
 - a) Sirius; and
 - b) Xtext.
- 3) Install the TOP - TDL Open Source Project using the Eclipse Marketplace and select the desired components.
- 4) Alternatively, install the additional TDL-components from the TOP update site directly (currently <https://tdl.etsi.org/eclipse/latest/>):
 - a) Open the menu item 'Help' and select the item 'Install new software'.
 - b) Click the 'Add...' button to add a new repository.
 - c) Insert the required information: Name: TOP Plugins, Location: <https://tdl.etsi.org/eclipse/latest/>.
 - d) Click Ok. In the window, a new set of plugins called TDL should appear.
 - e) Click on the checkbox to select the desired plugins (or simply all), then click 'Next'.
 - f) Now follow the instructions to complete the installation.
 - g) Restart Eclipse when prompted at the end.

Once the TOP is installed, the following steps should be taken before new models can be created with the graphical editor:

- 1) Make sure an explorer view is open (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], '.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 B of ETSI ES 203 119-4 [i.16]. 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 TOP

6.1 Usage Scenarios

TDL and TOP can be used in different ways. Depending on the specific goals, different parts of TDL and TOP may be relevant for a given usage scenario. For different starting points and end goals, the following common use cases may come into question:

- Defining structured test objectives (or test purposes) with the help of TDL-TO.
- Transforming existing structured test objectives in TDL-TO into TDL test descriptions.
- Defining test descriptions with the help of TDL.
- Transforming existing test descriptions in TDL into TTCN-3 test cases.
- Transforming existing test descriptions in TDL into a target execution language (see clause 9).

- Using existing interface specifications in OpenAPI™ with TDL (see clause 8.2).
- Using existing protocol specifications in ASN.1 with TDL (see clause 8.3).

6.2 Defining Structured Test Objectives

6.2.0 Overview

TDL Structured Test Objective (TDL-TO) may be used in several ways in the test developments process. The process illustrated in this clause is based on the test development process defined in ETSI EG 203 130 (V1.1.1) [i.21]. The TDL-TO specifies a refinement of a 'TestObjective' and defines a formal description of a test objective, that may be the basis for a transformation to a TDL test description.

Developing a test specification from a base standard the first step after identifying the requirements to be tested, is to define the test objectives. The entities and events to check the test objectives may then be specified and finally arguments of events (data values) and timing constraints may be specified. The context in which the required behaviour executes is defined in the test configuration.

Then the parts of a complete TDL-TO specification are:

- Domain part.
- Data.
- Configuration.
- Test purpose behaviour.

The domain, data, and configuration parts are common to a set of test purpose behaviour descriptions, while each test purpose behaviour is specific to a single test objective. Test purpose behaviours are typically grouped based on different criteria, e.g. test for normal behaviour and test for invalid behaviour to form a test suite structure. In TDL-TO this structuring is supported by grouping of test purpose behaviours. To further structure a TDL-TO specification, the domain, data, configuration and test purpose behaviours may be also separated using the TDL package concept, to support re-use of basic data definitions and configurations.

6.2.1 Domain part of TDL-TO

The domain part of a TDL-TO specification defines the PICS elements, entities, and events relevant for a set of TDL-TOs.

```

Domain {
    pics:
        - NONE
    ;
    entities:
        - EPC_PCRF_A
        - EPC_PCRF_B
        - EPC_PGW_A
        - EPC_PGW_B
        - EPC_MME_A
        - EPC_MME_B
        - IMS_HSS_A
        - IMS_HSS_B
    ;
    events:
        - receives
        - sends
        - triggers
        - detachment
        - invokes
        - create_session_request
        - delete_session_request
        - termination_SIP_signalling_session
    ;
}

```

Figure 6.2.1-1: TDL-TO Domain example

In Figure 6.2.1-1 an example of a domain specification is shown. The example illustrates the definition of a single PICS value. The example also contains the definition of a list of entities that can be referenced in test configuration definitions and in event occurrences in the behaviour part. Finally, the example shows definition of events that may be referenced in the event occurrence parts of TDL-TO behaviour descriptions.

6.2.2 Data definitions

In TDL-TO data may be used in the behaviour part without explicit declaration. However, in the data part of the TDLTO definition structured data types and structured data values may be specified.

```

Data {
    type DiameterMessage;
}

```

Figure 6.2.2-1: TDL-TO data definition example

Figure 6.2.2-1 illustrates the specification of a single data type.

6.2.3 Configuration

The configurations part of the TDL-TO specification defines by reference the context in which a TDL-TO is to be executed. The Configuration part may contain any number of test configuration as needed for the TDL-TOs to which it may be associated.


```

Configuration {
  Interface Type defaultGT accepts DiameterMessage;
  Component Type DiameterComp with gate g of type defaultGT;

  Test Configuration CF_VxLTE_INT
    containing
      Tester component EPC_PGW_A of type DiameterComp
      Tester component EPC_PCRF_A of type DiameterComp
      SUT component IMS_A of type DiameterComp
      connection between EPC_PGW_A.g and EPC_PCRF_A.g
    ;

  Test Configuration CF_VxLTE_RMI
    containing
      Tester component EPC_PCRF_A of type DiameterComp
      Tester component EPC_PCRF_B of type DiameterComp
      SUT component IMS_A of type DiameterComp
      connection between EPC_PCRF_A.g and EPC_PCRF_A.g
    ;
}

```

Figure 6.2.3-1: TDL-TO configuration example

The configuration part in Figure 6.2.3-1 shows the definition of two test configurations "CF_VxLTE_INT" and "CF_VxLTE_RMI". Both test configurations are based on the same component type "DiameterComp" and gate type "defaultGT". The 'defaultGT' is specified to accept instances of the datatype 'DiameterMessage'. The test configurations also specifies the role of involved entities, as tester or SUT component.

6.2.4 Test purpose behaviour

The test behaviour defines the behaviour of a TDL-TO to check a single test objective in terms of a sequence of event occurrences in a referenced test configuration and with data values and timing constraints.

```

Package TP_RX {
  import all from Sip_Common;
  import all from Diameter_Common;

  Test Purpose {
    TP Id TP_RX_PCSCF_STR_05
    //TP_EPC_7002_21 from ETSI TS 103 029 V5.1.1
    Test objective "Verify that IUT after reception of 486 response sends an ST-Request at originating
leg."

    Reference
      "ETSI TS 129 214 (V15.6.0) [i.27], clauses 4.4.4"

    Config Id CF_VxLTE_INT

    PICS Selection NONE

    Initial conditions with {
      the UE_A entity isAttachedTo the EPC_A and
      the UE_A entity isRegisteredTo the IMS_A
    }

    Expected behaviour
      ensure that {
        when {
          the IMS_P_CSCF_A entity receives a 486_Response_INVITE
          from the IMS_S_CSCF_A entity
        }
        then {
          the IMS_P_CSCF_A entity sends the STR containing
          Session_Id_AVP;
          to the EPC_PCRF_A entity
        }
      }
  }
}

```

Figure 6.2.4-1: TDL-TO behaviour example

A test purpose behaviour example is shown in Figure 6.2.4-1. The test purpose behaviour references the other parts of a TDL-TO, that is the domain, data and configuration part, that in this example are all imported from the two packages 'SIP_Common' and 'Diameter_Common'.

A test purpose is assigned a unique id often reflecting its association within a test suite structure. In this example indicating in the TP name the interface 'RX', the component 'PCSCF', and the message 'STR' relevant for this TP.

The TDL-TO test purpose allows a reference to the base standard from where the requirement and test objective is derived. The test objective may can be defined as an informal text string in the field "Test objective". The condition for the applying the test purpose in a test execution can be specified in "PICS selection" field. The PICS selection expression may consist of a list of PICS references combined by logical operators.

The test configuration 'CF_VxLTE_INT' referenced in the example specifies the test configuration used in the test behaviour specification part.

The event occurrence sequences of the TDL-TO constitutes the core of the test behaviour part. It is split in three optional parts, the initial condition, the expected behaviour, and final conditions. The example illustrates an initial part where the 'UE_A' and 'IMS_A' entities are brought into the state required to check the expected behaviour. The events 'isAttachedTo' and 'isRegisteredTo' may be abstract operations which may often be used in the initial phase of the TDL-TO specification, to allow for further refinements in later phases. In the Expected behaviour part the conditional event occurrence sequences are specified that is assigned the verdict of the test purpose, explicitly or implicitly as is the case in this example. The event occurrences in the example illustrates the use of undeclared data instances '486_Response_INVITE' and 'STR', where the latter is further specified to contain the data value 'Session_Id_AVF'. In case the test purpose needs to perform operations after the test objective is achieved, such behaviour may be specified in the final conditions part.

6.3 Transforming Test Objectives into Test Descriptions

6.3.1 Overview

Structured test objectives can be used as a starting point for test descriptions or even for executable test cases. As the abstraction gap between structured tests objectives and executable test cases is often too large, it is recommended to refine the structured test objectives into test descriptions in a stepwise manner, where at each step there is a smaller abstraction gap in comparison to the preceding step.

While structured test objectives provide many building blocks for a test description, structured test objectives can abstract away many of the important details that are essential for the specification of a test description. Some of the details can be inferred easily, while others allow for different interpretations. In order to narrow down the spectrum of possible interpretations, it is recommended that guidelines and conventions are defined in advance and enforced during the specification of the structured test objectives. This can streamline the refinement process and pave the way for standardized refinement of structured test objectives into test descriptions.

The examples are provided in the textual representation for brevity and convenience. The graphical representation can be used instead as well.

6.3.2 Data

TDL-TO provides different means for the use of data within a 'StructuredTestObjective'. In addition to the use of defined 'DataInstances' and 'SpecialValueUse's, TDL-TO provides means for the specification of literal 'Value's inline, within a 'StructuredTestObjective'.

The use of defined 'DataInstance's and 'SpecialValueUse's does not require particular handling as the same 'DataInstance's can be used in the resulting 'TestDescription's. While the concrete syntax may be different, the underlying model elements are the same. The corresponding 'DataType's may need to be taken into account if a 'TestConfiguration' needs to be inferred from the 'StructuredTestObjective'. Additionally, any qualifying 'Comment's used to describe further details related to the context of its usage may need to be interpreted according to the existing conventions (if defined). The example in Figure 6.3.2-1 illustrates the definition of 'DataType's and 'DataInstance's in TDL-TO. The corresponding data definitions in the textual representation of TDL are shown in Figure 6.3.2-2. Apart from minor syntactical differences, the underlying model structures are the same. In fact, the TOP tools enable cross referencing between both notations so that data definitions in TDL can be reused in TDL-TO and vice-versa.

```

Package data {
  Data {
    type float;
    type position with x of type float, y of type float;
    float -22;
    float -21;
    position startingPosition containing x indicating value -21;
  }
}

```

Figure 6.3.2-1: Predefined data example in TDL-TO

```

Package data {
  Type float ;
  Type position ( x of type float , y of type float ) ;
  float -22 ;
  float -21 ;
  position startingPosition ( x = -21 ) ;
}

```

Figure 6.3.2-2: Corresponding data in TDL

TDL-TO permits the use of 'LiteralValue's as a flexible way for specifying the arguments of 'EventOccurrenceSpecification's. This can be useful, especially at an early stage, where the data structures and contents are not fixed yet. 'LiteralValue's may contain descriptions of the structure and contents of the 'LiteralValue'. Additionally, 'LiteralValue's may be referenced within the same 'StructuredTestObjective'. Any qualifying 'Comment's used to describe further details related to the context of its usage may need to be interpreted according to the existing conventions (if defined).

In order to transform 'LiteralValue's, first corresponding 'DataType's and 'DataInstance's need to be inferred. Consider the following example illustrated in Figure 6.3.2-3, showing a 'LiteralValue' specification within an 'EventOccurrenceSpecification'. The basic structure is the same, but there are no predefined 'DataType's and 'DataInstance's. The inferred 'DataType's and 'DataInstance's are illustrated in Figure 6.3.2-4. The inferred 'DataElement's are prefixed with 'inferred_' for illustrative purposes. The contextual information may provide hints for more appropriate naming. Apart from the inference, type compatibility and merging needs to be considered. In this example, it is assumed that 'x' and 'y' are of the same type, otherwise distinct 'DataType's need to be inferred as well. If a 'StructuredDataInstance' is used only once, it is also possible to specify it as inline 'DataInstanceUse' in TDL.

```

when {
  the Controller entity sends the start position containing
    x indicating value 22,
    y indicating value 21
  ;
} then {
  the Object entity moves_to the received start position
}

```

Figure 6.3.2-3: Literal data in TDL-TO

```

Package inferred_data {
  Type inferred_simple ;
  Type inferred_position ( x of type inferred_simple, y of type inferred_simple ) ;
  inferred_simple 22 ;
  inferred_simple 21 ;
  inferred_position inferred_start_position ( x = 22, y = 21 ) ;
}

```

Figure 6.3.2-4: Inferred literal data in TDL

In case existing 'DataInstance's are used and corresponding 'DataMapping's exist, these can be reused as well. Otherwise, the 'DataMapping's can be defined as part of the refinement process for both existing data specifications and data specifications inferred from inline literal data specifications.

