Object Management Group

109 Highland Avenue
Needham, MA 02494
USA

Telephone: +1-781-444-0404
Facsimile: +1-781-444-0320
rfp@omg.org

UTP 2 WG

UML Testing Profile 2 (UTP 2)

Revised Submission

OMG Document: ad/2016-10-01

Date: 15 March 2017

UML Testing Profile 2 (UTP 2)

Revised Submission

In response to: UML Testing Profile 2 (UTP 2) RFP (OMG Document ad/2013-12-08)

OMG Document Number: **ad/2016-10-01**

Normative reference: **-**

Machine readable file(s):

- **ad/2016-10-02** http://www.omg.org/spec/UTP/20161101/utp.xmi
- **ad/2016-10-03** http://www.omg.org/spec/UTP/20161101/utptypes.xmi
- **ad/2016-10-04** http://www.omg.org/spec/UTP/20161101/utpauxiliary.xmi

Inventory file: **ad/2016-10-05**

The following OMG member organizations submitted this specification (in alphabetic order):

- Fraunhofer FOKUS, Germany.
- SOFTEAM, France.

The following OMG and external member organizations supported this specification (in alphabetic order):

- PTC Inc., United Kingdom and USA.
- Hamburg University of Applied Science, Germany.
- KnowGravity Inc., Switzerland.
- Grand Software Testing, USA.
- SELEX SI, Italy.
- Simula Research Lab, Norway.

State:	under review
Document version:	1.3
Generated on:	March 15, 2017
Model name:	UTP 2
Model version:	6

USE OF SPECIFICATION - TERMS, CONDITIONS & NOTICES

LICENSES

PATENTS

GENERAL USE RESTRICTIONS

OMG's Issue Reporting Procedure

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page http://www.omg.org, under Documents, Report a Bug/Issue (http://www.omg.org/technology/agreement.)

Table of Contents

List of Figures

0 Submission-specific Material

0.1 Contact Information

Primary Contact

The primary contact for this specification is Marc-Florian Wendland, marc-florian.wendland@fokus.fraunhofer.de.

Submission Team

The following people have been active members of the UTP 2 submission team:

- Shaukat Ali, Simula Research Lab, Norway
  shaukat@simula.no
- Alessandra Bagnato, SOFTEAM, France
  alessandra.bagnato@softeam.fr
- Etienne Brosse, SOFTEAM, France
  etienne.brosse@softeam.fr
- Gabriella Carrozza, SELEX ES, Italy
  gabriella.carrozza@selex-es.com
- Zhen Ru Dai, Hamburg University of Applied Science, Germany
  dai@informatik.haw-hamburg.de
- Rolf Gubser, KnowGravity Inc., Switzerland
  rolf.gubser@knowgravity.com
- Jon D. Hagar, Grand Software Testing, USA
  embedded@ecentral.com
- Andreas Hoffmann, Fraunhofer FOKUS, Germany
  andreas.hoffmann@fokus.fraunhofer.de
- Markus Schacher, KnowGravity Inc., Switzerland
  markus.schacher@knowgravity.com
- Ina Schieferdecker, Fraunhofer FOKUS, Germany
  ina.schieferdecker@fokus.fraunhofer.de
- Marc-Florian Wendland, Fraunhofer FOKUS, Germany
  marc-florian.wendland@fokus.fraunhofer.de
- Tao Yue, Simula Research Lab, Norway
  tao@simula.no

Supporters

The following people are supporters of the UTP 2 submission:

- Andreas Korff, PTC Inc., United Kingdom and USA
  akorff@ptc.com

0.2 Guide to the Material in the Submission

The material accompanied by this initial submission consists of four normative artifacts:

- The specification document containing the normative specification of UTP 2 (ad/2016-10-01): The section How to read this document provides the reader with a brief guidance through the specification document.
- UTP profile specification, a machine-readable file: ad/2016-10-02
- UTP types library, a machine-readable file: ad/2016-10-03
- UTP auxiliary library, a machine-readable file: ad/2016-10-04
- Inventory file: ad/2016-10-05

0.3 Statement of Proof of Concept

Since UTP 2 is an extension of UTP 1 and all submitters and supporters have extensive experience in applying UTP 1, many concepts of UTP 2 have already been applied in practice. The examples shown in section Annex A (Informative): Examples illustrate an early application of UTP 2.

Furthermore, at least the following companies have committed to provide implementations of UTP 2:

- Fraunhofer FOKUS, Germany.
- KnowGravity Inc., Switzerland.
- Simula Research Lab, Norway.
- SOFTEAM, France.

0.4 Mandatory Requirements

Testing Conceptual Model and Vocabulary	
Requirement id	MR-0001
Description	Submissions shall define and provide a conceptual model that includes important test artifacts such as Test Plan, Test Model, Test Suite, Test Case, Test Data, Test Design Directive, Test Execution Log, and Test Result.
Resolution	Section (Informative) Conceptual Model defines the conceptual model of UTP 2 by means of an SBVR vocabulary.

UML Profile	
Requirement id	MR-0002
Description	Submissions shall provide a UML Profile as an implementation of the conceptual model as requested by Testing Conceptual Model and Vocabulary. The UML Profile shall be compatible to UML 2.5.
Resolution	Section Profile Specification and Model Libraries describe the implementation of the conceptual model as UML 2.5-compliant UML profile.

XMI	
Requirement id	MR-0003
Description	Submissions shall provide XMI compatible machine-readable files of the UML Profile (requested in UML Profile) compliant with version 2.5 of XMI.
Resolution	Machine-readable files do supplement this submission.

Reuse Terminology	
Requirement id	MR-0004
Description	Submissions shall reuse existing terminology of industry-relevant testing standards including ISO 29119, ISTQB Glossary, and UTP 1.2 and explicitly mention the source of each term.
Resolution	The conceptual model has been built first and foremost on adopted and slightly modified terms and their definitions of

	ISO 29119 [ISO29119] and ISTQB [ISTQB]. Copyright issues with ISO have been carefully addressed and resolved by not relying on ISO definitions.

Model Structure	
Requirement id	MR-0005
Description	Submissions shall enable the Test Modeler to group test artifacts by different levels of structure.
Resolution	This requirement is realized in two ways. The implicit containment and association structure is obtained from the underlying UML base elements. The explicit model structure is defined on top of UML by means of tag definitions of the UTP 2 stereotypes.

Levels of Abstractions	
Requirement id	MR-0006
Description	Submissions shall enable the Test Modeler to Model at different levels of abstraction (such as MDA's CIM, PIM, and PSM) for specifying test artifacts.
Resolution	This requirement is fulfilled by relying on UML. UML allows for different levels of abstraction for a given modeling methodology. UTP 2 inherits these characteristics.

Test Architecture	
Requirement id	MR-0007
Description	Submissions shall enable the Test Modeler to specify Test Configurations, including Test Components and their interfaces.
Resolution	This requirement is satisfied on the one hand by the concepts test configuration, test item, test component, test item configuration, test component configuration (see Section Test Architecture), on the other hand by means of UML class and composite structures.

SUT Interfaces	
Requirement id	MR-0008

Description	Submissions shall enable the Test Modeler to specify or refer to the interfaces of the System Under Test.
Resolution	This requirement is satisfied by relying on UML and the concept test item. UML allows for both specification of interfaces from scratch and the import of existing interfaces. These interfaces can be realized by test items (see Section Test Architecture)

Quality Models	
Requirement id	MR-0009
Description	Submissions shall be expressive enough to address different Software Quality Models, such as ISO 25010.
Resolution	This requirement is satisfied by using the metaclass ValueSpecification as possible value for the tagged value of TestContext.testType. This enables users to define their own schema and naming for test types.

Functional Testing	
Requirement id	MR-0010
Description	Submissions shall enable the Test Modeler to capture the artifacts, relationships, and behaviors used for functional testing.
Resolution	This requirement is satisfied by relying on UML. UTP 2 concentrates on modelling and testing functionality in the first place, however, non-functional properties could be modeled by using e.g., MARTE in combination with UTP 2.

Non-functional Testing	
Requirement id	MR-0011
Description	Submissions shall enable the Test Modeler to capture the artifacts, relationships, and behaviors used for non-functional testing.
Resolution	This requirement is realized by the profile mechanism UML comes along with. Additional profiles for modeling non-functional properties can be combined with UTP 2 in a way that UTP 2 stays focused on the testing aspects and all other aspects are integrated via additional profiles.

Black-Box Testing	
Requirement id	MR-0012
Description	Submissions shall provide means to support the Test Modeler in designing black-box tests.
Resolution	This requirement is satisfied by the fact that UTP 2 is a modeling language. White-box (or glass-box) approaches to testing are defined to be based on the implementation code of a test item, which is actually missing on modeling level. This gives rise to the fact that UTP 2 is first and foremost dedicated to black-box approaches to test design. Nonetheless, white-box test design techniques (such as structural coverage criteria) can be applied on model level as well.

Test Objectives	
Requirement id	MR-0013
Description	Submissions shall enable the Test Modeler to specify objectives and strategies for the design of Test Cases.
Resolution	This requirement is satisfied by means of test design technique, test objective and test requirement (see Test Behavior)

Arbitration Specification	
Requirement id	MR-0014
Description	Submissions shall enable the Test Modeler to specify the arbitration logic for assigning final verdicts to sets of Test Cases (see Arbitration Specification in Appendix A References & Glossary Specific to this RFP).
Resolution	This requirement is satisfied by means of arbitration specification (see Section Test Evaluation)

Test Data Aspects	
Requirement id	MR-0015
Description	Submissions shall enable the Test Modeler to specify data aspects to stimulate inputs sent to the SUT, Responses from the SUT, or for the generation of Test Data.
Resolution	This requirement is satisfied on the one hand by reusing the Type and Classifier concepts of UML. On the other hand UTP

	2 introduces the concepts of data specification and data values that enables the user to model data on different levels of abstractions and with different degrees of completeness (see section Test Data)

Test Log Structure	
Requirement id	MR-0016
Description	Submissions shall allow the Test Modeler to specify the structure of Test Execution Logs.
Resolution	This requirement is satisfies by means of the concept test log structure (see section test logging)

Test Log	
Requirement id	MR-0017
Description	Submissions shall allow the Test Modeler to integrate and visualize Test Execution Logs based on the structure defined in requirement "Test Log Structure".
Resolution	This requirement is satisfies by means of test log (see section test logging)

Test Model Independence	
Requirement id	MR-0018
Description	Submissions must ensure that UTP 2 Test Models can be specified standalone (i.e., that the Test Model must be specifiable without the concurrent existence of other models such as requirements models or design models).
Resolution	This requirement is satisfied by relying on UML. Since UML allows for both importing existing models and creating isolated models from scratch and UTP 2 does not restrict either of these possibilities, users can decide which approach to apply.

Test Coverage Goals	
Requirement id	MR-0019
Description	Submissions shall enable the Test Modeler to specify test

	coverage goals.
Resolution	This requirement is satisfied by introducing the concept test design technique. A test design technique specifies both the model/elements (i.e. test design input) that should be used for test case derivation and the strategy that are applied on the test design input.

Test Management	
Requirement id	MR-0020
Description	Submissions shall be expressive enough to support Test Management activities with respect to Test Planning, Test Analysis, Test Design, Test Execution, and Test Evaluation.
Resolution	This requirement is satisfied by multiple means such as the concepts test context, test case, test requirement, test log, test objective, test design technique, test design directive, test set, test execution schedule. In general, UTP 2 is not a specification made for modelling test management processes, but could be used for controlling and steering test processes based on information obtained from the UTP 2 model. More specific information about how some of the earlier mentioned concepts might be used to support test control and steering is given next: At first, UTP 2 introduces the concept of a test context, which denotes a container for information relevant to activities of the same test-subprocess (synonyms are test plans, test level plans, test phase or test-subprocess). A test context is dedicated to enable the test manager to prescribe or specify relevant information for the respective testing activities (such as test analysis, test design, test planning). Secondly, UTP 2 introduces the concepts test set (synonym test suite) which allows for grouping test cases that should be executed together as they share a common purpose. Being able to define such sets is a very basic test management concept which was actually missing in UTP 1.x. A related concept introduced by UTP 2 is the notation of test execution schedule (synonym test control or test procedure) that enables testers to order the execution of a set of test cases according to certain criteria. Although the implementation of those criteria is not in the scope of UTP 2, the concept itself is a vital means to formally express the execution order that was

	actually calculated by external entities (such as a test designer, an intelligent scheduling and reasoning engine etc.). Finally, the introduction of a test design technique enables test managers to prescribe which strategy should be applied in order to derive test cases. The actual test design might be executed manually by a human, or automatically by means of a test generator.

Test Documentation	
Requirement id	MR-0021
Description	Submissions shall enable the Test Modeler to capture all information required to produce Test Documentation for industry-relevant standards such as ISO 29119.
Resolution	This requirement is satisfied by all the stereotypes that allow capturing information most relevant for test reports such as progress reports or final reports. UTP 2 was designed that at least the information required by ISO 29119-3 [ISO29119] were covered without depending on that ISO standard. In general, this requirement was satisfied by means of the concepts test context, test objective, test requirement, test case, test set, test execution schedule, test case log etc.

Change Tracking	
Requirement id	MR-0022
Description	Submissions shall explain changes from UTP version 1.2.
Resolution	Each introduced stereotype contains a short description about its deviation from a corresponding UTP 1.2 stereotype.

0.5 Non-mandatory Requirements

Grey-Box Testing	
Requirement id	NMR-0001
Description	Submissions may provide means to support the Test Modeler in designing grey-box tests.
Resolution	None.

White-Box Testing	
Requirement id	NMR-0002
Description	Submissions may provide means to support the Test Modeler in designing white-box tests.
Resolution	This requirement is satisfied by relying on UML. Since UTP 2 does not restrict the accessibility of the test item during test case execution at all, UTP 2 can easily be used to support white box test case execution (i.e., access to internal information, variables, memory etc. of the test item or related components during test case execution in order to evaluate whether the test item behaved as expected).

Internal Traceability	
Requirement id	NMR-0003
Description	Submissions may enable the Test Modeler to specify mechanisms for explicitly specifying traceability links between different test artifacts to support additional analysis such as impact analysis.
Resolution	This requirement is satisfied by relying on UML. UML offers facilities to establish trace dependencies among various UML elements (in general, any NamedElements). This is supplemented by UTP 2 in terms of associations among stereotypes such as TestCase::realizes:TestRequirement, TestDesignStrategy::testDesignModel:Element,TestDesignStrategy::derivedTestDesignElement Element

Graphical Representation	
Requirement id	NMR-0004
Description	Submissions may provide graphical notation (e.g. diagram shapes, icons, etc.) for UTP 2 elements.
Resolution	None.

TestIF Mapping	
Requirement id	NMR-0006

Description	Submissions may provide a Mapping from a UTP 2 model to a TestIF representation (e.g. by means of model2text transformations, QVT, spreadsheet, etc.).
Resolution	None.

Test Deployment	
Requirement id	NMR-0007
Description	Submissions may enable the Test Modeler to specify means relevant for the deployment of test-relevant artifacts.
Resolution	This requirement is satisfied by relying on UML. UML does offer concepts to define deployment specifications. This feature of UML provides helpful concepts for modelling the deployment of test artifacts.

MOF Model or OWL/RDF Model	
Requirement id	NMR-0008
Description	Submissions may provide an MOF model or OWL/RDF model of the conceptual model as requested by Testing Conceptual Model and Vocabulary.
Resolution	None.

Formal Semantics	
Requirement id	NMR-0009
Description	For defining the conceptual model and vocabulary a formal semantics such as MOF 2.4.1 or ODM may be used.
Resolution	This requirement is satisfied by utilizing SBVR as the language for the conceptual model and vocabulary of UTP 2.

0.6 Issues to be discussed

Other Specifications	
Requirement id	I-0001
Description	Submissions shall discuss the relationship to other, related specifications such as SysML and BMM (see Relationship to

	other OMG Specifications and activities)
Resolution	**Relationship to MOF:** The UML profile implementation of UTP 2 is based on the MOF specification. The serialization and de-serialization of model instances is prescribed by XMI according to MOF [MOF]. **Relationship to UML:** The UML profile implementation is defined by using the UML profile mechanism. Thus, UTP 2 is depending on UML as its base language [UML]. **Relationship to UTP 1.x:** As its predecessor, UTP 2 tries to stay as close as possible to UTP 1.x in concepts, vocabulary and definitions. Since UTP 2 has a much broader intention and scope, it is not always possible to be backward compatible with UTP 1.x [UTP]. **Relationship to SysML:** SysML [SysML] is related to UTP in two ways. First, SysML re-specified the concept of test cases and verdicts as defined by UTP 1.x. With the modified version of test cases and verdicts in UTP 2, SysML is not up-to-date with UTP 2 and might need to be updated accordingly. Secondly, SysML offers the concept "Requirement". Since requirements are most important for testing activities, it was important to not prevent the establishing of a trace link among the UTP 2 concepts test case, test requirement or test objective and the SysML concept "Requirement". The current version of UTP 2 is defined independently from SysML. Nevertheless, whenever cohesion exists between UTP 2 and SysML, especially when defining test requirements, both languages can be applied simultaneously. This is achieved by setting UML::Class as base metaclass for test objective and test requirement. **Relationship to TestIF:** UTP 2 might employ TestIF [TestIF] as non-MOF-based ex-

	change format. If such a mapping would be available, UTP 2-based tools would become interoperable with TestIF-based toolings. Since TestIF is not finally adopted yet, UTP 2 does not specify any mapping to TestIF so far. **Relationship to BMM:** UTP 2 reuses the notion of BMM [BMM] "Objective" and "Strategy" so far. Since there is no UML profile for BMM defined, the implementation of test objective and test design technique does not contradict any specification. It is assumed that further submission of UTP 2 will reuse additional concepts from BMM such as "Directives" and "Goals". **Relationship to SBVR:** UTP 2 utilizes SBVR [SBVR] for the definition of its conceptual model as a community-specific vocabulary. Except this application, there is not further relationship to SBVR noticeable for UTP 2. **Relationship to Essence:** There is no relationship identified between UTP 2 and Essence. **Relationship to Diagram Definition (DD):** There is currently no relationship between UTP 2 and DD [DD], since UTP 2 does not specify any further graphical elements so far. **Relationship to fUML:** There is no explicit relationship to fUML [FUML] defined for UTP 2 yet. UTP 2 is not supposed to build executable models, although it does not prevent building such test models whatsoever.

Traceability	
Requirement id	I-0002
Description	Submissions shall discuss the specification of traces between UTP 2 elements and model elements outside UTP 2.
Resolution	UTP 2 was designed to express testing-relevant information in a domain-independent manner. The premise of UTP 2 was to

	result in a solution that might be reused by different domains (such as finance, avionics, transportation etc.) and UML-based modeling languages (such as pure UML, SysML, MARTE etc.). In its pure essence, the technical realization of tracing is not a concept that is relevant to testing in the first place, even though it is essential to trace between relevant information (e.g., risk information, requirements specifications, defect reports, PLM information etc.). However, the realization of such traces is not in scope of UTP 2, and thus, not covered by the submission whatsoever. There are many tools and frameworks available that allow for establishing traces between UTP 2 (as a representative of MOF-based models) and further relevant artifacts (even outside the MOF realm).

Non-SW Properties	
Requirement id	I-0003
Description	Submissions shall discuss the specification of properties to be tested that are not software-specific. Examples of such properties include physical quantities of complex Systems like engine power or oven temperature.
Resolution	The test item could represent any item that is modeled by means of UML. When SysML is used on to of the UML, this allows specifying test cases for non-software items such as hardware or heterogeneous systems as well and also to refer to corresponding properties of such systems [SysML].

1 Scope

In 2001, a working group at the OMG started developing a UML Profile dedicated to Model-based testing, called UML Testing Profile (UTP). It is a standardized language based on OMG's Unified Modeling Language (UML) for designing, visualizing, specifying, analyzing, constructing, and documenting the artifacts commonly used in and required for various testing approaches, in particular model-based testing (MBT) approaches. UTP has the potential to assume the same important role for model-based testing approaches as UML assumes for model-driven system engineering.

UTP is a part of the UML ecosystem (see figure below), and as such, it can be combined with other profiles of that ecosystem in order to associate test-related artifacts with other relevant system artifacts, e.g. requirements, risks, use cases, business processes, system specifications etc. This enables requirements engineers, system engineers and test engineers to bridge the communication gap among different engineering disciplines.

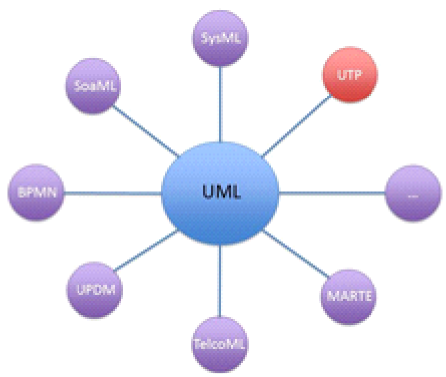


Figure 1 - The UML Ecosystem

As the interest in Model-based testing approaches and languages of the industry increases, UTP started to attract more and more users. Until today, UTP is the only available, standardized language for Model-based approaches to help in the validation and verification of software-intensive systems. Model-based test specifications expressed with the UML Testing Profile are independent of any methodology, domain, environment or type of system.

Eight years later, the UTP working group (WG) has agreed on consolidating the experiences and achievements of UTP in order to justify the move from UTP 1.2 to a successor specification. These efforts resulted in a Request For Information (RFI) for UML Testing Profile 2 (UTP 2), which was aimed at eliciting and gathering the shortcomings of the current UTP and the most urgent requirements for a successor specification from the OMG and model-based testing community.

Some of the main issues in the RFI responses are that UTP 2 should:

- be able to design test models of different test levels.
- address testing of non-functional requirements.
- be able to reuse test logs for further test evaluation and test generation.
- meet industry-relevant standards.
- integrate with SysML for requirements traceability.
- and so forth.

The UML Testing Profile 2 (UTP 2) was designed to meet the requirements derived from the RFI responses.

People may use the UML Testing Profile in addition to UML to:

- Specify the design and the configuration of a test system: Designing a test system includes the identification of the test item (also known as system under test or abbreviated as SUT), its boundaries, the derivation of test components, and the identification of communication channels between interconnected test items test components over which data can be exchanged.
- Build the model-based test plans on top of already existing system models: The possibility to reuse already existing (system) artifacts, e.g. requirements, interface definitions, type definitions etc.
- Model test cases: The specification of test cases is an essential task of each test process in order to assess the quality of the test item and to verify whether the test item complies with its specification.
- Model test environments: A test environment contains hardware, instrumentation, simulators, software tools, and other support elements needed to conduct a test (according to IEEE 610).
- Model deployment specifications of test-specific artifacts: By relying on the UML's deployment specification capabilities, the actual deployment of a test system can be done in a model-based way.
- Model data: Modeling of data includes the data values being used as stimuli into the test item as well as for responses expected from the test item such as the test oracle.
- Provide necessary information pertinent to test scheduling optimization: Test scheduling optimization can be based on priorities, risk-related information, costs etc.
- Document test case execution results: To associate test cases with the actual outcome of their execution within the very same model in order to perform further analysis, calculate specific metrics, etc.
- Document traceability to requirements and other UML model artifacts: Requirements traceability within test specification is important to document and evaluate test coverage and to calculate other metrics such as progress reports. Native traceability is given by the underlying UML capabilities. UTP does not offer different

concepts for traceability other than that provided by UML,

The intended audience for the UML Testing Profile are users who are able to read model-based test specifications expressed within the UML Testing Profile models including:

- Test engineers
- Requirements Engineers
- System/Software Engineers
- Domain experts
- Customer/Stakeholder
- Certification authorities
- Testing tools (test case generators, data generators, schedulers, reporting engines, test script generators, etc.).

The intended audience of this UML Testing Profile specification itself includes, among others:

- People who want to implement UML Testing Profile-compliant tools.
- People who need to/want to/like to teach the UML Testing Profile.
- People who want to improve the UML Testing Profile specification.
- People who want to tailor the UML Testing Profile to satisfy needs of their specific project/domain/process.

Ver.	Date	Author	Description
1.0	May 19, 2014	UTP 2 WG	Initial version
1.1	May 24, 2016	UTP 2 WG	Major revision of almost all sections; some more examples added
1.2	November 7, 2016	UTP 2 WG	Many spelling errors fixed; resolutions in Submission-specific Material updated; sources in Terms and Definitions and in (Informative) Conceptual Model added/updated; section Conformance and introduction to (Informative) Conceptual Model extended; References updated; section UTP 2 Use Cases added; constraints in section Profile Specification updated; ISO 25010, ISO 9126, and FURPS in Quality Models Library removed due to copyright issues; Annex A (Informative): Examples updated
1.3	March 15, 2017	UTP 2 WG	Conceptual Model declared as informative; introductory text of sections "How to read this

			document" and "(Informative) Conceptual Model" slightly updated to make it clear that the conceptual model is informative; colors removed from diagrams

2 Conformance

As a native profile specification of the UML, the UTP 2 has to abide by the conformance types declared for compliant UML profiles. The corresponding conformance types of UML can be found in section 2 "Conformance" of the current UML specification [UML]. This guarantees that the underlying environment of any UTP 2 implementation is a UML modeling environment that is conformant with the UML. The UTP 2 adopted version of UML's conformance types are defined as follows:

- Abstract syntax conformance: All concrete stereotypes and tags are implemented in the profile implementation
- Concrete syntax conformance: Support for the visual representation (i.e. icons) of the UTP concepts is provided by the profile implementation
- Model interchange conformance: (delegated to underlying UML)
- Diagram interchange conformance: (delegated to underlying UML)
- Semantic conformance: All UTP constraints are enforced, either directly in the model with OCL (assuming underlying OCL support) or indirectly by any other suitable means of the underlying modeling environment

In addition to the fundamental conformance types of the UML and its profiling mechanism, UTP 2 specifies two compliance levels for its respective concepts:

- mandatory: concepts that are deemed mandatory have to be implemented in order to claim UTP 2 compliance;
- optional: concepts that are deemed optional might be implemented. If they are implemented, they have to be implemented exactly how they have been specified by the UTP 2 specification - i.e., optional concepts are still normative, but when they are implemented, they have to abide by the conformance types imposed by the underlying UML and its profiling mechanism.

The decisions, which concepts are considered as mandatory and optional, have been based on the typical use cases of UTP 2 (see section 6.3 Typical Use Cases of UTP 2). The main objective of UTP 2 is to design test cases, potentially in an automated manner, and to describe the test architecture in order to execute test cases, potentially in an automated manner. Except from that, UTP 2 provides further helpful concepts for the design and implementation of a test environment that supports various activities of the test process, such as test analysis, manual and automated test design, test execution and evaluation. The concepts required for these activities are grouped by corresponding sections within this specification. The following relates the test process activities with the respective sections of the UTP 2 specification and indicates whether a feature (a set of concepts grouped in a setion) is normative, mandatory or optional:

Test Process Phase	Normative	Mandatory
• Test Analysis Activities		
- Section 8.3.1 Test Analysis	X	-
• Test Design Activities		
- Section 8.3.2 Test Design	X	-
- Section 8.4 Test Architecture	X	X
- Section 8.5.1 Test-specific Procedures	X	X
- Section 8.5.1 Procedural Elements	X	X
- Section 8.5.1 Test-specific Actions	X	X
- Section 8.6.1 Data Specifications	X	-
• Test Execution and Evaluation Activities		
- Section 8.6.2 Data Values	X	-
- Annex C Non-normative data value extensions	-	-
- Section 8.7.1 Arbitration Specifications	X	-
- Section 8.7.2 Test Logging	X	-

In addition to these concepts, UTP 2 specifies three model libraries for UTP 2. The conformance considerations for the libraries are as follows:

UTP 2 Model Libraries	Normative	Mandatory
• Section 9.1 UTP Types Library	X	X
• Section 9.2 UTP Auxiliary Library	X	-
• Section 9.3 UTP Arbitration Library	X	-

Any implementation that wants to claim conformance with UTP 2 specification has to abide by the adopted UTP 2 conformance types for each normative concept. If the concept is deemed mandatory in addition, any implementation that wants to claim conformance with the UTP 2 specification, has to provide those mandatory concepts to the user.

3 References

3.1 Normative References

[MOF]	Object Management Group: **"Meta Object Facility Specification"**
[OCL]	Object Management Group: "**Object Constraint Language - Version 2.4**", February 2014, formal/2014-02-03
[UML]	Object Management Group: "**OMG Unified Modeling Language™ (OMG UML) - Version 2.5**", March 2015, formal/2015-03-01
[XMI]	Object Management Group: **"XML Metadata Interchange Specification"**

3.2 Informative References

[BMM]	Object Management Group: "**Business Motivation Model - Version 1.3**", May 2015, formal/2015-05-19
[DD]	Object Management Group: **"Diagram Definition (DD)"**
[ES20187301]	ETSI ES 201 873-1: **"Methods for Testing and Specifications (MTS) - The Testing and Test Control Notation version 3 - Part 1: TTCN-3 Core Language"**; V4.7.1 (2015-06)
[ES202951]	ETSI ES 202 951: **"Requirements for Modeling Notations. ETSI Standard, Methods for Testing and Specification (MTS)"**; Model-Based Testing (MBT). V1.1.1 (2011-07)
[ES20311901]	ETSI ES 203 119-1: **"Methods for Testing and Specifications (MTS) - The Test Description Language (TDL) - Part 1: Abstract Syntax and Associated Semantics"**; V1.2.1 (2015-06)
[ES20311902]	ETSI ES 203 119-1: **"Methods for Testing and Specifications (MTS) - The Test Description Language (TDL) - Part 2: Graphical Syntax"**; V1.1.1 (2015-06)
[ES20311903]	ETSI ES 203 119-1: **"Methods for Testing and Specifications (MTS) - The Test Description Language (TDL) - Part 3: Exchange Format"**; V1.1.1 (2015-06)
[ES20311904]	ETSI ES 203 119-1: **"Methods for Testing and Specifications (MTS) - The Test Description Language (TDL) - Part 4: Structured Test Objective Specification (Extension)"**; V1.1.1 (2015-06)
[FUML]	Object Management Group: **"Semantics of a Foundational Subset for Executable UML Models (fUML)"**
[HWT2012]	R. Hametner, D. Winkler, and A. Zoitl, "**Agile testing concepts**

	based on keyword-driven testing for industrial automation systems" in IECON 2012-38th Annual Conference on IEEE Industrial Electronics Society, 2012, pp. 3727-3732
[IEC61508]	IEC: **"Functional safety of electrical/electronic/programmable electronic safety-related** **systems -- Part 1: General Requirements"**, Edition 2.0, IEC 61508-1, 2010-04
[ISO1087-1]	ISO: "**Terminology work - Vocabulary - Part 1: Theory and application**", ISO 1087-1:2000(E/F), 15-OCT-2000
[ISO25010]	ISO/IEC: **"System and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - Systems and software quality models"**, ISO/IEC 25010:2011, ISO, 2011-03-01
[ISO29119]	ISO/IEC: **"System and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - Systems and software quality models"**, ISO/IEC 25010:2011, ISO, 2011-03-01 ISO/IEC/IEEE: "**The International Software Testing Standard**", (website)
[ISO9126]	ISO/IEC: **"Software engineering -- Product quality -- Part 1: Quality model"**, ISO/IEC 9126-1:2001, ISO, 2001
[ISTQB]	ISTQB: "**International Software Testing Qualifications Board**", (website)
[MDA]	Object Management Group: "**MDA Guide - Version 1.0.1**", June 2003, omg/2003-06-01
[MDAa]	Object Management Group: **"OMG Architecture Board, "Model Driven Architecture - A Technical Perspective""**
[MDAb]	Object Management Group: **"Developing in OMG's Model Driven Architecture (MDA)"**
[MDAc]	Object Management Group: **"MDA Guide"**
[MDAd]	Object Management Group: **"MDA "The Architecture of Choice for a Changing World""**
[OSLC]	OSLC Wiki: "**Open Services for Lifecycle Collaboration** **Quality Management Specification Version 2.0**", (website)
[SBVR]	Object Management Group: "**Semantics of Business Vocabularies and Business Rules (SBVR) - Version 1.3**", May 2015, formal/2015-05-07
[SEP2014a]	Marquis, Jean-Pierre, "**Category Theory**", The Stanford Encyclopedia of Philosophy (Winter 2015 Edition), Edward N. Zalta (ed.)

[SysML]	Object Management Group: "**OMG Systems Modeling Language (OMG SysML™) - Version 1.4**", September 2015, formal/2015-06-03
[TCM2008]	J. Tang, X. Cao, and A. Ma, "**Towards adaptive framework of keyword driven automation testing**" in Automation and Logistics, 2008. ICAL 2008. IEEE International Conference on, 2008, pp. 1631-1636
[TestIF]	Object Management Group: "**Test Information Interchange Format (TestIF) Specification - FTF Beta 1**", February 2013, dtc/2013-02-01
[UL2007]	Utting, M., Legeard, B.: **"Practical Model-Based Testing: A Tools Approach"**, Morgan-Kaufmann, 2007
[UPL2012]	Utting, M., Pretschner, A., and Legeard, B.: **"A taxonomy of model-based testing approaches**", in Softw. Test. Verif. Reliab. 22, 5, August 2012, p. 297-312
[UTP]	Object Management Group: "**UML Testing Profile - Version 1.2**", April 2013, formal/2013-04-03
[WikiCT]	Wikipedia entry on "**Category theory**", as visited on January 26, 2016
[WikiM]	Wikipedia entry on "**M**orphism", as visited on October 04, 2016
[WN]	WordNet of the "Cognitive Science Laboratory" at the University of Princeton

4 Terms and Definitions

Name	Description	Source
abstract test case	A test case that declares at least one formal parameter.	UTP 2 WG
abstract test configuration	A test configuration that specifies the test item, test components and their interconnections as well as configuration data that should be abstract test data.	UTP 2 WG
actual data pool	A specification of an actual implementation of a data pool.	UTP 2 WG
actual parameter	A concrete value that is passed over to the procedure and replaces the formal parameter with its concrete value.	UTP 2 WG
alternative	A compound procedural element that executes only a subset of its contained procedural elements based on the evaluation of a boolean expression.	UTP 2 WG
arbitration specification	A set of rules that calculates the eventual verdict of an executed test case, test set or procedural element.	UTP 2 WG
artifact	An object produced or modified during the execution of a process.	UTP 2 WG
atomic procedural element	A procedural element that cannot be further decomposed.	UTP 2 WG
boolean expression	An expression that may be evaluated to either of these values: "TRUE" or "FALSE".	UTP 2 WG
check property action	A test action that instructs the tester to check the conformance of a property of the test item and to set the procedural element verdict according to the result of this check.	UTP 2 WG
complement	A morphism that inverts data)i.e., that replaces the data items of a given set of data items by their opposites).	UTP 2 WG
compound procedural element	A procedural element that can be further decomposed.	UTP 2 WG
concrete test case	A test case that declares no formal parameter.	UTP 2 WG
concrete test	A test configuration that specifies the test item, test components and their interconnections as well as	UTP 2 WG

Name	Description	Source
configuration	configuration data that should be concrete data.	
constraint	An assertion that indicates a restriction that must be satisfied by any valid realization of the model containing the constraint.	[UML]
create log entry action	A test action that instructs the tester to record the execution of a test action, potentially including the outcome of that test action in the test case log.	UTP 2 WG
create stimulus action	A test action that instructs the tester to submit a stimulus (potentially including data) to the test item.	UTP 2 WG
data	A usually named set of data items.	UTP 2 WG
data item	Either a value or an instance.	UTP 2 WG
data partition	A role that some data plays with respect to some other data (usually being a subset of this other data) with respect to some data specification.	UTP 2 WG
data pool	Some data that is an explicit or implicit composition of other data items.	UTP 2 WG
data provider	A test component that is able to deliver (i.e., either select and/or generate) data according to a data specification.	UTP 2 WG
data specification	A named boolean expression composed of a data type and a set of constraints applicable to some data in order to determine whether or not its data items conform to this data specification.	UTP 2 WG
data type	A type whose instances are identified only by their value.	[UML]
duration	The duration from the start of a test action until its completion.	UTP 2 WG
Error	An indication that an unexpected exception has occurred while executing a specific test set, test case, or test action.	UTP 2 WG
executing entity	An executing entity is a human being or a machine that is responsible for executing a test case or a test set.	UTP 2 WG
expect response action	A test action that instructs the tester to check the occurrence of one or more particular responses from the test item within a given time window and to set the procedural element verdict according to the result of this check.	UTP 2 WG

Name	Description	Source
extension	A morphism that increases the amount of data (i.e., that adds more data items to a given set of data items).	UTP 2 WG
Fail	A verdict that indicates that the test item did not comply with the expectations defined by a test set, test case, or test action during execution.	UTP 2 WG
Fault	A verdict that indicates that the tester did not comply with the procedural specifications defined by a procedure during execution of a test set, test case, or test action.	UTP 2 WG
formal parameter	A placeholder within a procedure that allows for execution of the procedure with different formal parameters that are provided by the procedure invocation.	UTP 2 WG
Inconclusive	A verdict that indicates that the compliance of a test item against the expectations defined by a test set, test case, or test action could not be determined during execution.	UTP 2 WG
loop	A compound procedural element that repeats the execution of its contained procedural elements.	UTP 2 WG
main procedure invocation	A procedure invocation that is considered as the main part of a test case by the test case arbitration specification.	UTP 2 WG
morphism	A structure-preserving map from one mathematical structure to another.	[WikiM]
negative	A compound procedural element that prohibits the execution of its contained procedural elements in the specified structure.	UTP 2 WG
None	A verdict that indicates that the compliance of a test item against the expectations defined by a test set, test case, or test action has not yet been determined (i.e., it is the initial value of a verdict when a test set, test case, or test action was started).	UTP 2 WG
parallel	A compound procedural element that executes its contained procedural elements in parallel to each other.	UTP 2 WG
Pass	A verdict that indicates that the test item did comply with the expectations defined by a test set, test case, or test action during execution.	UTP 2 WG
PE end duration	The duration between the end of the execution of a procedural element and the end of the execution of	UTP 2 WG

Name	Description	Source
tion	the subsequent procedural element.	
PE start duration	The duration between the end of the execution of a procedural element and the beginning of the execution of the subsequent procedural element.	UTP 2 WG
postcondition	A boolean expression that is guaranteed to be True after a test case execution has been completed.	UTP 2 WG
preconditon	A boolean expression that must be met before a test case may be executed.	UTP 2 WG
procedural element	An instruction to do, to observe, and/or to decide.	UTP 2 WG
procedural element verdict	A verdict that indicates the result (i.e., the conformance of the actual properties of the test item with its expected properties) of executing a test action on a test item.	UTP 2 WG
procedure	A specification that constrains the execution order of a number of procedural elements.	UTP 2 WG
procedure invocation	An atomic procedural element of a procedure that invokes another procedure and wait for its completion.	UTP 2 WG
property	A basic or essential attribute shared by all members of a class of test items.	UTP 2 WG
refinement	A morphism that decreases the amount of data (i.e., that removes data items from a given set of data items).	UTP 2 WG
response	A set of data that is sent by the test item to its environment (often as a reaction to a stimulus) and that is typically used to assess the behavior of the test item.	UTP 2 WG
sequence	A compound procedural element that executes its contained procedural elements sequentially.	UTP 2 WG
setup procedure invocation	A procedure invocation that is considered as part of the setup by the arbitration specification and that is invoked before any main procedure invocation.	UTP 2 WG
stimulus	A set of data that is sent to the test item by its environment (often to cause a response as a reaction) and that is typically used to control the behavior of the test item.	UTP 2 WG
suggest verdict action	A test action that instructs the tester to suggest a particular procedural element verdict to the arbitration specification of the test case for being taken	UTP 2 WG

Name	Description	Source
	into account in the final test case verdict.	
teardown procedure invocation	A procedure invocation that is considered as part of the teardown by the responsible arbitration specification and that is invoked after any main procedure invocation.	UTP 2 WG
test action	An atomic procedural element that is an instruction to the tester that needs to be executed as part of a test procedure of a test case within some time frame.	UTP 2 WG
test case	A procedure that includes a set of preconditions, inputs and expected results, developed to drive the examination of a test item with respect to some test objectives.	UTP 2 WG
test case log	A test log that captures relevant information on the execution of a test case.	UTP 2 WG
test case verdict	A verdict that indicates the result (i.e., the conformance of the actual properties of the test item with its expected properties) of executing a test case against a test item.	UTP 2 WG
test component	A role of an artifact within a test configuration that is required to perform a test case.	UTP 2 WG
test component configuration	A set of configuration options offered by an artifact in the role of a test component chosen to meet the requirements of a particular test configuration.	UTP 2 WG
test configuration	A specification of the test item and test components, their interconnection as well as their configuration data.	UTP 2 WG
test context	A set of information that is prescriptive for testing activities that are organized and managed together for deriving or selecting test objectives, test design techniques, test design inputs and eventually test cases.	UTP 2 WG
test design directive	A test design directive is an instruction for a test designing entity to derive test artifacts such as test sets, test cases, test configurations, data or test execution schedules by applying test design techniques on a test design input. The set of assembled test design techniques are referred to as the capabilities a test designing entity must possess in order to carry out the test design directive, regardless whether it is carried out by a human tester or a test genera-	UTP 2 WG

Name	Description	Source
	tor. A test design directive is a means to support the achievement of a test objective.	
test design input	Any piece of information that must or has been used to derive testing artifacts such as test cases, test configuration, and data.	UTP 2 WG
test design technique	A specification of a method used to derive or select test configurations, test cases and data. test design techniques are governed by a test design directive and applied to a test design input. Such test design techniques can be monolithically applied or in combination with other test design techniques. Each test design technique has clear semantics with respect to the test design input and the artifacts it derives from the test design input.	UTP 2 WG
test execution schedule	A procedure that constrains the execution order of a number of test cases.	UTP 2 WG
test item	A role of an artifact that is the object of testing within a test configuration.	UTP 2 WG
test item configuration	A set of configuration options offered by an artifact in the role of a test item chosen to meet the requirements of a particular test configuration.	UTP 2 WG
test level	A specification of the boundary of a test item that must be addressed by a specific test context.	UTP 2 WG
test log	A test log is the instance of a test log structure that captures relevant information from the execution of a test case or test set. The least required information to be logged is defined by the test log structure of the test log.	UTP 2 WG
test log structure	A test log structure specifies the information that is deemed relevant during execution of a test case or a test set. There is an implicit default test log structure that prescribes at least the start time point, the duration, the finally calculated verdict and the executing entity of a test case or test set execution should be logged.	UTP 2 WG
test objective	A desired effect that a test or set of test cases intends to achieve.	UTP 2 WG
test procedure	A procedure that constrains the execution order of a number of test actions.	UTP 2 WG
test require-	A desired property on a test case or test set, referring to some aspect of the test item to be tested.	UTP 2 WG

Name	Description	Source
ment		
test set	A set of test cases that share some common purpose.	UTP 2 WG
test set log	A test log that captures relevant information from the execution of a test set.	UTP 2 WG
test set purpose	A statement that explains the rationale for grouping test cases together.	UTP 2 WG
test set verdict	A verdict that indicates the result (i.e., the conformance of the actual properties of the test item with its expected properties) of executing a test set against a test item.	UTP 2 WG
test type	A quality attribute of a test item that must be addressed by a specific test context.	UTP 2 WG
time point	The time point at which a test action is initiated.	UTP 2 WG
verdict	A statement that indicates the result (i.e., the conformance of the actual properties of the test item with its expected properties) of executing a test set, a test case, or a test action against a test item.	UTP 2 WG

5 Symbols

No special symbols have been used in this specification.

6 Additional Information

6.1 How to read this document

This specification is intended to be read by the audience listed below in order to learn, apply, implement and support UTP 2. To understand how UTP 2 relates to other testing standards, all readers are encouraged to read Clause 6 (Additional Information). In order to learn more about the conformance of UML and UTP 2 as well as the compliance levels between the UTP 2 specification and the UTP 2 tool implementation, please read Clause 2 (Conformance). Some references to other standards are listed in Chapter 3 (References). Important definition of test terms, acronyms and additional information are listed in the Clause 4 (Terms and Definitions).

The definition of the UML Testing Profile itself can be found in the Chapters 7-9. Clause 7 ((Informative) Conceptual Model) starts with the definition of a pure conceptual model of UTP 2 independent of any implementation measures. The conceptual model is informative (i.e. non-normative) but provides the big picture of the intended scope of UTP 2. The mapping of the conceptual model to the UML profile specification is described in Clause 8 (Profile Specification). The stereotype mappings abide by the semantics of the conceptual elements in general. Only additional aspects of the semantics regarding the integration of a stereotype with related UML metaclasses will be added in Clause 8.

Clause 9 (Model Libraries) describes the predefined UTP 2 model libraries. The UTP Arbitration Library provides predefined types and elements required for the definition of the profile itself. The UTP Auxiliary Library provides predefined elements for reuse across multiple modeling projects. The UTP Types Library provides additional types that have been proven helpful for the definition of tests.

The Annex sections provide further informative material for UTP 2, in particular an examples section that shows different methodologies how to apply UTP 2 technically and conceptually. The Annex sections are living sections that means they may change among future versions.

Modeling tool vendors should read the whole document, including the annex chapters. Modelers and engineers are encouraged to read Annex A to understand how the language is applied to examples.

This document may be read in both sequential and non-sequential manner.

6.2 Typographical conventions

A set of typographical conventions have been applied to the editorial part of this specification that should help the reader in understanding and relating things to their proper context. These conventions are subsequently explained:

- Concepts of the conceptual model are written in lower letters and colored blue, indicating a link to the section of the conceptual element. Example: test context
- UML metaclasses start with an upper case letter and are written in camel-case. Example: Constraint, BehavioredClassifier
- Stereotypes are start with an upper case letter and are written in camel-case, surrounded by guillemets. Example: «TestContext»
- Properties of metaclasses or tag definitions of stereotypes are stated in italic: Examples: *constrainedElement* (from UML metaclass Constraint), *arbitrationSpecification* (from stereotype «ProceduralElement»)
- Values of Properties or tagged values of tag definitions are stated italic: Examples: *false*, *true*
- OCL constraints as formalization of natural language Constraint descriptions are set in Courier. Example:

```
context TestComponent:
not
self.base_Property.class.getAppliedStereotype('UTP::TestItem')->
oclIsUndefined()
```

6.3 Typical Use Cases of UTP 2

This section briefly summarizes typical use cases of UML Testing Profile V2 (UTP 2) by means of a simple UML use case model. It is intended to give the interested reader an initial idea of who and what for UTP 2 may be used in the context of developing and testing complex systems.

The following use case diagram summarizes typical UTP 2 users and their use cases of UTP 2.

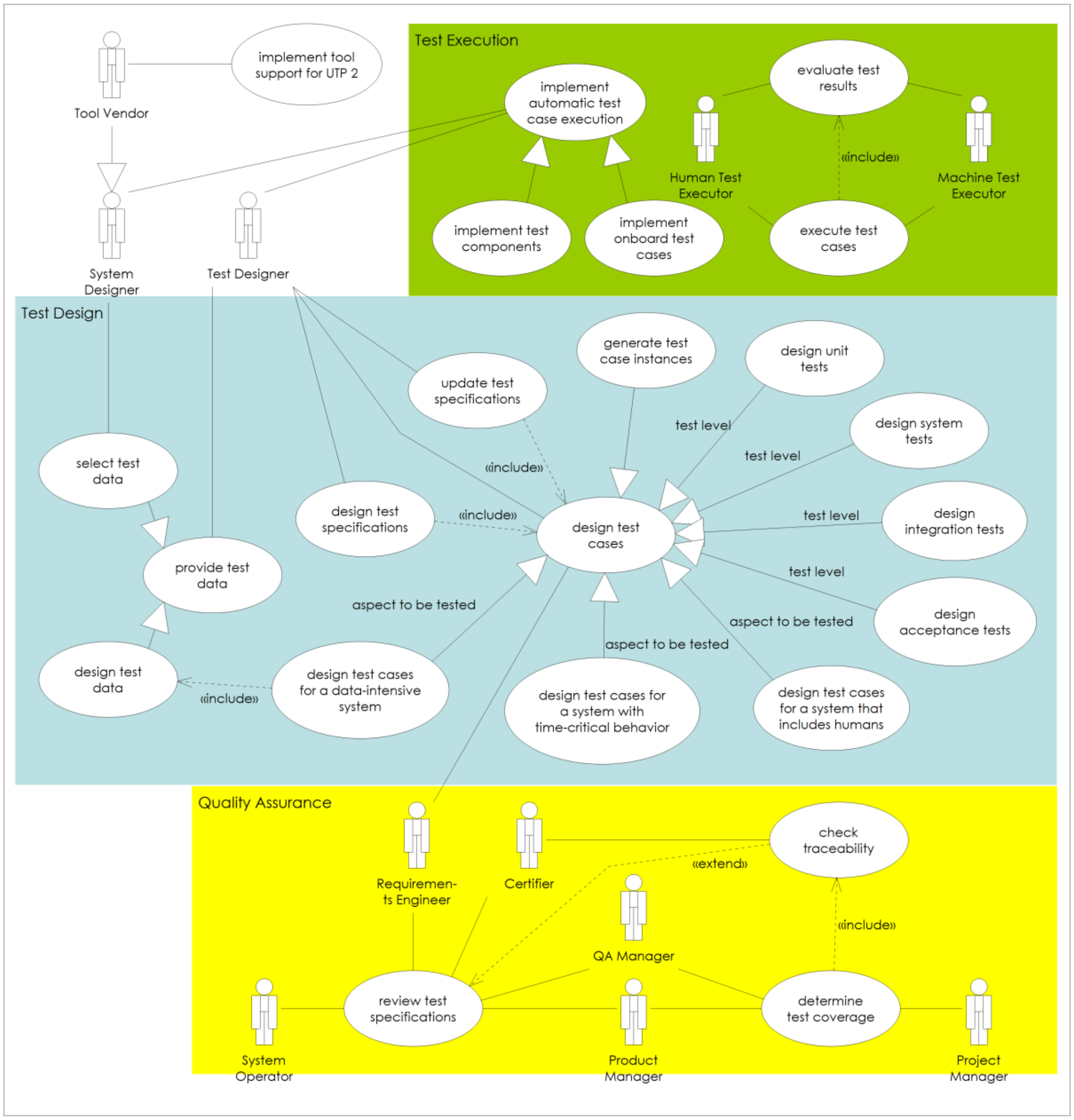


Figure 2 - UTP 2 Use Cases

The following table characterizes the users (represented as UML actors) introduced in the diagram above and lists for each user the use cases related to UTP 2 she or he may directly or indirectly carry-out.

User Type	Description	Use Cases
Certifier	A role of a person responsible for certifying a safety-critical or mission-critical system or product.	• check traceability • review test specifications
Human Test Executor	A role of a person responsible for executing test cases and/or evaluating their outcomes.	• evaluate test results • execute test cases
Machine Test Executor	A machine or device that executes test cases and/or evaluates their outcomes.	• evaluate test results • execute test cases
Product Manager	A role of a person having the overall responsibility for a system or product.	• determine test coverage • check traceability • review test specifications
Project Manager	A role of a person having the overall responsibility for the development, procurement, implementation, or adaption of a system or product or a part of it.	• determine test coverage • check traceability
QA Manager	A role of a person responsible to guarantee the appropriate quality of a system or product.	• determine test coverage • check traceability • review test specifications
Requirements Engineer	A role of a person responsible for gathering, expression and managing the requirements on a system or product.	• design test cases • design acceptance tests • design integration tests • design system tests • design test cases for a data-intensive system • design test data • design test cases for a system that includes humans • design test cases for a system with time-critical behavior • design unit tests • generate test case instances

		• review test specifications • check traceability
System Designer	A role of a person that designs, builds, extends, maintains or updates a system or product.	• implement automatic test case execution • implement onboard test cases • implement test components • select test data
System Operator	A role of a person that utilizes a system or product.	• review test specifications • check traceability
Test Designer	A role of a person that designs, builds, extends, maintains or updates test specifications of a system.	• design test cases • design acceptance tests • design integration tests • design system tests • design test cases for a data-intensive system • design test data • design test cases for a system that includes humans • design test cases for a system with time-critical behavior • design unit tests • generate test case instances • design test specifications • implement automatic test case execution • implement onboard test cases • implement test components • provide test data • select test data • update test specifications
Tool Vendor	A role of a person that develops a tool implementing at least some aspects of the UTP 2 specification.	• implement tool support for UTP 2 • implement automatic test case execution

		• implement onboard test cases • implement test components • select test data

Table 1 - Typical UTP 2 Users

The following table briefly describes the use cases introduced in the diagram above.

Use Case	Description
check traceability	Verification of the traceability between requirements and test cases in order to determine the coverage of a system by a set of test cases.
design acceptance tests	The design of test cases that are used to perform an acceptance test of a system or product, i.e. that the sponsor/customer may decide on the acceptance of that system or product.
design integration tests	The design of test cases that are used to perform an integration test of a system or product, i.e. the verification of the interoperability among its internal components as well as with its environment conforms to its specification.
design system tests	The design of test cases that are used to perform a system test of a system or product, i.e. the verification that the system or product (typically viewed as a black box) fulfills its requirements.
design test cases	The design, elaboration and adaptation of test sets comprising test cases in order to verify the requirements and/or to validate the goals of a system or product.
design test cases for a data-intensive system	The design of test cases for a system whose functionality includes complex processing of data that is of a highly complex structure and/or of large data volumes.
design test cases for a system that includes humans	The design of test cases for a sociotechnical system that includes technical systems as well as humans collaboratively performing complex processes.
design test cases for a system with time-critical behavior	The design of test cases for a system that must comply to soft or hard real-time constraints on its behavior.
design test data	The design and production of data that is of a highly complex structure and/or of large data volumes.
design test specifica-	The elaboration and compilation of all information neces-

tions	sary for carrying-out verification and validation procedures of a system or product. This includes specifying test objectives, test strategies, test procedures, test data, test configurations, evaluation criteria and more.
design unit tests	The design of test cases that are used to perform functional tests of an individual component of a system or product.
determine test coverage	The examination of test sets and test cases with the focus on the coverage provided by of those test sets and test cases with respect to the requirements and/or implementation aspects of a system or product in order to determine the suitability of the test sets and test cases for a given purpose.
evaluate test results	The examination of the results of an executed test set or executed test case in order to determine the verdict of the test set or test case.
execute test cases	The manual or automatic execution of test procedures according to a given test specification composed of sets and/or test cases.
generate test case instances	The manual or automatic production of specific test case instances from a given test specification composed of generic sets and/or test cases.
implement automatic test case execution	The implementation, provisioning and configuration of test infrastructure required to perform and evaluate test sets or test cases automatically.
implement onboard test cases	The implementation of test components and test procedures as part of a system or product in order to make it able to perform self-tests while it is in operation.
implement test components	The implementation, provisioning and configuration of auxiliary test components in order to automate or at least to simplify the execution of test sets or test cases.
implement tool support for UTP 2	The implementation, provisioning or configuration of a tool in order to supports the utilization of UTP 2. This could e.g. be a UML Profile implementing UTP 2 for a particular UML modeling tool or a test execution tool that supports the concepts of UTP 2.
provide test data	The provisioning of dedicated data that is used to perform test sets or test cases.
review test specifications	The quality assurance of a particular test specification in order to fulfill given quality goals.
select test data	The selection and potentially transformation of available operational data in order to use this data during the execution of test sets or test cases.
update test specifica-	The adaption of test objectives, test strategies, test proce-

tions	dures, test data, test configurations, evaluation criteria etc. according to changing requirements and goals of an already existing system or product.

Table 2 - Typical UTP 2 Use Cases

6.4 Relation to testing-relevant standards

The landscape of software/system testing standards is diversified. Many domain-specific standards (e.g., [IEC61508]) set requirements on how a test process should be conducted. In addition, there are a number of domain- and methodology-independent testing-relevant standards (e.g., [ISO29119]), to which UTP 2 can define integration points. In the following section, the specification describes some of these standards and discusses how they can be integrated with UTP 2.

ISO/IEC/IEEE 29119 Software Testing Standard

The ISO/IEC/IEEE 29119 Software Testing Standard is a family of standards for software testing, which consists of five parts:

- Concepts and definitions
- Test processes
- Test documentation
- Test techniques
- Keyword-driven testing

[ISO29119] is a conceptual standard, in the sense that it does not define technical solutions, specific languages or methodologies, in contrast to UTP 2. Instead, [ISO29119] standardizes a number of concepts and definitions, some of which have been adopted by UTP 2. [ISO29119]-2 specifies the structure of test processes and distinguishes different levels for test processes: organizational, test management and dynamic test processes. The first two processes deal with management-related aspects of test processes, and the dynamic test process is mainly about deriving test cases, implementing and executing test cases and evaluating executed test cases.

UTP 2 is designed to support the dynamic test process. That means, it provides concepts that enable the derivation/generation, specification, visualization and documentation of test artifacts such as test cases, data, test configurations, test sets and test contexts. Furthermore, UTP 2 provides necessary concepts to generate [ISO29119]-3-compliant test reports and documentations out of a UTP 2 model.

A set of standardized test design techniques, such as equivalence partitioning or state-based testing, has been adopted in [ISO29119]-4 made technically explicit as part of the UTP 2 language. Test engineers can utilize UTP 2 to specify test design techniques to be applied on a certain test design input (e.g., a description of the intended behavior of the test item, which is represented as a state machine or interaction). In addition to these standardized test design techniques, test engineers may define additional test design techniques if required.

The relation to [ISO29119]-5, which deals with standardizing the concepts of the keyword-driven testing paradigm, is of an implicit nature. UTP 2 can be effectively employed to setup and drive keyword-driving testing approaches. For further information on the relation of UTP 2 to keyword-driven testing see section Relation to keyword-driven testing.

ISTQB and its glossary

The ISQTB [ISTQB] and its glossary defines a set of globally standardized terminologies and definitions of testing-related concepts. The ISTQB nomenclature was deemed equally important for the definition of UTP 2 concepts as the [ISO29119] definitions. Hence, UTP 2 adopted a set of definitions, terminologies and even test design techniques from the ISTQB glossary and syllabi.

To keep the analogy with [ISO29119], UTP 2 is designed to support activities of test analysis and test design of the ISTQB fundamental test process. Test implementation and test execution are supported rather indirectly by means of arbitration specifications, precise semantics of test actions and the definition of test execution schedules.

Test evaluation activities are supported by means of the test logging capability of UTP 2, which enables a system-independent representation of a test execution. For example, UTP 2 test logs can be exploited for metrics calculations or supporting other analysis.

ETSI Testing and Test Control Notation 3 (TTCN-3)

ETSI TTCN-3 [ES20187301] standardizes a test programming language and architecture of a test execution system. It enables a platform-independent implementation of executable test cases. As such, it provides test engineers a set of language features that has been proven efficient in the development of large and complex test suites for software-intensive systems of various domains, including telecommunication, transportation, and automotive airborne software. In addition, TTCN-3 provides concepts that address reusability and simplicity in the specification of large test suites, such as using wildcard values to ease the definition of expected responses from the test item.

UTP 2, as a successor of UTP 1, is influenced by the capabilities of TTCN-3. UTP 2 adopts some TTCN-3 concepts such as test components, test configurations and test actions. Moreover, some of the TTCN-3 wildcards definitions (e.g., regular expression, any value) have been adopted by UTP.

Although UTP 2 defines test cases (due to being dependent on UML) at a much higher level of abstraction than TTCN-3, it is possible (and has been done in numerous approaches) to generate TTCN-3 modules from UTP 2 test models.

ETSI Test Description Language (TDL)

The Test Description Language (TDL) standardized by ETSI ([ES20311901], [ES20311902],[ES20311903], [ES20311904]) is a MOF-based graphical modeling language for describing test scenarios (not test cases) by a similar notation to Message sequence Charts (MSC) or UML sequence Diagrams (SD). TDL represents the next generation of testing languages in the ETSI testing technology stack and exploits the advantages of MBT. TDL is used primarily - but not exclusively - for functional testing. According to ETSI, TDL can bring a number of benefits, including:

- higher quality tests through better design
- easier to review by non-testing experts
- better, faster test development
- seamless integration of methodology and tools

TDL and UTP 2 share a set of common concepts such as test component, test configuration and procedural elements. This is partially due to the same origin of TDL and UTP 2: TTCN-3. In that regard the two languages are compatible. However, UTP 2 has a bigger scope than TDL, which so far mainly focuses on functional testing and the manual definition of test scenarios. UTP 2 offers several features beyond the capability of TDL, such as specifying test design techniques and application thereof onto a test design input. UTP 2 offers explicit concepts for test generation. Another feature of UTP 2 is the flexible handling of arbitration specifications. Finally, UTP 2 offers concepts to organize testing activities based on test management concepts such as test contexts, which resemble the semantics of [ISO29119] test process or test sub-process, test types, test objectives and test sets.

6.5 Relation to model-based testing

Model-Based Testing (MBT) is a testing technique that uses models of a software-intensive system under test to perform certain testing activities such as test analysis, test design and test implementation in both an automated (e.g., generation of test cases and data) and manual manner. Such a system under test is called a test item in the context of the UTP.

The UTP definition of MBT is adopted and slightly adjusted from the [ES202951] definition. "Model-based testing (MBT) is an umbrella of techniques that uses semi-formal models as engineering artifacts in order to specify and/or generate testing-relevant artifacts, such as test cases, test scripts, and reports." Other valid definitions of MBT are:

- "Testing based on or involving models" ([ISTQB], Glossary)
- "An umbrella of techniques that generates tests from models" [ES202951]

MBT has been thoroughly investigated in the academic literature and has also been of great interest in a variety of industry domains [UPL2012], [UL2007]. The idea of MBT is to utilize models (so called test models in the context of UTP 2) that represent the expected behavior of the test item or test cases of the test item at a higher level of abstraction. Such abstraction enables test engineers to focus exclusively on the logical aspects of the test item, instead of being bothered by technical details of the eventual implementation. Low level details of test cases, for example, syntactical details of a scripting language or completeness of data, can be taken care of by domain specific generators eventually producing executable test cases, which can finally be executed against the test item.

UTP 2 is an industrial standard that dedicatedly supports MBT by relying on UML. UTP covers a variety of concepts that are deemed mandatory such test case, data, and Arbitration & verdict. It also dedicatedly and exclusively defines concepts to govern the derivation of test-relevant information (such as test cases, data etc.) by means of test directives and test design techniques. Additionally, it also provides a few test management-related concepts that are required for defining complete test specification documents (compatible with [ISO29119]) such as test contexts (called test process/test sub-process in [ISO29119]), test level, test type and test logs.

UTP 2 is agnostic of any MBT methodology, and thus, supports a variety of MBT approaches. Some of the key aspects include: 1) Modeling test cases for a test item using stereotypes from the profile; 2) Modeling the expected behavior of the test item for test derivation using stereotypes from the profile; 3) Modeling test case specifications in domain specific languages implementing UTP.

Based on the philosophy of (test) modeling, UTP allows creating test models at various levels of abstraction ranging from test models that have no concrete data, test models that have some data, and test models that have all concrete data available.

6.6 Relation to keyword-driven testing

Keyword-driven testing (KDT) is an industrial de-facto standard that is suitable for both manual and automated test execution. KDT methodologies define logical functions that can be performed on the test item in an implementation-independent format (i.e., keyword) at a higher level of abstraction. Keywords are used to design so called keyword test cases (see [ISO29119]-5). In order to execute the keyword test cases against the test item, it is required that implementations of the keywords can be executed by a keyword-based test execution system. Keyword implementations are usually organized in a test library. The keyword-based test execution system is responsible to establish a connection between the keyword implementations and the actual implementation of the test item, run keyword test cases, and execute the keyword implementations against the actual implementation of the test item.

In the literature, there exist a number of keyword-driven testing frameworks. For example, Tang et al. [TCM2008] proposed a keyword-driven testing framework to transform keyword-based test cases into different kinds of test scripts. Hametner et al. [HWT2012] proposed a keyword-driven testing approach to specify keyword test cases in a high abstraction level, as tabular format using predefined keywords, and automatically generated executable test cases from the keyword test case. There are a number of commercial and open source tools available for KDT.

UTP 2 is defined to facilitate MBT but it does not explicitly cope with the design and implementation of test execution systems. However, UTP 2 defines concepts such as, abstract test cases and data specification explicitly to enable automated generation of concrete test cases and data from abstract ones. This idea conforms to the idea of KDT in terms of raising the level of abstractions by defining keyword test cases.

Keywords can be represented by numerous concepts of the underlying UML within UTP 2. For example, Operations of Interfaces may be interpreted as the logical functions that can be performed on the test item. Additionally, UTP 2 can be used to define or generate test cases that are based on these UML-based keyword representations. UML behaviors such as Activities or Interactions are suitable means to represent keywords in test cases in UTP 2, which are eventually exported into the keyword format required by the utilized keyword-based test execution system. As such, UTP 2 is suitable to be used as a standardized and visual language for keywords and keyword test cases.

UTP 2 could even go one step further. Due to the fact that UTP 2 is based on UML, it is even possible to provide an executable specification of the test library (i.e., the implementation of a keyword) by means of other standards such as fUML.

As a summary, UTP 2 can be efficiently leveraged as the language for the (automated or manual) design, visualization, documentation and communication of keywords, keyword test cases and even implementations thereof.

6.7 Relation to the MARTE Profile

Modeling and Analysis of Real-Time and Embedded Systems (MARTE) is a UML profile that is specifically designed for modelling and supporting analyses (e.g., performance and schedulability) for real-time and embedded systems. MARTE is developed to replace its predecessor UML profile, i.e., the UML profile for the Schedulability, Performance, and Time specification (SPTP).

At a very high level, the MARTE profile is organized into four main packages: MARTE foundations, MARTE design model, MARTE analysis model, and MARTE annexes including: MARTE model libraries, Value Specification Language, and Repetitive Structure Modeling. Out of these four packages MARTE analysis model is outside the scope of UTP since it doesn't aim to support analyses such as performance and schedulability but rather focuses on the test case generation. Nonetheless, UTP may be used for supporting model-based performance and schedulability testing and such modelling can be supported with MARTE foundation package on which MARTE analysis model relies on.

The most relevant packages for UTP from MARTE include Non-Functional Properties Modeling (NFP), Time Modelling (Time), and MARTE Library. The NFP package provides a generic framework for modelling NFPs using UML modeling elements. The package defines stereotypes such as «Nfp» to define new NFPs for a particular application and «Unit» for defining new measurement units by extending the existing ones provided in the MARTE model library such as TimeUnitKind and PowerUnitKind. Notice that NFPs defined in MARTE can be used together with UTP to support test case generation.

The Time package is specifically designed for modelling time and its related concepts specifically for real-time and embedded systems. Since Time and behavior are tightly coupled, MARTE's Time modelling can be used in conjunction with the UTP for supporting model-based testing of real-time embedded software/system with a focus on time behavior. The extensive model library of MARTE provides extended basic data types such as Real and DateTime and a rich collection of operations on them. In addition, it also provides a wide variety of measurement units such as TimeUnitKind and LengthUnitKind, general data types such as IntegerVector and IntegerInterval, predefined data types such as NFP_Percentage and NFP_DataSize and TimeLibrary supporting modelling such as logical and ideal clocks. These types can be used for modelling test items and test components that require extended data types rather than the basic data types supported by the UML. In addition, the modelling support for a variety of clocks, i.e., logical and ideal clocks, can be used for modelling complex time behavior of test items and test components.

6.8 Acknowledgements

The following OMG member organizations submitted this specification (in alphabetic order):

- Fraunhofer FOKUS, Germany.
- SOFTEAM, France.

The following OMG and non-OMG member organizations supported this specification (in alphabetic order):

- PTC Inc., United Kingdom and USA.
- Hamburg University of Applied Science, Germany.
- KnowGravity Inc., Switzerland.
- Grand Software Testing, USA.
- SELEX ES, Italy.
- Simula Research Lab, Norway.

Special Acknowledgments

The following persons were members of the core teams that contributed to the content of this document (in alphabetic order):

- Shaukat Ali, shaukat@simula.no
- Alessandra Bagnato, alessandra.bagnato@softeam.fr
- Etienne Brosse, etienne.brosse@softeam.fr
- Gabriella Carrozza, gcarrozza@sesm.it
- Zhen Ru Dai, dai@informatik.haw-hamburg.de
- Rolf Gubser, rolf.gubser@knowgravity.com
- Jon D. Hagar, embedded@ecentral.com
- Andreas Hoffmann, andreas.hoffmann@fokus.fraunhofer.de
- Andreas Korff, akorff@ptc.com
- Markus Schacher, markus.schacher@knowgravity.com
- Ina Schieferdecker, ina.schieferdecker@fokus.fraunhofer.de
- Marc-Florian Wendland, marc-florian.wendland@fokus.fraunhofer.de
- Tao Yue, tao@simula.no

7 (Informative) Conceptual Model

This section is ***informative***, i.e. non-normative and not relevant for actual profile implementations. However, it is included here to help the reader to get a better understanding of the concepts behind UTP 2. This section illustrates some of the semantics for the concepts defined in this document by means of a pragmatic application of the OMG specification "Semantics of Business Rules and Vocabularies" [SBVR]. This pragmatic application of SBVR includes the following:

- A number of concept diagrams visualize the concepts as well as their interrelationships (in SBVR called "verb concepts") organized around different subject areas. Furthermore, any SBVR definitional rule related to the concepts shown is also visualized on the diagram.
- For each concept diagram, the rule statements of each definitional rule shown are listed. The styling of those rule statements is simplified compared to [SBVR] in the sense that no colors/formatting is used. The only styling that is shown is that concepts defined within the document are shown underlined and represent an intra-document hyperlink.
- For each concept diagram, the semantics of each concept shown on the diagram is defined, usually by means of an intensional definition as suggested by [ISO1087-1]. Here underlined words also represent hyperlinks to the mentioned concepts. When defined, additional properties of concepts such as synonyms, examples, generalizations, specialization, etc. are also listed. Furthermore, for each concept the source of its definition is specified.

7.1 Test Planning

7.1.1 Test Analysis

7.1.1.1 Test Context Overview

The following concept diagram represents important semantic aspects of test context and associated other concepts such as test set, test case, data and test design input.

A test context is defined as a hub for information that specifies test type, test level, prescribes test design technique, and refers to data, data pool, test design input, arbitration specification, test set and test case. A test context also refers to other important test model elements, such as the set of test cases, data and the test design input. A test context also provides information for test management, where planning and strategies for the test are defined.

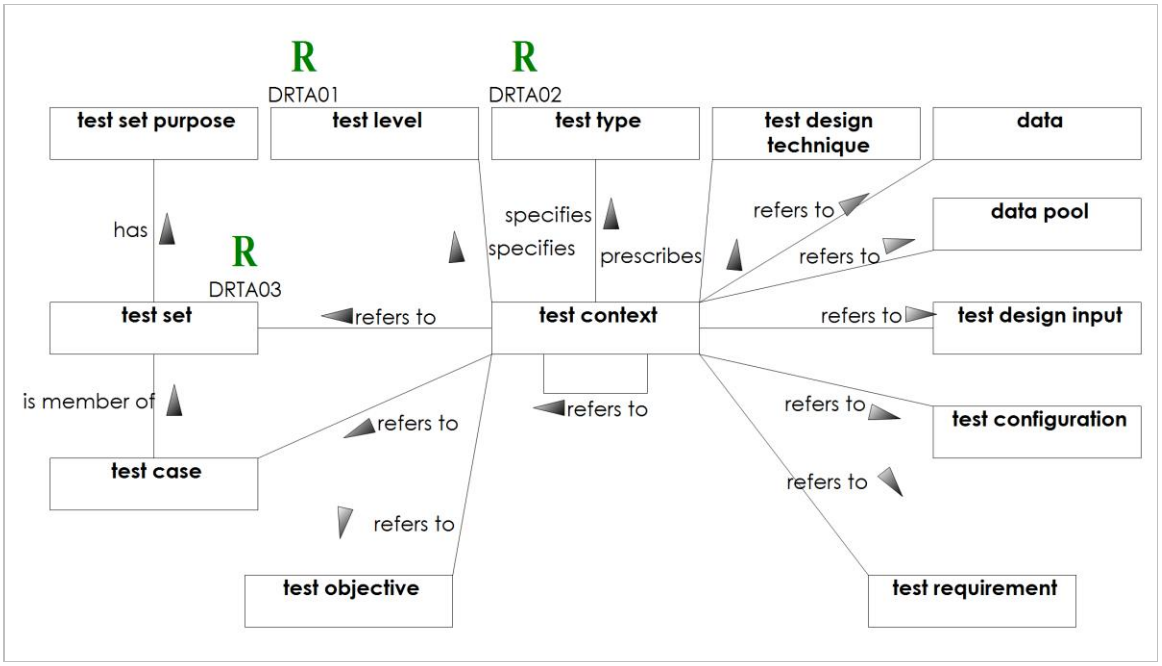


Figure 3 - Test Context Overview

Definitional Rules shown on "Test Context Overview"

Name	Rule statement
DRTA01	It is necessary that each test context specifies at most one test level.
DRTA02	It is necessary that each test context specifies at most one test type.
DRTA03	It is necessary that each test set refers to at most one arbitration specification.