6.3.3 Configurations

Similar to the use of data, TDL-TO provides different means for the specification of the entities related to an 'EventOccurrenceSpecification'. Abstract entities can be useful, especially at an early stage, where the 'TestConfiguration's are not fixed yet. If 'TestConfiguration's are already specified, the corresponding 'ComponentInstances' can be used in 'EventOccurrenceSpecification's. An example for a simple 'TestConfiguration' and corresponding 'ComponentType's and 'GateType's specified in TDL-TO is shown in Figure 6.3.3-1. The corresponding definitions in the textual representation of TDL are showing in Figure 6.3.3-2. The use of defined 'ComponentInstance's does not require particular handling as the same 'TestConfiguration's can be used in the resulting 'TestDescription's. Apart from minor syntactical differences, the underlying model structures are the same. The TOP tools enable cross referencing between both notations so that definitions related to 'TestConfiguration's in TDL can be reused in TDL-TO.

```
Package base_configuration {
  import all from data;
  Configuration {
    Interface Type wireless accepts position;
    Component Type unit with gate wifi of type wireless;
    Test Configuration basic containing
      Tester component controller of type unit
      SUT component object of type unit
      connection between controller.wifi and object.wifi
  };
}
```

Figure 6.3.3-1: Predefined configuration example in TDL-TO

```
Package base_configuration {
  import all from data ;
  Gate Type wireless accepts position ;
  Component Type unit having {
    gate wifi of type wireless ;
  }
  Test Configuration basic {
    create Tester controller of type unit ;
    create SUT object of type unit ;
    connect controller.wifi to object.wifi ;
  }
}
```

Figure 6.3.3-2: Corresponding configuration in TDL

The use of abstract entities provides more flexibility early on, however, it requires clear guidelines and conventions for the interpretation of the abstract entities. An 'Entity' may be transformed into a 'ComponentInstance' or a 'GateInstance' depending on the intended interpretation. Hints and conventions regarding the desired interpretation of an 'Entity' can be provided in the 'Entity' definition, in the context of its use, or outside the TDL-TO specification. Considering the example illustrated in Figure 6.3.2-3, it can be inferred that the 'Controller' entity and the 'Object' entity have some means to interact without this being explicitly specified. Figure 6.3.3-3 illustrates one possible 'TestConfiguration' which can be inferred from the behaviour specification in Figure 6.3.2-3. First, one or more 'GateType's need to be inferred, then the corresponding 'ComponentType's and their 'GateInstance's. For simplicity, it is assumed that 'Controller' and 'Object' are of the same 'ComponentType', conventions may be put in place to indicate that. Alternatively, subsequent refinement may further differentiate the 'ComponentType's where appropriate. Finally, the 'TestConfiguration' is inferred, including assigning 'ComponentInstance's with corresponding roles, as well as 'Connection's between the inferred 'GateInstance's. In this example, the roles and 'Connections' are inferred based on the 'EventOccurrenceSpecification's and their context (e.g. when / then clauses, etc.). Similar to 'DataType's, compatible inferred 'TestConfiguration's, 'ComponentType's, and 'GateType's need to be identified and merged where applicable to avoid unnecessary duplication.

```

Package inferred_configuration {
  Import all from data ;
  Gate Type inferred_gate_type accepts inferred_position ;
  Component Type inferred_component_type having {
    gate inferred_gate of type inferred_gate_type ;
  }
  Test Configuration inferred_move_object {
    create Tester Controller of type inferred_component_type ;
    create SUT Object of type inferred_component_type ;
    connect Controller.inferred_gate to Object.inferred_gate ;
  }
}

```

Figure 6.3.3-3: Inferred configuration in TDL

6.3.4 Behaviour

Initial conditions, expected behaviour, and final conditions in TDL-TO are expressed by means of 'EventSequences'. 'EventSequences' are comprised of 'EventOccurrenceSpecification's. This provides simple generic high-level construct with loose semantics indicated by the referenced 'Event'. The interpretation of the 'Event' can be indicated in the domain description and/or refined in the 'EventOccurrenceSpecification'. It is recommended to establish well-defined specification conventions in order to ensure consistent interpretation. TDL 'TestDescriptions' require more differentiated specification of behaviour, distinguishing between 'Interaction's, 'Action's, and other kinds of 'Behaviour's. While some assumptions regarding the mapping of 'Event's to 'AtomicBehaviour's can be made intuitively, it is recommended to define explicit conventions in order to ensure consistent interpretation and transformation. The example in Figure 6.3.4-1 illustrates a minimal 'StructuredTestObjective' containing only the specification of the expected behaviour. Assuming the data and configuration related information has been inferred as illustrated in the previous examples, the corresponding 'TestDescription' inferred from the 'StructuredTestObjective' is shown in Figure 6.3.4-2. In this scenario, the first 'EventOccurrenceSpecification' in the 'whenClause' is interpreted as an 'Interaction' between the 'Controller' and the 'Object', the latter is assumed to be the implicit opposite entity in the 'EventOccurrenceSpecification'. It is recommended to make opposite entities explicit whenever possible. The second 'EventOccurrenceSpecification' in the 'thenClause' is interpreted as an 'ActionReference'. In this case, it is also necessary to infer a definition for the action.

```

Test Purpose {
  TP Id TP_MOVE_OBJECT_LITERAL
  Test objective "Move object to destination with literal values."
  Expected behaviour
  ensure that {
    when {
      the Controller entity sends the start position containing
      x indicating value 22,
      y indicating value 21
    ;
    } then {
      the Object entity moves_to the received start position
    }
  }
}

```

Figure 6.3.4-1: Expected behaviour specification in TDL-TO

```

Action move_to (position of type inferred_position);
Test Description TD_MOVE_OBJECT_LITERAL uses configuration inferred_move_object {
  Controller.inferred_gate sends inferred_start_position to Object.inferred_gate;
  perform action move_to (position = inferred_start_position) on Object;
}

```

Figure 6.3.4-2: Corresponding behaviour specification in TDL

If desired, especially when 'StructuredDataInstance's are used only once, it is also possible to specify them as inline 'DataInstanceUse's in TDL. The resulting 'TestDescription' for the example in Figure 6.3.4-1 is shown in Figure 6.3.4-3, where instead of the 'inferred_start_position' 'DataInstance', the corresponding data is specified inline. Since the 'DataInstance' is used twice in this case, it results in some duplication.

```

Action move_to (position of type inferred_position);
Test Description TD_MOVE_OBJECT_LITERAL_INLINE uses configuration inferred_move_object {
  Controller.inferred_gate sends new inferred_position (x = 22, y = 21) to Object.inferred_gate;
  perform action move_to (position = new inferred_position (x = 22, y = 21)) on Object;
}

```

Figure 6.3.4-3: Corresponding behaviour specification in TDL using inline data

The steps for the derivation of 'TestDescription's from 'StructuredTestObjective's can be translated to other notations as well. Fundamentally, the process remains the same, starting with the data definitions, through the test configurations, and finally the behaviour specifications. The above guidelines can be used as a template for deriving 'TestDescription's from other kinds of documents and artifacts as a starting point.

6.3.5 Transformation Conventions and Assumptions

The transformation of 'LiteralValue's involves the following conventions:

- If the 'Content' of the 'LiteralValue' is empty, the 'LiteralValue' is mapped to a 'SimpleDataType' with a 'name' corresponding to the 'name' of the 'LiteralValue'. If qualifier 'Comment's are present, a 'SimpleDataInstance' is created with a 'name' corresponding to the concatenated 'body's of the qualifier 'Comment's. Alternatively, the 'name' of the 'DataType' can also be prefixed with the concatenated 'body's of the qualifier 'Comment's. If the corresponding 'DataType' or 'DataInstance' exists, no action is taken.
- If the 'Content' of the 'LiteralValue' is not empty, the 'LiteralValue' is mapped to a 'StructuredDataType' with a 'name' corresponding to the 'name' of the 'LiteralValue'. If qualifier 'Comment's are present, a 'StructuredDataInstance' is created with a 'name' corresponding to the concatenated 'body's of the qualifier 'Comment's. Alternatively, the 'name' of the 'DataType' can also be prefixed with the concatenated 'body's of the qualifier 'Comment's. If the corresponding 'DataType' or 'DataInstance' exists, no action is taken.
- Each 'Content' element of the 'LiteralValue' is mapped to a 'Member' within the corresponding 'StructuredDataType' with a 'name' corresponding to the 'name' of the 'Content'. If a 'Member' with the same 'name' exists, no action is taken. The 'DataType' of the 'Member' corresponds to:
 - A new 'DataType' corresponding to the 'Content' with a 'name' prefixed with the 'name' of the containing 'StructuredDataType' in case a 'Content' is directly contained within the 'Content'.
 - A (default) 'SimpleDataType' corresponding to the 'LiteralValue' in case a 'LiteralValue' is directly contained within the 'Content'.
 - The 'DataType' corresponding to the 'dataType' of the 'DataInstanceUse' in case a 'DataReference' is directly contained within the 'Content'.
- Each nested 'Content' or 'LiteralValue' element is transformed according to the conventions above.

For example, as shown in Figure 6.3.5-1 and Figure 6.3.5-3, for the 'EventOccurrenceSpecification' taken from [i.37], a corresponding 'StructuredDataType' is created for both the 'LiteralValue' 'message' and for the 'payload' 'Content' of the 'message'. The 'dataType' for the 'payload' 'Member' is then set accordingly. Instead, since the 'value's for the 'filterLength' and 'topic_filter' Content's correspond to 'DataReference's to defined 'DataInstance's, as shown in Figure 6.3.5-2, the 'dataType's for the corresponding 'Member's in the derived 'SUBSCRIBE_message_payload' 'DataType' are: set to the 'dataType's of the referenced 'DataInstance's.

```

the IUT entity receives a SUBSCRIBE message containing
  payload containing
    filterLength corresponding to TOPIC_FILTER_LEN_SEC_CVE_01,
    topic_filter corresponding to TOPIC_FILTER_LEN_SEC_CVE_01;;
from the ATTACKER_CLIENT entity

```

Figure 6.3.5-1: 'LiteralValue's and 'DataReference's example (from [i.37])

```

Data {
  UTF8String TOPIC_FILTER_SEC_CVE_01; // topic filter used in TP_MQTT_BROKER_SEC_CVE_001
  Int16 TOPIC_FILTER_LEN_SEC_CVE_01; // corresponds to lengthof(TOPIC_FILTER_SEC_CVE_01) + 1
}

```

Figure 6.3.5-2: Corresponding 'DataInstance' definitions (from [i.37])

```

Type SUBSCRIBE_message (
  payload of type SUBSCRIBE_message_payload
);
Type SUBSCRIBE_message_payload (
  filterLength of type Int16,
  topic_filter of type UTF8String
);

```

Figure 6.3.5-3: Resulting 'DataType' specifications in TDL

For the transformation of 'EventOccurrence's into 'Behaviour's, it is necessary to first derive the corresponding 'TestConfiguration' if no 'TestConfiguration' is specified for the 'StructuredTestObjective'. The derivation of the 'TestConfiguration' involves the following conventions:

- Each 'EntityReference' is transformed a 'ComponentType' and a 'GateType', where the 'GateType' accepts the 'DataType's resulting from the transformation of the 'LiteralValue's specified as arguments of the 'EventOccurrence'. The 'ComponentType' contains a 'GateInstance' of the transformed 'GateType'. If the 'ComponentType', 'GateType', or 'GateInstance' already exists, no action is taken.
- A 'TestConfiguration' is constructed with 'ComponentInstance's of the 'ComponentType's resulting from the transformation above. It is recommended to use naming conventions, annotations, or other conventions to indicate the roles of the 'ComponentInstance's, otherwise the roles are set to a default value of 'Tester' or 'SUT' and need to be adjusted afterwards. While opposite 'EntityReference's are optional, it is recommended that they are explicitly specified, otherwise an implicit 'Tester' 'ComponentInstance' needs to be assumed and constructed. If the assumption of an implicit 'Tester' does not hold, the transformed 'TestConfiguration' may need to be adjusted.
- Within the 'TestConfiguration', 'Connection's need to be created between interacting 'ComponentInstance's for every 'EventOccurrence'. If a 'Connection' already exists, no action is taken.
- Conventions or comparison between 'TestConfiguration's are recommended to avoid duplicate identical configurations.