Table 3 - Structural rules shown on Test Context Overview

7.1.1.2 Test Requirement and Test Objective Overview

The following concept diagram represents important semantic aspects of test objectives and test requirements and how they relate to requirements on a system to be tested.

A test requirement is designed to meet test objectives and test context specifies test objectives. A test case is designed to meet one or more test objectives and thus the test case must satisfy the associated test requirements of test objectives. In other words, a test objective specifies the goal of a test case and is defined for a certain test context. A test objective is realized by test requirement and implemented by test cases.

The diagram below also shows how test requirements are related to concepts in [SysML]. A test requirement refers to system specification item and associated with requirements of the system. A requirement is further specialized into functional requirement and non-functional requirement.

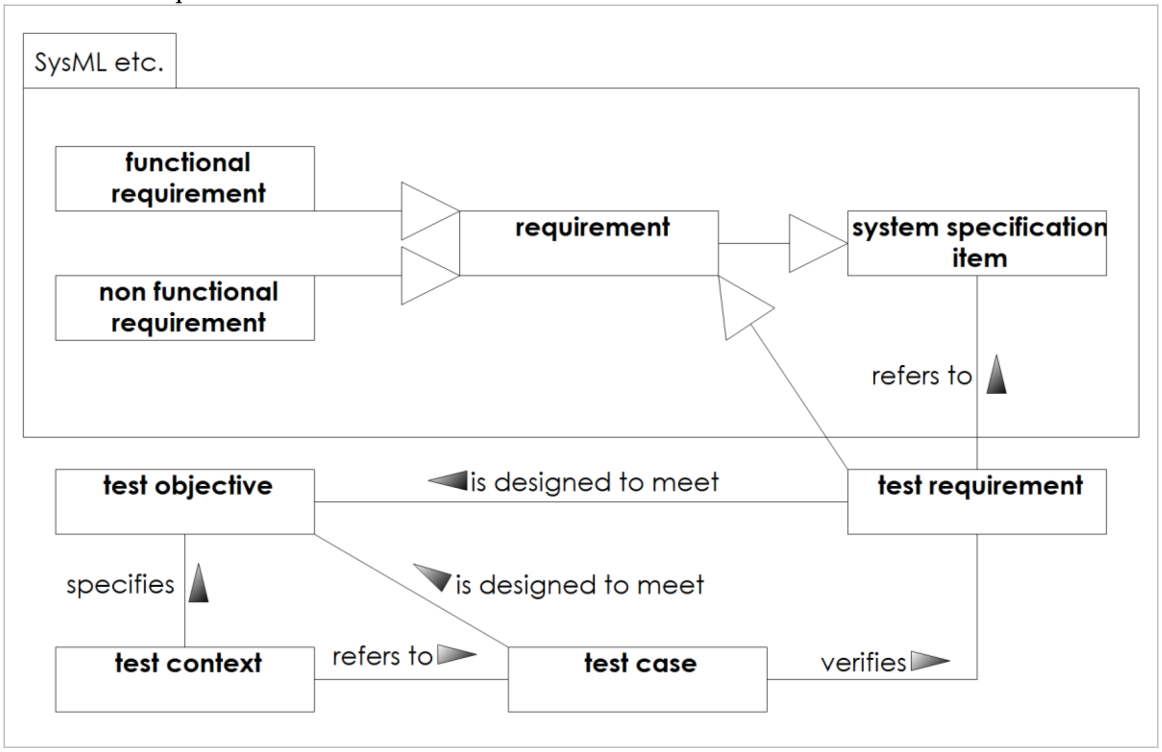


Figure 4 - Test Requirement and Test Objective Overview

7.1.1.3 Concept Descriptions

test context	
Definition	A set of information that is prescriptive for testing activities that are organized and managed together for deriving or selecting test objectives, test design techniques, test design inputs and eventually test cases.
Examples	acceptance test, smoke test, system test, ...
Source	UTP 2 WG

test level	
Definition	A specification of the boundary of a test item that must be addressed by a specific test context.
Examples	integration test, system test, component test, ...
Source	UTP 2 WG

test objective	
Definition	A desired effect that a test or set of test cases intends to achieve.
Examples	• Provision of information about the qualities of the product to a certification authority or other stakeholders • Provision of information that the product has met stakeholder expectations • Provision of information that requirements of a product are fulfilled (i.e. regulatory, design, contractual, etc.)
Source	UTP 2 WG

test requirement	
Definition	A desired property on a test case or test set, referring to some aspect of the test item to be tested.
Synonyms	test condition
Examples	• Test case must ensure 80% path coverage of use case XY. • Test case must check that an IPv6 multicast message is carried out over a GeoBroadcast message into the correct geographical area, with a GVL manually configured.
Source	UTP 2 WG
Is a	requirement

test set	
Definition	A set of test cases that share some common purpose.
Source	UTP 2 WG

test set purpose	
Definition	A statement that explains the rationale for grouping test cases together.
Source	UTP 2 WG

test type	
Definition	A quality attribute of a test item that must be addressed by a specific test context.
Examples	functionality test, usability test, conformance test, interoperability test, performance test, ...
Source	UTP 2 WG

7.1.2 Test Design

7.1.2.1 Test Design Facility Overview

The following diagram summarizes the concepts of UTP 2 test design facility. The test design facility enables the specification of test design techniques that must be applied on a test design input in order to derive test artifacts such as test sets, test cases, test configurations, required data or test execution schedules. Whether the test derivation process according to the specified test design techniques is carried out manually or automatically does not matter whatsoever. Such test design techniques are assembled and governed by a test design directive. Thus, the test design directive is a specification of the capabilities a test designing entity (e.g. a human tester or test generator) must offer in order to perform the derivation activities according to the assembled test design techniques. The UTP 2 test design facility is agnostic of any implementation- or tool-specific details and simply offers the ability to describe, select and extend the set of potentially available and applicable test design techniques.

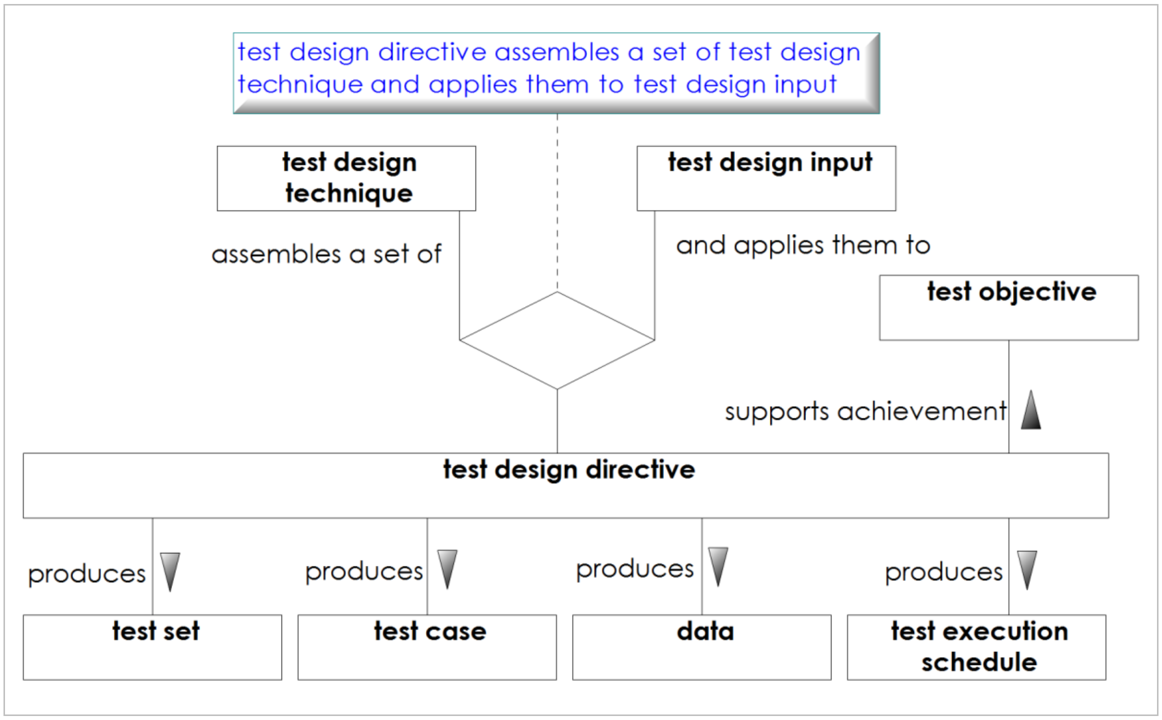


Figure 5 - Test Design Facility Overview

7.1.2.2 Concept Descriptions

test design directive	
Definition	A test design directive is an instruction for a test designing entity to derive test artifacts such as test sets, test cases, test configurations, data or test execution schedules by applying test design techniques on a test design input. The set of assembled test design techniques are referred to as the capabilities a test designing entity must possess in order to carry out the test design directive, regardless whether it is carried out by a human tester or a test generator. A test design directive is a means to support the achievement of a test objective.
Source	UTP 2 WG

test design input	
Definition	Any piece of information that must or has been used to derive testing artifacts such as test cases, test configuration, and data.
Examples	a state machine specifying some expected behavior of the test item used to derive some test cases, a requirements catalog used to derive some test cases, ...

Source	UTP 2 WG
Is a	model

test design technique	
Definition	A specification of a method used to derive or select test configurations, test cases and data. test design techniques are governed by a test design directive and applied to a test design input. Such test design techniques can be monolithically applied or in combination with other test design techniques. Each test design technique has clear semantics with respect to the test design input and the artifacts it derives from the test design input.
Examples	equivalence testing, structural coverage,
Source	UTP 2 WG

7.2 Test Architecture

7.2.1 Test Architecture Overview

The following concept diagram represents important semantic aspects in the context of test configuration and associated other concepts such as test component, test items and test cases. A test case relies on at least one test configuration to execute. A test configuration specifies how the test item and test components are interconnected and what configuration data are needed. Configuration data are specified as part of the test item configuration and test component configuration for the test item and each test component.

We explicitly classify test configuration into two categories: abstract test configuration and concrete test configuration such that enabling the generation of concrete test configurations from an abstract test configuration would be possible.

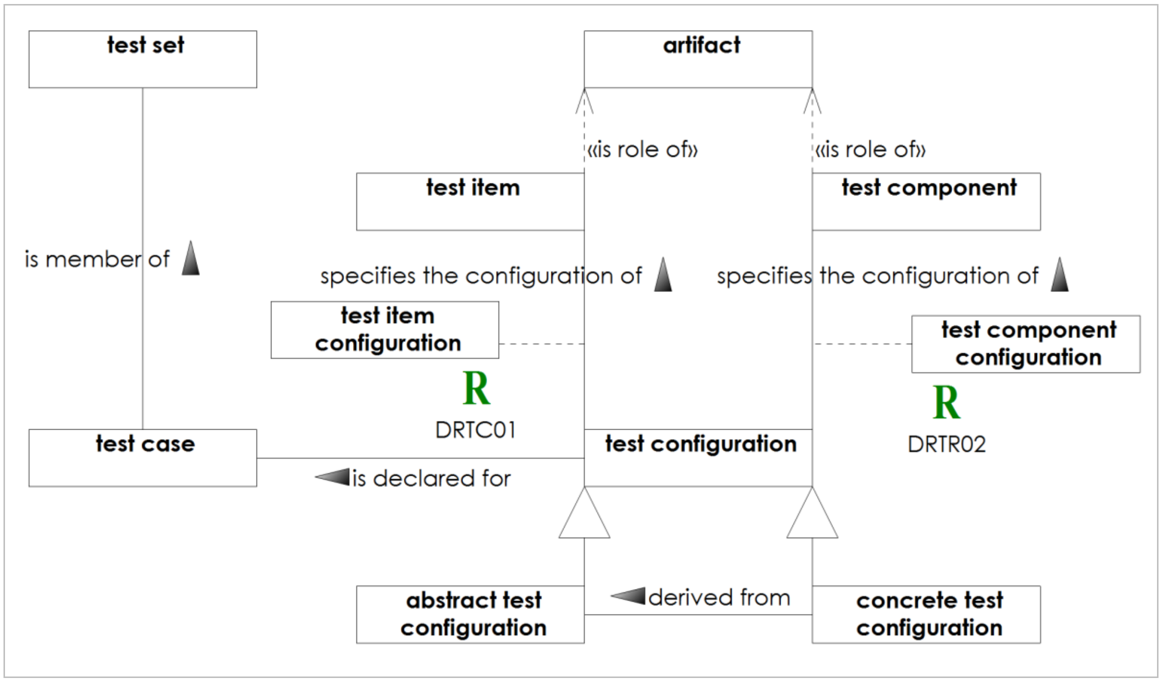


Figure 6 - Test Architecture Overview

Definitional Rules shown on "Test Architecture Overview"

Name	Rule statement
DRTC01	It is necessary that each test item configuration specifies the configuration of at least one test item.
DRTR02	It is necessary that each test component configuration specifies the configuration of at least one test component.

Table 4 - Structural rules shown on Test Architecture Overview

7.2.2 Concept Descriptions

abstract test configuration	
Definition	A test configuration that specifies the test item, test components and their interconnections as well as configuration data that should be abstract test data.
Source	UTP 2 WG
Is a	test configuration

artifact	
Definition	An object produced or modified during the execution of a

	process.
Synonyms	work product
Examples	• Software XY. • Software Requirements Specification. • Coffee machine. • Coffee bean.
Source	UTP 2 WG

concrete test configuration	
Definition	A test configuration that specifies the test item, test components and their interconnections as well as configuration data that should be concrete data.
Source	UTP 2 WG
Is a	test configuration

test component	
Definition	A role of an artifact within a test configuration that is required to perform a test case.
Examples	• A test driver • A test stub • Coffee machine that grinds the coffee beans to be tested.
Source	UTP 2 WG
Sub categories	data provider
Is role of	artifact

test component configuration	
Definition	A set of configuration options offered by an artifact in the role of a test component chosen to meet the requirements of a particular test configuration.
Source	UTP 2 WG

test configuration	
Definition	A specification of the test item and test components, their interconnection as well as their configuration data.
Source	UTP 2 WG
Sub categories	• abstract test configuration • concrete test configuration

test item	
Definition	A role of an artifact that is the object of testing within a test configuration.
Synonyms	System Under Test, SUT
Examples	• Software XY to be tested. • Software Requirements Specification to be reviewed. • Coffee machine to be tested. • Coffee beans to be tested.
Abbreviation	SUT
Source	UTP 2 WG
Is role of	artifact

test item configuration	
Definition	A set of configuration options offered by an artifact in the role of a test item chosen to meet the requirements of a particular test configuration.
Source	UTP 2 WG

7.3 Test Behavior

7.3.1 Test Cases

7.3.1.1 Test Case Overview

The following concept diagram represents important semantic aspects in the context of what a test case is and what its components are. A test case invokes a procedure describing the execution order of individual test actions (not shown here, see Test Procedures and Test-specific Actions for details). A test case is specialized into abstract test case and concrete test case depending on the availability of data. If all the data required for a test case is available, it is classified as a concrete test case and abstract test case otherwise.

As shown in Test Context Overview, test cases may be grouped into test sets. A test execution schedule prescribes execution order of this set of test cases. All, test cases, test procedure, and test execution schedule may require a preconditon and may guarantee a postcondition, each of which play the role of boolean expression.

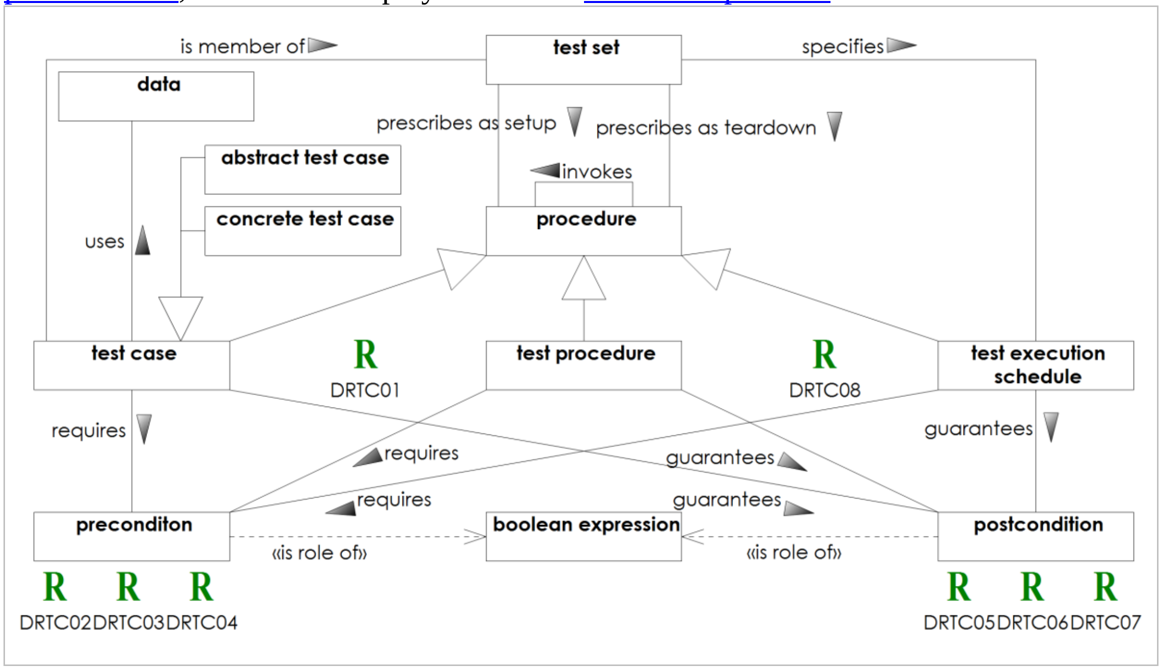


Figure 7 - Test Case Overview

Definitional Rules shown on "Test Case Overview"

Name	Rule statement
DRTC01	It is necessary that each test case invokes at least one test procedure.
DRTC02	It is necessary that each test execution schedule requires at most one preconditon.
DRTC03	It is necessary that each test case requires at most one preconditon.
DRTC04	It is necessary that each test procedure requires at most one preconditon.
DRTC05	It is necessary that each test execution schedule guarantees at most one postcondition.
DRTC06	It is necessary that each test case guarantees at most one postcondition.
DRTC07	It is necessary that each test procedure guarantees at most one postcondition.

Name	Rule statement
DRTC08	It is impossible that a test execution schedule invokes a test procedure.

Table 5 - Structural rules shown on Test Case Overview

7.3.1.2 Concept Descriptions

abstract test case	
Definition	A test case that declares at least one formal parameter.
Source	UTP 2 WG
Is a	test case

boolean expression	
Definition	An expression that may be evaluated to either of these values: "TRUE" or "FALSE".
Synonyms	predicate
Source	UTP 2 WG

concrete test case	
Definition	A test case that declares no formal parameter.
Source	UTP 2 WG
Is a	test case

postcondition	
Definition	A boolean expression that is guaranteed to be True after a test case execution has been completed.
Source	UTP 2 WG
Is role of	boolean expression

preconditon	
Definition	A boolean expression that must be met before a test case may be executed.

Source	UTP 2 WG
Is role of	boolean expression

test case	
Definition	A procedure that includes a set of preconditions, inputs and expected results, developed to drive the examination of a test item with respect to some test objectives.
Source	UTP 2 WG
Is a	procedure
Sub categories	• abstract test case • concrete test case

test execution schedule	
Definition	A procedure that constrains the execution order of a number of test cases.
Source	UTP 2 WG
Is a	procedure

7.3.2 Test-specific Procedures

7.3.2.1 Test Procedures

The following concept diagram represents important semantic aspects of procedures as they are used in UTP. UTP distinguishes three different types of procedures: test execution schedules, test cases and test procedures, which are all special forms of procedures. In general, procedures may invoke other procedures. Furthermore, all procedures may declare one or more formal parameters which are replaced by actual parameters upon procedure invocation.

A procedure prescribes the execution order of a set of procedural elements, which are either atomic procedural elements (such as procedure invocations or individual test actions) or compound procedural elements. A compound procedural element is a container that groups a set of procedural elements into sequences, loops, and other control structures.

Any procedural element may be constrained by time which is expressed by its possible fact statements of time points and durations. A procedural element may be constrained on when it is to be performed as well as how long it is to be performed by the tester.

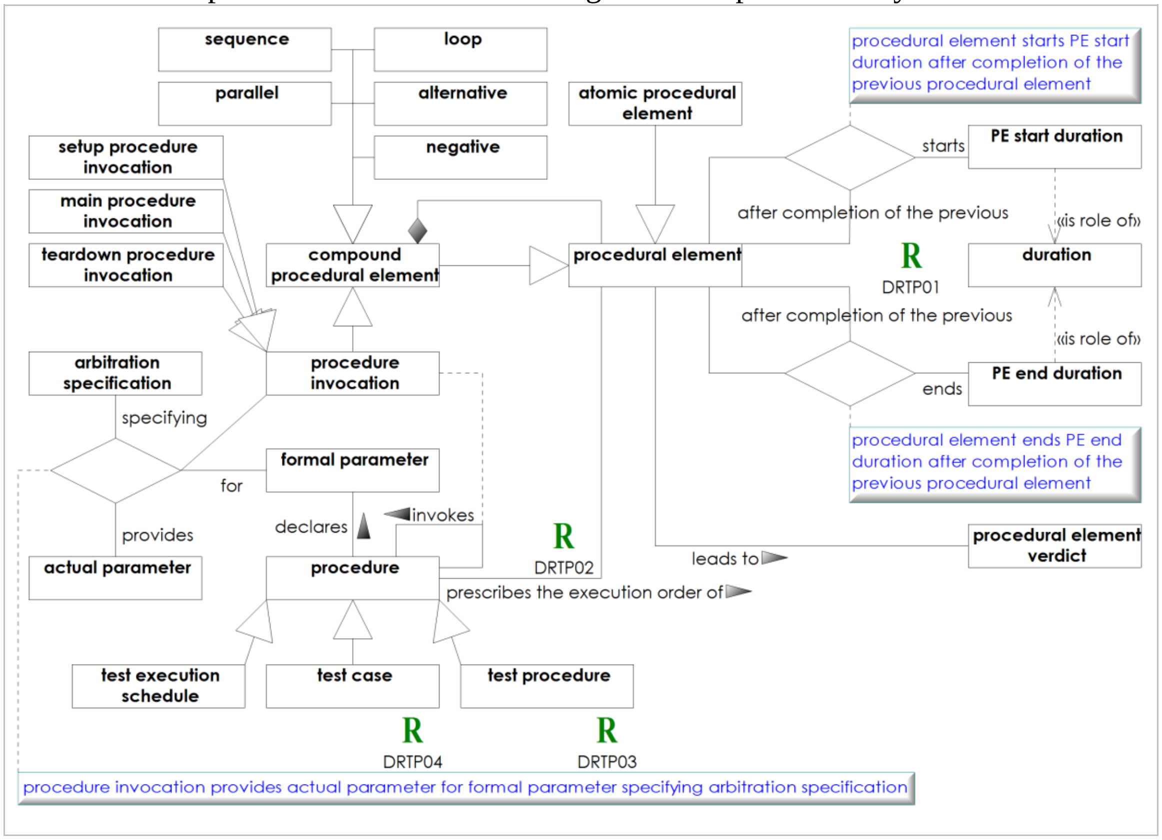


Figure 8 - Test Procedures

Definitional Rules shown on "Test Procedures"

Name	Rule statement
DRTP01	It is necessary that the PE start duration of a procedural element is smaller than the PE end duration of the same procedural element.
DRTP02	It is necessary that each procedure prescribes the execution order of at least one procedural element.
DRTP03	It is necessary that each test procedure prescribes the execution order of at least one test action.
DRTP04	It is necessary that each test case invokes at least one test procedure as a main procedure invocation.

Table 6 - Structural rules shown on Test Procedures

7.3.2.2 Concept Descriptions

actual parameter	
Definition	A concrete value that is passed over to the procedure and re-

	places the formal parameter with its concrete value.
Source	UTP 2 WG

alternative	
Definition	A compound procedural element that executes only a subset of its contained procedural elements based on the evaluation of a boolean expression.
Source	UTP 2 WG
Is a	compound procedural element

atomic procedural element	
Definition	A procedural element that cannot be further decomposed.
Source	UTP 2 WG
Is a	procedural element
Sub categories	test action

compound procedural element	
Definition	A procedural element that can be further decomposed.
Source	UTP 2 WG
Is a	procedural element
Sub categories	• alternative • loop • negative • parallel • procedure invocation • sequence

duration	
Definition	The duration from the start of a test action until its completion.
Source	UTP 2 WG

Is a	duration

formal parameter	
Definition	A placeholder within a procedure that allows for execution of the procedure with different formal parameters that are provided by the procedure invocation.
Source	UTP 2 WG

loop	
Definition	A compound procedural element that repeats the execution of its contained procedural elements.
Source	UTP 2 WG
Is a	compound procedural element

main procedure invocation	
Definition	A procedure invocation that is considered as the main part of a test case by the test case arbitration specification.
Source	UTP 2 WG
Is a	procedure invocation

negative	
Definition	A compound procedural element that prohibits the execution of its contained procedural elements in the specified structure.
Source	UTP 2 WG
Is a	compound procedural element

parallel	
Definition	A compound procedural element that executes its contained procedural elements in parallel to each other.
Source	UTP 2 WG
Is a	compound procedural element

PE end duration	
Definition	The duration between the end of the execution of a procedural element and the end of the execution of the subsequent procedural element.
Source	UTP 2 WG
Is role of	duration

PE start duration	
Definition	The duration between the end of the execution of a procedural element and the beginning of the execution of the subsequent procedural element.
Source	UTP 2 WG
Is role of	duration

procedural element	
Definition	An instruction to do, to observe, and/or to decide.
Source	UTP 2 WG
Sub categories	• atomic procedural element • compound procedural element

procedure	
Definition	A specification that constrains the execution order of a number of procedural elements.
Source	UTP 2 WG
Sub categories	• test case • test execution schedule • test procedure

procedure invocation	
Definition	An atomic procedural element of a procedure that invokes another procedure and wait for its completion.

Source	UTP 2 WG
Is a	compound procedural element
Sub categories	• main procedure invocation • setup procedure invocation • teardown procedure invocation

sequence	
Definition	A compound procedural element that executes its contained procedural elements sequentially.
Source	UTP 2 WG
Is a	compound procedural element

setup procedure invocation	
Definition	A procedure invocation that is considered as part of the setup by the arbitration specification and that is invoked before any main procedure invocation.
Source	UTP 2 WG
Is a	procedure invocation

teardown procedure invocation	
Definition	A procedure invocation that is considered as part of the teardown by the responsible arbitration specification and that is invoked after any main procedure invocation.
Source	UTP 2 WG
Is a	procedure invocation

test procedure	
Definition	A procedure that constrains the execution order of a number of test actions.
Source	UTP 2 WG
Is a	procedure

time point	
Definition	The time point at which a test action is initiated.
Source	UTP 2 WG
Is a	time point

7.3.3 Test-specific Actions

7.3.3.1 Overview of test-specific actions

The following concept diagram represents important semantic aspects of test actions as parts of test procedures. A test action is a specialization of an atomic procedural element and is to be interpreted as an instruction to the tester responsible for executing a test case. Any test action leads to a procedural element verdict (i.e., influences the final test case verdict).

Most test actions check certain aspects of the test item. The most important aspects of the test item are its observable behavior (i.e., its responses) and its measurable properties.

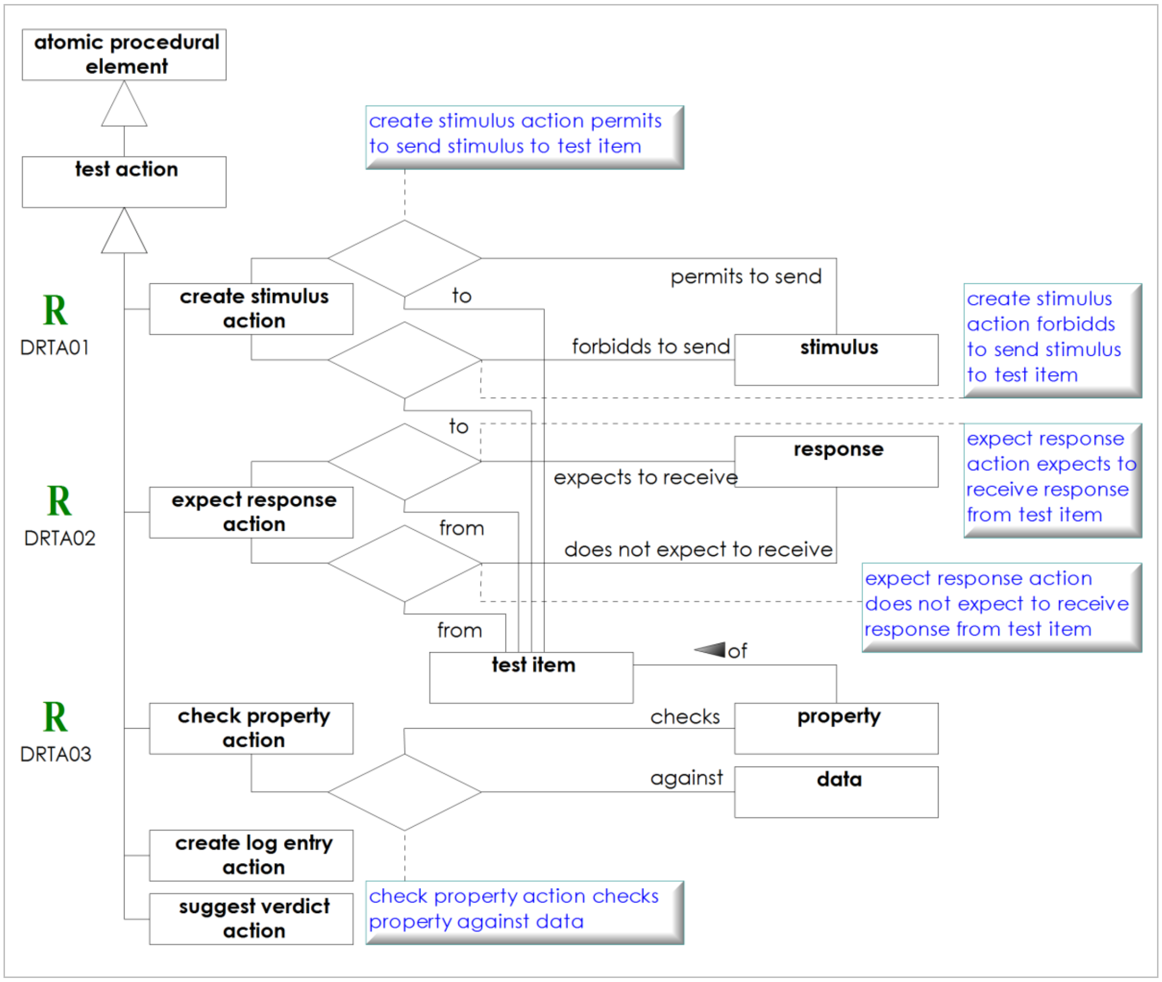


Figure 9 - Overview of test-specific actions

Definitional Rules shown on "Overview of test-specific actions"

Name	Rule statement
DRTA01	It is necessary that a create stimulus action permits to send at least one stimulus.
DRTA02	It is necessary that a expect response action expects to receive at least one response.
DRTA03	It is necessary that a check property action checks at least one property of the test item against the data.

Table 7 - Structural rules shown on Overview of test-specific actions

7.3.3.2 Concept Descriptions

check property action	
Definition	A test action that instructs the tester to check the conformance of a property of the test item and to set the procedural element verdict according to the result of this check.
Source	UTP 2 WG
Is a	test action

create log entry action	
Definition	A test action that instructs the tester to record the execution of a test action, potentially including the outcome of that test action in the test case log.
Source	UTP 2 WG
Is a	test action

create stimulus action	
Definition	A test action that instructs the tester to submit a stimulus (potentially including data) to the test item.
Source	UTP 2 WG
Is a	test action

expect response action	
Definition	A test action that instructs the tester to check the occurrence of one or more particular responses from the test item within a given time window and to set the procedural element verdict according to the result of this check.
Source	UTP 2 WG
Is a	test action

property	
Definition	A basic or essential attribute shared by all members of a class of test items.

Source	UTP 2 WG

response	
Definition	A set of data that is sent by the test item to its environment (often as a reaction to a stimulus) and that is typically used to assess the behavior of the test item.
Source	UTP 2 WG

stimulus	
Definition	A set of data that is sent to the test item by its environment (often to cause a response as a reaction) and that is typically used to control the behavior of the test item.
Source	UTP 2 WG

suggest verdict action	
Definition	A test action that instructs the tester to suggest a particular procedural element verdict to the arbitration specification of the test case for being taken into account in the final test case verdict.
Source	UTP 2 WG
Is a	test action

test action	
Definition	An atomic procedural element that is an instruction to the tester that needs to be executed as part of a test procedure of a test case within some time frame.
Synonyms	test step
Source	UTP 2 WG
Is a	atomic procedural element
Sub categories	• check property action • create log entry action • create stimulus action • expect response action

	• suggest verdict action

7.4 Test Data

7.4.1 Test Data Concepts

The following concept diagram represents important semantic aspects of test data. Test data or more generally just data may be modeled at two different levels:

- **Extensional level**: model elements that actually represent some data composed as a set of individual data items
- **Intensional level**: model elements that specify some criteria that some data must comply with, i.e. the specification of the meaning of data

At the *extensional level* data always represents a specific set of data items and is covered by concepts such as data pool, actual data pool, and data partition. The concepts data pool and actual data pool represent containers of data, the former is a logical container, the latter a physical container such as a concrete database. A data partition represents a subset of another set of data items in which all data item are conformant to a particular data specification.

In contrast, at the *intensional level* data is represented by a boolean expression that may be used to qualify data items as member of data, i.e. it represents the intended meaning of data and is covered by concepts such as data specification, data type, and constraint. A data specification is composed of a basic data type plus a set of constraints on that data type. The entire concept of a data specification may be considered as a category in the sense of "Category Theory" in mathematics (see for example [WikiCT] or [SEP2014a]). Thus, two data specifications might be interpreted as categories that are related to each other by means of different dependencies called "morphisms". These may be considered as structure-preserving maps supporting the following three informal semantics:

- A morphism of type "extension" increases the amount of data, i.e. they add more data items to a given set of data items
- A morphism of type "refinement" decreases the amount of data, i.e. they remove data items from a given set of data items
- A morphism of type "complement" inverts data, i.e. it replaces the data items of a given set of data items by their opposites.

A data provider is a test component that is able to deliver (i.e. either select and/or generate) data according to a data specification.

In the context of a test case, different places of a test case typically refer to different levels of test data

- test cases typically refer to data used as preconditions as well as data to be supplied with stimuli to be sent to the test item

- test cases typically refer to data specifications in postconditions or data returned by responses in order to determine or influence the verdict of the test case.