Once a suitable 'TestConfiguration' is derived, 'EventOccurrence's can be transformed into 'Behaviour's. There can be different strategies, ranging from transforming 'EventOccurrence's into detailed 'Interaction's, to transforming 'EventOccurrence's into abstract 'TestDescription' skeletons as containers for manually specified 'Behaviour's. This may depend on the general test specification process and/or the level of detail of the structured test objective specifications. Assuming detailed 'Message's as the target, the following conventions can be applied:

- A 'TestDescription' using the 'TestConfiguration' resulting from the transformation above is constructed as a container for the 'Behaviour's. For traceability and in general, it is recommended that the 'TestDescription' references the 'StructuredTestObjective' as its 'testObjective'.
- In the simplest case, the 'TestDescription' contains one 'CompoundBehaviour' which contains the individual 'Behaviour's corresponding to the 'EventOccurrence's.
- If desired, the 'CompoundBehaviour' may contain further 'CompoundBehaviour's corresponding to the structural blocks of the 'StructuredTestObjective', e.g. "Initial conditions", "Expected behaviour", "when/then"-clauses, etc., so that the resulting 'TestDescription' more closely reflects the structure of the corresponding 'StructuredTestObjective'. The additional 'CompoundBehaviour's may be annotated to clearly specify their purpose and potentially also influence how tools represent or treat the individual 'Behaviour's within the additional 'CompoundBehaviour's.
- The individual 'EventOccurrence's are transformed into corresponding 'Behaviour's. In the absence of further information, it needs to be specified what the default 'Behaviour' should be. In general, suitable conventions and 'Annotation's are recommended to ensure easier transformation. Assuming that the target 'Behaviour' is a 'Message', a 'Message' is constructed with the source and target 'GateReference's corresponding to the 'EntityReference's of the 'EventOccurrence'. The assumptions for the transformation of 'TestConfiguration's apply in this case as well, particularly when no opposite 'EntityReference' is specified. A 'DataUse' is then constructed as the argument of the 'Message' based on the 'LiteralValue' specified as the argument of the 'EventOccurrence' and the 'DataType's resulting from the transformation above.

6.4 Defining Test Descriptions

6.4.1 Overview

In the absence of structured test objectives or other documents which can serve as a starting point, test descriptions can be defined from scratch. The fundamental steps in the process involve the definition of data first, then configurations, finally the behaviour. The following examples illustrate the different steps by means of the graphical syntax for TDL with the help of the graphical editor.

6.4.2 Data and Configuration

Since the 'Generic TDL' diagram accommodates both the specification of data- and configuration-related elements, both are contemplated together. If necessary, separate diagrams can be created instead to capture only data-related or configuration-related elements separately. In the example shown in Figure 6.4.2-1 the one diagram approach is shown for conciseness and also to show a complete overview of all relevant elements in one place. On the top-left side the predefined simple data types are shown. In the bottom part the verdict-related types and instances are shown. On the right side behaviour-related definitions for an 'Action' and a 'TestDescription' are shown. Finally, in the middle part, the data types, data instances, as well as component and gate types and the test configuration are shown. The graphical editor does provide some more flexibility with regard to the order of creation of the different elements. However, the fundamental order remains the same - data, configuration, behaviour.

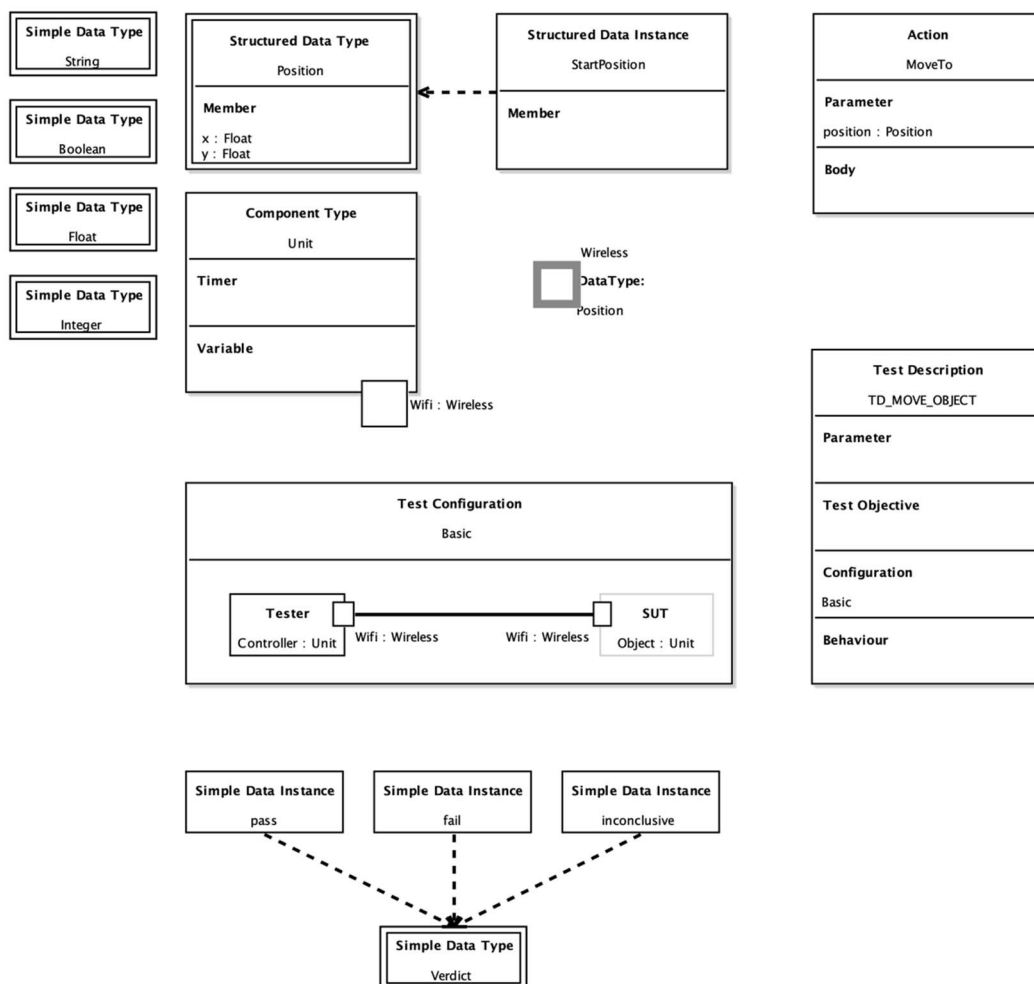


Figure 6.4.2-1: Data and configuration specification in TDL using the graphical editor

6.4.3 Test Behaviour and Time

The 'TDL Behaviour' diagram allows the visualization and specification of the behaviour of an individual 'TestDescription'. While the 'TestDescription' itself is defined within a 'Generic TDL' diagram, including its name, parameters, test configuration, and test objectives, the specifics of the behaviour are shown in a separate 'TDL Behaviour' diagram. The example shown in Figure 6.4.3-1 illustrates the behaviour of the 'TD_MOVE_OBJECT' 'TestDescription'. In addition to the basic behaviour, temporal properties of the behaviour are illustrated with the help of a 'TimeLabel' and a 'TimeConstraint'.

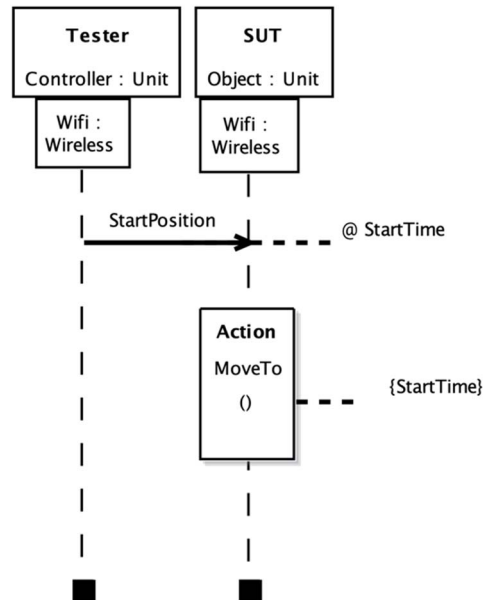


Figure 6.4.3-1: Test behaviour in TDL using the graphical editor

'TestDescriptionReference's enable the reuse of behaviour definitions. While in some other high-level test specification languages the use of so-called "data tables" has been gaining some popularity, TDL provides more sophisticated facilities both for the definition of data and for the reuse of behaviour. A parameterized 'TestDescription' can be invoked multiple times with different data instances as shown in the example in Figure 6.4.3-2. In the 'TC_MOVE_AROUND' 'TestDescription', the 'TC_MOVE_TO' 'TestDescription' is invoked four times to describe a test sequence where the 'Object' needs to move to four positions.

```
Test Description TC_MOVE_TO (target_position of type inferred_position)
  uses configuration inferred_move_object {
    Controller.inferred_gate sends parameter target_position to Object.inferred_gate;
    perform action move_to (position = parameter target_position) on Object;
  }
```

```
Test Description TC_MOVE_AROUND
  uses configuration inferred_move_object {
    execute TC_MOVE_TO (target_position = start_position);
    execute TC_MOVE_TO (target_position = open_position);
    execute TC_MOVE_TO (target_position = closed_position);
    execute TC_MOVE_TO (target_position = end_position);
  }
```

Figure 6.4.3-2: Test behaviour reuse in TDL using test description references

6.5 Transforming Test Descriptions into TTCN-3 Test Cases

6.5.1 Overview

One way to obtain executable test cases from TDL is to transform the test descriptions into TTCN-3 code. The standardized mapping to TTCN-3 in ETSI ES 203 119-6 [i.18] specifies in great detail all the peculiarities that need to be considered for the derivation of executable test cases in TTCN-3 from TDL. The basic steps remain fundamentally the same, involve transforming the data definitions, the configuration-related definitions, as well as the behaviour specifications. All the transformations have to take into account the semantic gaps between both languages, as well as the intrinsic differences in the levels of abstraction. The standardized mapping is defined for locally ordered test descriptions only. Thus, if totally ordered test descriptions are the starting point, these first need to be transformed into locally ordered test descriptions, keeping in mind the differences in semantics and the additional constraints that are imposed by locally ordered test descriptions. The prototypical implementation of the mapping within the TOP provides automated translation for the essential parts necessary for the transformation of TDL 'TestDescription's to TTCN-3 test cases.

6.5.2 Data

To illustrate the mapping of the data-related elements, consider the example in Figure 6.5.2-1. It illustrates different data definitions and data uses. The corresponding equivalents in TTCN-3 are shown in Figure 6.5.2-2. The mappings for data are pretty straightforward in this example. Although the use of data mappings is recommended, in which case the respective mapping targets are used instead, it is also possible to generate basic data definitions in case no data mappings are present. Annotations can be used to override assumptions.

```
//data types
Type SESSIONS (id1 of type Integer, id2 of type Integer);
Type MSG (ses of type SESSIONS, content of type String);

//data instances
SESSIONS s1{id1 = 1, id2 = 2};
SESSIONS s2{id1 = 11, id2 = 22};
MSG msg1{ses = s1, content = m1};

//value data instances
SESSIONS c_s1{id1 = 1, id2 = 2} with {VALUE;};
MSG c1{ses = s1, content = c1} with {VALUE;};

ComponentType ct having {
  //variables
  variable v1 of type MSG with {VALUE;};
  variable v2 of type MSG;
  gate g of type gt;
}
```

Figure 6.5.2-1: Test data example in TDL

```

//data types
type record SESSIONS {
    integer id1,
    integer id2
}
type record MSG {
    SESSIONS ses,
    charstring content
}

//templates
template SESSIONS s1 := {id1:=1, id2:=2}
template SESSIONS s2 := {id1:=11, id2:=22}
template MSG msg1 := {ses := s1, content := "m1"}

//value -> constant
const SESSIONS c_s1 := {id1:=1, id2:=2}
const MSG c1 := {ses := c_s1, content := "c1"}

type component ct {
    //variables
    var MSG v1;
    var template MSG v2;
    port gt g;
}

```

Figure 6.5.2-2: Test data equivalents for Figure 6.5.2.1 in TTCN-3

6.5.3 Configuration

With regard to test configurations, there are several concerns to address. TTCN-3 provides means for the dynamic instantiation and management of test configurations. The essential parts of a configuration include the main test component which plays a special role, zero or more parallel test components, as well as a unified system interface. There is a distinction between connecting and mapping ports and there are some restrictions with regard to these. In TDL the test configuration is defined upfront and remains static. TDL also provides a holistic view where the SUT can be decomposed into multiple interconnected components. The example in Figure 6.5.3-1 illustrates a minimal test configuration in TDL. The corresponding mapping in TTCN-3 is illustrated in Figure 6.5.3-2. A unified system interface needs to be inferred in case there are multiple SUT components. The steps for instantiating and mapping / connecting the components are encapsulated in a function.

```

Gate Type defaultGT accepts
    ACK, PDU, PDCCH, C_RNTI, CONFIGURATION ;

Component Type defaultCT having {
    gate g of type defaultGT;
}

Test Configuration defaultTC {
    create Tester SS of type defaultCT;
    create SUT UE of type defaultCT ;
    connect UE.g to SS.g ;
}

```

Figure 6.5.3-1: Test configuration example in TDL

```

type port defaultGT_to_map message {
  //this is a port type for SUT-Tester connections
  inout charstring, PDCCH /* ACK, PDU, C_RNTI, CONFIGURATION ; */
}

type port defaultGT_to_connect message {
  //this is a port type for Tester-Tester connections
  inout charstring, PDCCH /* ACK, PDU, C_RNTI, CONFIGURATION ; */
}

type component MTC_CT {
  //component type for MTC
  //variable for the PTC(s) --TESTER component(s) in TDL
  var defaultCT TESTER_SS;
}

type component defaultCT {
  port defaultGT_to_map g_to_map;
  port defaultGT_to_connect g_to_connect;
}

function defaultTC() runs on MTC_CT {
  // Test Configuration defaultTC, mappings, connections
  TESTER_SS := defaultCT.create;
  map (TESTER_SS:g_to_map,system:g_to_map);
}

```

Figure 6.5.3-2: Test configuration related equivalents for Figure 6.5.3.1 in TTCN-3

6.5.4 Behaviour

In terms of behaviour, TTCN-3 and TDL also have different assumptions. In TTCN-3, the focus is on the test system view, where all components execute their behaviour concurrently and independently unless there is explicit synchronization among them. TDL aims to provide a global view with the possibility to specify both locally and totally ordered behaviour, with explicit or implicit synchronization respectively. For the standardized mapping to TTCN-3 only the local ordering is taken into consideration. User-defined mappings may also tackle the totally ordered behaviour. In the example shown in Figure 6.5.4-1 a locally ordered 'TestDescription' is illustrated. The corresponding mappings in TTCN-3 are shown in Figures 6.5.4-2, 6.5.4-3, and 6.5.4-4. First, the default handling needs to be taken care of. This involves the definition of altsteps to handle deviations from the specified behaviour as well as quiescence, which is illustrated in Figure 6.5.4-2. Then the actual test behaviour from the test system's point of view is translated to a function as illustrated in Figure 6.5.4-3. Finally, in Figure 6.5.4-4 a test case is defined which takes care of activating the default behaviour, instantiating the test configuration, as well as starting the actual test behaviour.

```

Test Description Implementation TD_7_1_3_1
uses configuration defaultTC {

  SS.g sends pdcch (c_rnti=ue) to UE.g;
  SS.g sends mac_pdu to UE.g;
  UE.g sends harq_ack to SS.g with {
    test objectives : TP1 ;
  };

  set verdict to PASS ;
  SS.g sends pdcch (c_rnti=unknown) to UE.g;
  SS.g sends mac_pdu to UE.g;

  alternatively {
    UE.g sends harq_ack to SS.g ;
    set verdict to FAIL ;
  } or {
    gate SS.g is quiet for five ;
    set verdict to PASS ;
  } with {
    test objectives : TP2 ;
  }
}

```

Figure 6.5.4-1: Test behaviour example in TDL

```

altstep to_handle_deviations_from_TDL_description_AS() {
  [] any port.receive {
    setverdict(fail);
    mtc.stop;
  }
  //if nothing happens, a timer is started
  //before every receive instruction
  //and the timer is here
  //or we can leave the timeout for
  //the execute instruction called with the optional
  //timer parameter - but in this case
  //the final verdict will be 'error'
}

altstep quiescence_handler_AS(timer quiescence) {
  //for all quiescence that is not connected to a gate
  [] any port.receive {
    setverdict(fail);
    mtc.stop;
  }
  [] quiescence.timeout {
    setverdict(pass);
  }
}

```

Figure 6.5.4-2: Required altstep definitions in TTCN-3

```

function behaviourOfTESTER_SS() runs on defaultCT {
  timer quiescence;
  activate(to_handle_deviations_from_TDL_description_AS());

  g_to_map.send(modifies pdcch := {c_rnti := ue})
  g_to_map.send(mac_pdu);
  g_to_map.receive(harq_ack);
  setverdict(pass);
  /*Test Objective Satisfied: TP2 */

  g_to_map.send(modifies pdcch := {c_rnti := unknown});
  g_to_map.send(mac_pdu);

  quiescence.start(five);
  alt{
    [] g_to_map.receive(harq_ack){
      setverdict(fail);
    }
    [] quiescence_handler_AS(quiescence);
    /*Test Objective Satisfied: TP2 */
  }
}

```

Figure 6.5.4-3: Behaviour mapping for Figure 6.5.4-1 in TTCN-3

```

testcase TD_7_1_3_1() runs on MTC_CT
  system defaultCT
  {
    activate(to_handle_deviations_from_TDL_description_AS());
    defaultTC();
    TESTER_SS.start(behaviourOfTESTER_SS());
    all component.done;
  }

```

Figure 6.5.4-4: Test case integrating all steps for mapping Figure 6.5.4-1 to TTCN-3

7 UML Profile Editor

7.1 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 are not aligned with the latest version of the TDL specification parts, but are related to an earlier release of the TDL specification parts.

7.2 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.

7.3 Implemented Facilities

7.3.1 Applying the Profile

Overview

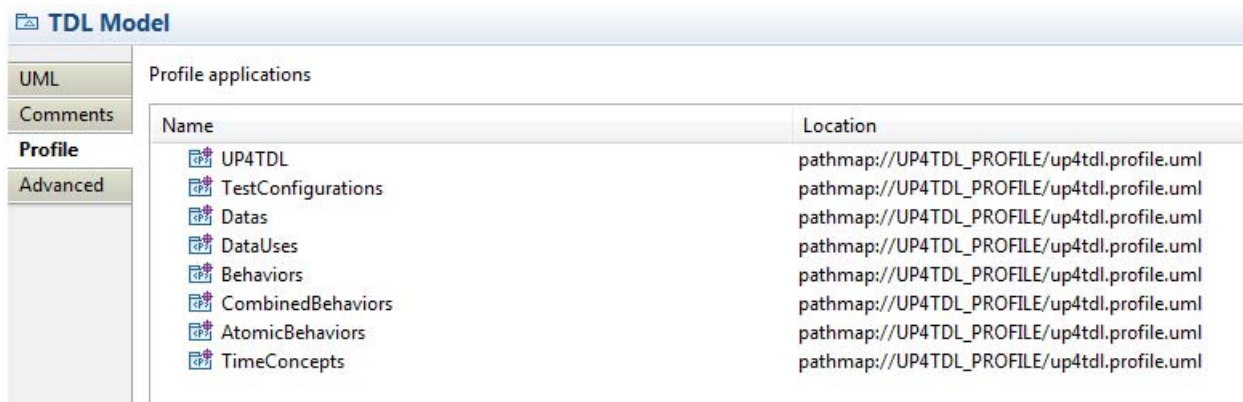
A UML profile allows users to build models with additional constraints and specific properties, while still relying on the UML meta-model. A UP4TDL model is then a UML model with additional constraints and properties tailored towards the domain of TDL.

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 7.3.1-1.



TDL Model	
UML	Profile applications
Comments	
Profile	
Advanced	
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 7.3.1-1: UML profile application

The stereotype applied on the UML model allow the specification of stereotype properties. In Figure 7.3.1-2, the stereotype 'ComponentInstance' is applied to a 'UML::Property'. This allows the user to specify the role property, in this case, 'tester'.

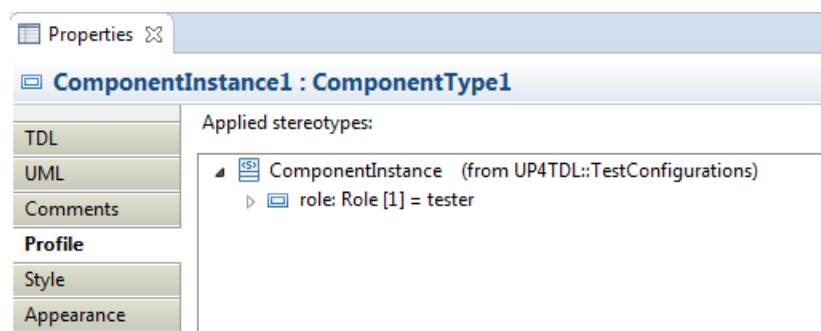


Figure 7.3.1-2: Stereotype property specification

7.3.2 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'.

7.3.3 Editing Models with the Model Explorer

As shown in clause 6.2.1, 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 7.3.3-1.

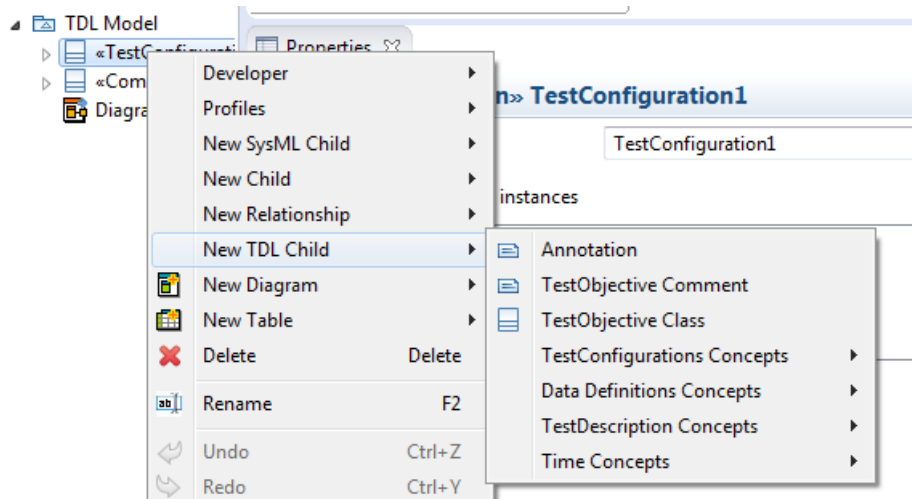


Figure 7.3.3-1: Adding TDL-stereotyped elements

7.3.4 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 7.3.4-1 illustrates the property view of a 'ComponentInstance', which contains its 'Name', 'Type', and 'Role' properties.

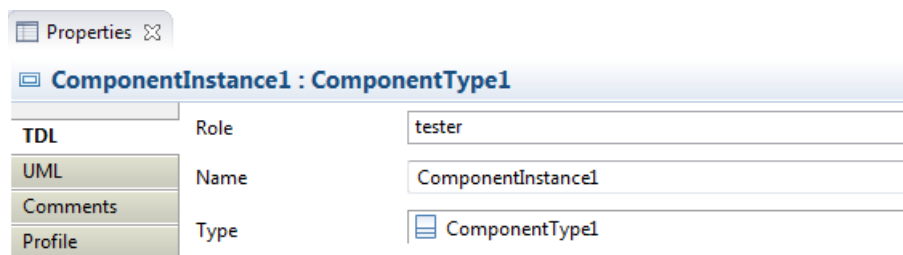


Figure 7.3.4-1: Editing TDL-specific properties

7.3.5 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 7.3.5-1.

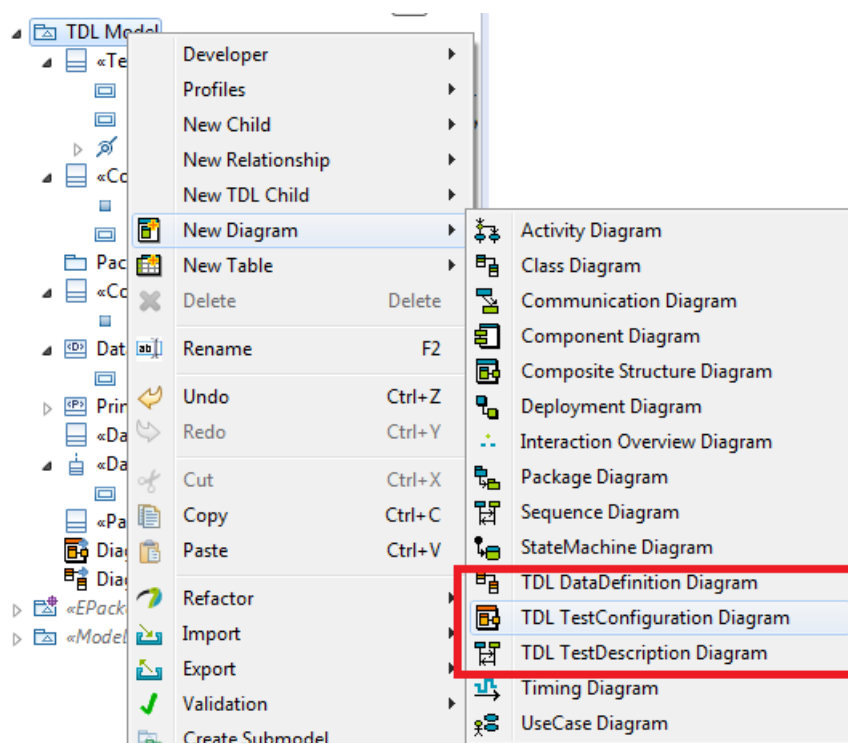


Figure 7.3.5-1: Creating TDL-specific diagrams

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 7.3.5-2.