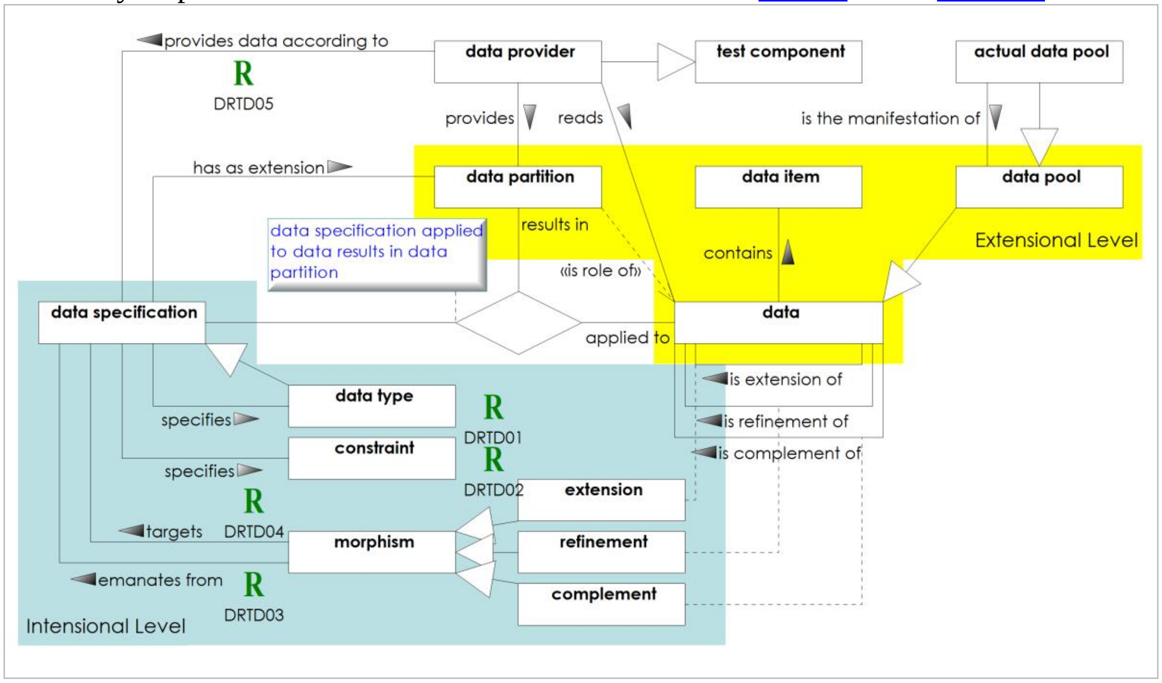


Figure 10 - Test Data Concepts

Definitional Rules shown on "Test Data Concepts"

Name	Rule statement
DRTD01	It is necessary that each data specification specifies at least one data type.
DRTD02	It is necessary that each data specification specifies at least one constraint.
DRTD03	It is necessary that a morphism emanates from exactly one data specification.
DRTD04	It is necessary that a morphism targets exactly one data specification.
DRTD05	It is necessary that each data provider provides data according to at least one data specification.

Table 8 - Structural rules shown on Test Data Concepts

7.4.2 Concept Descriptions

actual data pool	
Definition	A specification of an actual implementation of a data pool.
Examples	• the specification of the database of type "Customers"

	on disk DK13 on machine XYZ.
Source	UTP 2 WG
Is a	data pool

complement	
Definition	A morphism that inverts data)i.e., that replaces the data items of a given set of data items by their opposites).
Source	UTP 2 WG
Is a	morphism

constraint	
Definition	An assertion that indicates a restriction that must be satisfied by any valid realization of the model containing the constraint.
Source	[UML]

data	
Definition	A usually named set of data items.
Synonyms	concrete data
Examples	• 42. • "John". • "Some people": {"John", "Greg", "Barb", "Aline"} • "Example customer": Sherlock Holmes, living at Baker Street in London • The contents of a database "CUST-PRD" containing customers.
Source	UTP 2 WG
Sub categories	data pool
Is instance of	data structure

data item	
Definition	Either a value or an instance.
Source	UTP 2 WG

data partition	
Definition	A role that some data plays with respect to some other data (usually being a subset of this other data) with respect to some data specification.
Source	UTP 2 WG
Is role of	data

data pool	
Definition	Some data that is an explicit or implicit composition of other data items.
Examples	• the specification of a database type named "Customers"
Source	UTP 2 WG
Is a	data
Sub categories	actual data pool

data provider	
Definition	A test component that is able to deliver (i.e., either select and/or generate) data according to a data specification.
Source	UTP 2 WG
Is a	test component

data specification	
Definition	A named boolean expression composed of a data type and a set of constraints applicable to some data in order to determine whether or not its data items conform to this data specification.

Synonyms	abstract data
Examples	• 40...50. • "Jo(h)?n". • "odd numbers", i.e. numbers where self mod 2 = 1 • "right-angled triangles", i.e. triangles where a^2 + b^2 = c^2 • "young, German-speaking customers" i.e., customers, where language= 'German' and age < 18 • any/all/295 customers having the forename "John" and living in London.
Source	UTP 2 WG
Sub categories	data type

data type	
Definition	A type whose instances are identified only by their value.
Source	[UML]
Is a	data specification

extension	
Definition	A morphism that increases the amount of data (i.e., that adds more data items to a given set of data items).
Source	UTP 2 WG
Is a	morphism

morphism	
Definition	A structure-preserving map from one mathematical structure to another.
Source	[WikiM]
Sub categories	• complement • extension • refinement

refinement	
Definition	A morphism that decreases the amount of data (i.e., that removes data items from a given set of data items).
Source	UTP 2 WG
Is a	morphism

7.5 Test Evaluation

7.5.1 Arbitration Specifications

7.5.1.1 Arbitration & Verdict Overview

The following concept diagram represents important semantic aspects of verdicts and how they are derived.

An arbitration specification is defined as a set of rules that should be followed to determine the instance of a verdict of an executed test case. An arbitration specification should be specified for a procedure which describes the behavior of test case (test procedure) or a test execution schedule (associated to the execution of a set of test cases). An arbitration specification calculates a verdict which can be Fail, Pass, Inconclusive and None.

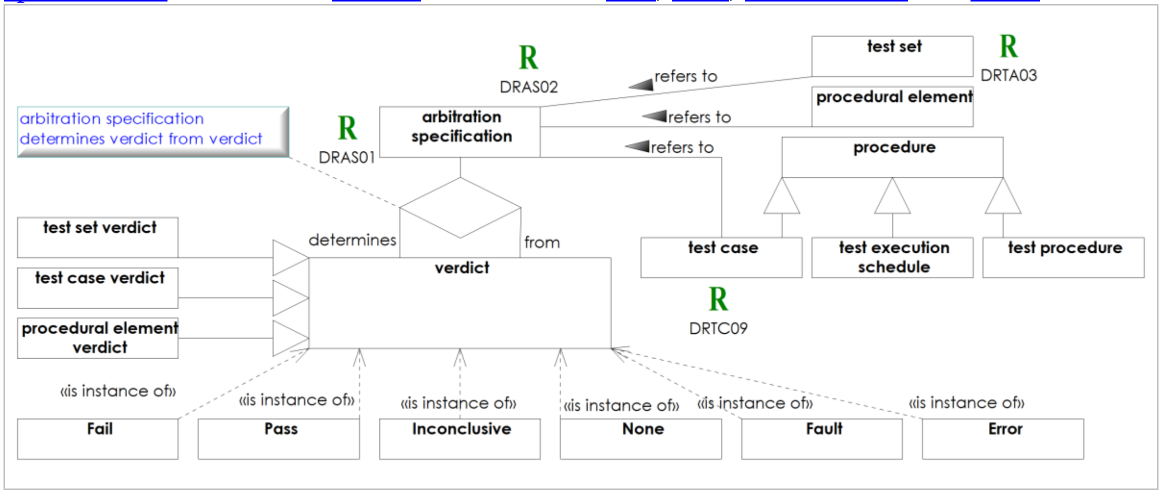


Figure 11 - Arbitration & Verdict Overview

Definitional Rules shown on "Arbitration & Verdict Overview"

Name	Rule statement
DRAS01	It is necessary that an arbitration specification determines exactly one verdict.

Name	Rule statement
DRAS02	It is necessary that a arbitration specification determines exactly one of a test set verdict, a test case verdict or a procedural element verdict.
DRTA03	It is necessary that each test set refers to at most one arbitration specification.
DRTC09	It is necessary that each test case refers to at most one arbitration specification.

Table 9 - Structural rules shown on Arbitration & Verdict Overview

7.5.1.2 Concept Descriptions

arbitration specification	
Definition	A set of rules that calculates the eventual verdict of an executed test case, test set or procedural element.
Source	UTP 2 WG

Error	
Definition	An indication that an unexpected exception has occurred while executing a specific test set, test case, or test action.
Source	UTP 2 WG
Is instance of	verdict

Fail	
Definition	A verdict that indicates that the test item did not comply with the expectations defined by a test set, test case, or test action during execution.
Source	UTP 2 WG
Is instance of	verdict

Fault	
Definition	A verdict that indicates that the tester did not comply with the procedural specifications defined by a procedure during execution of a test set, test case, or test action.
Source	UTP 2 WG

Is instance of	verdict

Inconclusive	
Definition	A verdict that indicates that the compliance of a test item against the expectations defined by a test set, test case, or test action could not be determined during execution.
Source	UTP 2 WG
Is instance of	verdict

None	
Definition	A verdict that indicates that the compliance of a test item against the expectations defined by a test set, test case, or test action has not yet been determined (i.e., it is the initial value of a verdict when a test set, test case, or test action was started).
Source	UTP 2 WG
Is instance of	verdict

Pass	
Definition	A verdict that indicates that the test item did comply with the expectations defined by a test set, test case, or test action during execution.
Source	UTP 2 WG
Is instance of	verdict

procedural element verdict	
Definition	A verdict that indicates the result (i.e., the conformance of the actual properties of the test item with its expected properties) of executing a test action on a test item.
Source	UTP 2 WG
Is a	verdict

test case verdict	
Definition	A verdict that indicates the result (i.e., the conformance of the actual properties of the test item with its expected properties) of executing a test case against a test item.
Source	UTP 2 WG
Is a	verdict

test set verdict	
Definition	A verdict that indicates the result (i.e., the conformance of the actual properties of the test item with its expected properties) of executing a test set against a test item.
Source	UTP 2 WG
Is a	verdict

verdict	
Definition	A statement that indicates the result (i.e., the conformance of the actual properties of the test item with its expected properties) of executing a test set, a test case, or a test action against a test item.
Source	UTP 2 WG
Sub categories	• procedural element verdict • test case verdict • test set verdict
Instances	• Pass • Inconclusive • None • Error • Fault • Fail

7.5.2 Test Logging

7.5.2.1 Test Log Overview

As defined by [ISTQB] a test log is “a chronological record of relevant details about the execution of tests” and as such is an important means for test evaluation and reporting activities. Thus, the purpose of the UTP 2 test logging facility is twofold:

1.) It helps establish a trace link between a test case or an entire test set and one or potentially more executions thereof. Essential information of a test log are, for example, the date and the duration when the corresponding test case was executed; the executing entity (i.e., a human tester or automated test execution system) or entities (in some domains it is not uncommon that test cases are executed over several days by potentially more than one executing entity), and finally, the test case verdict. These so called test log header information are the minimal required information in order to achieve full traceability between test objectives, test requirements, test cases/test sets and finally the execution thereof. Full traceability among those artifacts enables the computation of test metrics such as the status of test execution (how many test cases have eventually been executed at a certain point in time), coverage of requirements (not part of UTP), test requirements or test objectives, etc.

2.) It supports a deeper analysis of what was going on during the execution of a test case or test set. Since the execution of test case or test set is a transient set of test actions performed by an executing entity against the test item, the capturing of detailed information about the performed test actions in a test log is the only way for a stakeholder, usually a test analyst or test manager, to be able to comprehend what has really happened during execution without being part of the executing entities. Such a chronological record of detailed information of an executed test case or test set is in UTP 2 called test log body information. They optionally supplement the test log header information of UTP.

Since the understanding of what information is really relevant during the execution of a test case or test set heavily depends on domain- and/or project-specific requirements, UTP 2 enables the definition of user-defined test log structures that specify what information or data deemed relevant in the respective (test) context and additionally the minimal required header information mentioned above.

Representing test logs on model level contributes to a harmonized and homogeneous view on relevant test log information in the dynamic test process. Usually, a test execution toolscape comprises more than just one tool. Tools for functional testing might be complemented by specialized tools such as those for performance testing (stress, load etc.), security testing or UI testing. The test logs of such heterogeneous toolscapes are basically heterogeneous, too. Thus, a comprehensive, detailed analysis (e.g., for the calculation of metrics over tools etc.) requires access to the proprietary structures of each tool’s test log format. The UTP 2 test logging facility mitigates the heterogeneity of test logs by offering an extensible framework to describe arbitrary complex and structured test log formats. The following use cases depict the scenarios the UTP 2 test logging facility was intended to cope with:

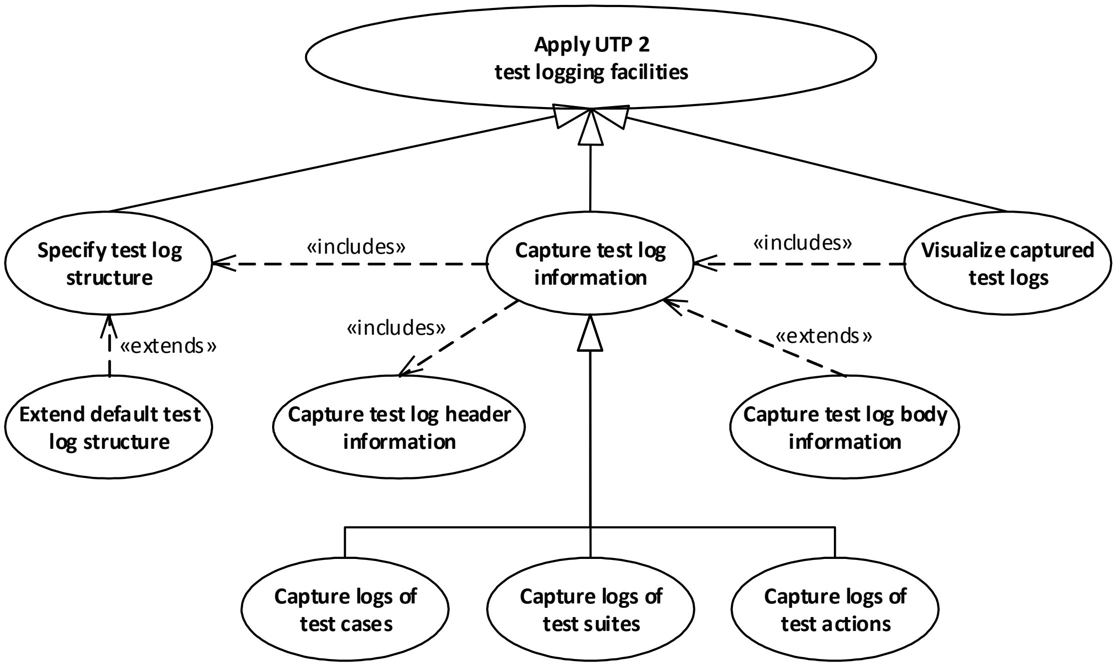


Figure 12 - Use Cases of UTP 2 test logging Facility

The use case "Specify test log structure" enables testers to specify which information is deemed relevant during the execution of in the given test process in addition to the predefined minimal required information. If no additional information is desired, the tester can rely on the implicit default test log structure. This ensures that testers can employ the UTP 2 test logging facilities immediately out of the box.

The use case "Capture test log information" is about capturing the information deemed as relevant that actually appeared during the execution of a test case, test set or even a test action in accordance with the test log structure. Incorporating the test log header information is mandatory, while representing the body part, in contrast, is optional.

The use case "Visualize captured test logs" deals with exposing the captured test log information in an appropriate representation. Since there is no common definition of the most appropriate format of test logs, UTP 2 does not prescribe how that information must be visualized. Thus, it is up to tool vendors to decide about the most appropriate and helpful visual representation(s) of captured test log information.

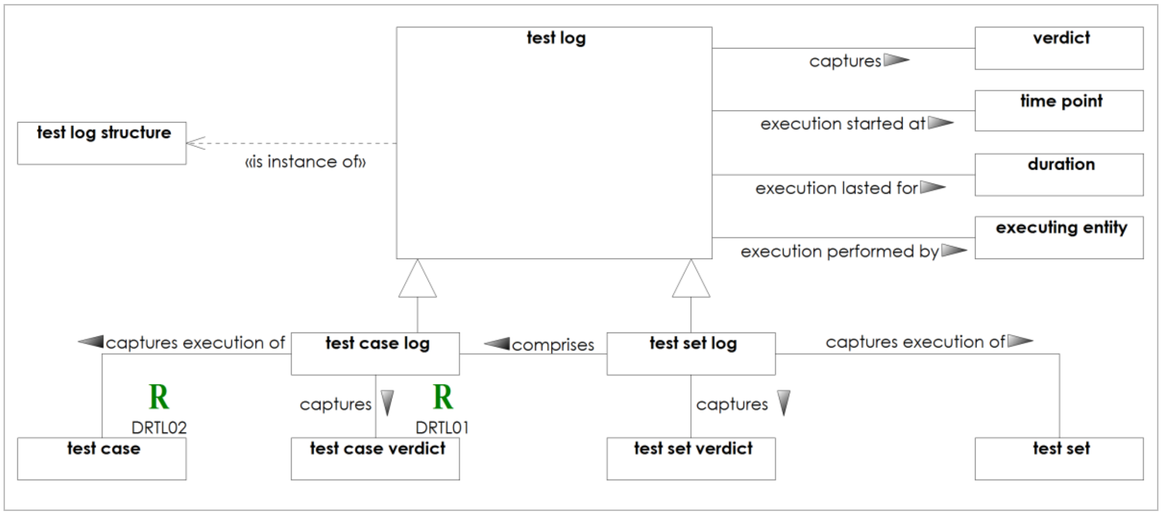


Figure 13 - Test Log Overview

Definitional Rules shown on "Test Log Overview"

Name	Rule statement
DRTL01	It is necessary that each test case log captures exactly one test case verdict.
DRTL02	It is necessary that each test case log captures execution of exactly one test case.

Table 10 - Structural rules shown on Test Log Overview

7.5.2.2 Concept Descriptions

executing entity	
Definition	An executing entity is a human being or a machine that is responsible for executing a test case or a test set.
Source	UTP 2 WG

test case log	
Definition	A test log that captures relevant information on the execution of a test case.
Source	UTP 2 WG
Is a	test log

test log	
Definition	A test log is the instance of a test log structure that captures relevant information from the execution of a test case or test set. The least required information to be logged is defined by the test log structure of the test log.
Source	UTP 2 WG
Sub categories	• test case log • test set log
Is instance of	test log structure

test log structure	
Definition	A test log structure specifies the information that is deemed relevant during execution of a test case or a test set. There is an implicit default test log structure that prescribes at least the start time point, the duration, the finally calculated verdict and the executing entity of a test case or test set execution should be logged.
Source	UTP 2 WG
Instances	test log

test set log	
Definition	A test log that captures relevant information from the execution of a test set.
Source	UTP 2 WG
Is a	test log

8 Profile Specification

This section specifies the stereotypes that are defined by the UML Testing Profile.

8.1 Language Architecture

The UML Testing Profile consists of the profile definition and three normative model libraries, which can be imported and applied if required. The profile itself is independent of these libraries, and is a self-contained package. The normative model libraries UTP Auxiliary Library and UTP Arbitration Library uses concepts from UTP and defines concepts that can be used, extended or specialized by the users.

The UTP Types Library offers helpful types and values, in particular the default verdict type and the default verdict instances. Since some of the definitions and constraints in the profile are based on predefined types, the profile imports the UTP Types Library.

The UTP Auxiliary Library offers the following concepts:

- Predefined set of quality models
- ISTQB terms for test levels and test set purposes
- Predefined test design techniques and test design technique structures.

The UTP Arbitration Library is part of the normative specification for it specifies the exact semantics of the default arbitration specifications offered by UTP 2. The semantics of the default arbitration specifications for test cases, test sets and procedural elements are given by means of xUML (i.e., State Machines). Even though the semantics of the default arbitration specifications are normative, the implementation of such a library is not mandatory for omission of any explicit arbitration specification implicitly assigns the corresponding default arbitration specification. If, however, a tool vendor provides an implementation of the default arbitration specifications, it has to abide by the semantics (not necessarily the syntax, i.e., an implementation would still comply to the standard if the default arbitration specifications are defined by means of fUML or even natural language) that is specified by the UTP 2 specification.

Overview of the technical, high-level UML Testing Profile language architecture is given next.

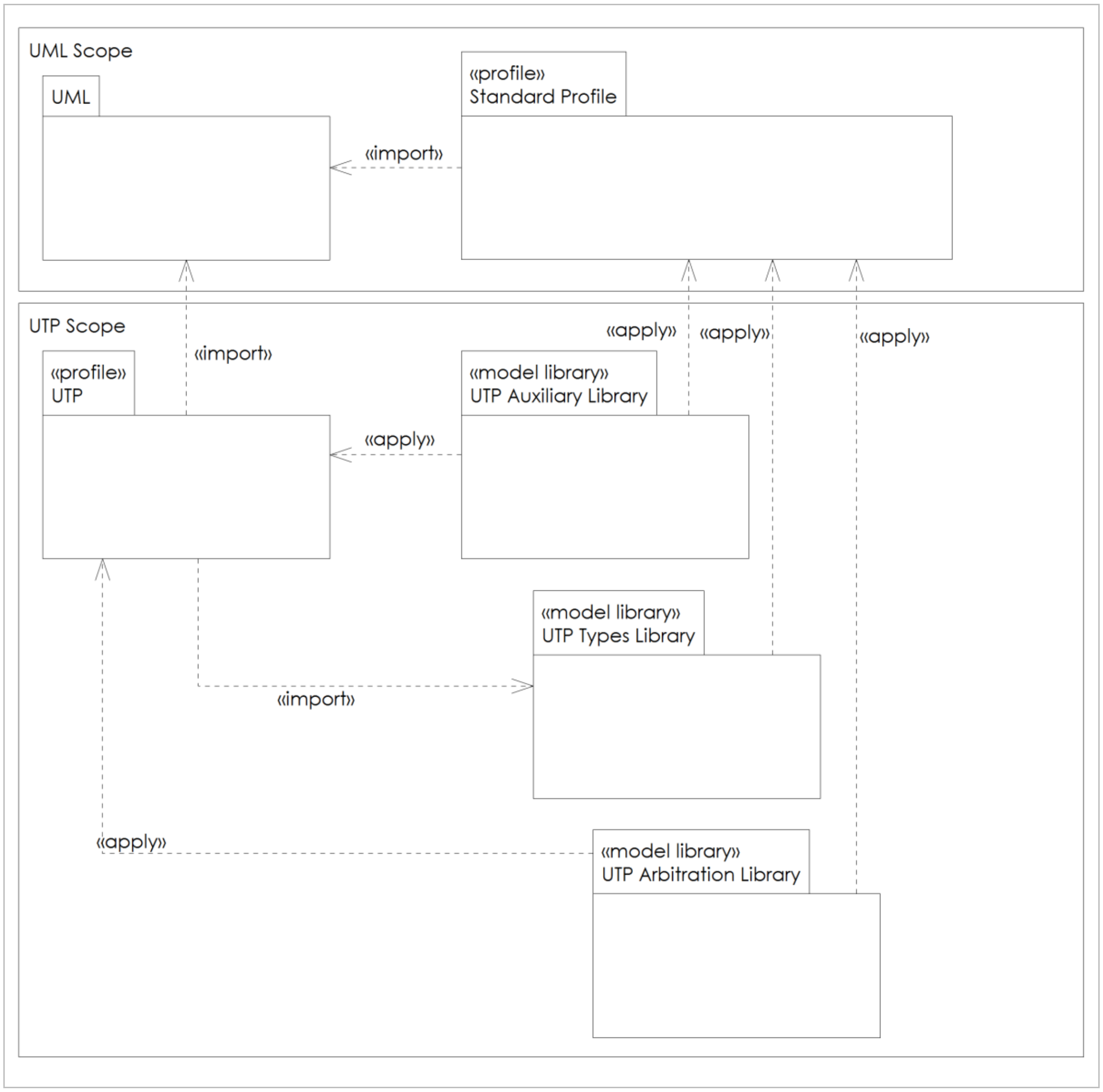


Figure 14 - Language Architecture

8.2 Profile Summary

The following table gives a brief summary on the stereotypes introduced by the UML Testing Profile 2 (listed in the second column of the table). The first column specifies the mapping to the conceptual model shown in the previous section and the third column specifies the UML 2.5 metaclasses that are extended by the stereotypes.

Stereotype	UML 2.5 Metaclasses	Concepts
Alternative	CombinedFragment, StructuredActivityNode	alternative

AlternativeArbitration-Specification	BehavioredClassifier	arbitration specification
AnyValue	Expression	data specification
ArbitrationSpecification	BehavioredClassifier	arbitration specification
AtomicProceduralEle-ment		atomic procedural ele-ment
AtomicProceduralEle-mentArbitrationSpecifi-cation	BehavioredClassifier	arbitration specification
BoundaryValueAnalysis	InstanceSpecification	test design technique
CauseEffectAnalysis	InstanceSpecification	test design technique
ChecklistBasedTesting	InstanceSpecification	test design technique
CheckPropertyAction	Constraint, ObjectFlow	check property action
CheckPropertyArbitra-tionSpecification	BehavioredClassifier	arbitration specification
Classification-TreeMethod	InstanceSpecification	test design technique
CombinatorialTesting	InstanceSpecification	test design technique
Complements	Dependency	complement
CompoundProce-duralElement	CombinedFragment, Structure-dActivityNode	compound procedural element
CompoundProce-duralElementArbitra-tionSpecification	BehavioredClassifier	arbitration specification
CreateLogEntryAction	InvocationAction	create log entry action
CreateLogEntryArbitra-tionSpecification	BehavioredClassifier	arbitration specification
CreateStimulusAction	InvocationAction, Message	create stimulus action
CreateStimulusArbitra-tionSpecification	BehavioredClassifier	arbitration specification
DataPartition	Classifier	data pool

DataPool	Classifier	data pool
DataProvider	Classifier, Property	data provider
DataSpecification	Constraint	data specification
DecisionTableTesting	InstanceSpecification	test design technique
EquivalenceClassPartitioning	InstanceSpecification	test design technique
ErrorGuessing	InstanceSpecification	test design technique
ExpectResponseAction	Message, Trigger	expect response action
ExpectResponseArbitrationSpecification	BehavioredClassifier	arbitration specification
ExperienceBasedTechnique	InstanceSpecification	test design technique
ExploratoryTesting	InstanceSpecification	test design technique
Extends	Dependency	extension
GenericTestDesignDirective	InstanceSpecification	test design directive
GenericTestDesignTechnique	InstanceSpecification	test design technique
Loop	CombinedFragment, StructuredActivityNode	loop
LoopArbitrationSpecification	BehavioredClassifier	arbitration specification
Morphing	Dependency	morphism
Negative	CombinedFragment, StructuredActivityNode	negative
NegativeArbitrationSpecification	BehavioredClassifier	arbitration specification
NSwitchCoverage	InstanceSpecification	test design technique
OpaqueProceduralElement	NamedElement	procedural element
overrides	Dependency	morphism

PairwiseTesting	InstanceSpecification	test design technique
Parallel	CombinedFragment, StructuredActivityNode	parallel
ParallelArbitrationSpecification	BehavioredClassifier	arbitration specification
ProceduralElement		procedural element
ProceduralElementArbitrationSpecification	BehavioredClassifier	arbitration specification
ProcedureInvocation	CallBehaviorAction, CombinedFragment, InteractionUse, StructuredActivityNode	procedure invocation
ProcedureInvocationArbitrationSpecification	BehavioredClassifier	arbitration specification
Refines	Dependency	refinement
RegularExpression	Expression	data specification
RoleConfiguration	Constraint	test configuration
Sequence	CombinedFragment, StructuredActivityNode	sequence
SequenceArbitrationSpecification	BehavioredClassifier	arbitration specification
StateCoverage	InstanceSpecification	test design technique
StateTransitionTechnique	InstanceSpecification	test design technique
SuggestVerdictAction	InvocationAction	suggest verdict action
SuggestVerdictArbitrationSpecification	BehavioredClassifier	arbitration specification
TestCase	Behavior, BehavioredClassifier	• test case • abstract test case • concrete test case
TestCaseArbitrationSpecification	BehavioredClassifier	arbitration specification
TestCaseLog	InstanceSpecification	test case log

TestComponent	Classifier, Property	test component
TestComponentConfiguration	Constraint	test component configuration
TestConfiguration	StructuredClassifier	test configuration
TestConfigurationRole	Classifier, Property	test configuration
TestContext	Package	test context
TestDesignDirective	InstanceSpecification	Test Design Directive
TestDesignDirectiveStructure	Classifier	test design directive
TestDesignInput	NamedElement	test design input
TestDesignTechnique	InstanceSpecification	test design technique
TestDesignTechniqueStructure	Classifier	test design technique
TestExecutionSchedule	Behavior	test execution schedule
TestItem	Classifier, Property	test item
TestItemConfiguration	Classifier, Constraint, Property	test item configuration
TestLog	InstanceSpecification	test log
TestLogStructure	Classifier	test log structure
TestLogStructureBinding	Dependency	test log structure
TestObjective	Class	test objective
TestProcedure	Behavior	test procedure
TestRequirement	Class	test requirement
TestSet	Package	test set
TestSetArbitrationSpecification	BehavioredClassifier	arbitration specification
TestSetLog	InstanceSpecification	test set log
TransitionCoverage	InstanceSpecification	test design technique
TransitionPairCoverage	InstanceSpecification	test design technique

UseCaseTesting	InstanceSpecification	test design technique
verifies	Dependency	

8.3 Test Planning

Test analysis and test design deals with determining the identifying test basis for specific testing activities, determination of test objectives, and eventually the selection and application of appropriate the test design techniques to achieve those test objectives. UTP organizes concepts provided for carrying out test analysis and design activities into two parts: concepts for describing test contexts, test objectives, test requirements, and concepts to specify test design activities.

8.3.1 Test Analysis

The test analysis concepts are means to argue and justify why certain testing activities have to be carried out as well as how these testing activities with all required or helpful artifacts are organized.

In order to group artifacts and information that are deemed necessary for certain testing activities, the test context concept (represented by the stereotype «TestContext») is introduced. It offers the capability to bundle artifacts (e.g., any PackageableElement) in a shared scope (e.g., the Namespace), to hide information from other scopes and to import elements from other scopes. This enables a high degree of organizational reusability of information.

In dynamic testing, test cases are eventually produced by the test design activities in order to execute them. For certain reasons, test cases are often assembled and executed together in a test set (or test suite, which is a synonym of a test set). In UTP, a test set is represented by the stereotype «TestSet» which has the ability to assemble, import and reuse test cases.

The definition of certain coverage criteria and/or objectives that the testing activities have to meet is essential for test planning. In UTP, the planning activities are supported by means of the concepts test objective (implemented by the stereotype «TestObjective»), test requirement (implemented by the stereotype «TestRequirement»), a verification dependency among development artifacts and test objectives or test requirements (represented by the stereotype «verifies»). In order to stay as close as possible to the SysML definition of requirements [SysML], both test objective and test requirements are designed as extensions to the UML metaclass Class. Such a stereotyped Class is capable of defining new properties solely, whereas most of the capabilities of the metaclass Class are forbidden by constraint, such as owning Ports, Operations, Behaviors etc.. The stereotype «verifies» extends the UML metaclass Dependency in order to be technically compatible with SysML [SysML], too.

These concepts enable testers to adhere to well-known and established industrial testing standards such as ISTQB [ISTQB] or ISO 29119 [ISO29119] when creating model-based test specifications. Whereas test objectives are intended to describe higher level goals the testing activities have to achieve in a certain context (e.g., coverage of all high priority requirements at system level testing), test requirements are intended to pinpoint a single and testable aspect of the test item. As such, test objectives describe often the test ending criteria for the testing activities in a certain context (e.g., system level testing), and test requirements leverage the development of test design input definitions or test cases. Eventually, test requirements are realized by test cases, which is similar to the coverage of test requirements. Test requirements contribute to the fulfilment of test objectives.

Both test objectives and test requirements can be used independently of each other or in joint manner or not at all. This is contextually up to the respective testing methodology. UTP does not prescribe the use of these concepts.

8.3.1.1 Test Context Overview

The stereotypes «TestContext» and «TestSet» are defined in UTP. Both represent a container for dedicated elements, thus, they are extensions of the UML Package. As such they inherit the concept of nested Packages, Package templates, owned and imported members as well as visibility. However, it is not prescribed that the visibility concepts have to be respected by any conforming UTP tooling. The decision whether or not to utilize the visibility and import mechanism of UML is up to the tool implementation. However, the derived associations of «TestContext» and «TestSet», however, are based on UML visibility and import.

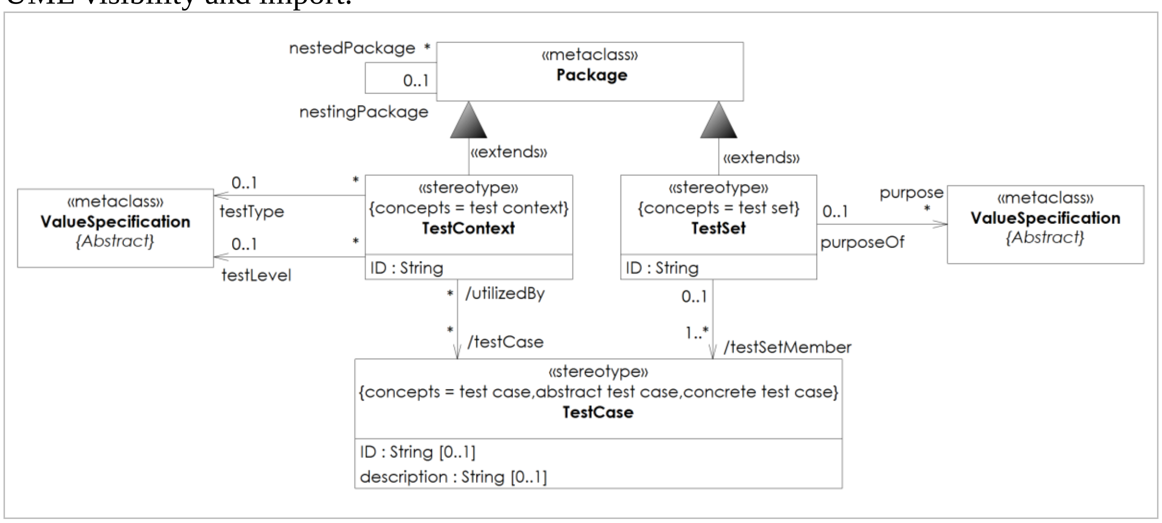


Figure 15 - Test Context Overview

8.3.1.2 Test-specific Contents of Test Context

The UML profile specification for the test context concepts is shown in the following diagram. Most of the relationships among the concepts of the Conceptual Model are already covered by the underlying UML metamodel. In order to allow users of the UTP an easy access to related elements, a set of derived associations is defined that retrieves the desired element for a currently processed stereotype. As an example for the design decision, please see the derived associations between «TestContext» and «TestCase». In the Conceptual Model it is stated that a test context refers to a set of test cases. Since «TestContext» extends the UML metaclass Package and «TestCase» extends a subclass of a PackageableElement, there are several native (i.e., given by the UML metamodel) possibilities on how to reflect the conceptual 'refers to' relationship. First, a Package may contain PackageableElements; second, a Package may import PackageableElement, either by using ElementImport (i.e., only that specific element) or by PackageImport (i.e., all visible and accessible elements in the imported Package). The derived associations of the UTP stereotypes follow the UML metamodel capabilities to collect all concrete PackageableElements stereotyped with «TestCase» that are either contained in or imported by the underlying «TestContext» Package. The advantage is that the test engineer does not have to implement or even know the details of the UML metamodel to retrieve the desired elements.