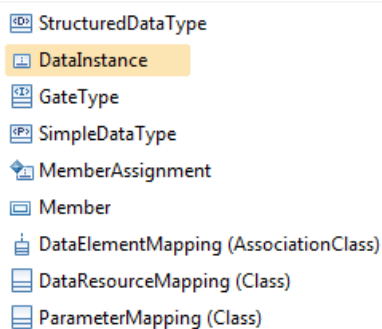


Figure 7.3.5-2: DataDefinition Diagram palette

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 7.3.5-3. An example of the TDL TestConfiguration Diagram is shown in Figure 7.3.5-4.

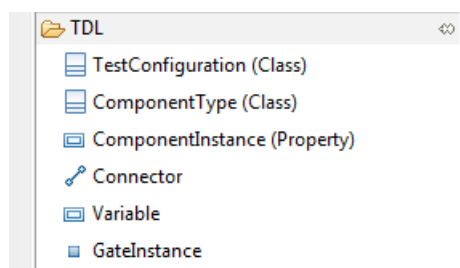


Figure 7.3.5-3: TestConfiguration Diagram palette

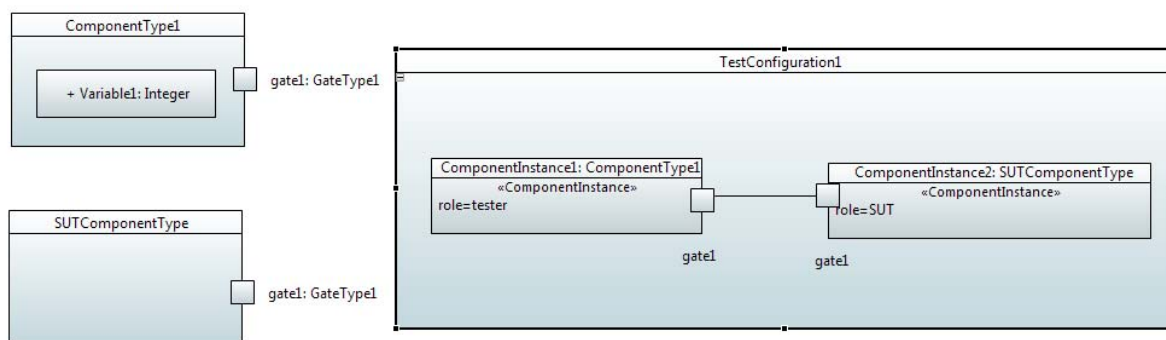


Figure 7.3.5-4: TestConfiguration Diagram example

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 7.3.5-5.

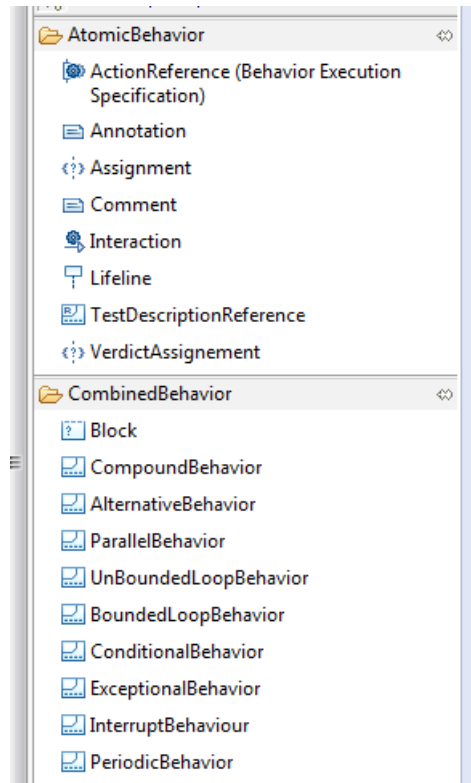


Figure 7.3.5-5: TestDescription Diagram palette

8.1 Generalized Process

8.1.1 Process Overview

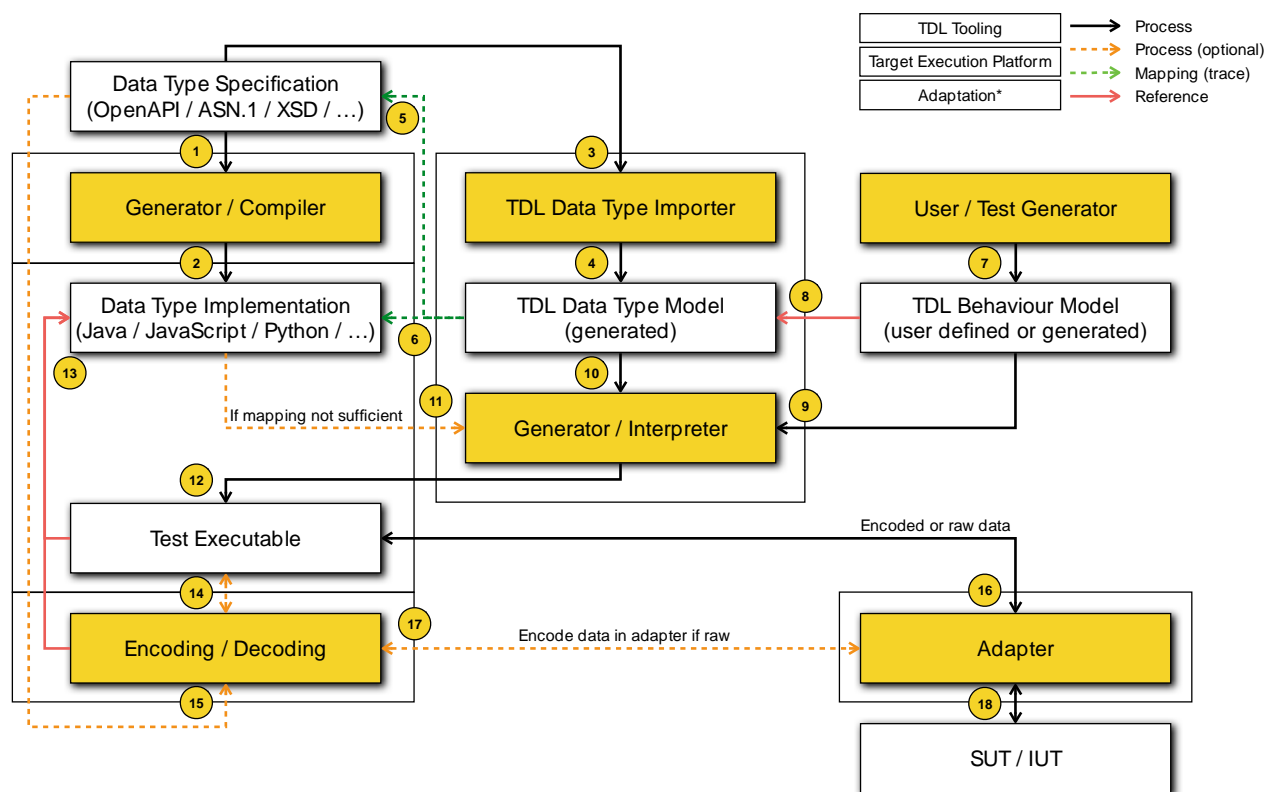


Figure 8.1.1-1: Overall process for importing existing external data type specifications in TDL

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™ and ASN.1 are provided in the subsequent clauses.

The overall process for the importing and use of data types from OpenAPI™, ASN.1, and other specifications is outlined in Figure 8.1.1-1. 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.

8.1.2 Example Instantiation

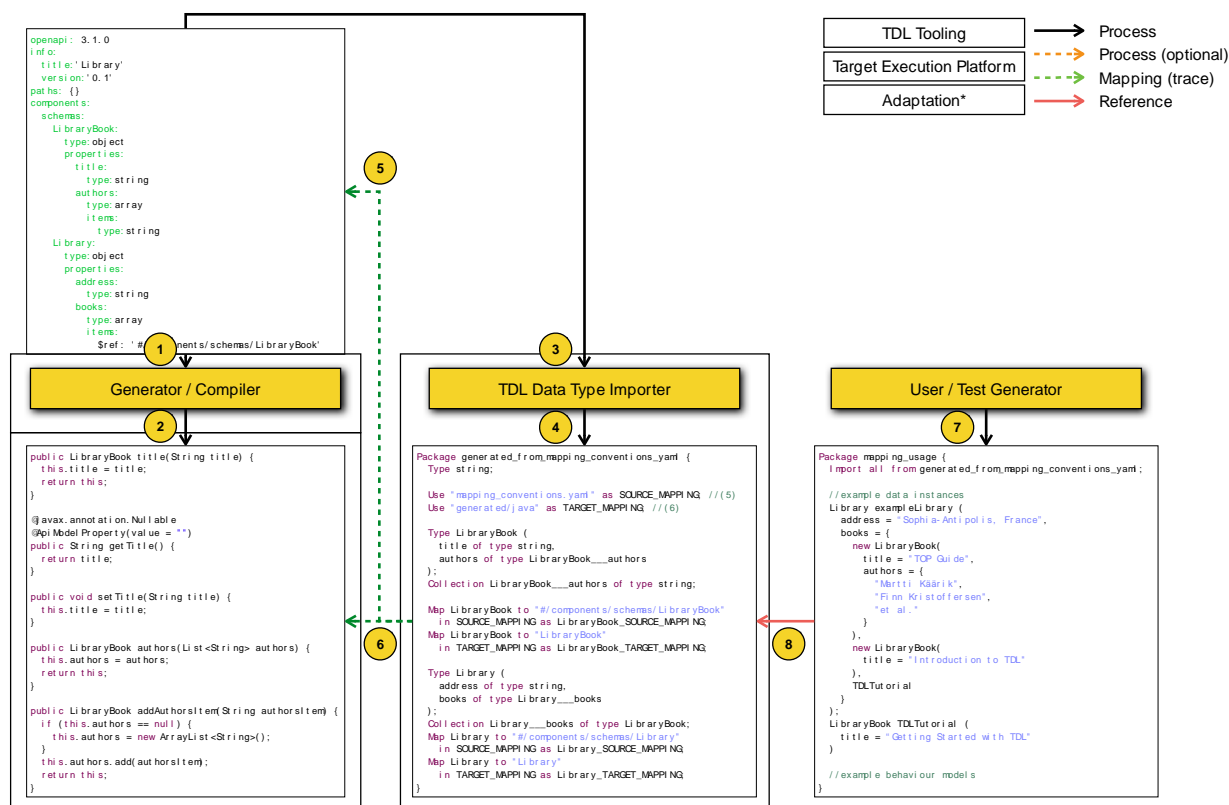


Figure 8.1.2-1: Example instantiation of the overall process from Figure 8.1.1-1

A concrete example for the outlined process is illustrated in Figure 8.1.2-1. 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 ASN.1 specifications. The TOP provides basic capabilities for importing data type specifications from OpenAPI™ and ASN.1, which can be optionally installed in addition to the core meta-model implementation and the different editors.

8.2 Using TDL with OpenAPI™ Specifications

8.2.1 Overview

The OpenAPI™ Specification [i.31] (previously known as the Swagger Specification) is a notation for the specification of interfaces for RESTful web services. In addition to data-related information, OpenAPI™ specifications also include paths to identify resources by means of URLs, along with applicable operations, and corresponding request and response specifications. While these can be used to derive skeletons for structured test objectives and test descriptions as outlined in ETSI EG 203 647 [i.32], within the present document, the focus is solely on data-related information. Further information and guidelines regarding the use of OpenAPI™ for specification and testing at ETSI can be found in ETSI EG 203 647 [i.32].

In addition to a set of primitive data types, OpenAPI™ provides means for defining structured data types. The specification is an extension of the JSON schema [i.33]. Data type schemas may be defined inline or in a *schemas* object which enables reuse of those definitions. In the present document, only the latter is considered. Future editions may add guidelines for inline data definitions as well.

The built-in primitive types in OpenAPI™ are mapped to TDL according to the conventions in Table 8.2.1-1. The mapping relies on a TDL library of predefined types and constraints. As OpenAPI™ specifications may include format specifications for types, a generic constraint (OpenAPIFormat) with corresponding quantifiers may be used to capture this information in the derived TDL data model. Non-standard formats may be present in an OpenAPI™ specification as well. The generic constraint can be used for such formats as well.

Table 8.2.1-1: OpenAPI™ Built-in Type Mapping

OpenAPI™ Type	Type in TDL	Constraints	Formats and Patterns
integer	Integer	OpenAPIFormat	int32, int64
number	Real	OpenAPIFormat	float, double
string	String	OpenAPIFormat	e-mail, password
boolean	Boolean		

A structured type in OpenAPI™ is either an 'array' with member type declaration ('items' object) or an 'object' with a set of properties ('properties' object). Consequently, the transformation of OpenAPI™ data types into TDL data types involves the following conventions:

- If the data type corresponds to one of the primitive data types within the OpenAPI™ library as indicated in Table 8.2.1-1, the 'Type' is mapped to the corresponding 'SimpleDataType' from Table 8.2.1-1.
- If the data type is an 'object', it is mapped to a 'StructuredDataType' with a 'name' corresponding to the name of the OpenAPI™ data type.
- If the data type is an 'array', it is mapped to a 'CollectionDataType' with a 'name' corresponding to the name of the OpenAPI™ data type. The data type indicated in the 'items' object is mapped to the corresponding 'DataType' as the 'itemType' of the 'CollectionDataType'. If 'minItems' and 'maxItems' are specified for the 'array', the corresponding predefined constraints need to be added to the 'CollectionDataType'.
- Each item in the 'properties' object of the 'object' object is mapped to a 'Member' within the corresponding 'StructuredDataType' with a 'name' corresponding to the property. If a 'Member' with the same 'name' exists, no action is taken. All 'Member's are to be marked as optional, except for 'Member's corresponding to properties which are listed in the 'required' array of the 'object'. The 'dataType' of the 'Member' corresponds to:
 - A new 'DataType' corresponding to the 'type' of the property with a 'name' prefixed with the 'name' of the containing 'StructuredDataType' in case a property is of 'type' 'object'.
 - A 'SimpleDataType' corresponding to the 'type' of the property in case the 'SimpleDataType' is one of the predefined 'DataTypes' within the OpenAPI™ library as indicated in Table 8.2.1-1.
- Nested 'objects' are transformed according to the conventions above.
- If the property contains an 'enum' array, it is mapped to an 'EnumDataType' with a 'name' corresponding to the name of the property. The items contained in the 'enum' array are mapped to 'SimpleDataInstance's of the 'EnumDataType' that are contained in the 'EnumDataType'.

- Corresponding 'DataElementMapping's are created for the defined data types. 'DataElementMapping's for 'DataType's derived from anonymous (inline) data types are not created. The 'DataElementMapping's may include target platform mappings in addition to the source mappings to the OpenAPI™ specifications.

8.2.2 Examples

As an example consider the OpenAPI™ snippet shown in Figure 8.2.2-1 and the derived TDL data type model snippet showing in Figure 8.2.2-2. Corresponding 'StructuredDataType's are created for both the 'Library' and 'LibraryBook' data types, as well as for the nested anonymous 'object's and 'array's, which are prefixed with 'Library____' and 'LibraryBook____' accordingly. This would also apply to additional anonymous 'object's and 'array's nested further within the 'object's. The 'dataTypes' for the corresponding 'Member's are then set accordingly. Finally, both source and target (for Java in this example) 'DataElementMapping's are provided.

```
components:
  schemas:
    LibraryBook:
      type: object
      properties:
        title:
          type: string
        authors:
          type: array
          items:
            type: string
        reviewers:
          type: array
          items:
            type: string
    Library:
      type: object
      properties:
        address:
          type: string
        books:
          type: array
          items:
            $ref: '#/components/schemas/LibraryBook'
```

Figure 8.2.2-1: OpenAPI™ example including nested anonymous data types

```
Type LibraryBook (
  String title,
  LibraryBook___authors authors,
  LibraryBook___reviewers reviewers
)
Collection LibraryBook___authors of String
Collection LibraryBook___reviewers of String

Type Library (
  String address,
  Library___books books
)
Collection Library___books of LibraryBook

Use "mapping_conventions.yaml" as SOURCE_MAPPING
Use "generated/java" as TARGET_MAPPING
Map LibraryBook to "#/components/schemas/LibraryBook"
in SOURCE_MAPPING as LibraryBook_SOURCE_MAPPING
Map LibraryBook to "LibraryBook"
in TARGET_MAPPING as LibraryBook_TARGET_MAPPING
Map Library to "#/components/schemas/Library"
in SOURCE_MAPPING as Library_SOURCE_MAPPING
Map Library to "Library"
in TARGET_MAPPING as Library_TARGET_MAPPING
```

Figure 8.2.2-2: Corresponding flattened TDL definitions for Figure 8.2.2-1

8.3 Using TDL with ASN.1 Specifications

8.3.1 Overview

ASN.1 (Abstract Syntax Notation One) Recommendation ITU-T X.680 [i.34] is a standardized language for the specification of data types and data structures. As the name implies, the specifications are abstract and therefore independent of a specific target platform. The specifications provide the information about the structure and encoding of the data which can be processed by generators or compilers to produce data type implementations for the desired target language and platform, including codecs for encoding and decoding the data for transmission. While TDL is not concerned with the encoding and decoding at the specification level, in many cases the test execution platform needs to include codes for the operationalisation of the tests.

When ASN.1 specifications are imported in TDL, the level of detail may vary from the bare essentials, including the data types only, to including additional constraints, and even encoding information (where applicable). The additional information can be utilised for early validation of the TDL specifications. While it is also possible to specify constant values in ASN.1, these are not covered in the guidelines at present.

ASN.1 includes a set of built-in types, some of which are mapped to TDL according to the conventions in Table 8.3.1-1. The mapping relies on a TDL library of predefined types and constraints. The generic constraints (ASN1String, ASN1DateTime, ASN1Real, ASN1ObjectIdentifier) may be used to provide additional patterns for the contents of data instances of the corresponding data types to facilitate validation. Alternatively, a tool may implement implicit validation based on the underlying types.

Table 8.3.1-1: ASN.1 Built-in Type Mapping

ASN.1 Type	Type in TDL	Constraints	Examples and Patterns
BITSTRING	BITSTRING	ASN1String	"1101'B", also "Named BITS" () : [0 1]+B
OCTETSTRING	OCTETSTRING	ASN1String	"A3B2'H", "10010'B": [A-F 0-9]+'H
BMPString	BMPString	ASN1String	"": 16 bit Character
IA5String	IA5String	ASN1String	"Hallo": 8 bit ASCII
GeneralString	GeneralString	ASN1String	: all graphic/character sets, SPACE, DELETE
GraphicString	GraphicString	ASN1String	: all graphic sets, SPACE
NumericString	NumericString	ASN1String	"34 8": [0-9, SPACE]+
PrintableString	PrintableString	ASN1String	"Black, Blue + Brown": [a-z,A-Z,'()]+,.-?:/=,SPACE]
TeletexString	TeletexString	ASN1String	: CCITT T.101
T61String	T61String	ASN1String	: CCITT T.101
UniversalString	UniversalString	ASN1String	: ISO10646
UTF8String	UTF8String	ASN1String	: ASCII + Control
VideotexString	VideotexString	ASN1String	: CCITT T.100, T.101
VisibleString and ISO646String	VisibleString	ASN1String	: ASCII Printing
UTCTime	UTCTime	ASN1DateTime	"991231235959+0200": YYMMDDhhmm[ss]Z
GeneralizedTime	GeneralizedTime	ASN1DateTime	"20200425175522.214+0200": YYYYMMDDHH[MM[SS[.fff]]]Z (ISO 8601 [i.38])
DATE	Date	ASN1DateTime	"1636-09-18": YYYY-MM-DD
TIME-OF-DAY	TimeOfDay	ASN1DateTime	"18:30:23": HH:mm:ss
DATE-TIME	DateTime	ASN1DateTime	"2000-11-22T18:30:23": YYYY-MM-DDThh:mm:ss
INTEGER	Integer		
REAL	Real	ASN1Real	
BOOLEAN	Boolean		
NULL	Null		
OBJECT IDENTIFIER	ObjectIdentifier	ASN1ObjectIdentifier	id-ssp OBJECT IDENTIFIER ::= { itu-t (0) identified-organization (4) etsi (0) smart-secure-platform (3666) part1 (1) }
RELATIVE OBJECT IDENTIFIER	ObjectIdentifier	ASN1ObjectIdentifier	Relative_id-ssp RELATIVE-OID ::= { smart-secure-platform (3666) part1 (1) }

The transformation of ASN.1 data types into TDL data types involves the following conventions:

- If the data type corresponds to one of the predefined 'DataType's within the ASN.1 library as indicated in Table 8.3.1-1, the ASN.1 'Type' is mapped to the corresponding TDL 'SimpleDataType' from Table 8.3.1-1. For the supported ASN.1 types the following additional restrictions apply:
 - NULL type is only used in the scope of a Choice type where if there is no information, the corresponding alternative is activated.
- If the ASN.1 data type is a 'SequenceType', a 'SetType', or a 'ChoiceType', it is mapped to a 'StructuredDataType' with a 'name' corresponding to the name of the ASN.1 data type. In the case of 'ChoiceType', a 'Constraint' with the predefined 'union' 'ConstraintType' is applied to the corresponding 'StructuredDataType', and all 'Member's are marked as optional.
- If the ASN.1 data type is a 'SequenceOfType' or a 'SetOfType', it is mapped to a 'CollectionDataType' with a 'name' corresponding to the name of the ASN.1 data type. The 'Type' indicated for the 'SequenceOfType' or 'SetOfType' is mapped to the corresponding 'DataType' as the 'itemType' of the 'CollectionDataType'.
- Each 'ComponentType' in the 'ComponentTypeList' of the 'SequenceType', 'SetType', or 'ChoiceType' is mapped to a 'Member' within the corresponding 'StructuredDataType' with a 'name' corresponding to the 'identifier' of the 'ComponentType'. If a 'Member' with the same 'name' exists, no action is taken. The 'dataType' of the 'Member' corresponds to:
 - A new 'DataType' corresponding to the 'Type' of the 'ComponentType' with a 'name' prefixed with the 'name' of the containing 'StructuredDataType' in case a 'ComponentType' is directly contained within another 'ComponentType'.
 - A 'SimpleDataType' corresponding to the 'Type' of the 'ComponentType' in case the 'SimpleDataType' is one of the predefined 'DataType's within the ASN.1 library as indicated in Table 8.3.1-1.
- Nested 'ComponentType's are transformed according to the conventions above.
- If the ASN.1 data type is an 'EnumeratedType', it is mapped to an 'EnumDataType' with a 'name' corresponding to the name of the ASN.1 data type. The contained 'EnumerationItem's are mapped to 'SimpleDataInstance's of the 'EnumDataType' that are contained in the 'EnumDataType'. There are no guidelines for 'NamedNumber's at present.
- Corresponding 'DataElementMapping's are created for the defined data types. 'DataElementMapping's for 'DataType's derived from anonymous (inline) data types are not created. The 'DataElementMapping's may include target platform mappings in addition to the source mappings to the ASN.1 specifications.

For the following built-in types not mentioned in Table 8.3.1-1, the transformations to TDL types are done as follows:

- UnrestrictedCharacterStringType: Replace the CHARACTER STRING type with its associated type obtained by expanding inner subtyping in the associated type of the CHARACTER STRING type (see clause 44.5 of Recommendation ITU-T X.680 [i.34]) to the corresponding TDL type.
- EmbeddedPDVType: Replace any EMBEDDED PDV type with its associated type obtained by expanding inner subtyping in the associated type of the EMBEDDED PDV type (see clause 36.5 of Recommendation ITU-T X.680 [i.34]) to the corresponding TDL type.
- ExternalType: replace the EXTERNAL type with its associated type obtained by expanding inner subtyping in the associated type of the EXTERNAL type (see clause 37.5 of Recommendation ITU-T X.680 [i.34]) to the corresponding TDL type.
- InstanceOfType: Replace the INSTANCE OF type with its associated type obtained by substituting INSTANCE OF DefinedObjectClass by its associated ASN.1 type (see clause C.7 of Recommendation ITU-T X.681 [i.36]) and map all ASN.1 types to their TDL types according to Table 8.3.1-1.