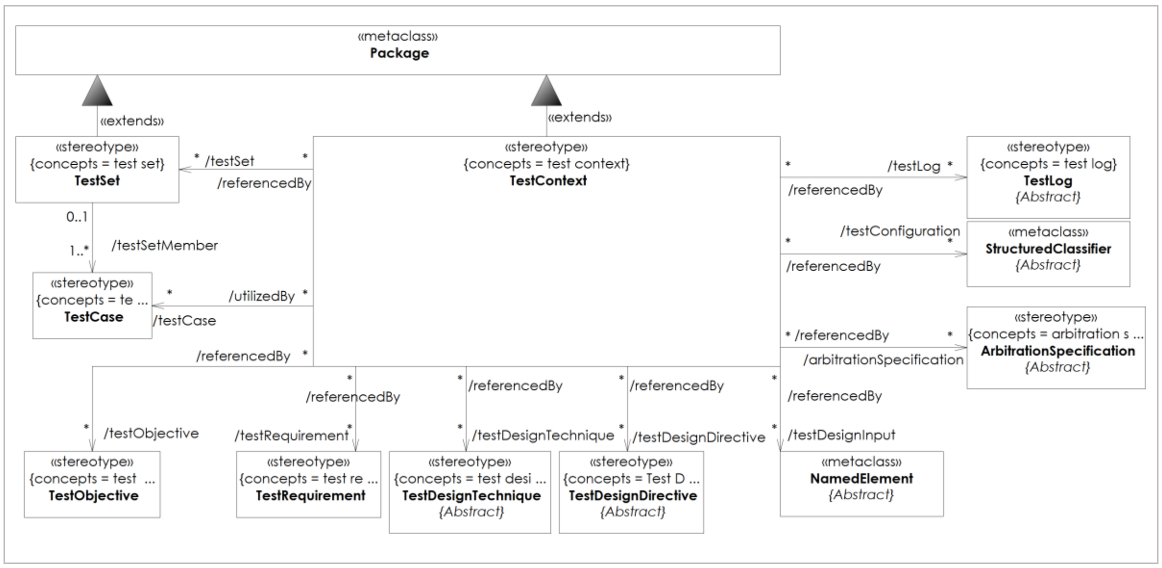


Figure 16 - Test-specific Contents of Test Context

8.3.1.3 Test Objective Overview

The following diagram shows the abstract syntax for the test objectives concepts.

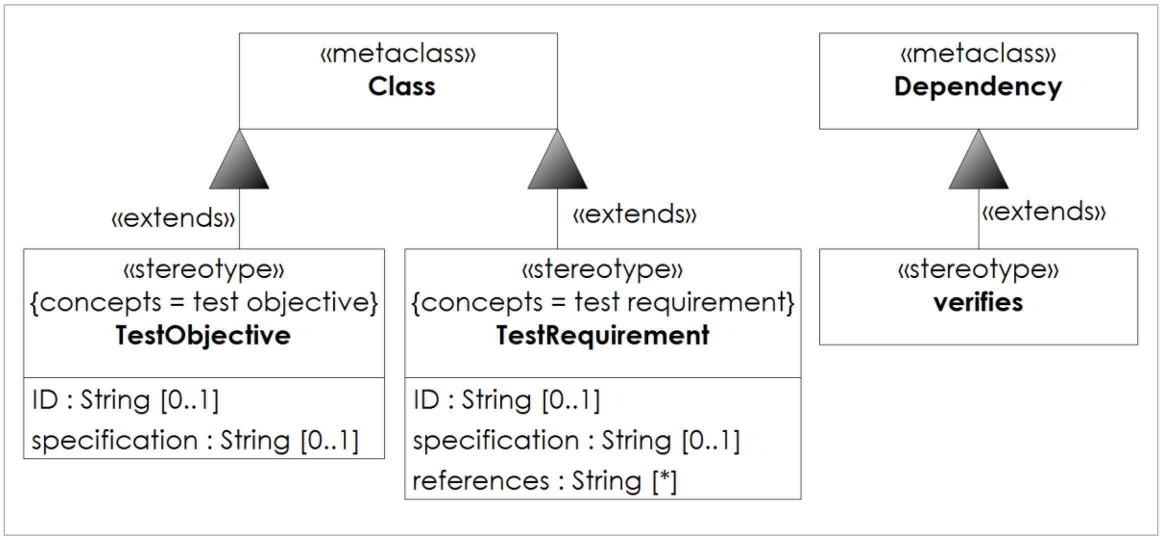


Figure 17 - Test Objective Overview

8.3.1.4 Stereotype Specifications

8.3.1.4.1 TestContext

Description	TestContext: A set of information that is prescriptive for testing activities that are organized and managed together for deriving or selecting test objectives, test design techniques, test design inputs and eventually test cases. A test context may import the packaged elements of another test context in order to access and reuse visible elements of the imported test context. This is inherently given by the native UML concepts PackageImport or ElementImport. Whether or not the visibility of elements contained in a test context is respected is up to the tool implementation. Since a «TestContext» is an extended Package, it is possible to decompose test contexts into more fine-grained test contexts. For example, a test context defined for the test level 'System testing' might be decomposed in accordance to the test types that are addressed at that test level (e.g., functional system testing, security system testing etc.).
Extension	Package
Attributes	ID : String [1] A unique identifier to unambiguously distinguish between any

	two test contexts.
Associations	/testCase : TestCase [*] The test cases that are accessible by the given «TestContext». This feature is derived by the set of directly owned or via ElementImport or PackageImport for imported test cases.
	testLevel : ValueSpecification [0..1] The test levels that the testing activities within the given «TestContext» have to cope with.
	testType : ValueSpecification [0..1] The test types that the testing activities within the given «TestContext» have to cope with.
	/testSet : TestSet [*] Refers to the test sets that are known by this test context. It is derived from both contained and imported Packages with «TestSet» applied.
	/testObjective : TestObjective [*] Refers to the test objectives that are known by this test context. It is derived from both contained and imported Classes with «TestObjective» applied.
	/testRequirement : TestRequirement [*] Refers to the test requirements that are known by this test context. It is derived from both contained and imported Classes with «TestRequirement» applied.
	/testConfiguration : StructuredClassifier [*] Refers to the test configurations that are known by this test context. It is derived from both contained and imported StructuredClassifier with «TestConfiguration» applied.
	/testDesignInput : NamedElement [*] Refers to the test design inputs that are known by this test context. It is derived from both contained and imported NamedElements with «TestDesignInput» applied and the NamedElements that are referenced by all known «TestDesignDirective» as their test design input (i.e., referenced by the tag definition *testDesignInput*). The latter part of

	the derivation algorithm is necessary, because the use of the «TestDesignInput» stereotype is not mandatory, and sometimes even not possible.
	/testDesignDirective : TestDesignDirective [*] Refers to the test design directives that are known by this test context. It is derived from both contained and imported InstanceSpecifications with a concrete subclass of «TestDesignDirective» applied.
	/testDesignTechnique : TestDesignTechnique [*] Refers to the test design techniques that are known by this test context. It is derived from both contained and imported InstanceSpecifications with a concrete subclass of «TestDesignTechnique» applied.
	/arbitrationSpecification : ArbitrationSpecification [*] Refers to the arbitration specifications that are known by this test context. It is derived from both contained and imported BehavioredClassifiers with «TestDesignTechnique» applied.
	/testLog : TestLog [*] Refers to the test logs that are known by this test context. It is derived from both contained and imported InstanceSpecification with a concrete subclass of «TestLog» applied

Constraints	«Test Context» not applicable to Profile
	
	/* A stereotype «TestContext» must not be applicable to UML 2.5::Profile metaclass. */
	
	`context TestContext inv:`
	`self.base_Profile->oclIsUndefined()`

Change from UTP 1.2	Changed from UTP 1.2. In UTP 1.2 «TestContext» extended StructuredClassifier and BehavioredClassifier as well as incorporated the concepts TestSet, TestExecutionSchedule and TestConfiguration into a single concept.

8.3.1.4.2 TestObjective

Description	TestObjective: A desired effect that a test or set of test cases intends to achieve. The stereotype «TestObjective» extends Class. test objectives enables tester to define the test ending criteria for the testing activities in a certain test context. A test objective can be expressed with detail or very abstractly, depending on the underlying methodology. As pure test analysis concept, it is very likely that test objectives have to be traceable to and from test environment tools, which first and foremost would be test management tools. Therefore, test objectives have the ability to specify a unique identifier represented by the tag definition ID. However, the use of the explicit identifier is optional and simply enables the most primitive kind of traceability within a test environment. The specification of a test objective, i.e., the reason why test cases are created and eventually executed, is expressed by means of the tag definition specification. Although it is typed by the PrimitiveType String, the test objective might be specified by means of a formal or structured language. If a BMM profile (see [BMM]) is also loaded into a model containing the UTP 2.0 profile, this stereotype may be considered as a BMM objective (i.e., merged with a BMM objective).
Extension	Class
Attributes	ID : String [0..1] A unique identifier that unambiguously identifies the test objective.
	specification : String [0..1] This is the textual specification of the test objective.
Associations	: TestDesignDirective
	/referencedBy : TestContext [*]

Constraints	No BehavioralFeatures allowed
	
	/* A Class with «TestObjective» applied must not declare Operations or Receptions. */
	
	`context TestObjective inv:`
	`self.base_Class.ownedOperation->size() = 0 and self.base_Class.ownedReception->size()=0`

	No InterfaceRealization allowed
	
	/* A Class with «TestObjective» applied must not realize Interfaces. */
	
	`context TestObjective inv:`
	`self.base_Class.interfaceRealization->size()=0`

	No Associations allowed
	
	/* A Class with «TestObjective» applied must not establish Associations to other Classifiers. */
	
	`context TestObjective inv:`
	`self.base_Class.ownedAttribute->forAll(p:Property|p.association->size()=0)`

	No nested classifiers allowed
	
	/* A Class with «TestObjective» applied must not nest Classifiers. */
	
	`context TestObjective inv:`
	`self.base_Class.nestedClassifier->size()=0`

	Containment of «TestObjective»s
	
	/* A Class with «TestObjective» applied must only be contained in Package having «TestContext» applied. */
	
	`context TestObjective inv:`
	`self.base_Class.owner->selectByType(uml::Package)->forAll(p:Package|p.getAppliedStereotypes()-`

	`>one(s:Stereotype|s.name="TestContext"))`

	Restriction of extendable metaclasses
	
	/* A stereotype «TestObjective» must only be applied to UML 2.5::Class metaclasses and nothing else. */
	
	`context TestObjective inv:`
	`self.base_Class.getAppliedStereotype('UML 2.5::Class')<>null`

Change from UTP 1.2	Changed from UTP 1.2. In UTP 1.2, «TestObjective» was called «TestObjectiveSpecification».

8.3.1.4.3 TestRequirement

Description	TestRequirement: A desired property on a test case or test set, referring to some aspect of the test item to be tested. The stereotype «TestRequirement» extends Class (for integration with the SysML stereotype «requirement»). A test requirement enables testers to decompose single and distinct testable aspects of the test item prior to test design. As such, it is part of the test analysis facility of UTP. test requirements are deemed helpful for both the derivation of test cases, test procedures and in particular test design input definitions. test requirements are said to be realized by test design input definitions, test case or test procedures. The default UML metaclass Realize is intended to be utilized to express this relationship. As a pure test analysis concept, it is very likely that test requirements have to be traceable to and from test environment tools, first and foremost test management tools. Therefore, test requirements have the ability to specify a unique identifier represented by the tag definition ID. However, the use of the explicit identifier is optional and simply enables the most primitive kind of traceability within a test environment. The specification of a test requirement (i.e., the textual description of a single testable aspect of a test requirement) is expressed by means of the tag definition specification. Although it is typed by the PrimitiveType String, the test requirement might be specified by means of a more formal or structured language (e.g., using the Test Purpose Language (TPLan) standardized by ETSI). Additional references to external resources (e.g., relevant standards, guidelines, documents, websites etc.) can be added via the tag definition references. If SysML [SysML] is also loaded into a model containing the UTP 2.0 profile, this stereotype may be considered as (i.e., merged with) the SysML stereotype «requirement».
Extension	Class
Attributes	ID : String [0..1]

	A unique identifier that unambiguously identifies the test requirement.
	specification : String [0..1] This is the specification of the test requirement. It might be represented in both unstructured and structured text.
	references : String [*] Includes any additional references that are deemed relevant for the definition of the test requirement (such as relevant standards, papers, or any other meaningful artifact)
Associations	/realizedBy : TestCase [*] References the test cases that realize the given test requirement. They are derived from the set of UML Realization dependencies that point to the base Class of this stereotype and stem from a BehavioredClassifier or Behavior stereotyped with «TestCase».
	/referencedBy : TestContext [*]

Constraints	No BehavioralFeatures allowed
	
	/* A Class with «TestRequirement» applied must not declare Operations or Receptions. */
	
	`context TestRequirement inv:`
	`self.base_Class.ownedOperation->size() = 0 and self.base_Class.ownedReception->size()=0`

	No InterfaceRealization allowed
	
	/* A Class with «TestRequirement» applied must not realize Interfaces. */
	
	`context TestRequirement inv:`
	`self.base_Class.interfaceRealization->size()=0`

	No Associations allowed /* A Class with «TestRequirement» applied must not establish Associations to other Classifiers. */ `context TestRequirement inv:`

	`self.base_Class.ownedAttribute->forAll(p:Property|p.association->size()=0)`

	Only nested «TestRequirement»s are allowed
	
	/* A Class with «TestRequirement» applied must only nest Classes that have «TestRequirement» applied. * /
	
	`context TestRequirement inv:`
	`self.base_Class.nestedClassifier->forAll(c:Classifier|c.oclIsKindOf(TestRequirement))`

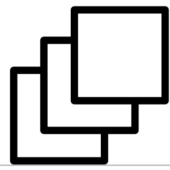
	Restriction of extendable metaclasses
	
	/* A stereotype «TestRequirement» must only be applied to UML 2.5::Class metaclasses and nothing else. */
	
	`context TestRequirement inv:`
	`self.base_Class.getAppliedStereotype('UML 2.5::Class')<>null`

	References to UseCases not allowed
	
	/* A Class with «TestRequirement» applied must neither refer to UseCase nor own UseCase. */
	
	`context TestRequirement inv:`
	`self.base_Class.getAllAttributes()->forAll(p:Property|p.type <> uml::UseCase) and self.base_Class.ownedElement->forAll(e:Element|e.oclAsType(uml::UseCase) = null)`

Change from UTP 1.2	«TestRequirement» has been newly introduced into UTP 2.

8.3.1.4.4 TestSet

Description	TestSet: A set of test cases that share some common purpose. A test set assembles test cases either via ownership or import. These test cases are called the members of the test set. Ownership assembly is based on the ability of UML Packages to nest any PackageableElement. Import assembly is based on the ability of UML Packages to import PackageableElements either directly or indirectly by importing the Package that contains the PackageableElement to be imported. A test case is transitively an extension of PackageableElement, thus, the import mechanisms given by UML can be reused to group test cases in test sets by either assembly kind. Visibility of test cases within a test set is defined in accordance with the visibility of NamedElement in Namespaces as defined by UML. Since the use of visibility is not mandatory by UML, it is also not mandatory to utilize visibility in UTP. However, if visibility is desired, it must comply with the UML semantics. A test set can have an arbitrary number of test execution schedules (extends Behavior) either by ownership or import, similar to test case assembly. A test execution schedule must only schedule the execution of test cases that are members of the respective test sets. If a test set does not contain an explicit test execution schedule, it is semantically equivalent to an implicitly owned test execution schedule that schedules the execution of all test cases assembled by the current test set in an arbitrary order. If a test set is supposed to be executed, the decision which test execution schedule will be taken into account for scheduling is not defined UTP, since a test set may have more than just one test execution schedule defined. A viable method is to use the UML deployment specification to implement the desired test execution schedule for eventual execution by an executing entity. If a test set assembles another test set, the assembling test set has access to all visible test cases assembled by the assembled test set. In addition, the assembling test set has access to all visible test execution schedules of the assembled test set. This enables the composition and decomposition of test sets and their respective test execution schedules.

	The purpose of a test set is set of a ValueSpecifications that can be shared with other test sets. If a test set has more than one purpose, the purposes are logically combined by AND (i.e., if a test set has the two purposes 'Manual Testing' and 'Regression Testing' it should be read as follows 'The test set's purpose is 'manual regression testing').
Graphical syntax	
Extension	Package
Attributes	ID : String [1] This is a unique identifier to unambiguously distinguish between any two test contexts.
Associations	purpose : ValueSpecification [*] Denotes the purposes why the test set has been assembled.
	/testSetMember : TestCase [1..*] Refers to the TestCases that are assembled, either via ownership or import, by the given TestSet, and thus, are members of that TestSet. A TestCase can be a member of more than one TestSet.
	: TestSetLog [*]
	testSetAS : TestSetArbitrationSpecification [0..1]
	/referencedBy : TestContext [*]

Constraints	Restriction of extendable metaclass
	
	/* A stereotype «TestSet» must not be applied to UML 2.5::Profile metaclasses. */
	
	`context TestSet inv:`
	`self.base_Profile->oclIsUndefined()`

	Restrictions on contained elements /* A Package with «TestSet» applied must only contain BehavioredClassifiers having «TestCase» applied or imports thereof, or Behaviors having «ArbitrationSpecification» applied or imports thereof. */

	`context TestSet inv:`
	`self.base_Package->forAll(p:Package|p.allOwnedElements()->selectByType(BehavioredClassifier)->forAll(b:BehavioredClassifier|b.getAppliedStereotypes()->exists(s:Stereotype|s.name='TestCase'))) or`
	`self.base_Package->forAll(p:Package|p.importedMember()->selectByType(BehavioredClassifier)->forAll (b:BehavioredClassifier|b.getAppliedStereotypes()->exists(s:Stereotype|s.name='TestCase'))) or`
	`self.base_Package->forAll(p:Package|p.allOwnedElements()->selectByType(Behavior)->forAll(b:Behavior|b.getAppliedStereotypes()->exists(s:Stereotype|s.name='ArbitrationSpecification'))) or`
	`self.base_Package->forAll(p:Package|p.importedMember()->selectByType(Behavior)->forAll(b:Behavior|b.getAppliedStereotypes()->exists(s:Stereotype|s.name='ArbitrationSpecification')))`

Change from UTP 1.2	«TestSet» has been newly introduced by UTP 2. It was part of the TestContext in UTP 1.2.

8.3.1.4.5 verifies

Description	The stereotype «verifies» extends Dependency and is intended to express relationships among elements that are supposed to be verified (e.g., a requirement, an interface operation, a use case, a user story, a single transition or state, and so forth) and elements that support the verification thereof (e.g., a test objective, a test requirement, a test case, a test set). A «verifies» Dependency as a means to establish traceability within UML-based model elements. It weakens the constraints applied on SysML «Verify» in a sense that UTP «verifies» allows targeting elements different than SysML «requirement». This limitation is too restrictive for UTP, in particular in setups where, for example, use cases are the elements to be verified. Since the semantics of Dependencies with respect to n:m-ary in contrast to binary, 1:m-ary, or n:1-ary Dependencies are not precisely defined, UTP considers by default no difference among all the different ways on how «verifies» Dependencies can be expressed between more than two elements. If a SysML profile (see [SysML]) is also loaded into a model containing the UTP 2.0 profile, this stereotype may be considered as the SysML «Verify» stereotype (i.e. merged with the SysML «Verify» stereotype).
Extension	Dependency
Change from UTP 1.2	«verifies» has been newly introduced into UTP 2. In UTP 1.2 the «verify» stereotype from SysML was recommended.

8.3.2 Test Design

The UTP 2 test design facility describes a language framework for the specification of test design techniques and their application to a test design input element. This includes behavioral descriptions (e.g., UML state machines), or structural information (e.g., interface definitions). test design techniques are usually assembled by so called test design directive which is responsible for establishing the associations between a set of test design techniques and the test design input element those test design techniques must operate on. A test design directive may also link the test design outputs elements that have been generated or derived by the set of applied test design techniques. This allows for a more comprehensible test design phase and is the key to comprehensive traceability among test objectives/test requirements, test design techniques, test design input and eventually test design output elements.

The UTP 2 test design facility only represents the very core of the language framework. Since the stereotypes of the core framework are based on abstract stereotypes and mostly derived (and read-only unions) associations, it is possible to concretize and extend the test design facility as required by using stereotype specialization and property subsetting. A built-in concretization of the core framework was done by means of the generic test design capabilities and the predefined test design techniques. It enables test engineers to immediately utilize the test design facility or develop proprietary test design directives and test design techniques. Tailoring of the UTP test design facility can be done at metalevel M1 (model level) and metalevel M2 (metamodel level). The different mechanism for tailoring are:

- Tailoring through structural features: Both «TestDesignTechnique» and «TestDesignDirective» extend the UML metaclass InstanceSpecification with implicit attributes predefined by the respective stereotypes. In addition to these predefined attributes, user may add additional attributes to these two elements by using the genuine InstanceSpecification-Classifier association. Since both stereotypes extend InstanceSpecification, it is possible to classify these InstanceSpecifications with multiple Classifiers. For this purpose, UTP provides the stereotypes «TestDesignDirectiveStructure» and «TestDesignTechniqueStructure». As a result, the user may add as many additional attributes as desired or required to a «TestDesignDirective» and «TestDesignTechnique».
- Tailoring through use of «GenericTestDesignDirective» and «GenericTestDesignTechnique»: By means of the predefined stereotypes «GenericTestDesignTechnique» and «GenericTestDesignDirective», users can build on proprietary test design directives and test design techniques by simply providing dedicated names to the underlying InstanceSpecification (i.e., the InstanceSpecification with «GenericTestDesignDirective» or «GenericTestDesignTechnique» applied. In combination with the extension through structural features as just described above, the use of «GenericTestDesignTechnique» and «GenericTestDesignDirective» provides a flexible and powerful mechanism to tailor the UTP test design facility for user-specific purposes. For example, an InstanceSpecification with «TestDesignTechnique» applied and name set to 'PathCoverage' is one way to provide the test engineer with a new test design techniques that represents path coverage.
- Profile extension: The third and most powerful tailoring to user-specific needs comes along with profile extension. Similar to the provision of specialized stereotypes of the abstract stereotypes «TestDesignTechnique» and «TestDesignDirective» as predefined concepts of the language itself, users or vendors may introduce proprietary stereotypes that specialize the abstract stereotypes provided by the test design facility of UTP.

8.3.2.1 Test Design Facility

The following picture shows the abstract syntax of the very core of the UTP test design facility.

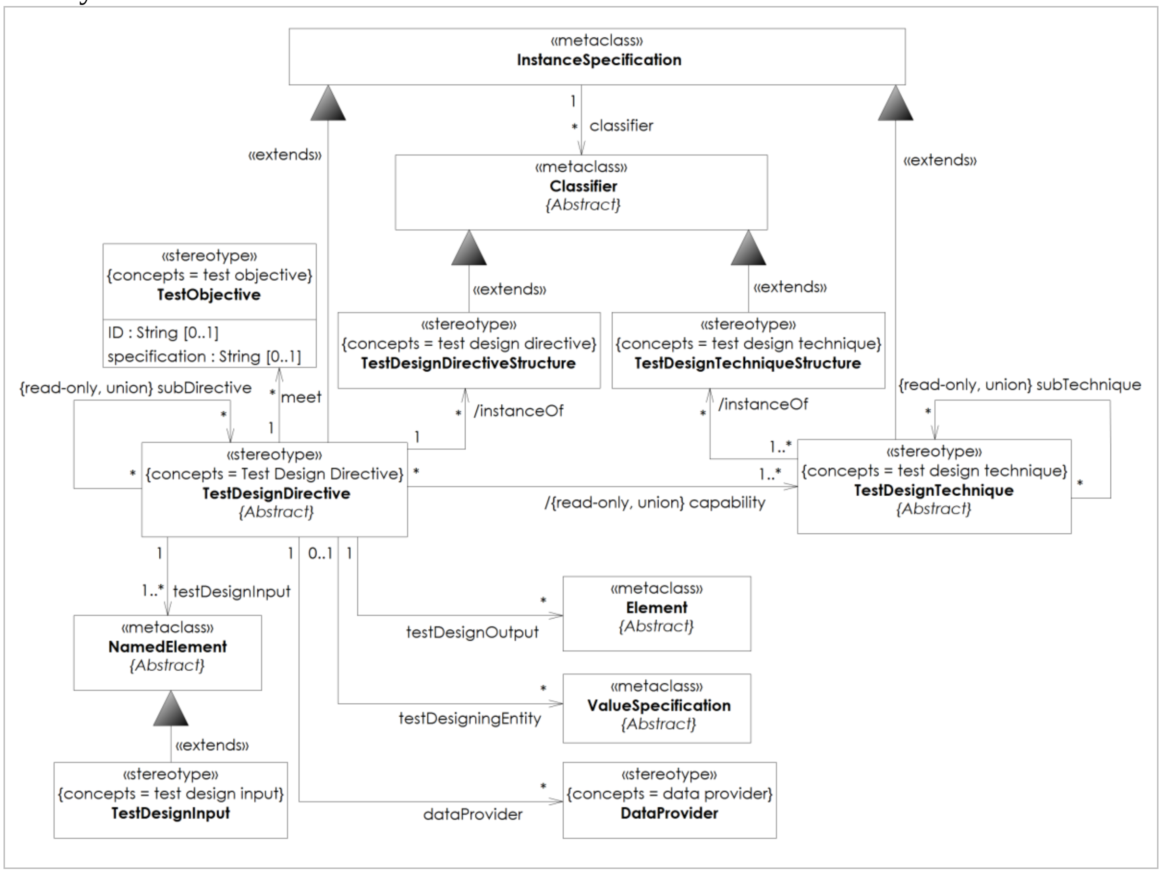


Figure 18 - Test Design Facility

8.3.2.2 Generic Test Design Capabilities

The generic test design capabilities of UTP 2 enable tester to immediately start off with specifying test design directives and defining proprietary, user-defined or project-specific test design techniques, if the predefined test design techniques does not suffice.

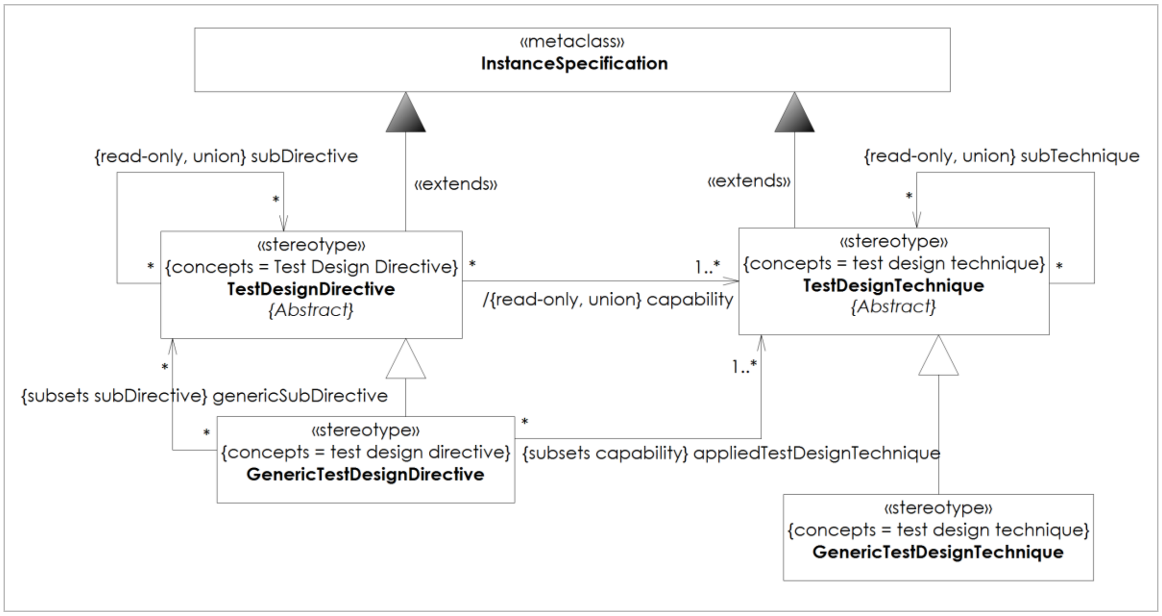


Figure 19 - Generic Test Design Capabilities

8.3.2.3 Predefined high-level Test Design Techniques

The following diagram shows the predefined high-level test design techniques. They belong to the so called specification-based test design techniques as categorized by [ISO29119]-4.

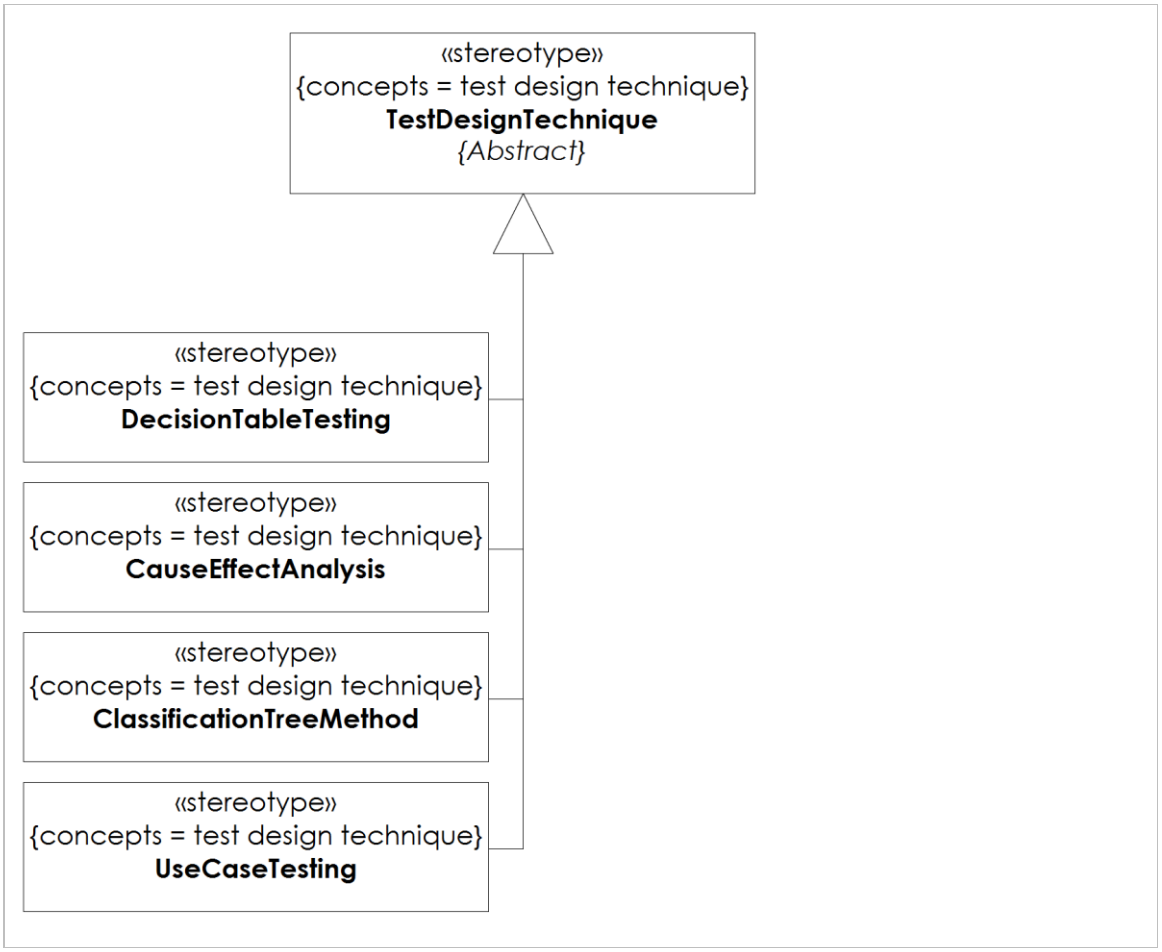


Figure 20 - Predefined high-level Test Design Techniques

8.3.2.4 Predefined data-related Test Design Techniques

The following diagram shows the predefined data-related test design techniques. They belong to the so called specification-based test design techniques as categorized by [ISO29119]-4.

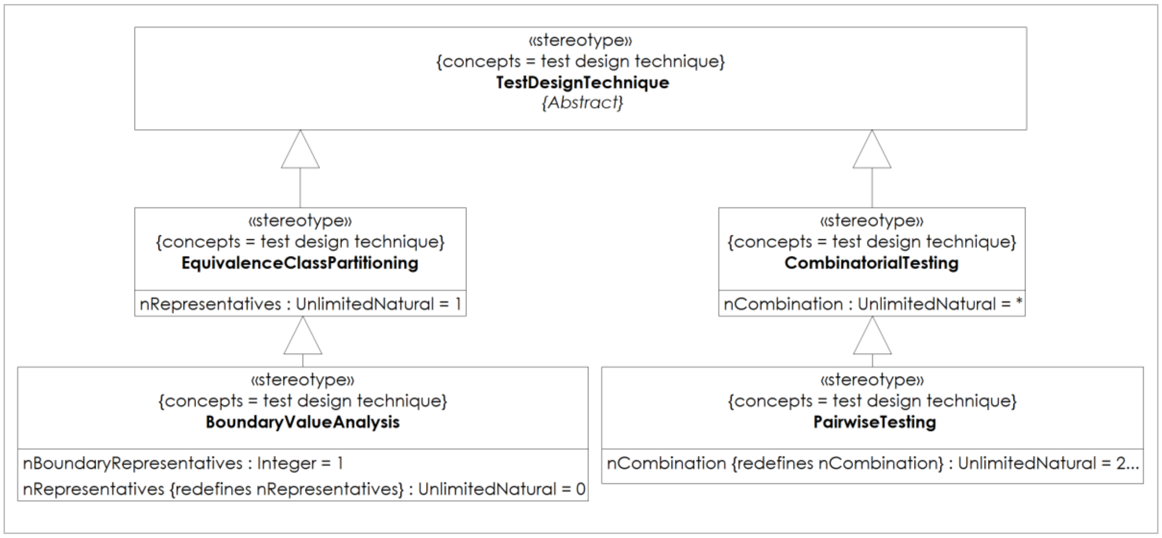


Figure 21 - Predefined data-related Test Design Techniques

8.3.2.5 Predefined state-transition-based Test Design Techniques

The following diagram shows the predefined state-transition based test design techniques. They belong to the so called specification-based test design techniques as categorized by [ISO29119]-4.