8.3.2 Examples

For example, as shown in Figure 8.3.2-1 and Figure 8.3.2-2, for the 'NodeDescriptor' and related type definitions taken from [i.35], a corresponding 'StructuredDataType' is created, including derived 'DataType's for the nested 'aNode' anonymous 'ContentType', as well as the 'aLink', 'aFile', and 'aDirectory' anonymous 'ContentType's nested further within the 'aNode' 'ContentType'. The 'dataType's for the corresponding 'Member's are then set accordingly. The 'DataType' for the 'NodeIdentity' type as well as the derived 'DataType' for the 'aNode' 'Member' are assigned a 'Constraint' with the 'union' 'ConstraintType'.

```
NodeDescriptor ::= SEQUENCE
{
  aNodeName NodeName, -- Node name
  aShortName UUID, -- Short node name
  aNode CHOICE
  {
    aLink SEQUENCE
    {
      aLinkedFileIdentity NodeIdentity, -- Identity of the linked SSP file
      aLinkedFileSize FileSize -- Size of the linked SSP file
    },
    aFile SEQUENCE
    {
      aFileSize FileSize -- Size of the SSP file
    },
    aDirectory SEQUENCE
    {
    }
  },
  aMetaData SEQUENCE OF MetaDatum OPTIONAL, -- Optional meta data
  aACL SET OF AccessControl OPTIONAL -- Access Control List attribute
}

/* Node identity */
NodeName ::= UTF8String (SIZE(1..16)) -- node name encoded in UTF-8
NodeReference ::= SEQUENCE (SIZE(1..6)) OF NodeName -- pathname and node name

NodeIdentity ::= CHOICE
{
  aShortName UUID, -- UUID of file reference using absolute pathname
  aNodeReference NodeReference -- Node reference
}
```

Figure 8.3.2-1: ASN.1 example including nested anonymous data types (excerpt from [i.35])

```

Type NodeDescriptor (
  aNodeName of type NodeName,
  aShortName of type UUID,
  aNode of type NodeDescriptor___aNode,
  optional aMetaData of type NodeDescriptor___aMetaData,
  optional aACL of type NodeDescriptor___aACL
);
Type NodeDescriptor___aNode { union } (
  aLink of type NodeDescriptor___aNode___aLink,
  aFile of type NodeDescriptor___aNode___aFile,
  aDirectory of type NodeDescriptor___aNode___aDirectory
);

Collection NodeDescriptor___aMetaData of type MetaDatum;
Collection NodeDescriptor___aACL of type AccessControl;

Type NodeIdentity { union } (
  aShortName of type UUID,
  aNodeReference of type NodeReference
);
Collection NodeReference of type NodeName;
Type NodeDescriptor___aNode___aLink (
  aLinkedFileIdentity of type NodeIdentity,
  aLinkedFileSize of type FileSize
);
Type NodeDescriptor___aNode___aFile (
  aFileSize of type FileSize
);

```

Figure 8.3.2-2: Corresponding TDL definitions (excerpt) for Figure 8.3.2-1

As an example consider the ASN.1 snippet shown in Figure 8.3.2-3 and the derived TDL data type model snippet showing in Figure 8.3.2-4. Corresponding 'StructuredDataType's are created for both the 'Library' and 'Document' data types, as well as for the nested anonymous 'ContentType's, which are prefixed with 'Library___' and 'Document___' accordingly. This would also apply to additional anonymous 'ContentType's nested further within the 'ContentType's. The 'dataType's for the corresponding 'Member's are then set accordingly. The derived 'DataType' 'Document___number' for the 'number' 'ContentType' of type 'CHOICE' is assigned a 'Constraint' with the 'union' 'ConstraintType'. Default values are not present in the derived TDL data model, as TDL does not support default values for type definitions. However, a data type implementation in the target platform may include default values. In TDL it is possible to define a data instance which provides default values which can be overridden when the data instance is used. Finally, source 'DataElementMapping's are provided.

```

Library ::= SEQUENCE {
  address  UTF8String DEFAULT "Sophia-Antipolis, France",
  documents SEQUENCE OF Document
}

Document ::= SEQUENCE {
  title  UTF8String (SIZE(1..128)),
  status  ENUMERATED {draft, published, historical},
  authors SEQUENCE OF UTF8String,
  number  CHOICE {
    es  INTEGER,
    eg  INTEGER,
    tr  INTEGER
  } OPTIONAL,
  updated DATE
}

```

Figure 8.3.2-3: ASN.1 example including nested anonymous data types

```

Type Library (
  address of type UTF8String,
  documents of type Library___documents
);
Type Document (
  title of type UTF8String,
  status of type Document___status ,
  authors of type Document___authors ,
  optional number of type Document___number ,
  updated of type Date
);
Collection Library___documents of type Document;
Collection Document___authors of type UTF8String;
Type Document___number { union } {
  es of type Integer,
  eg of type Integer,
  tr of type Integer
};
Enumerated Document___status {
  Document___status draft;
  Document___status published;
  Document___status historical;
}
Use "example-1-library.asn" as SOURCE_MAPPING;
Map Library to "Library" in SOURCE_MAPPING as Library_MAPPING;
Map Document to "Document" in SOURCE_MAPPING as Document_MAPPING;

```

Figure 8.3.2-4: Corresponding flattened TDL definitions (excerpt) for Figure 8.3.2-3

9 TDL Runtime / Execution

9.1 Java: Code generator

9.1.1 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.

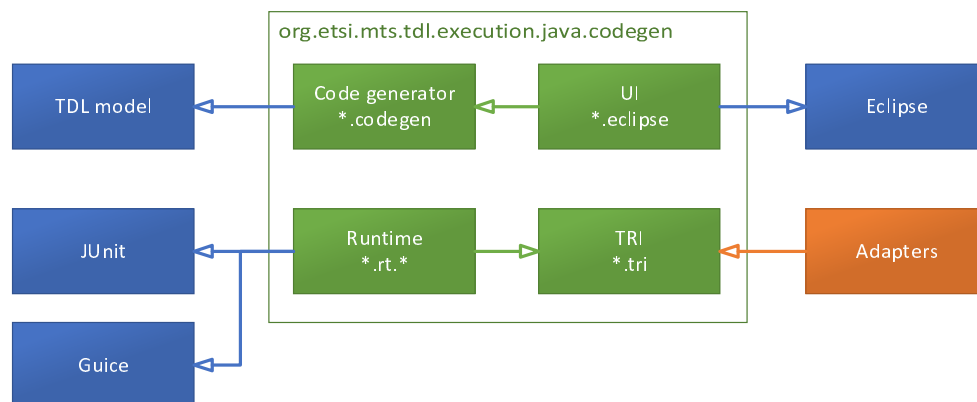


Figure 9.1.1-1: Project structure and dependencies of Java code generator

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.

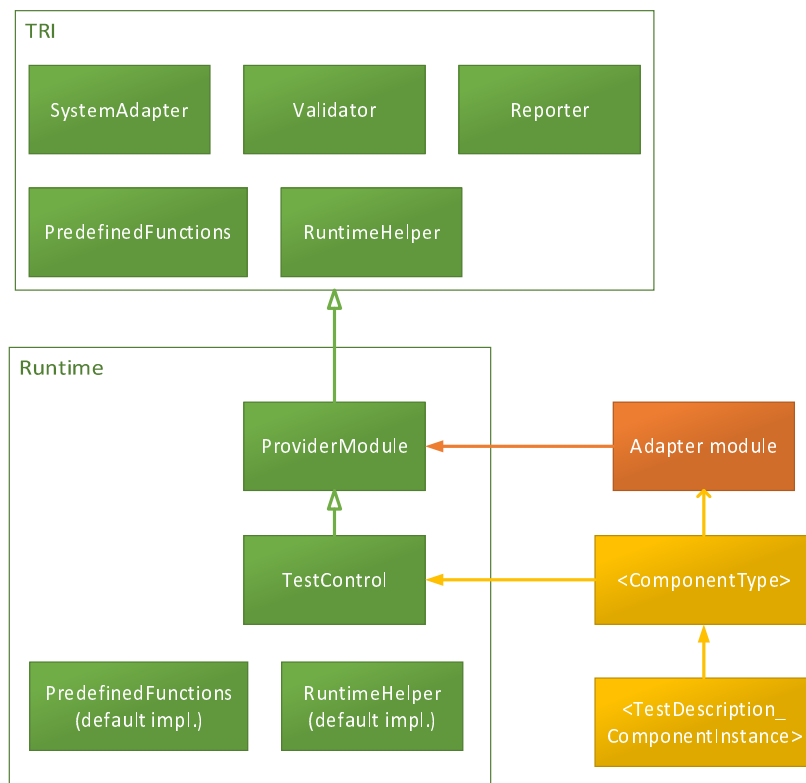


Figure 9.1.1-2: Structure of runtime classes of Java code generator

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.

9.1.2 Test Runtime Interface (TRI) for Java

9.1.2.1 Overview

The TRI consists of functional interfaces that are used to perform environment specific operations and various classes of information objects that are passed to or returned from as arguments of those operations.

The info classes are used to specify the following:

- gate and connection information for 'SystemAdapter';
- data types and values as arguments for interactions;
- test execution verdicts; and
- annotations for the above.

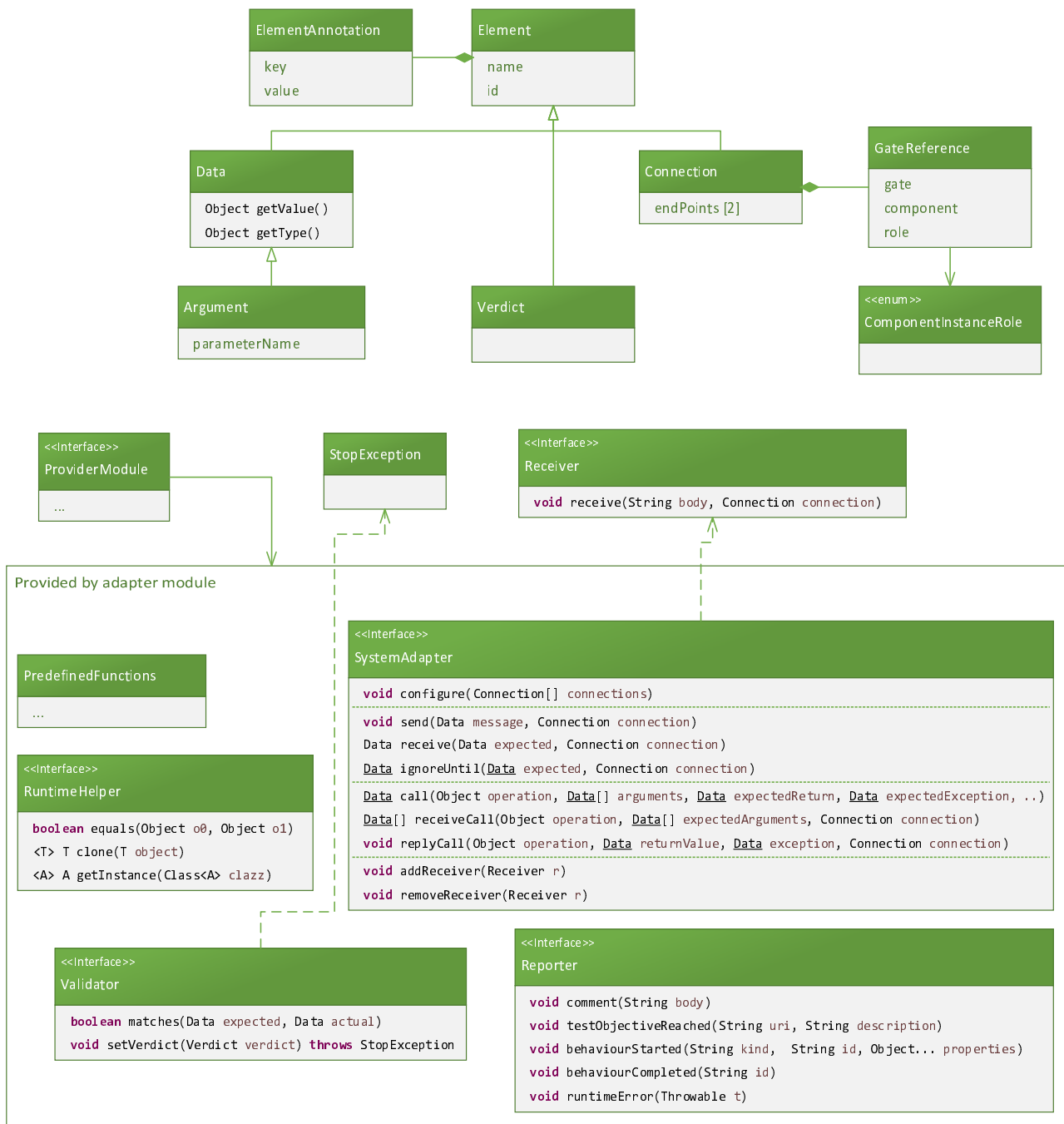


Figure 9.1.2.1-1: Java TRI interfaces and classes

'PredefinedFunctions' class implements all predefined functions as specified in TDL [i.13].

'RuntimeHelper' class provides utilities for cloning Java objects and determining object equality possibly in environment dependent manner. It also provides instances of Java classes for the runtime where required (see clause 9.1.3).

'Validator' provides matching of received and expected data and manages test execution verdicts. 'Validator' may stop the test execution if a calculated or provided verdict so dictates by throwing a 'StopException'.

'Reporter' may be provided for logging or other purposes and it receives information from runtime about executed behaviours with comments and covered test objectives.

9.1.2.2 TRI: SystemAdapter

'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'.

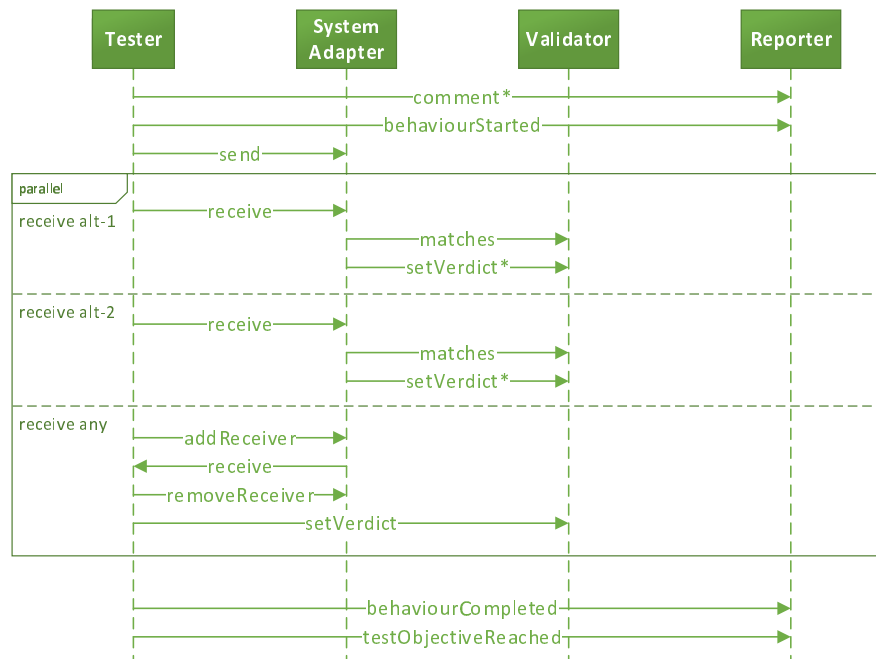


Figure 9.1.2.2-1: Example of method calls involving the SystemAdapter

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 9.1.3). This code generation feature is configurable in settings.

9.1.3 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
- 'Member's 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.

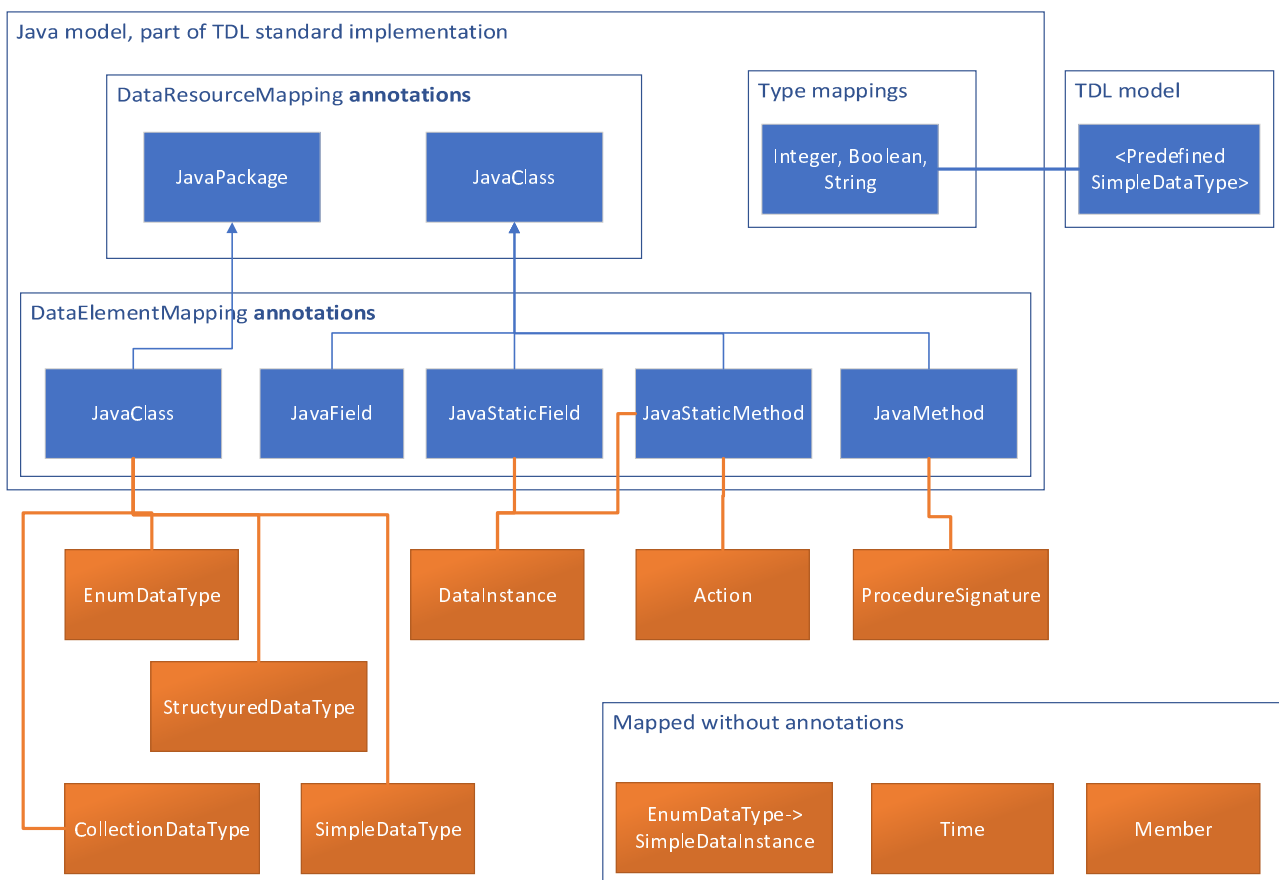


Figure 9.1.3-1: Required annotations for Java mappings

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.

9.1.4 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).

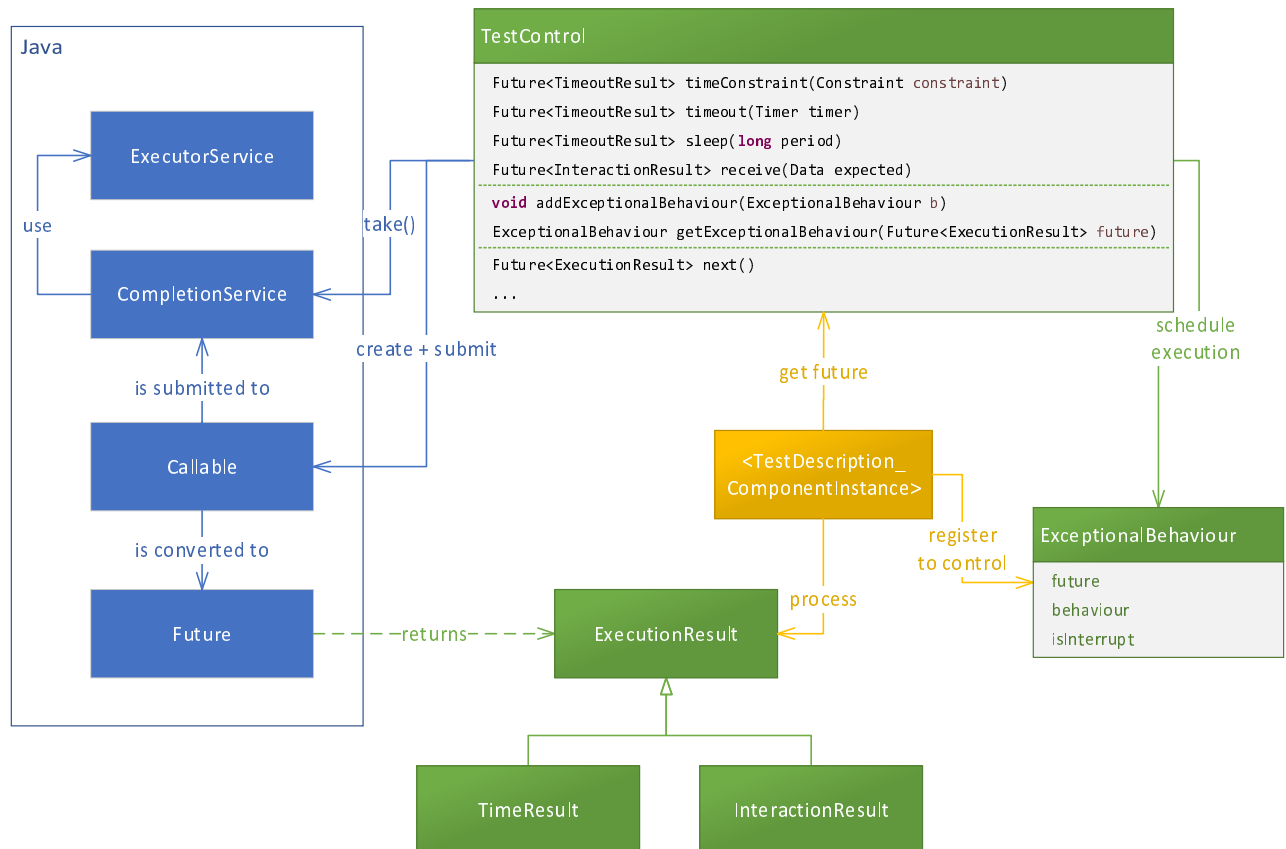


Figure 9.1.4-1: Event handling component dependencies in TDL Java runtime

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 9.1.4-2 describes an example scenario of receiving a message with time constraint while a default behaviour is activated.

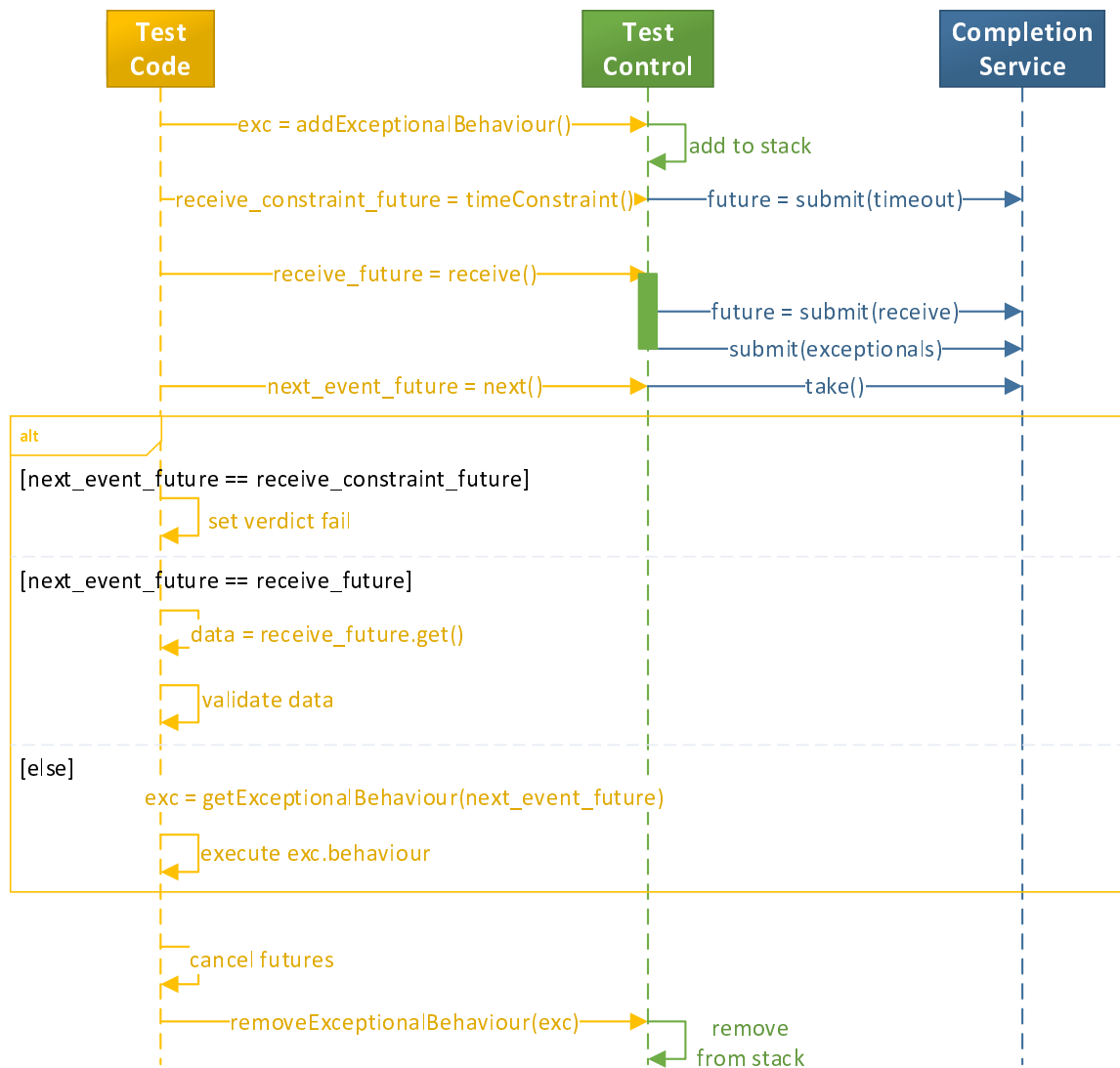


Figure 9.1.4-2: Event handling process in TDL Java runtime

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

Annex A:

Technical Realisation of the Reference Implementation

The technical representation of the TDL reference implementation is available as an open source project available at <http://top.etsi.org/>. 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 at <https://tdl.etsi.org/index.php/open-source>.

History

Document history		
V1.1.1	February 2018	Publication
V1.2.1	September 2020	Publication
V1.3.1	March 2022	Publication