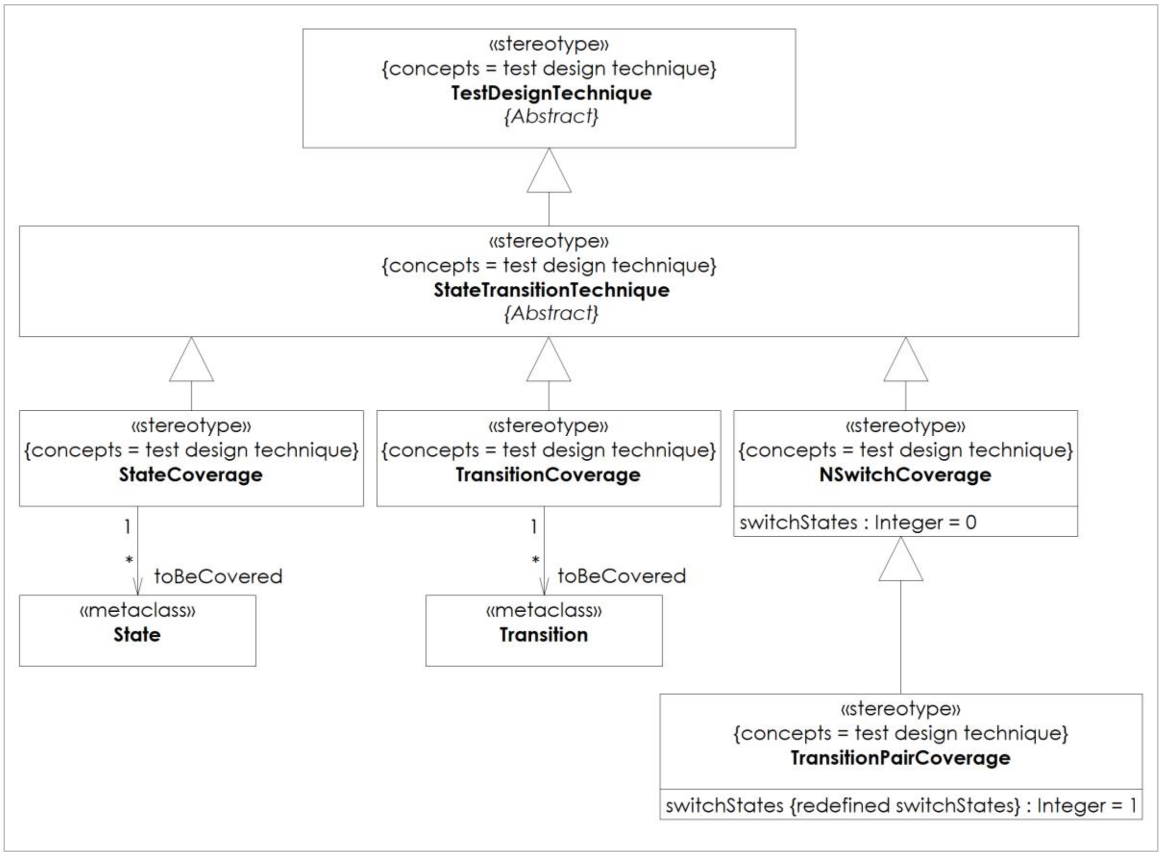


Figure 22 - Predefined state-transition-based Test Design Techniques

8.3.2.6 Predefined experience-based Test Design Techniques

The following diagram shows the predefined experienced-based test design techniques as categorized by [ISO29119]-4.

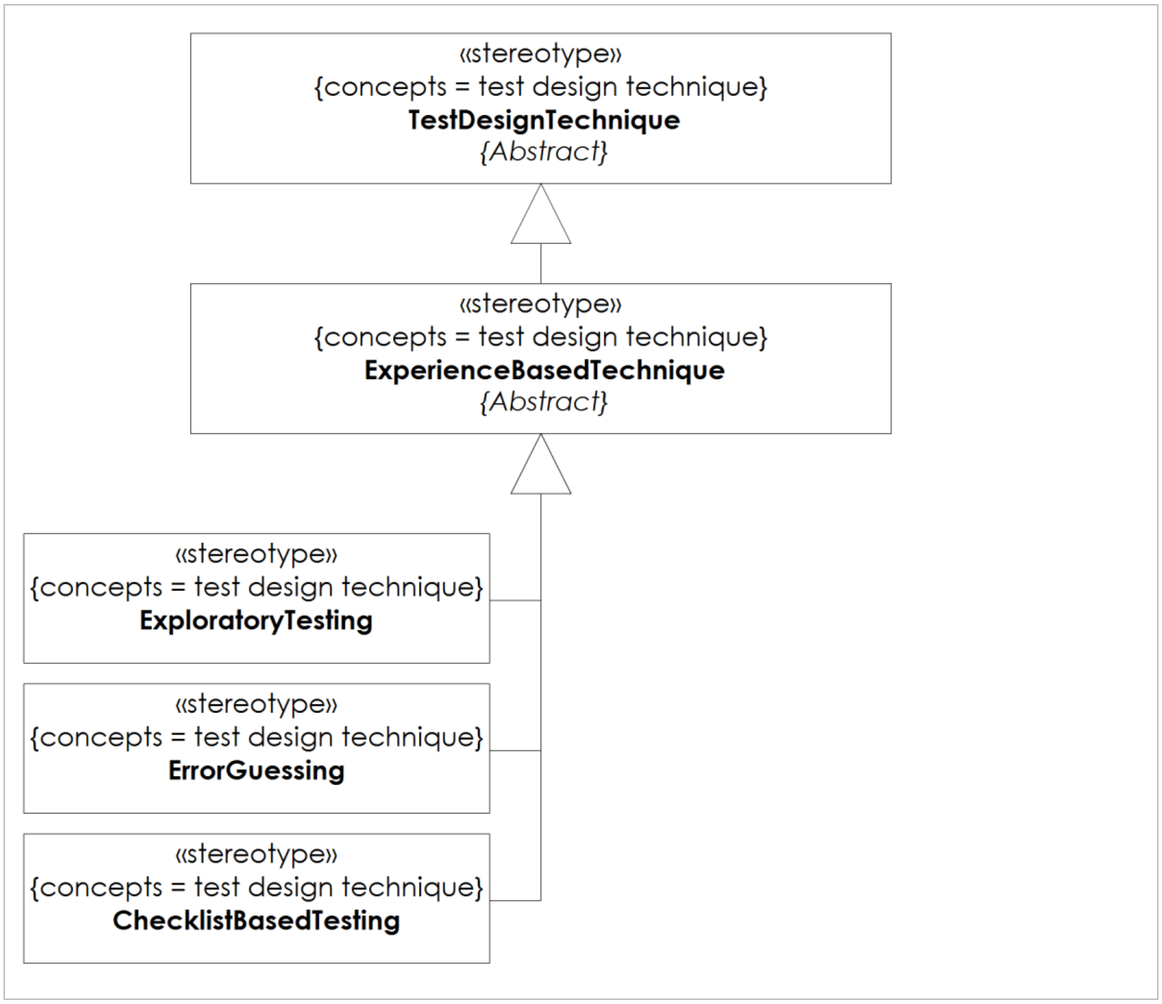


Figure 23 - Predefined experience-based Test Design Techniques

8.3.2.7 Stereotype Specifications

8.3.2.7.1 BoundaryValueAnalysis

Description	According to [ISTQB]: Black box testing is a test design technique in which test cases are designed based on boundary values. «BoundaryValueAnalysis» is an extension of «EquivalenceClassPartitioning» that takes also values at the boundaries (left and right or upper and lower boundary) into account. A boundary value is defined by ISTQB as "an input value or output value which is on the edge of an equivalence partition or at the smallest incremental distance on either side of an edge, for example the minimum and maximum value of a range." Since the boundary values already define representatives of an equivalence class, the ordinary (i.e. non-boundary) representatives are usually of less interest. Therefore, the inherited property nRepresentatives is redefined to obtain the default value 0. This ensures that no additional ordinary representatives of the equivalence class are selected. However, it is still possible to specify that in addition to the boundary values, ordinary representatives of the corresponding equivalence class will be selected by setting the value of nRepresentatives to a value greater than 0. See [ISO29119]-4 clause 5.2.3 BoundaryValueAnalysis for further information.
Extension	InstanceSpecification
Super Class	EquivalenceClassPartitioning
Attributes	nBoundaryRepresentatives : Integer [1] = 1 Specifies the number of boundary representatives that have to be covered by the resulting test cases. Default is 1.
	nRepresentatives {redefines nRepresentatives} : UnlimitedNatural [1] = 0 Redefines the number of representatives to 0, in addition to the boundary values, meaning that by default only the boundary values will be selected.

Change from UTP 1.2	«BoundaryValueAnalysis» has been newly introduced by UTP 2.

8.3.2.7.2 CauseEffectAnalysis

Description	According to [ISTQB]: A black box test design technique in which test cases are designed from cause-effect graphs. See also [ISO29119]-4, clause 5.2.7 Cause-Effect Graphing for further information.
Extension	InstanceSpecification
Super Class	TestDesignTechnique
Change from UTP 1.2	«CauseEffectAnalysis» has been newly introduced by UTP 2.

8.3.2.7.3 ChecklistBasedTesting

Description	According to [ISTQB]: An experience-based test design technique whereby the experienced tester uses a high-level list of items to be noted, checked, or remembered, or a set of rules or criteria against which a product has to be verified.
Extension	InstanceSpecification
Super Class	ExperienceBasedTechnique
Change from UTP 1.2	«ChecklistBasedTesting» has been newly introduced by UTP 2.

8.3.2.7.4 ClassificationTreeMethod

Description	According to [ISTQB]: A black box test design technique in which test cases, described by means of a classification tree, are designed to execute combinations of representatives of input and/or output domains. A classification tree is a tree showing equivalence partitions hierarchically ordered, which are used to design test cases in the classification tree method. See also [ISO29119]-4, clause 5.2.2 Classification Tree Method for further information.
Extension	InstanceSpecification

Super Class	TestDesignTechnique
Change from UTP 1.2	«ClassificationTreeMethod» has been newly introduced by UTP 2.

8.3.2.7.5 CombinatorialTesting

Description	According to [ISTQB]: A means to identify a suitable subset of test combinations to achieve a predetermined level of coverage when testing an object with multiple input parameters and where those parameters themselves each have several values. The Property nCombinations specifies the number of how many parameters must be combined with each other. The higher the number of combinations, the higher the number of derived test cases. By default, all combinations of input parameters will be covered, which is indicated by the asterisk (*). However, the value of the Property nCombination has to be less than the number of the input parameters. See [ISO29119]-4 clause 5.2.5 Combinatorial Test Design Technqiues for further information.
Extension	InstanceSpecification
Super Class	TestDesignTechnique
Sub Class	PairwiseTesting
Attributes	nCombination : UnlimitedNatural [1] = * The number of combinations of input parameters
Change from UTP 1.2	«CombinatorialTesting» has been newly introduced by UTP 2.

8.3.2.7.6 DecisionTableTesting

Description	According to [ISTQB]: A black box test design technique in which test cases are designed to execute combinations of inputs and/or stimuli (causes) shown in a decision table. A decision table is a table showing combinations of inputs and/or stimuli (causes) with their associated outputs and/or actions (effects), which can be used to design test cases. See also [ISO29119]-4, clause 5.2.6 Decision Table Testing for further information.
Extension	InstanceSpecification
Super Class	TestDesignTechnique
Change from UTP 1.2	«DecisionTableTesting» has been newly introduced by UTP 2.

8.3.2.7.7 EquivalenceClassPartitioning

Description	According to [ISTQB]: A black box test design technique in which test cases are designed to execute representatives from equivalence partitions. In principle test cases are designed to cover each partition at least once. Usually, the number of the representatives of each equivalence class that will be used to derive the test cases is set to 1 in order to keep the number of test cases as low as possible. In certain situations it might be, for whatever reason, desired to select more than just one representative per equivalence class. The property nRepresentatives enables the tester to set any number desired number of representatives per equivalence class. By default, the value is set to 1 (reflecting the usual application of that test design technique). If the value is set to unlimited (i.e., the asterisk (*)), all possible representatives of an equivalence class have to be selected. See [ISO29119]-4 clause 5.2.1 Equivalence Partitioning for further information.
Extension	InstanceSpecification
Super Class	TestDesignTechnique
Sub Class	BoundaryValueAnalysis

Attributes	nRepresentatives : UnlimitedNatural [1] = 1 Indicates the desired number of minimal representatives that should be derived for a given equivalence class.
Change from UTP 1.2	«EquivalenceClassPartitioning» has been newly introduced by UTP 2.

8.3.2.7.8 ErrorGuessing

Description	According to [ISTQB]: A test design technique where the experience of the tester is used to anticipate what defects might be present in the component or test item as a result of Errors made and to design tests specifically to expose them. See [ISO29119]-4 clause 5.4 Error Guessing for further information.
Extension	InstanceSpecification
Super Class	ExperienceBasedTechnique
Change from UTP 1.2	«ErrorGuessing» has been newly introduced by UTP 2.

8.3.2.7.9 ExperienceBasedTechnique

Description	According to [ISTQB]: A procedure to derive and/or select test cases based the tester's experience, knowledge and intuition. Experienced-based test design techniques are usually informal techniques potentially supported by checklists or Error taxonomies.
Extension	InstanceSpecification
Super Class	TestDesignTechnique
Sub Class	ChecklistBasedTesting, ErrorGuessing, ExploratoryTesting
Change from UTP 1.2	«ExperienceBasedTechnique» has been newly introduced by UTP 2.

8.3.2.7.10 ExploratoryTesting

Description	According to [ISTQB]: An informal test design technique where the tester actively controls the design of the tests as those tests are performed and uses information gained while testing to design new and better tests.
Extension	InstanceSpecification
Super Class	ExperienceBasedTechnique
Change from UTP 1.2	«ExploratoryTesting» has been newly introduced by UTP 2.

8.3.2.7.11 GenericTestDesignDirective

Description	A predefined test design directive that is able to assemble any test design technique available or known in a certain context, including any user-defined «GenericTestDesignTechnique». As such, the generic test design directive makes no assumptions about the capabilities of a test designing entity a priori. Additional required information can be introduced by utilizing the test design directive structure concept.
Extension	InstanceSpecification
Super Class	TestDesignDirective
Associations	{subsets capability} appliedTestDesignTechnique : TestDesignTechnique [1..*] Enables a generic test design directive to apply any known test design technique for the test design activity.
	{subsets subDirective} genericSubDirective : TestDesignDirective [*] Enables a generic test design directive to be potentially refined by any other known test design directive.
Change from UTP 1.2	«GenericTestDesignDirective» has been newly introduced by UTP 2.

8.3.2.7.12 GenericTestDesignTechnique

Description	The predefined generic test design technique is a semantic-free test design technique that is intended to be used to specify proprietary test design techniques that are not part of the pre-defined UTP 2 test design facility. The name of the underlying InstanceSpecification determines the name of the test design technique, potentially extended by structural information.
Extension	InstanceSpecification
Super Class	TestDesignTechnique
Change from UTP 1.2	«GenericTestDesignTechnique» has been newly introduced by UTP 2.

8.3.2.7.13 NSwitchCoverage

Description	According to [ISTQB]: A form of state transition testing in which test cases are designed to execute all valid sequences of N+1 transitions. N-Switch coverage was initially developed by [Chow], where n defines the number of switch states among a sequence of consecutive transitions. The default is 0, meaning that a test case may only consist of a single transition. However, the entirety of all transitions will be captured by the resulting test cases.
Extension	InstanceSpecification
Super Class	StateTransitionTechnique
Sub Class	TransitionPairCoverage
Attributes	switchStates : Integer [1] = 0 Specifies the number of switch states, and thus, implicitly the sequence of transitions that will at least be covered by the resulting test cases.
Change from UTP 1.2	«NSwitchCoverage» has been newly introduced by UTP 2.

8.3.2.7.14 PairwiseTesting

Description	According to [ISTQB]: A black box test design technique in which test cases are designed to execute all possible discrete combinations of each pair of input parameters. «PairwiseTesting» is a specialized «CombinatorialTesting» test design technique whose property nCombination is refined and set to the read-only value 2, meaning, that at least each pair of input parameters will be covered in the resulting test cases. See [ISO29119]-4 clause 5.2.5.4 Pair-wise Testing for further information.
Extension	InstanceSpecification
Super Class	CombinatorialTesting
Attributes	nCombination {redefines nCombination} : UnlimitedNatural [1] = 2 The number of combinations for each input parameter is set to exactly 2 (i.e., each combination of every pair of input parameters must at least be covered).
Change from UTP 1.2	«PairwiseTesting» has been newly introduced by UTP 2.

8.3.2.7.15 StateCoverage

Description	According to [ISTQB]: A black box test design technique in which test cases are designed that cover at least the execution of a set of referenced states. If no State is referenced by the property toBeCovered, all States in the related state machine will be covered.
Extension	InstanceSpecification
Super Class	StateTransitionTechnique
Associations	toBeCovered : State [*] Refers to a set of States that will at least be covered by the test designer.

Change from UTP 1.2	«StateCoverage» has been newly introduced by UTP 2.

8.3.2.7.16 StateTransitionTechnique

Description	According to [ISTQB]: A black box test design technique in which test cases are designed to execute valid and invalid state transitions. Test design directives that assemble a concrete state-transition technique must refer to at least one state machine as its test design input. If more than one state machine is referenced as test design input, the concrete state-transition techniques are applied to all state machines. See also [ISO29119]-4, clause 5.2.8 State-Transition Testing for further information.
Extension	InstanceSpecification
Super Class	TestDesignTechnique
Sub Class	NSwitchCoverage, StateCoverage, TransitionCoverage
Constraints	State Machine as test design input /* A TestDesignDirective that assembles a StateTransition-Technique should have at least one TestDesignInput element referenced that is a StateMachine. */
Change from UTP 1.2	«StateTransitionTechnique» has been newly introduced by UTP 2.

8.3.2.7.17 TestDesignDirective

Description	TestDesignDirective: A test design directive is an instruction for a test designing entity to derive test artifacts such as test sets, test cases, test configurations, data or test execution schedules by applying test design techniques on a test design input. The set of assembled test design techniques are referred to as the capabilities a test designing entity must possess in order to carry out the test design directive, regardless whether it is carried out by a human tester or a test generator. A test design directive is a means to support the achievement of a test objective. The abstract stereotype «TestDesignDirective» extends InstanceSpecification and brings all relevant information together that is required for automatically or manually derive test artifacts from a test design input. The derivation process is steered by the set of test design techniques, which the current test design directives refers to. Each test design directive has a basic set of structural elements, given by the tag definitions of the «TestDesignDirective» stereotype. The fundamental and implicit structure can be extended by means of UML. Since «TestDesignDirective» extends InstanceSpecification, it is possible to add Classifiers to the underlying InstanceSpecification which then define additional structural information deemed necessary in a specific context. This is the easiest and UML native mechanism to tailor test design directive to specific needs. The test design techniques that will be applied on the test design input are captured in the association end capabilities. This is a derived union, since it cannot be foreseen which test design techniques are required. Concrete subtypes have to subset the derived union capabilities (see for example «GenericTestDesignDirective») in order to enable certain test design techniques for a test design directive. Those test design techniques can be combined with each other by a test design directive. A test design directive refers to a set of NamedElements as the input for the eventual test design activities performed by a test designing entity. This input yields the association end

	TestDesignInput. It is not required that a referenced NamedElement has the stereotype «TestDesignInput» applied. The assembled test design techniques by the given test design directive are then applied on the test design input in order to produce the test design output artifacts. A test design directive may provide sub-directives by means of the association end subDirective. Providing a sub test design directive enables testers to refine the test design activities for certain elements contained in the test design input. As an example, this specification assumes a parent test design directive refers to a StateMachine as its test design input. The test design directive also assembles a set of state-transition and data-related test design techniques that will be applied to the StateMachine by a test designing entity. This specification further assume that the StateMachine contains a submachine State (i.e., a reference of another StateMachine that is considered to be copied to the location of the submachine State) which is referred to as test design input by a sub test design directive. This enables the composition of different kinds of test design directives in order to meet different test objectives.
Extension	InstanceSpecification
Sub Class	GenericTestDesignDirective
Associations	meet : TestObjective [*] The test objectives that have to be fulfilled by putting the given test design directive into effect.
	/{read-only, union} capability : TestDesignTechnique [1..*] Refers to the set test design techniques that are assembled by the given test design directive. The set is referred to as the capabilities a test designing entity (e.g., a generator in automated test design or human tester in manual test design) has to offer in order to be able to perform the test design activities imposed by the test design directive.
	: TestDesignDirective [*]
	{read-only, union} subDirective : TestDesignDirective [*] Refers to one or more test design directives that further refine the instructions given by the parent test design directive.
	: GenericTestDesignDirective [*]

	testDesignOutput : Element [*] The outcome of the test design activities produced by the given test design directives.
	testDesigningEntity : ValueSpecification [*] Identifies the test designing entity (e.g. a generator in automated test design or a human tester in manual test design) that has produced (parts of) the test design output.
	/instanceOf : TestDesignDirectiveStructure [*] Refers to the test design directive structure of which the given test design directive is an instance of. The test design directive structure is derived from all Classifiers with «TestDesignDirectiveStructure» applied that are referred as classifiers by the underlying InstanceSpecification.
	testDesignInput : NamedElement [1..*] Refers to the model elements that have to be incorporated by the test designer (e.g. a generator in automated test design or a human tester in manual test design) as input to the derivation process.
	/referencedBy : TestContext [*]
	dataProvider : DataProvider [*] References the data providers that are supposed to deliver or produce the required test data.
Change from UTP 1.2	«TestDesignDirective» has been newly introduced by UTP 2.

8.3.2.7.18 TestDesignDirectiveStructure

Description	A TestDesignDirectiveStructure describes user-defined or context-specific additional information that may augment any given TestDesignDirective. A Classifier with «TestDesignDirectiveStructure» applied might be of arbitrary complexity. It enables the provision of information that are deemed relevant in a certain context but not required in a different context.
Extension	Classifier
Associations	: TestDesignDirective

Change from UTP 1.2	«TestDesignDirectiveStructure» has been newly introduced by UTP 2.

8.3.2.7.19 TestDesignInput

Description	TestDesignInput: Any piece of information that must or has been used to derive testing artifacts such as test cases, test configuration, and data. The stereotype «TestDesignInput» is an explicit, yet optional means to indicate that the purpose of a given model element is to use it for test design activities (i.e., usually the derivation of test cases, test data, test configurations etc.). The application of this stereotype is declared as optional, because in general any kind of model element might be used as input for the test design activities.
Extension	NamedElement
Change from UTP 1.2	«TestDesignInput» has been newly introduced by UTP 2.

8.3.2.7.20 TestDesignTechnique

Description	TestDesignTechnique: A specification of a method used to derive or select test configurations, test cases and data. test design techniques are governed by a test design directive and applied to a test design input. Such test design techniques can be monolithically applied or in combination with other test design techniques. Each test design technique has clear semantics with respect to the test design input and the artifacts it derives from the test design input. The abstract stereotype «TestDesignTechnique» extends InstanceSpecification and integrates test design techniques with test design directives. A test design technique is a concrete action, technique or procedure to derive test design output from a test design input. A test design technique is basically independent of a dedicated test design input element, but can be reused across multiple test design input elements. Some test design techniques only make sense if a certain test design input element was selected (e.g., state-transition test design techniques make only sense if the test design input element is a StateMachine). Each test design technique has a basic set of structural elements given by the tag definitions of the «TestDesignTechnique» stereotype. The fundamental (and implicit) structure can be extended by means of UML. Since «TestDesignTechnique» extends InstanceSpecification, it is possible to add Classifiers to the underlying InstanceSpecification which then define additional structural information deemed necessary in a specific context. This is the easiest and UML native mechanism to tailor test design techniques to specific needs. A test design technique may provide sub-techniques by means of the association end subTechnique. Providing a sub test design technique enables testers to refine the test design techniques for certain elements contained in the test design input and also to enrich existing (potentially pre-defined) test design techniques in a certain context.
Extension	InstanceSpecification
Sub Class	CauseEffectAnalysis, ClassificationTreeMethod, CombinatorialTesting, DecisionTableTesting, EquivalenceClassPartitioning, ExperienceBasedTechnique, GenericTestDesign-

	Technique, StateTransitionTechnique, UseCaseTesting
Associations	: TestDesignDirective [*]
	: TestDesignTechnique [*]
	{read-only, union} subTechnique : TestDesignTechnique [*] Refers to one or more test design techniques that may further refine the parent test design technique.
	: GenericTestDesignDirective [*]
	/instanceOf : TestDesignTechniqueStructure [*] Refers to additional structural information of the given test design technique. The test design technique structures are derived from all Classifiers with «TestDesignTechniqueStructure» applied that are referred to as classifiers by the underlying InstanceSpecification.
	/referencedBy : TestContext [*]
Change from UTP 1.2	«TestDesignTechnique» has been newly introduced by UTP 2.

8.3.2.7.21 TestDesignTechniqueStructure

Description	A test design technique structure describes user-defined or context-specific additional information that may augment any given test design technique. A Classifier with «TestDesignTechniqueStructure» applied might be of arbitrary complexity. It enables the provision of information that is deemed relevant in a certain context but not required in a different context.
Extension	Classifier
Associations	: TestDesignTechnique [1..*]
Change from UTP 1.2	«TestDesignTechniqueStructure» has been newly introduced by UTP 2.

8.3.2.7.22 TransitionCoverage

Description	According to [ISTQB]: A black box test design technique in which test cases are designed that cover at least the execution of a set of references states. If no Transition is referenced by the property toBeCovered, all States in the related state machine will be covered.
Extension	InstanceSpecification
Super Class	StateTransitionTechnique
Associations	toBeCovered : Transition [*] Refers to a set of Transitions that will at least be covered by the test designer.
Change from UTP 1.2	«TransitionCoverage» has been newly introduced by UTP 2.

8.3.2.7.23 TransitionPairCoverage

Description	The «TransitionPairCoverage» test design technique is a specific (and often used) «NSwitchCoverage» test design technique that redefines the Property switchStates to the read-only value 1. That means that the resulting test cases should at least cover all sequences of any two consecutive Transitions. The semantics of transition pair coverage and N-Switch coverage with nSwitches set to 1 is semantically equivalent.
Extension	InstanceSpecification
Super Class	NSwitchCoverage
Attributes	switchStates {redefined switchStates} : Integer [1] = 1 Restricts the number of switch states to exactly one, meaning, that every pair of subsequent Transitions will at least be covered.
Change from UTP 1.2	«TransitionPairCoverage» has been newly introduced by UTP 2.

8.3.2.7.24 UseCaseTesting

Description	According to [ISTQB]: A black box test design technique in which test cases are designed to execute scenarios of use cases. See also [ISO29119]-4, clause 5.2.9 Scenario Testing for further information.
Extension	InstanceSpecification
Super Class	TestDesignTechnique
Change from UTP 1.2	«UseCaseTesting» has been newly introduced by UTP 2.

8.4 Test Architecture

Test architecture concepts specify structural aspects of a test environment, including a test configuration, necessary to eventually execute test cases against the test item(s). The test environment comprises everything that is necessary to execute test cases (e.g., test components, hardware, simulators, test execution tools etc.). The test configuration describes how those parts of the test environment and represented test components, are connected with the test item.

Building a reliable test configuration is required for any test case, because it determines the test item(s) and how the test environment (in UTP represented by test components) interfaces to the test item(s).

Test architectures are mainly expressed by means of UML class and composite structure diagrams. In contrast to UTP 1.2, both test components and test items can be represented either as a standalone type or as a role that a certain type may assume in a specific test configuration. However, UTP does not prescribe which option to use for describing test architecture and both have advantages and disadvantages.

The test architecture concepts consist of

- test configuration, implemented by the stereotype «TestConfiguration»;
- test configuration role, implemented by the abstract stereotype «TestConfigurationRole» as a superclass for any known (even future) role a test configuration may assume;
- role configuration, implemented by the abstract stereotype «RoleConfiguration» as superclass for configurations of concrete roles;
- test component, implemented by the stereotype «TestComponent» that specializes «TestConfigurationRole»;

- test component configuration, implemented by the stereotype «TestComponentConfiguration» that specializes «RoleConfiguration»;
- test item, implemented by the stereotype «TestItem» that specializes «TestConfigurationRole»;
- test item configuration, implemented by the stereotype «TestItemConfiguration» that specializes «RoleConfiguration»;

8.4.1 Test Architecture Overview

The diagram below shows the abstract syntax of the test architecture concepts.

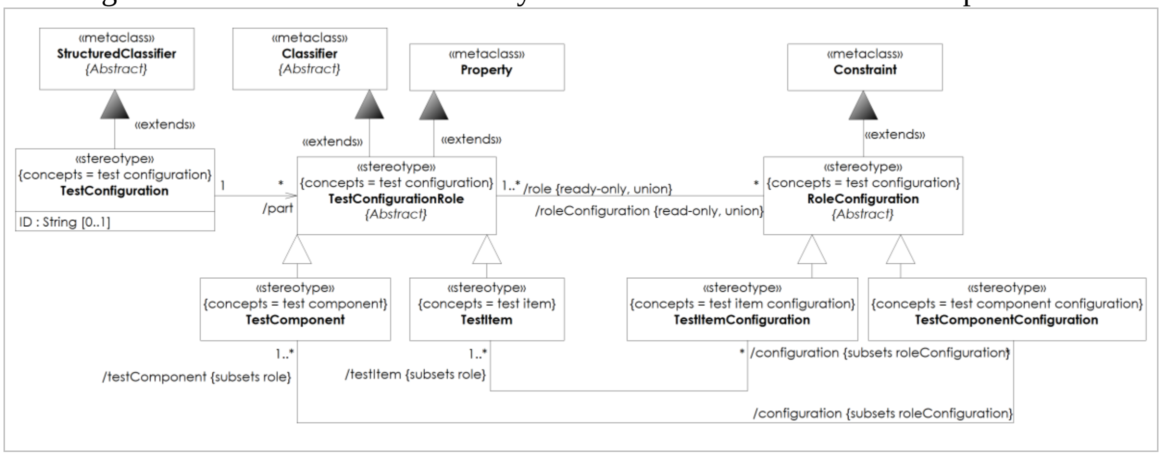


Figure 24 - Test Architecture Overview

8.4.2 Stereotype Specifications

8.4.2.1 RoleConfiguration

Description	The abstract stereotype «RoleConfiguration» extends the metaclass Constraint and is used to specify the configuration of test configuration role within a certain test configuration. There are at least two ways a role configuration can be associated with a test configuration role, both stemming from the underlying UML Constraints metamodel: • Classifier-oriented: A Constraint with a concrete substereotype of «RoleConfiguration» applied is contained by a Classifier as its context with a concrete substereotype of «TestConfigurationRole» applied, or it refers to a set of such Classifiers by means of the meta-association constrainedElement; and • Property-oriented: A Constraint with a concrete substereotype of «RoleConfiguration» applied refers to one or more Properties with «TestConfigurationRole» applied by means of the meta-association constrainedElement The Classifier-oriented way has the advantage that all parts of test configurations which are typed by a Classifier with a concrete substereotype of «TestConfigurationRole» applied, must abide by the configurations defined for that Classifier. On the downside, this might prevent reuse, because it is not possible to get rid of configurations (similar to the handling of Constraints in UML) expressed on Classifier level. The Property-oriented way has the advantage that it enables the dedicated configuration of single test component roles within a test configuration.
Extension	Constraint
Sub Class	TestComponentConfiguration, TestItemConfiguration
Associations	/role {ready-only, union} : TestConfigurationRole [1..*] Refers to the set of at least one test configuration roles.
Change from UTP 1.2	«RoleConfiguration» is newly introduced in UTP 2.

8.4.2.2 TestComponent

Description	TestComponent: A role of an artifact within a test configuration that is required to perform a test case. The stereotype «TestComponent» specializes «TestConfigurationRole» and declares that a certain element (i.e., either a Classifier or Property) is responsible for driving the execution of a test case. The use of the stereotype «TestComponent» on Classifier is optional but, if it is used, all Properties of that type must also have «TestComponent» applied, if they are used in a test configuration.
Extension	Classifier, Property
Super Class	TestConfigurationRole
Sub Class	DataProvider
Associations	/configuration {subsets roleConfiguration} : TestComponentConfiguration [*] Refers to the configurations that are defined for this «TestComponent». This set of configurations is derived from all Constraints with «TestComponentConfiguration» applied that are either owned rules (in case of «TestComponent» is applied on a Classifier) of the «TestComponent» or inversely referring to the «TestComponent» (in case of «TestComponentConfiguration» is applied on Constraint without having a context, but using *Constraint.constrainedElement* to refer to the «TestComponent»).

Constraints	«TestComponent» applied on Type of Property
	
	/* A Property that is typed by a Classifier with «TestComponent» applied, must also have «TestComponent» applied. */
	
	`context TestComponent`
	`not self.base_Property.class.getAppliedStereotype('TestComponent')->oclIsUndefined()`

Change from UTP 1.2	Changed from UTP 1.2. In UTP 1.2., «TestComponent» only extended Class.

8.4.2.3 TestComponentConfiguration

Description	TestComponentConfiguration: A set of configuration options offered by an artifact in the role of a test component chosen to meet the requirements of a particular test configuration. The stereotype «TestComponentConfiguration» specializes the abstract stereotype «RoleConfiguration». The eventual set of configurations for a NamedElement with «TestComponent» applied is derived from the union of all test component configurations declared for that NamedElement (i.e., either on Classifier or Property level).
Extension	Constraint
Super Class	RoleConfiguration
Associations	/testComponent {subsets role} : TestComponent [1..*] Refers to the set of at least one test components that are configured by the given test component configuration. The resulting set is derived from both the Classifier stereotyped with «TestComponent» that is the context of the underlying Constraint and all test components regardless of whether Classifier or Property that are referenced by the underlying *Constraint.constrainedElement*.

Constraints	Ownership of «TestComponentConfiguration»
	
	/* A Constraint with «TestComponentConfiguration» must only be contained by a Classifier with «TestComponent» applied, a StructuredClassifier (preferably stereotyped by «TestComponent» containing a Property with «TestComponent» or by a Package. */
	
	`context TestComponentConfiguration inv:`
	`not`
	`self.base_Constraint.owner.oclAsType(Classifier).getAppliedStereotype('UTP::TestComponent')->oclIsUndefined()`
	`or`
	`not`
	`self.base_Constraint.owner.oclAsType(uml::StructuredClassifier).getAppliedStereotype('UTP::TestComponent')->oclIsUndefined()`
	`and`

	`(self.base_Constraint.owner.oclAsType(uml::`StructuredClassifier`).attribute->collect(p:Property|p.getAppliedStereotype('UTP::`TestComponent`'))->size()=1)`
	`or`
	`not`
	`self.base_Constraint.owner.oclAsType(uml::`Package`).getAppliedStereotype('UTP::`TestComponent`')->oclIsUndefined()`

Change from UTP 1.2	«TestComponentConfiguration» has been newly introduced into UTP 2.

8.4.2.4 TestConfiguration

Description	TestConfiguration: A specification of the test item and test components, their interconnection as well as their configuration data. The stereotype «TestConfiguration» extends StructuredClassifier which effectively extends a variety of UML metaclasses such as Class, Collaboration, and Component, etc. The test configuration then refers to the composite structure of the underlying StructuredClassifier. Every test configuration must have at least one member stereotyped «TestItem» which is connected to at least one member stereotyped with «TestComponent». The test configurations of any two distinct test procedures that are intended to be executed together, as part of a potentially third test procedure, and must have a compatible test configuration. Compatibility of test configurations is partially defined by UML and the substitution principle of Liskov, but also by means of the idea of EncapsulatedClassifiers. The attempt to invoke test procedures together will most likely fail due to technical incompatibility. Test cases or test procedures may come along with their own test configurations expressed by means of their respective composite structures. In that case, the application of the «TestConfiguration» stereotype will be done in addition to «TestCase» or «TestProcedure». In case of shared test configurations it is recommended, though not required, to facilitate the UML concept of a «TestConfiguration» stereotyped Collaboration. Collaborations are meant to be reused by other StructuredClassifiers, including Behaviors, by means of CollaborationUse and role bindings. Inheritance and redefinition, as defined by UML, are additional means to express shared and reusable test configurations, as well.
Detailed semantics	A test configuration might be either explicitly represented by the use of this stereotype or implicitly by simply using the «TestItem» and «TestComponent» stereotypes on a StructuredClassifier.
Extension	StructuredClassifier
Attributes	ID : String [0..1] A unique identifier that unambiguously identifies the given

	test configuration.
Associations	/part : TestConfigurationRole [*] Refers to the test configuration parts that are involved in this test configuration. They are derived from all members of the underlying StructuredClassifier that has a subclass of the abstract stereotype «TestConfigurationRole» applied.

Constraints	Minimal test configuration
	
	/* A StructuredClassifier with «TestConfiguration» applied must at least specify one part having «TestItem» applied and one part having «TestComponent» applied. */
	
	`context TestConfiguration inv:`
	`self.base_StructuredClassifier.part->exists(p:Property|p.oclIsTypeOf(TestItem)) and self.base_StructuredClassifier.part->exists(p:Property|p.oclIsTypeOf(TestComponent))`

Change from UTP 1.2	«TestConfiguration» has been newly introduced into UTP 2. It was conceptually represented by the composite structure of a «TestContext» in UTP 1.2.

8.4.2.5 TestConfigurationRole

Description	The abstract stereotype «TestConfigurationRole» extends both Classifier and Property. The advantage of assigning the role to a certain part assumes in a test configuration that the very same Type of this part (i.e., Class or Component) can be reused in different test configuration with different roles. This entails that the application of a concrete subclass of «TestConfigurationRole» on a Classifier is not required at all and limits reusability of this Classifier. If a concrete substereotype of «TestConfigurationRole» is applied on a Classifier, any part of a test configuration must have the very same concrete substereotype applied.
Extension	Classifier, Property
Sub Class	TestComponent, TestItem, TestItemConfiguration
Associations	/roleConfiguration {read-only, union} : RoleConfiguration [*] Refers to the role configuration that is defined for this test

	configuration role. : TestConfiguration
Constraints	Application on Port not allowed /* Any subclass of «TestConfigurationRole» must not be applied on the metaclass Port. */ `context TestConfigurationRole inv: self.base_Property.oclAsType(uml::Port)->oclIsInvalid()`
Change from UTP 1.2	«TestConfigurationRole» is newly introduced in UTP 2.

8.4.2.6 TestItem

Description	TestItem: A role of an artifact that is the object of testing within a test configuration. The stereotype «TestItem» always indicates that a certain artifact (i.e., either applied on Classifier or Property) specifies (parts of) the system under test. The use of the stereotype «TestItem» on a Classifier is optional, but if it is used, all Properties of that type within a test configuration must also have «TestItem» applied, if they are used in a test configuration.
Extension	Classifier, Property
Super Class	TestConfigurationRole
Associations	/configuration {subsets roleConfiguration} : TestItemConfiguration [*] Refers to the configurations that are defined for this test item. This set of configurations is derived from all Constraints with «TestItemConfiguration» applied that are either owned rules of the «TestItem» (in case of «TestItem» is applied on a Classifier) or that refer to the given test item using the underlying Constraint's *constrainedElement* attribute.
Constraints	«TestItem» applied on Type of Property /* A Property that is typed by a Classifier with «TestItem»

	applied, must also have «TestComponent» applied. */
	
	`context TestComponent inv:`
	`not self.base_Property.class.getAppliedStereotype('UTP::TestItem')->oclIsUndefined()`

Change from UTP 1.2	«TestItem» has been newly introduced into UTP 2 and supersedes the «SUT» stereotype in UTP 1.

8.4.2.7 TestItemConfiguration

Description	TestItemConfiguration: A set of configuration options offered by an artifact in the role of a test item chosen to meet the requirements of a particular test configuration. The stereotype «TestItemConfiguration» specializes the abstract stereotype «RoleConfiguration». The eventual set of configurations for a NamedElement with «TestItem» applied is derived from the union of all test item configurations declared for that NamedElement (i.e., either on Classifier or Property level).
Extension	Classifier, Constraint, Property
Super Class	RoleConfiguration, TestConfigurationRole
Associations	/testItem {subsets role} : TestItem [1..*] Refers to the set of at least one test items that are configured by the given configuration. The resulting set is derived from both the Classifier stereotyped with «TestItem» that is the context of the underlying Constraint and all «TestItem» elements, regardless whether Classifier or Property, that are referenced by the underlying *Constraint.constrainedElement*.

Constraints	Ownership of «TestItemConfiguration»
	
	/* A Constraint with «TestItemConfiguration» must only be contained by a Classifier with «TestItem» applied, a StructuredClassifier(preferably with «TestConfiguration» applied) containing a Property with «TestItem» applied or by a Package. */
	
	`context TestItemConfiguration inv:`
	`not`

	self.base_Constraint.owner.oclAsType(Classifier).getAppliedStereotype('UTP::TestItem')->oclIsUndefined() or not self.base_Constraint.owner.oclAsType(StructuredClassifier).getAppliedStereotype('UTP::TestItem')->oclIsUndefined() and (self.base_Constraint.owner.oclAsType(StructuredClassifier).attribute->collect(p:Property|p.getAppliedStereotype('UTP::TestItem'))->size()=1) or not self.base_Constraint.owner.oclAsType(Package).getAppliedStereotype('UTP::TestItem')->oclIsUndefined()
Change from UTP 1.2	«TestItemConfiguration» has been newly introduced into UTP 2.

8.5 Test Behavior

Test behavior is a collective term for concepts that can be executed as part of a test set or test case. Since the behavioral descriptions of UML are orthogonal to each other to a certain extent, UTP introduces a set of test execution-relevant stereotypes independently of the underlying UML Behaviors or its constituting parts. Integration with these Behaviors is done via partially multiple extensions.

The concepts for test behaviors are separated into the following blocks:

- Concepts for test-specific procedures (see section Test-specific Procedures)
- Concepts for procedural element (see section Procedural Elements)
- Concepts for test-specific actions (see section Test-specific Actions)

8.5.1 Test-specific Procedures

The fundamental executable concept in UTP is a procedure. Any UML Behavior without «TestCase», «TestExecutionSchedule» or «TestProcedure» applied is considered as a procedure. A procedure comprises procedural elements regardless whether the building blocks are called InteractionFragments (if the procedure is realized as Interaction) or Action (if the procedure is realized as Activity). For example, the procedural element loop is represented by stereotype «Loop» and denotes the repeated iteration of procedural elements that are contained in that loop. «Loop» extends the UML metaclasses CombinedFragment (integrating with Interactions) and the StructuredActivityNode loop (integrating with Activities). Furthermore, it adds some test-specific information such as the ability to provide arbitration specifications, when the loop is part of a test procedure.

Test-specific procedures are procedures that deliver a verdict (i.e., they can, or must in the case of a test case, be arbitrated (see section Arbitration Specifications for further information about arbitration)). This includes that its constituting procedural elements, in particular the test actions, are arbitrated as well and provide their respective verdict to the test case arbitration specification, which potentially provides their verdicts to the test set arbitration specification. UTP defines three different test-specific procedures for:

- test procedure, represented by the stereotype «TestProcedure»;
- test case, represented by the stereotype «TestCase»; and
- test execution schedule, represented by the stereotype «TestExecutionSchedule»

A test procedure is a reusable behavior that comprises procedural elements and runs on a test configuration. A test case invokes one or more test procedures and assigns either of these roles: setup, main or teardown to the invoked test procedure. A test execution schedule represents the invocation order of a test set's test cases.

The allowed invocation scheme for test-specific procedures is as follows:

- test execution schedule must only invoke other test execution schedules, test cases or procedures. The invocation of test procedures by a test execution schedule is not allowed;
- test case must only invoke test procedures or procedures, but must invoke at least one test procedure as its main part. The invocation of test cases or test execution schedules is not allowed;
- test procedure must only invoke other test procedures or procedures. The invocation of test cases or test execution schedules is not allowed.

The test configuration of the invoking test case or test procedure must be compatible with the test configuration of the invoked test procedure. In the case of contained test configurations and inheritance thereof, compatibility is given by the substitution principle of Liskov. In the case of shared test configurations based on Collaboration, compatibility is defined by UML.

8.5.1.1 Test Case Overview

The following diagram shows the abstract syntax of the test-specific procedures.

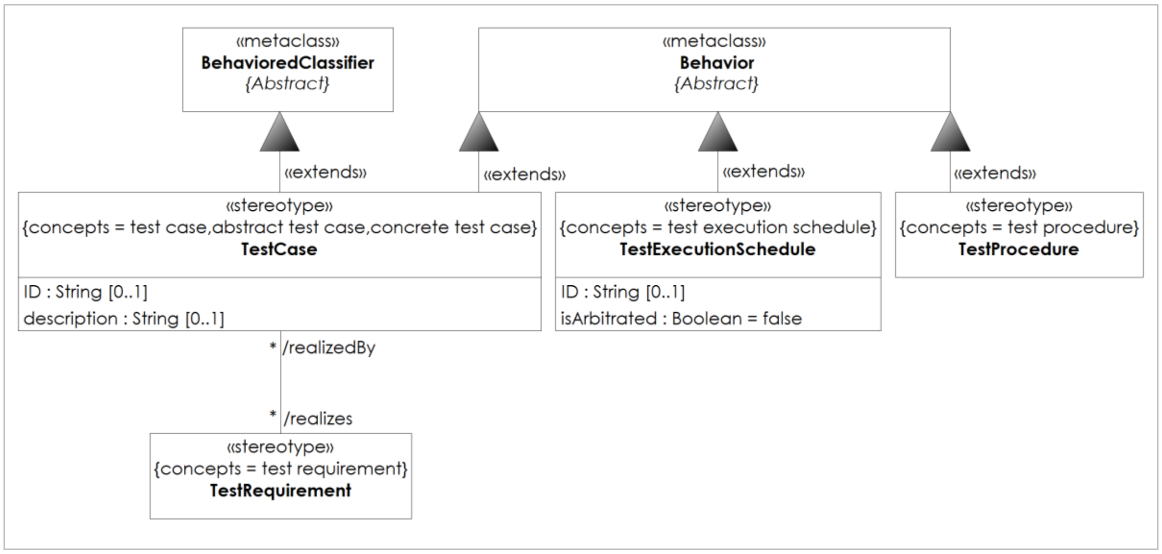


Figure 25 - Test Case Overview

8.5.1.2 Stereotype Specifications

8.5.1.2.1 TestProcedure

Description	TestProcedure: A procedure that constrains the execution order of a number of test actions. A test procedure is a reusable Behavior that constitutes the building blocks for other test procedures or test cases. A test procedure consists of procedural elements, in particular test actions. A test procedure must always run on a test configuration (i.e., its constituting procedural elements are either executed by a test component or a test item). Since «TestProcedure» extends Behavior (as such both StructuredClassifier as well as BehavioredClassifier), a test procedure may provide its own dedicated test configuration defined by its composite structures. In that case, compatibility with the test configuration of any invoking test-specific procedure (i.e., test procedure or test case) must be ensured. A test procedure must only invoke other test procedures or procedures and must only be invoked by other test procedures or test cases. If invoked by a test case, a test procedure may assume either of these roles: main, setup or teardown. If a test procedure invokes another test procedure by means of «ProcedureInvocation» the attribute role of «ProcedureInvocation» must not be set. A test procedure is not allowed to determine the role of other test procedures, because this role can only be determined by test cases. Implicitly, any test procedure assigns their current role assigned by the invoking test case to any other test procedure they invoke. This transitive assignment will be recursively continued until no more test procedures are available. This recursion ensures consistency for the invoking test case.
Extension	Behavior
Constraints	Test procedure operates on test configuration /* A TestProcedure must always run on a (potentially implicit) TestConfiguration comprising at least one instance of a TestComponent connected to a TestItem */

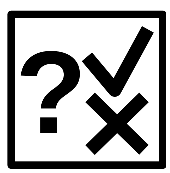
	Allowed invocation scheme /* A TestProcedure must only invoke other TestProcedures or procedures. */
	Use of «ProcedureInvocation» /* A TestProcedure must not make use of the *role* attribute of «ProcedureInvocation» when used as ProceduralElement of the given TestProcedure. */

	Test case invokes one main procedure
	
	/* DRTP04: It is necessary that each test case invokes at least one test procedure as a main procedure invocation. */
	
	`context TestProcedure inv:`
	`self.base_BehavioredClassifier.getAppliedStereotype('UTP::TestProcedure')->size() >=1`

	Procedure sequentializes procedural element /* DRTP02: It is necessary that each procedure prescribes the execution order of at least one procedural element. */
	Test procedure sequencializes test action /* DRTP03: It is necessary that each test procedure prescribes the execution order of at least one test action. */
Change from UTP 1.2	«TestProcedure» has been newly introduced by UTP 2.

8.5.1.2.2 TestCase

Description	TestCase: A procedure that includes a set of preconditions, inputs and expected results, developed to drive the examination of a test item with respect to some test objectives. «TestCase» extends both BehavioredClassifier and Behavior. According to the conceptual model, a test case must provide different functionality like defining pre-/postconditions, being executable etc., and the UML allows different ways for implementing the test case concept. In general, a test case can be either defined as a standalone Behavior stereotyped with «TestCase» or as a compound construct consisting of a «TestCase» BehavioredClassifier, and a «TestCase» Behavior set as the classifierBehavior of the «TestCase» BehavioredClassifier. In the second alternative, both the BehavioredClassifier and its classifierBehavior are semantically treated as a single concept. A test case describes the interplay of the test item with its controlled environment, the so called test environment, consisting of test components. A test case has to operate on a test configuration. The composite structure of a StructuredClassifier with «TestConfiguration» applied determines the different roles the composite structures assume for that test case. Test cases may define their own test configurations as part of their dedicated composite structure (e.g. in case the stereotype «TestCase» is applied on an instance of StructuredClassifier>, or it may operate on a shared «TestConfiguration» StructuredClassifier such as a Collaboration. If a «TestCase» Behavior invokes a «TestProcedure» Behavior, the invoked test procedure has to operate on the same or a compatible test configuration. The pre- and postconditions of a test case are always declared by the Behavior with «TestCase» applied by means of the underlying UML capability that each Behavior may contain a number of Constraints as pre- and postconditions. A test case must be parameterizable. This feature is also determined by the Behavior with «TestCase» applied. Again, the underlying capability of a UML Behavior is reused by UTP. A test case may only invoke test procedures as main, setup or teardown part or ordinary procedures. A test case must invoke

	at least one test procedure as its main part. This can be either done explicitly using the stereotype «ProcedureInvocation» or by using the underlying native UML elements for Behavior invocation (e.g., CallBehaviorAction, InteractionUse, BehaviorExecutionSpecification etc.) If a native UML Behavior invocation element is used and refers to a Behavior with «TestProcedure» applied, it is semantically equivalent with explicitly applying the stereotype «ProcedureInvocation» on the UML Behavior invocation element and setting the tagged value of role to main. Any procedural element that is directly contained in Behavior with «TestCase» applied is considered semantically equivalent to an explicit Behavior with «TestProcedure» applied that contains the procedural element and the use of «ProcedureInvocation» within the «TestCase» instead of the procedural elements. This ensures flexibility and guarantees simplicity when defining test cases. The semantics of test case depends on the arbitration specification that is applied to the test case. The default arbitration specification semantics of a test case is prescribed by the default precedence rules of the predefined verdict instances: none < pass < inconclusive < fail < fault < error. This is similar to the precedence rules of verdict calculation of [TTCN3].
Graphical syntax	
Detailed semantics	The preconditon of a test case are either represented by the preconditions of the classifier behavior of the UseCase, or by the preconditions of the Behavior directly. The postcondition of a test case are either represented by the postconditons of the classifier behavior of the UseCase, or by the postconditions of the Behavior directly. The formal parameters are either represented by the parameters of the classifier behavior of the UseCase, or by the parameters of the Behavior directly. The postcondition of a test case are either represented by the postconditons of the classifier behavior of the UseCase, or by

	the postconditions of the Behavior directly. The formal parameters are either represented by the parameters of the classifier behavior of the UseCase, or by the parameters of the Behavior directly. The test procedure of a test case is either represented as the classifier behavior of the UseCase (if applied to a UseCase) or directly via the respective behavioral elements of a concrete Behavior metaclass (e.g. InteractionFragments in case of Interactions, States and Transitions in case of a StateMachine) A test case may contain Properties stereotyped by «TestItem» or «TestComponent». In that case the test case specifies its own test configuration without the need for applying the «TestConfiguration» to itself.
Extension	Behavior, BehavioredClassifier
Attributes	ID : String [0..1] A unique identifier to unambiguously distinguish between any two test cases. This is mainly intended to interface easier with management tools such as test management tools.
	description : String [0..1] Usually, a narrative description of the given test case.
Associations	/utilizedBy : TestContext [*]
	/realizes : TestRequirement [*] The test requirements that are realized by the given test case. They are derived from the set of UML Realization dependencies that point from the base BehavioredClassifier to UML Classes stereotyped by «TestRequirement».
	: TestSet [0..1]
	: TestCaseLog [*]
	testCaseAS : TestCaseArbitrationSpecification [0..1] Refers to the explicit static test case arbitration specification that overrides the implicit default test case arbitration specification.

Constraints	Nested Classifier not allowed
	
	/* A Behavior with «TestCase» applied must not nest any other Behavior that has «TestCase» applied. */
	
	`context TestCase inv:`
	`self.base_Behavior <> null implies self.base_Behavior.nestedClassifier->isEmpty()`

	Owned UseCases not allowed
	
	/* A BehavioredClassifier or Behavior with «TestCase» applied must not own UseCases with «TestCase» applied. */
	
	`context TestCase inv:`
	`self.base_Behavior.ownedUseCase.getAppliedStereotype('UTP::TestCase')->oclIsUndefined()`
	`and`
	`self.base_BehavioredClassifier.ownedUseCase.getAppliedStereotype('UTP::TestCase')->oclIsUndefined()`

	Each test case returns a verdict statement /* Any Behavior stereotyped as «TestCase» always returns a ValueSpecification typed by verdict at runtime. */

	Use of BehavioredClassifier
	
	/* If «TestCase» is applied on BehavioredClassifier, the classifierBehavior of this BehavioredClassifier if set must also have «TestCase» applied. */
	
	`context TestCase inv:`
	`not self.base_BehavioredClassifier.classifierBehavior.getAppliedStereo-type('UTP::TestCase').oclIsUndefined()`

	Allowed invocation scheme /* A TestCase must only invoke TestProcedure or procedures, but not other TestCases or TestExecutionSchedule. */

	One precondition per test execution schedule
	
	/* DRTC02: It is necessary that each test execution schedule requires at most one preconditon. */
	
	`context TestExecutionSchedule inv:`
	`      self.base_Behavior.precondition->size()    =0`
	`xor self.base_Behavior.precondition->size() = 1`

	One precondition per test case
	
	/* DRTC03: It is necessary that each test case requires at most one preconditon. */
	
	`context TestCase inv:`
	`      self.base_Behavior.precondition->size()    =0`
	`xor self.base_Behavior.precondition->size() =1`

	One precondition per test procedure
	
	/* DRTC04: It is necessary that each test procedure requires at most one preconditon. */
	
	`context TestProcedure inv:`
	`      self.base_Behavior.precondition->size()  =  0`
	`xor self.base_Behavior->size() =1`

	One postcondition per test execution schedule
	
	/* DRTC05: It is necessary that each test execution schedule guarantees at most one postcondition. */
	
	`context TestExecutionSchedule inv:`
	`      self.base_Behavior.postcondition->size()   =0`
	`xor   self.base_Behavior.postcondition->size()   =1`

	One postcondition per test case
	
	/* DRTC06: It is necessary that each test case guarantees at most one postcondition. */
	
	`context TestCase inv:`
	`      self.base_Behavior.postcondition->size() = 0`
	`xor  self.base_Behavior.postcondition->size()  =  1`

	One postcondition per test procedure
	
	/* DRTC07: It is necessary that each test procedure guarantees at most one postcondition. */
	
	`context TestProcedure inv:`
	`self.base_Behavior.postcondition->size() =0`
	`xor self.base_Behavior.postcondition->size() =1`

	Test case invokes one test procedure /* DRTC01: It is necessary that each test case invokes at least one test procedure. */
	Test execution schedule must not invoke test procedure /* DRTC08: It is impossible that a test execution schedule invokes a test procedure. */
Change from UTP 1.2	Changed from UTP 1.2. «TestCase» extended Behavior and Operation in UTP 1.2.

8.5.1.2.3 TestExecutionSchedule

Description	TestExecutionSchedule: A procedure that constrains the execution order of a number of test cases. A test execution schedule is a Behavior with «TestExecutionSchedule» applied that schedules the execution order of a number of TestCases. A test execution schedule can be either defined standalone or related to one or more test sets. If a test execution schedule is related to a test set, the test execution schedule is only allowed to schedule the execution of test cases that belong to its related test set. This holds true, even if many test sets share the same test execution schedule. However, it is possible, due to the semantics of Behavior, to specialize, invoke or redefine test execution schedules. This enables the composition and decomposition of test execution schedules, which, in turn, fosters reusability. A standalone test execution schedule has the same semantics like defining a test set that owns the test execution schedule and assembles all the test cases scheduled for execution by the standalone test execution schedule. Standalone test execution schedules may specialize or invoke non-standalone test execution schedules. However, the semantics of the standalone test execution schedule remains the same. A test execution schedule may produce a test set verdict, calculated by an implicit or explicit arbitration specification for that test execution schedule. By default, a test execution schedule is not supposed to deliver a test set verdict. However, if it is desired to calculate such a test set verdict based on the test case verdicts of the test cases invoked by the given test execution schedule, the attribute isArbitrated must be set to true. This means that at run-time the effective arbitration specification for the test execution schedule will be determined by the execution entity and eventually activated. The eventual test set verdict is then calculated according to the effective arbitration specification. A test execution schedule may invoke other test execution schedules, test cases or auxiliary procedures (e.g., to retrieve required test data), however, a test execution schedule is not allowed to invoke a test procedure directly (see

«ProcedureInvocation» for further information on the allowed invocation schemes). Invocation of Behaviors relies on the underlying UML concepts for invoking Behaviors. These are for Activities and StateMachines CallBehaviorAction, StartObjectBehaviorAction and StartClassifierBehaviorAction, and for Interactions InteractionUse. If such an invocation element is stereotyped with «ProcedureInvocation», and part of a «TestExecutionSchedule» Behavior, e.g., such as an Activity, the following Behaviors can be invoked:

- Behaviors with «TestExecutionSchedule» applied: Useful for decomposing and reusing test execution schedules. If the user assigns a ProcedurePhaseKind to the invoked «TestExecutionSchedule», it will not have an effect.
- Behaviors with «TestCase» applied: Useful for decomposing and reusing test cases. If the user assigns a ProcedurePhaseKind to the invoked «TestCase», it will not have an effect.
- Behaviors without «TestExecutionSchedule», «TestCase» or «TestProcedure» applied: Such a Behavior invoked by a «ProcedureInvocation» is considered as auxiliary Behavior required to prepare the execution of succeeding «TestExecutionSchedules», and thus, «TestCase». The user may mark the invoked Behavior as setup or teardown activity by means of the role attribute.

In the last case, a role might be assigned to an invoked Behavior. This role is either of setup or teardown. If the role main is assigned, it will not have an effect. Behaviors executed as setup or teardown Behaviors will not be arbitrated by a corresponding arbitration specification. The meaning of the ProcedurePhaseKind in the context of an test execution schedule are as follows:

- Setup: A means to declare that the executed Behavior is responsible to prepare the execution of succeeding arbitrated test cases contained in that test execution schedule. UTP does not prescribe which verdict will be assigned in case something goes wrong while executing the setup phase of an arbitrated test execution schedule.
- Teardown: A means to declare that the executed Behavior is responsible to clean-up after the arbitrated

	test cases of this test execution schedule have been executed. UTP does not prescribe which verdict will be assigned in case something goes wrong while executing the teardown phase.
Extension	Behavior
Attributes	ID : String [0..1] A unique identifier to unambiguously distinguish between any two test execution schedules. This is mainly intended to interface easier with management tools such as test management tools.
	isArbitrated : Boolean [1] = false Indicated whether an executing entity must assign a test set verdict to the test execution schedule in the end. Default is 'false'.
Associations	testSetAS : TestSetArbitrationSpecification [0..1] Refers to the explicit static test set arbitration specification that overrides the implicit default test set arbitration specification. An explicit test set arbitration specification has only an effect, if the attribute *isArbitrated* is set to *true*.
Constraints	Allowed invocation scheme /* A TestExecutionSchedule must only invoke other TestExecutionSchedule, TestCases or procedures, but not TestProcedures directly. */
	An arbitrated test execution schedule returns a verdict statement /* If a TestExecutionSchedule is supposed to be arbitrated by explicitly setting the value of isArbitrated to *true*, the underlying Behavior stereotyped with «TestExecutionSchedule» must have a return parameter typed by verdict at runtime. */
	Owned UseCases not allowed /* Behavior with «TestExecutionSchedule» applied must not own UseCases with «TestExecutionSchedule» applied. */

	`context TestExecutionSchedule inv: self.base_Behavior.ownedUseCase.getAppliedStereotype('UTP::TestExecutionSchedule')->oclIsUndefined()`
Change from UTP 1.2	«TestExecutionSchedule» has been newly introduced by UTP 2. It was conceptually represented as the classifier behavior of a «TestContext» in UTP 1.2.

8.5.2 Procedural Elements

Procedural elements constitute the building blocks of procedures and test procedures. They can be realized by any building block of UML Behaviors (e.g., InteractionFragments in case of Interactions, Actions in case of Activities and Transitions/Vertices in case of StateMachines). The stereotypes for procedural elements reflect the minimal language concepts that are deemed necessary for testers to specify test-specific procedures. Each procedural element in a test-specific procedure has an effective arbitration specification assigned that delivers a procedural element verdict to the surrounding arbitration specification at runtime.

Since the UML Behavior building blocks outnumber the UTP procedural elements, test-specific procedures may consist of more than just the few predefined procedural elements. CombinedFragments of Interactions, for example, offer more than just the four predefined compound procedural elements of UTP. Such a plain UML Behavior building block provides implicitly the predefined verdict instances *inconclusive* to the surrounding arbitration specification. This default semantics can be overridden by means of «OpaqueProceduralElement».

Note: The different kinds of arbitration specification as well as the definition of the effective arbitration specification are stated in greater detail in the respective section.

In general, UTP provides the following procedural elements out of the box:

- procedural element represented by the abstract stereotype «ProceduralElement»
- atomic procedural element represented by the abstract stereotype «AtomicProceduralElements»
- compound procedural element represented by the abstract stereotype «CompoundProceduralElement»
- opaque procedural element represented by the stereotype «OpaqueProceduralElement»

Specialized compound procedural elements comprises:

- loop represented by the stereotype «Loop»

- sequence represented by the stereotype «Sequence»
- parallel represented by the stereotype «Parallel»
- alternative represented by the stereotype «Alternative»
- negative represented by the stereotype «Negative»
- procedure invocation represented by the stereotype «ProcedureInvocation»

Specialized atomic procedural elements are described by the test-specific actions (see section Test-specific Actions).

The procedural elements have been introduced by UTP to offer a harmonized view on technically different UML behavioral building blocks.

8.5.2.1 Procedural Elements Overview

The following diagram shows the abstract syntax of the core procedural elements.

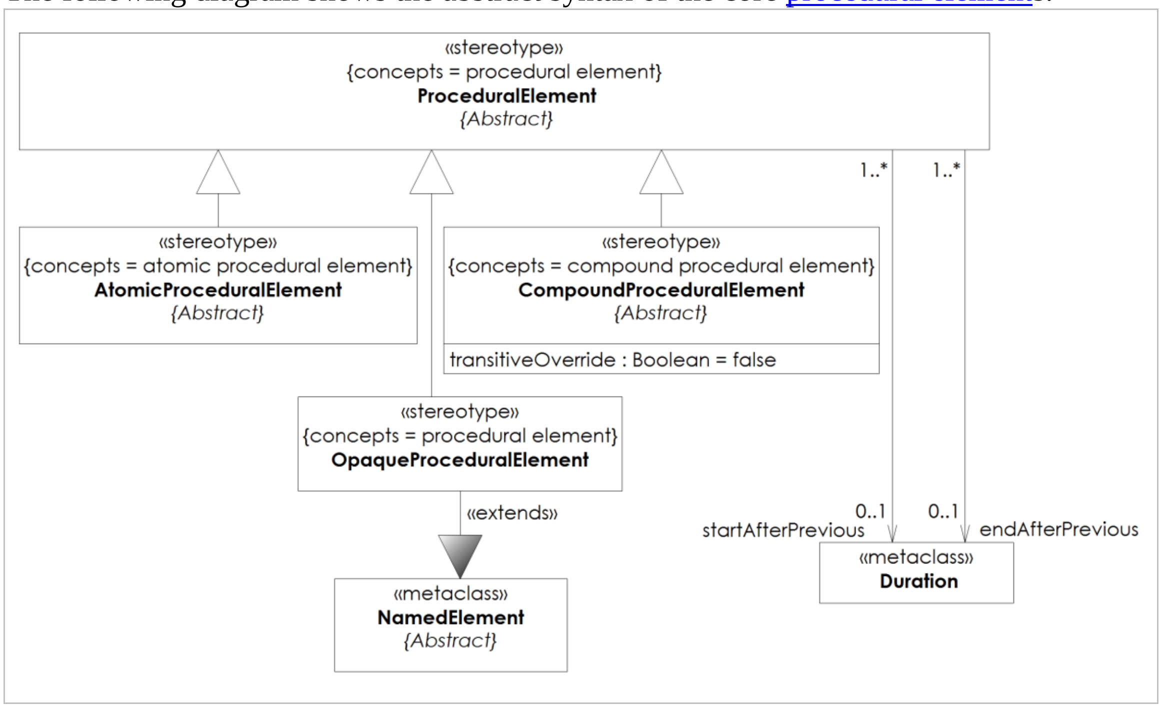


Figure 26 - Procedural Elements Overview

8.5.2.2 Compound Procedural Elements Overview

The following diagram shows the abstract syntax of the compound procedural elements.

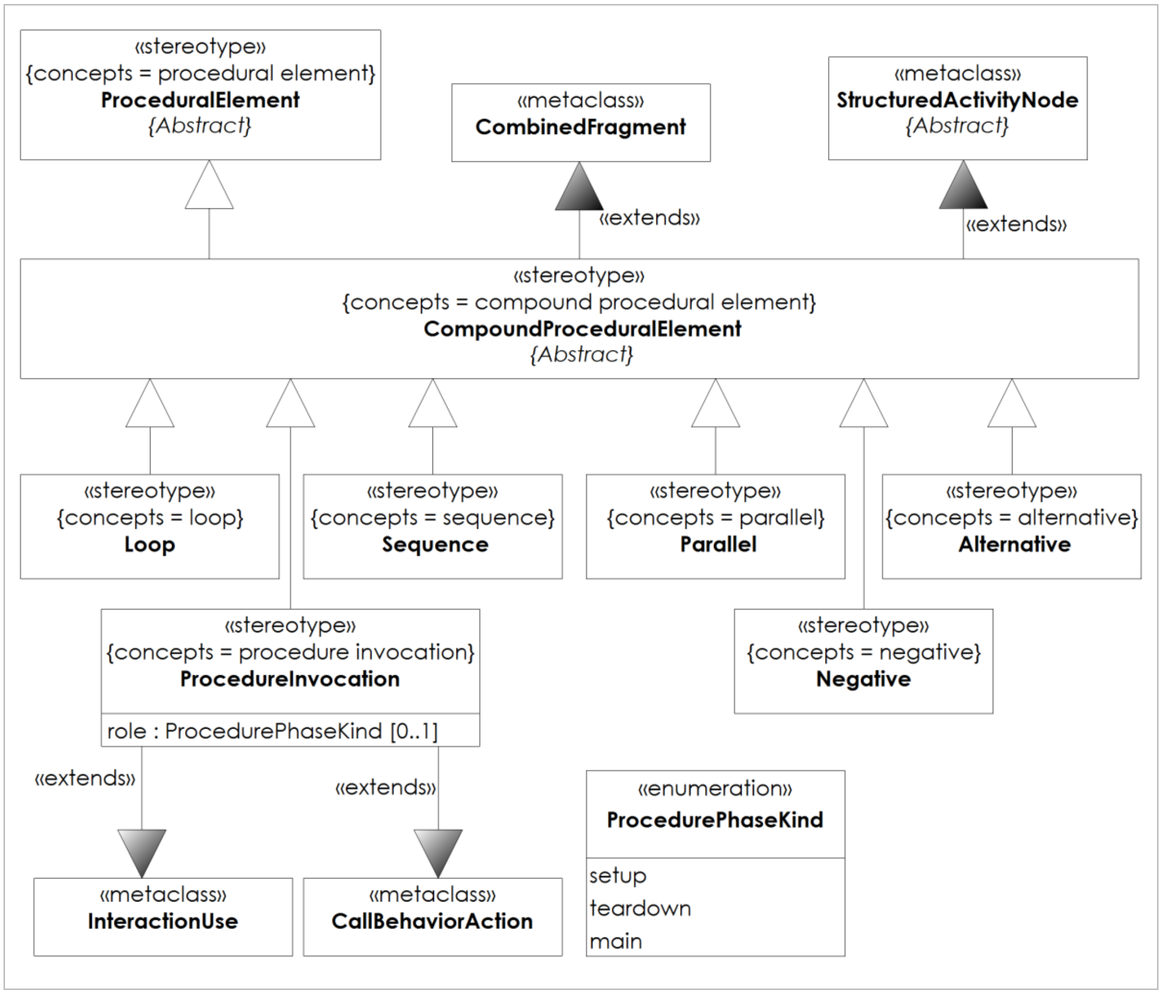


Figure 27 - Compound Procedural Elements Overview

8.5.2.3 Stereotype Specifications

8.5.2.3.1 Alternative

Description	Alternative: <Error: Could not resolve item> If «Alternative» is applied to CombinedFragement, the underlying CombinedFragment must have the InteractionOperatorKind *alt* or *opt* set. In an Activity, «Alternative» must only be applied to ConditionalNode.
Extension	CombinedFragment, StructuredActivityNode
Super Class	CompoundProceduralElement

Associations	arbitrationSpecification {redefines arbitrationSpecification} : AlternativeArbitrationSpecification [0..1] ——— Refers to an alternative arbitration specification that overrides the default and implicit arbitration specification, if set. It redefines the Property *arbitrationSpecification* of CompoundProceduralElement.
Constraints	Application in Interactions ——— /* If «Alternative» is applied to CombinedFragment, the underlying CombinedFragment must have the InteractionOperatorKind *alt* or *opt* set. */ context Alternative inv: self.base_CombinedFragment.interactionOperator = uml::InteractionOperatorKind::alt or self.base_CombinedFragment.interactionOperator = uml::InteractionOperatorKind::opt
	Application in Activities ——— /* In an Activity, «Alternative» must only be applied to ConditionalNode. */ context Alternative inv: self.base_StructuredActivityNode.oclIsTypeOf(uml::ConditionalNode)
Change from UTP 1.2	«Alternative» has been newly introduced by UTP 2.

8.5.2.3.2 AtomicProceduralElement

Description	AtomicProceduralElement: A procedural element that cannot be further decomposed. «AtomicProceduralElement» is an abstract stereotype that does not extend UML metaclass at all. This means that its substereotypes have to define suitable UML metaclass for extension. Atomic procedural elements resembles the semantics of UML Behavior building blocks that are not able to be further decomposed. Message and CallOperationAction are examples for concrete UML Behavior building block that adhere to the definition of atomic procedural element. In contrast, CombinedFragment or LoopNode are examples for compound procedural elements for they contain potentially further procedural elements.
Super Class	ProceduralElement
Sub Class	CheckPropertyAction, CreateLogEntryAction, CreateStimulusAction, ExpectResponseAction, ProcedureInvocation, SuggestVerdictAction
Associations	arbitrationSpecification {redefines arbitrationSpecification} : AtomicProceduralElementArbitrationSpecification [0..1] Refers to an atomic arbitration specification that overrides the default and implicit arbitration specification if set. It redefines the Property *arbitrationSpecification* of procedural element.
Change from UTP 1.2	«AtomicProceduralElement» has been newly introduced by UTP 2.

8.5.2.3.3 CompoundProceduralElement

Description	CompoundProceduralElement: A procedural element that can be further decomposed. «CompoundProceduralElement» is an abstract stereotype that extends CombinedFragment and StructuredActivityNode to interface with the UML Behaviors Interaction and Activity. A compound procedural element resembles the semantics of UML Behavior building blocks that consist of other procedural element. As such, it may obtain the verdicts of its contained executed procedural elements in order to calculate its own procedural element verdict. The difference between an atomic procedural element verdict and compound procedural element verdict is that the latter is potentially composed out of multiple atomic procedural element verdicts. As a container of other procedural elements, compound procedural element offers the possibility to dynamically override the arbitration specifications of its contained procedural elements by means of the tag definition *dynamicArbitrationSpecification*. This is similar to the way, procedure invocation offers dynamic arbitration specification override. If the tag definition *transitiveOverride* is set to *true* (default is *false*) the arbitration specification for dynamic override transitively override the arbitration specification of nested compound procedural elements or test procedure invoked by means of procedure invocation.
Extension	CombinedFragment, StructuredActivityNode
Super Class	ProceduralElement
Sub Class	Alternative, Loop, Negative, Parallel, ProcedureInvocation, Sequence
Attributes	transitiveOverride : Boolean [1] = false Indicates whether the override of arbitration specifications will be transitively propagated by nested compound procedural element or procedure invocations. Default is 'false'.
Associations	arbitrationSpecification {redefines arbitrationSpecification} : CompoundProceduralElementArbitrationSpecification [0..1] Refers to a compound procedural element arbitration specifi-

	cation that overrides the default and implicit arbitration specification for compound procedural elements.
	overridingArbitrationSpecification : ArbitrationSpecification [*] Specifies a set of arbitration specifications that override the default and static arbitration specifications of contained procedural elements or invoked test procedures.
Change from UTP 1.2	«CompoundProceduralElement» has been newly introduced by UTP 2.

8.5.2.3.4 Loop

Description	Loop: A compound procedural element that repeats the execution of its contained procedural elements. If «Loop» is applied to CombinedFragement, the underlying CombinedFragment must have the InteractionOperatorKind loop set. In an Activity, «Loop» must only be applied to LoopNode. The stereotype «Loop» can be omitted, if no static or dynamic arbitration specification is set. In other words, every instance of CombinedFragement with loop and every instance of LoopNode without «Loop» as part of a test procedure is considered semantically equivalent to the application of «Loop» but without setting any tagged values. The nature of the loop (i.e., counter-controlled loop, conditional-controlled loop or collection-controlled loop) is determine by the configuration of the underlying UML element for expressing loops.
Extension	CombinedFragment, StructuredActivityNode
Super Class	CompoundProceduralElement
Associations	arbitrationSpecification {redefines arbitrationSpecification} : LoopArbitrationSpecification [0..1] Refers to a loop arbitration specification that overrides the

	default and implicit arbitration specification if set. It redefines the Property *arbitrationSpecification* of CompoundProceduralElement.
Constraints	Application in Interactions /* If «Loop» is applied to CombinedFragment, the underlying CombinedFragment must have the InteractionOperatorKind *loop* set. */ context Loop inv: self.base_CombinedFragment.interactionOperator = uml::InteractionOperatorKind::loop
	Application in Activities /* In an Activity, «Loop» must only be applied to LoopNode. */ context Loop inv: self.base_StructuredActivityNode.oclIsTypeOf(uml::LoopNode)
Change from UTP 1.2	«Loop» has been newly introduced by UTP 2.

8.5.2.3.5 Negative

Description	Negative: A compound procedural element that prohibits the execution of its contained procedural elements in the specified structure. If «Negative» is applied to CombinedFragement, the underlying CombinedFragment must have the InteractionOperatorKind *neg* set. In an Activity, «Negative» must only be applied to StructuredActivityNode.
Extension	CombinedFragment, StructuredActivityNode
Super Class	CompoundProceduralElement
Associations	arbitrationSpecification {redefines arbitrationSpecification} :

	NegativeArbitrationSpecification [0..1]
Constraints	Application in Interactions /* If «Negative» is applied to CombinedFragment, the underlying CombinedFragment must have the InteractionOperatorKind *neg* set. */ context Negative inv: self.base_CombinedFragment.interactionOperator = uml::InteractionOperatorKind::neg
	Application in Activities /* In an Activity, «Negative» must only be applied to StructuredActivityNode. */ context Negative inv: self.base_StructuredActivityNode.oclIsTypeOf(uml::StructuredActivityNode)
Change from UTP 1.2	«Negative» has been newly introduced by UTP 2.

8.5.2.3.6 OpaqueProceduralElement

Description	«OpaqueProceduralElement» adds the possibility to assign arbitration specifications to UML Behavior building blocks that are not covered by UTP procedural elements. Thus, it is a plain technical stereotype introduced for flexibility of UTP. Similar to the semantics of opaque elements in UML (i.e., OpaqueBehavior, OpaqueExpression, OpaqueAction), there is no additional semantics for «OpaqueProceduralElement» given apart from the ability to assign arbitration specifications to UML elements for which no dedicated procedural element stereotype has been defined.
Extension	NamedElement
Super Class	ProceduralElement
Constraints	Only applicable to UML Behavior building blocks

	/* «OpaqueProceduralElement» must only be applied on instances of the UML metaclass Action, InteractionFragment, Vertex and Transition. */
	
	`context OpaqueProceduralElement inv:`
	`self.base_NamedElement.oclAsType(uml::Action)->size() =1`
	`xor`
	`self.base_NamedElement.oclAsType(uml::InteractionFragment) ->size() =1`
	`or`
	`self.base_NamedElement.oclAsType(uml::Vertex)->size() =1`
	`or`
	`self.base_NamedElement.oclAsType(uml::Transition)->size() =1`

Change from UTP 1.2	«OpaqueProceduralElement» has been newly introduced by UTP 2.

8.5.2.3.7 Parallel

Description	Parallel: <Error: Could not resolve item> If «Parallel» is applied to CombinedFragement, the underlying CombinedFragment must have the InteractionOperatorKind *par* set. If used in Activities, the metaclass ConditionalNode is reused to describe parallel execution of procedural elements (i.e., ExecutableNodes). The branches that must be executed in parallel are defined by the Clauses that are contained in a ConditionalNode with «Parallel» applied. If such a ConditionalNode is activated and ready for execution, the evaluation of the Clauses by executing the test parts are executed as described by UML. In contrast to a plain ConditionalNode, where at most one Clause's body part will be executed, even if more than one Clause's test part eventually enabled the Clause, all enabled Clause's body parts are executed in parallel, if the ConditionalNode has «Parallel» applied.
Extension	CombinedFragment, StructuredActivityNode
Super Class	CompoundProceduralElement

Associations	arbitrationSpecification {redefines arbitrationSpecification} : ParallelArbitrationSpecification [0..1] Refers to a parallel arbitration specification that overrides the default and implicit arbitration specification if set. It redefines the Property *arbitrationSpecification* of CompoundProceduralElement.
Constraints	Application in Interactions /* If «Parallel» is applied to CombinedFragment, the underlying CombinedFragment must have the InteractionOperatorKind *par* set. */ context Parallel inv: self.base_CombinedFragment.interactionOperator = uml::InteractionOperatorKind::par
	Application in Activities /* In an Activity, «Loop» must only be applied to SequenceNode */ context Parallel inv: self.base_StructuredActivityNode.oclIsTypeOf(uml::SequenceNode)
Change from UTP 1.2	«Parallel» has been newly introduced by UTP 2.

8.5.2.3.8 ProceduralElement

Description	ProceduralElement: An instruction to do, to observe, and/or to decide. «ProceduralElement» is an abstract stereotype that does not extend any UML metaclass. This means that its substereotypes have to define suitable UML metaclasses for extension. A procedural element is the lowest common denominator for the building blocks of the different UML Behaviors. If used as constituting part (possibly transitively) of a test case execution, every procedural element delivers a verdict depending on both the execution of the respective procedural element and the effective arbitration specification of that procedural element. Every procedural element has an effective arbitration specification assigned at runtime determined either implicitly (default arbitration specification), explicitly (static arbitration specification) or dynamically (dynamic arbitration specification). The default arbitration specification for each procedural element is defined by the respective substereotypes. If no explicit value is set for the tag definition *arbitrationSpecification*, the default arbitration specification (prescribed by UTP for that specific procedural element) is set. However, a procedural element may override the default arbitration specification with an explicit one by assigning a concrete arbitration specification to the procedural element via the tag definition *arbitrationSpecification*. If such an arbitration specification is explicitly set for a procedural element, it becomes the effective arbitration specification. A procedural element adds the ability to specify the expected starting and end point of the execution of procedural element related to a previously executed procedural element, represented by the tag definitions *startAfterPrevious* and *endAfterPrevious*. These timing-related characteristics are represented by means of explicit tag definitions in addition to the existing simple time concepts of UML and time-related information potentially available by further UML profiles such as MARTE. UTP 2 does not prescribe which of these timing-related concepts should be used. As a recommendation, users should not mix different mechanisms to express timing-related information.

	The semantics of a violation of these timing information regarding the delivery of the corresponding procedural element verdict of the executed procedural element is determined by the arbitration specification assigned to the procedural element.
Sub Class	AtomicProceduralElement, CompoundProceduralElement, OpaqueProceduralElement
Associations	arbitrationSpecification : ProceduralElementArbitrationSpecification [0..1] Refers to a procedural element arbitration specification that overrides the default and implicit arbitration specification for procedural elements.
	startAfterPrevious : Duration [0..1]
	endAfterPrevious : Duration [0..1]
Constraints	Valid duration /* DRTP01: It is necessary that the PE start duration of a procedural element is smaller than the PE end duration of the same procedural element. */
Change from UTP 1.2	«ProceduralElement» has been newly introduced by UTP 2.

8.5.2.3.9 ProcedureInvocation

Description	ProcedureInvocation: An atomic procedural element of a procedure that invokes another procedure and wait for its completion. The stereotype «ProcedureInvocation» specializes «CompoundProceduralElements» and is intended to invoke test-specific procedures from within test-specific procedures and adds the ability to assign arbitration specifications dynamically. Since the constituents of UML Behaviors are not based on an integrated metaclass, the concrete metaclasses for «ProcedureInvocation» depend on the Behavior kind in which the «ProcedureInvocation» is used. If it represents a building block of an Activity or StateMachine, «ProcedureInvocation» must only be applied on the metaclass CallBehaviorAction, StartObjectBehaviorAction or StartClassifierBehaviorAction. If it represents a building block of an Interaction, «ProcedureInvocation» must only be applied on the metaclass InteractionUse. The allowed invocation scheme for a «ProcedureInvocation» is as follows: • If it constitutes a procedural element of a test execution schedule, only test execution schedules, test cases or procedures must be invoked. • If it constitutes a procedural element of a test case, only test procedures and procedures must be invoked. • If it constitutes a procedural element of a test procedure, only test procedure or procedures must be invoked. If procedure invocation is part of a test case it must assign a role to the invoked test procedure. This role is either *main*, *setup* or *teardown*. The semantics of these roles in UTP are: • main: A test procedure that implements the reason why the invoking test case has been designed, i.e., it contribute to the coverage of a test objective or test requirement. The main part of a test case is relevant for calculating coverage and controlling the progress. The default semantics of a main test procedure with respect to verdict calculation is defined by the default UTP ar-

	bitration specification, but can be easily overridden by the user. • setup: A means to declare that the executed test procedure is responsible to prepare the main part of a test case. UTP does not prescribe which verdict must be assigned in case something goes wrong while executing the setup phase. This is up to the effective arbitration specification assigned to the invoked test procedure, and thus, can be easily changed by the user using the overriding arbitration specification capability. • teardown: A means to declare that the executed test procedure is responsible to clean-up after the main part of a test case has been executed. UTP does not prescribe which verdict must be assigned in case something goes wrong while executing the teardown phase. This is up to the effective arbitration specification assigned to the invoked test procedure, and thus, can be easily changed by the user using the overriding arbitration specification capability. If procedure invocation is part of a test execution schedule it may assign a role to an invoked Behavior. This role is either of *setup* or *teardown*. The semantics of these roles in UTP are: • setup: A means to declare that the executed Behavior is responsible to prepare the execution of arbitrated test cases contained in that test case. UTP does not prescribe which verdict must be assigned in case something goes wrong while executing the setup phase of an arbitrated test execution schedule. This is up to the effective arbitration specification for the invoked test procedure and, thus, can be easily changed by the user using the overriding arbitration specification capability. • teardown: A means to declare that the executed Behavior is responsible to clean-up after the arbitrated test cases of this test execution schedule have been executed. UTP does not prescribe which verdict must be assigned in case something goes wrong while executing the teardown phase. This is up to the effective arbitration specification for the invoked test procedure and, thus, can be easily changed by the user using the overriding arbitration specification capability.

	A procedure invocation enables the user to override the default or static arbitration specifications defined for the procedural elements owned by the invoked test-specific procedure by means of the tag definition *dynamicArbitrationSpecification*. If no such overriding arbitration specification is set, the test-specific procedure will be invoked with its genuine arbitration specifications (i.e., either the default or explicitly stated ones). The set of dynamic arbitration specifications must not contain two identic arbitration specification types in order to prevent ambiguities when executing the invoked test-specific procedure. If two identic arbitration specification types (e.g., two different BehavioredClassifier with «ExpectedResponseArbitrationSpecification» applied and with varying semantics) were supplied, it would not be clear which of these two or more arbitration specifications eventually represent the effective arbitration specification. If an invoked test procedure may invoke other test procedures by means of «ProcedureInvocation», it is possible to declare that the dynamic arbitration specifications must also be propagated to the nested test procedure. This can be achieved by setting the tag definition *transitiveOverride* to *true* (default is *false*). The types of arbitration specification that can be dynamically overridden by means of «ProcedureInvocation» depend on the invoked test-specific procedure: • Test set or test execution schedule: test set arbitration specification, test case arbitration specification; procedural elements or arbitration specifications can be assignment via «ProcedureInvocation», if the *transitiveOverride* attribute of the «ProcedureInvocation» stereotype is set to *true*. If so, any procedure invocation of test execution schedule that invokes a test case propagates those procedural element arbitration specifications, in addition to the test case arbitration specification. • Test case: test case arbitration specification and procedural element arbitration specification • Test procedure: procedural element arbitration specification Any other combination of arbitration specification and test-

	specific procedure must not be considered for the calculation of the effective arbitration specifications.
Extension	CallBehaviorAction, CombinedFragment, InteractionUse, StructuredActivityNode
Super Class	AtomicProceduralElement, CompoundProceduralElement
Attributes	role : ProcedurePhaseKind [0..1] The role, the invoked procedure assumes within the invoking test-specific procedure.
Associations	arbitrationSpecification {redefines arbitrationSpecification} : ProcedureInvocationArbitrationSpecification [0..1] Refers to a procedure invocation arbitration specification that overrides the default and implicit arbitration specification if set. It redefines the Property *arbitrationSpecification* of «CompoundProceduralElement».
Constraints	Role only in context of test cases relevant /* If «ProcedureInvocation» is part of a «TestProcedure» Behavior, the tag definition role must be empty. If it is empty, it will be ignored. */
	No overlapping dynamic arbitration specification /* If dynamic arbitration specifications are set they must be only one instance of each ArbitrationSpecification given. */
Change from UTP 1.2	«ProcedureInvocation» has been newly introduced by UTP 2.

8.5.2.3.10 Sequence

Description	Sequence: A compound procedural element that executes its contained procedural elements sequentially. If «Sequence» is applied to CombinedFragement, the underlying CombinedFragment must have the InteractionOperatorKind *strict* or *seq* applied. In an Activity, «Sequence» must only be applied to SequenceNode.
Extension	CombinedFragment, StructuredActivityNode
Super Class	CompoundProceduralElement
Associations	arbitrationSpecification {redefines arbitrationSpecification} : SequenceArbitrationSpecification [0..1] Refers to a SequenceArbitrationSpecification that overrides the default and implicit ArbitrationSpecification if set. It redefines the Property *arbitrationSpecification* of CompoundProceduralElement.
Constraints	Application in Interactions /* If applied on a CombinedFragment, the underlying CombinedFragment must have set InteractionOperatorKind::seq or InteractionOperatorKind::strict as the *interactionOperator*. */ context Sequence inv: self.base_CombinedFragment.interactionOperator = uml::InteractionOperatorKind::seq or self.base_CombinedFragment.interactionOperator = uml::InteractionOperatorKind::strict
	Application in Activities /* If applied on a StructuredActivityNode, the StructuredActivityNode must be a SequenceNode. */ context Sequence inv: self.base_StructuredActivityNode.oclIsTypeOf(uml::SequenceNode)
Change from UTP 1.2	«Sequence» has been newly introduced by UTP 2.

8.5.3 Test-specific Actions

UTP introduces dedicated test-specific actions that denote actions a tester, regardless whether this is an automated or human tester, can carry out in order to communicate with the test item. In context of dynamic testing, communicating with a test item either means to stimulate the test item with a create stimulus action (implemented as stereotype «CreateStimulusAction») or observing and evaluating its actual responses with the expected ones (represented by the stereotypes «ExpectResponseAction», «CheckPropertyAction»).

Test-specific actions are specialized procedural elements. As such, they contribute a dedicated procedural element verdict to the eventual calculation of a test case or test set verdict. The test-specific actions can be categorized by the entity that contributes information to the calculation of the respective procedural element verdict.

The procedural element verdicts of the following test-specific actions are calculated by taking into consideration the information provided by the test component or tester. These test-specific actions are henceforth called test component controlled actions, because an erroneous execution of these test actions indicates a misbehavior of the test component (submitting the wrong stimulus, performing a test-specific action too late/too early) or technical issues in the test environment (e.g., breakdown of connectivity etc.):

- Create stimulus action represented by the stereotype «CreateStimulusAction»
- Suggest verdict action represented by the stereotype «SuggestVerdictAction»
- Create log entry action represented by the stereotype «CreateLogEntryAction»

It is highly recommended that the verdicts calculated by these test component controlled actions should only result in the predefined verdict instances pass, fault or error.

The verdict of following test-specific actions is calculated by taken into consideration information received by the test items. These test-specific actions are henceforth called test item controlled actions, because the arbitration of these test-specific actions depend on the responses of the test items during execution and as such indicate deviations between the expected response and actual response:

- Expect response action represented by the stereotype «ExpectResponseAction»
- Check property action represented by the stereotype «CheckPropertyAction»

It is highly recommended that the test item controlled actions should only result in the predefined verdict instances pass, fail or inconclusive.

8.5.3.1 Test-specific actions Overview

The following diagram shows the abstract syntax of the test action.

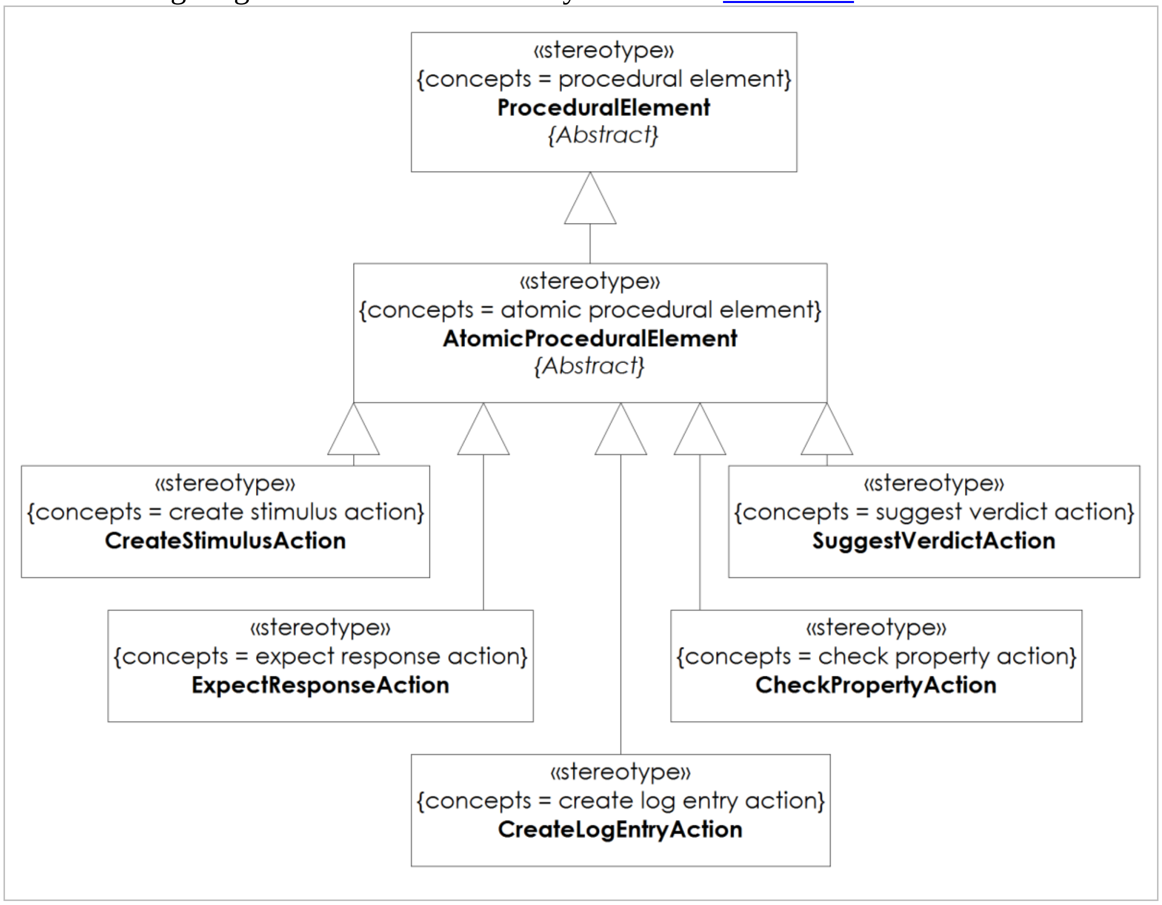


Figure 28 - Test-specific actions Overview

8.5.3.2 Tester Controlled Actions

The following diagram shows the details of the test component controlled test actions.

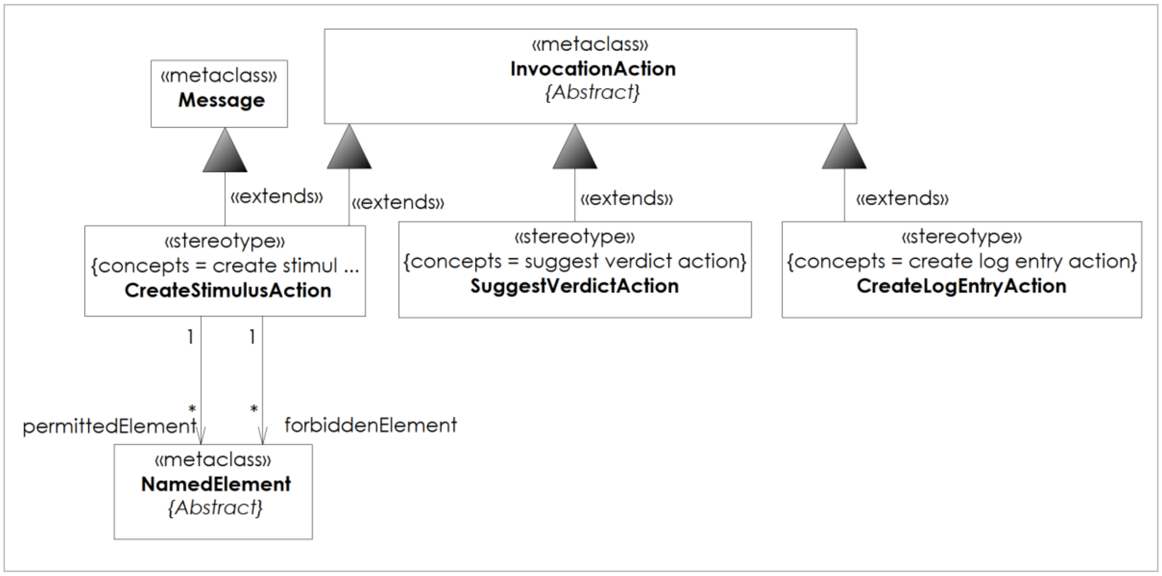


Figure 29 - Tester Controlled Actions

8.5.3.3 Test Item Controlled Actions

The following diagram shows the details of the test item controlled test actions.

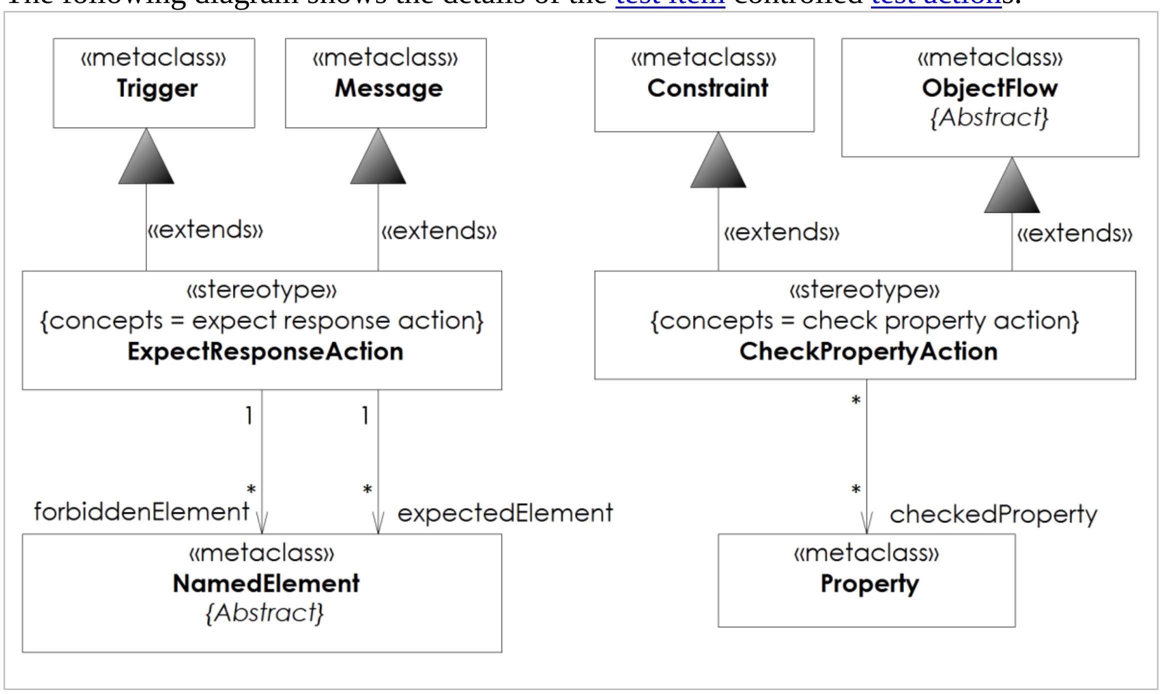
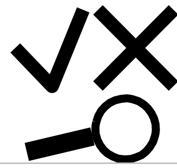


Figure 30 - Test Item Controlled Actions

8.5.3.4 Stereotype Specifications

8.5.3.4.1 CheckPropertyAction

Description	CheckPropertyAction: A test action that instructs the tester to check the conformance of a property of the test item and to set the procedural element verdict according to the result of this check. The stereotype «CheckPropertyAction» extends Constraint (for integration with Interaction's StateInvariant and StateMachines), and ObjectFlow (for integration with Activities) and enables the test component to check certain properties of the test item that cannot be checked via the publicly available or known APIs of the test item. Thus, it is not defined how the test component accesses the test item's property. If used in Interactions, check property action is used as Constraint of a StateInvariant that covers a test component. Such a Constraint must be contained by StateInvariants. The specification of the StateInvariant's «CheckPropertyAction» Constraint is intended to determine the Property of the test item that must be checked and the value the Property has to match with. As specification of the «CheckPropertyAction» Constraint, any kind of suitable ValueSpecification can be utilized. For example, the «CheckPropertyAction» Constraint may specify location expressions with OCL or Alf for declaring access and expected values of the test item's Property. If used in StateMachines, check property action is expressed as stateInvariant attribute of a State. Since the stateInvariant attribute is of type Constraint, the usage, application and semantics is similar to the check property action used in Interactions (i.e., use of StateInvariant in Interactions). If used in Activities, check property action is expressed as «CheckPropertyAction» ObjectFlow that emanates from a ReadStructuralFeatureAction and is used to access a StructuralFeature of the test item. The expected value of the checked Property is defined by the guard condition of the CheckPropertyAction» ObjectFlow. In addition, it is possible to point directly to the Property that will be checked by the check property action by means of the

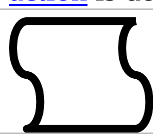
	tag definition checkedProperty. This information is helpful, if, for example, natural language is used to describe «CheckPropertyAction» Constraint. The default arbitration specification for the check property action is described by check property action AS.
Graphical syntax	
Extension	Constraint, ObjectFlow
Super Class	AtomicProceduralElement
Associations	arbitrationSpecification {redefines arbitrationSpecification} : CheckPropertyArbitrationSpecification [0..1] Refers to a check property action arbitration specification that overrides the default and implicit arbitration specification, if set. It redefines the Property *arbitrationSpecification* of test action.
	checkedProperty : Property [*] Refers to set of Properties of a test item that is supposed to be checked by the check property action.

Constraints	Owner of Constraint
	
	/* If applied on a Constraint, the owner of this Constraint must only be a State (referring to the Constraint as StateInvariant) or StateInvariant. */
	
	`context CheckPropertyAction inv:`
	`self.base_Constraint.owner.oclAsType(uml::State)->size() = 1`
	`xor`
	`self.base_Constraint.owner.oclAsType(uml::StateInvariant)->size() = 1`

	Owner of Property /* If checkedProperty is not empty, the referenced Property must belong to a TestItem participating in the current test-specific procedure. */ `context CheckPropertyAction inv:`

	`not self.checkedProperty.owner.getAppliedStereotype('UTP::TestItem')->oclIsUndefined()`
	At least one property /* DRTA03: It is necessary that a check property action checks at least one property of the test item against the data. */
Change from UTP 1.2	«CheckPropertyAction» has been newly introduced by UTP 2.

8.5.3.4.2 CreateLogEntryAction

Description	CreateLogEntryAction: A test action that instructs the tester to record the execution of a test action, potentially including the outcome of that test action in the test case log. The stereotype «CreateLogEntryAction» extends Invocation-Action which allows for using a variety of metaclasses for application. The create log entry action is a test action that instructs the tester or the test execution system to log certain information about the execution of a test case. This information is henceforth called content to be logged. The content to be logged has to be provided as the argument InputPin of the underlying InvocationAction. It is not specified how the variety of potentially logable contents is eventually be represented in the log. Test execution systems are responsible for eventually writing the content to be logged into the actual test log. If used in an Interaction, the InvocationAction that is stereotyped with «CreateLogEntryAction» should be referenced from an ActionExecutionSpecification that indirectly covers a Lifeline that represents a test component role in the underlying test configuration. Indirectly means that the corresponding start and end OccurenceSpecification of the ActionExecutionSpecification cover the test component lifeline. If used in Activities or StateMachines, e.g., CallOperationAction could be used to invoke a (not standardized, yet proprietary) logging interface operation. Another possibility is to use SendObjectAction without specifying the target Pin which has the semantics to submit the information to be logged to the logging facility of the test execution system without needing a dedicated interface. However, during test execution the create log entry action must be made executable and eventually carried out. This may include manually writing some information into a paper-based document. The default arbitration specification for the create log entry action is described by create log entry action AS.
Graphical syntax	

Extension	InvocationAction
Super Class	AtomicProceduralElement
Associations	arbitrationSpecification {redefines arbitrationSpecification} : CreateLogEntryArbitrationSpecification [0..1] Refers to a create log entry action arbitration specification that overrides the default and implicit arbitration specification if set. It redefines the Property *arbitrationSpecification* of test action.
Change from UTP 1.2	«CreateLogEntryAction» has been newly introduced by UTP 2.

8.5.3.4.3 CreateStimulusAction

Description	CreateStimulusAction: A test action that instructs the tester to submit a stimulus (potentially including data) to the test item. «CreateStimulusAction» extends Message (for integration with Interaction) and InvocationAction (for integration with Activities and StateMachines). The create stimulus action is performed by an instance of a test component and represents a set of possible invocations of the test item, potentially conveyed by a payload. Invocation means that either a BehavioralFeature of the test item is invoked (e.g. using a Message or a SendSignalAction respectively CallOperationAction) or by simply sending a stimulus to the test items (e.g., SendObjectAction or BroadcastSignalAction). The set of stimuli to be sent is derived from the arguments of the underlying UML element and the elements specified by the tag definition permittedElement. This set is then reduced by the elements to yield by forbiddenElement. If the set of stimuli is empty (i.e., neither the underlying UML element yields arguments nor the permittedElement tag definition yields an element), it is semantically equivalent to a situation where any possible and known by the invoking test component stimuli at this point in time can be send to the test item. This set of any possible and known stimuli is potentially reduced by the elements yield by forbiddenElement. In case the set of permitted elements and the set of forbidden elements are overlapping, the elements in the intersection belong to the set of forbidden elements. If both sets are empty, every known stimuli can be send to the test item. The default arbitration specification for the create stimulus action is described by create stimulus action AS.
Extension	InvocationAction, Message
Super Class	AtomicProceduralElement
Associations	arbitrationSpecification {redefines arbitrationSpecification} : CreateStimulusArbitrationSpecification [0..1] Refers to a create stimulus action arbitration specification that

	overrides the default and implicit arbitration specification if set. It redefines the Property *arbitrationSpecification* of test action.
	forbiddenElement : NamedElement [*] A set of elements that are explicitly removed from the set of stimuli to be sent.
	permittedElement : NamedElement [*] Additional set of stimuli that contribute to the set of permitted stimuli.

Constraints	Type of forbidden elements
	
	/* The tag definition forbiddenElement may only yield instances of the metaclasses Message, Event, Signal or InstanceSpecification of Signal, BehavioralFeature and Trigger. */
	
	`context CreateStimulusAction inv:`
	`self.forbiddenElement.oclAsType(uml::Message) ->size() =1`
	`xor`
	`self.forbiddenElement.oclAsType(uml::Event)->size() =1`
	`xor`
	`self.forbiddenElement.oclAsType(uml::Signal)->size() =1`
	`xor`
	`self.forbiddenElement.oclAsType(uml::InstanceSpecification) ->size() =1`
	`xor`
	`self.forbiddenElement.oclAsType(uml::BehavioralFeature) ->size() =1`
	`xor`
	`self.forbiddenElement.oclAsType(uml::Trigger) ->size() =1`

	Type of permitted elements
	
	/* The tag definition permittedElement may only yield instances of the metaclasses Message, Event, Signal or InstanceSpecification of Signal, BehavioralFeature and Trigger. */
	
	`context CreateStimulusAction inv:`
	`self.permittedElement.oclAsType(uml::Message) ->size() =1`
	`xor`
	`self.permittedElement.oclAsType(uml::Event)->size() =1`
	`xor`
	`self.permittedElement.oclAsType(uml::Signal)->size() =1`
	`xor`
	`self.permittedElement.oclAsType(uml::InstanceSpecification) ->size() =1`
	`xor`
	`self.permittedElement.oclAsType(uml::BehavioralFeature) ->size() =1`
	`xor`
	`self.permittedElement.oclAsType(uml::Trigger) ->size() =1`

	At least one stimulus /* DRTA01: It is necessary that a create stimulus action permits to send at least one stimulus. */
Change from UTP 1.2	«CreateStimulusAction» has been newly introduced by UTP 2.

8.5.3.4.4 ExpectResponseAction

Description	ExpectResponseAction: A test action that instructs the tester to check the occurrence of one or more particular responses from the test item within a given time window and to set the procedural element verdict according to the result of this check. The stereotype «ExpectResponseAction» extends Message (for integration with Interactions) and Trigger (for integration with StateMachines and Activities) and denotes the expectation of the test component to receive an actual response, potentially conveyed by some payload, from the test item at a certain point in time. In case of a Message, the expected response is defined by the Message's signature and its arguments, if any. If the signature is empty, the expect response action accepts and consumes any kind of actual response from the test item and continues execution. If more than one response types is expected at the same point in time, the tag definition expectedElement can be used to denote further expected responses in addition to the expected response denoted by the Message. The eventual number of expected responses is the union of the Message including the arguments and the entries obtained from the tag definition expectedElement, if any. In case of Trigger, the expected responses are the union of the MessageEvents obtained from the underlying Trigger and the expected responses yield by the expectedElement tag definition, if any. It is possible to mix Trigger with «ExpectResponseAction» and Trigger without «ExpectResponseAction» applied in Transitions or AcceptEventActions. A Trigger with «ExpectResponseAction» and AnyReceiveEvent set is semantically equivalent to except and consume any actual response. In both cases it is additionally possible to denote a set of elements that are explicitly not allowed to be received at that point in time. The set of not allowed responses is defined by the tag definition forbiddenElement. In case of overlapping sets of expected and forbidden elements, the forbidden elements have higher priority, i.e., as

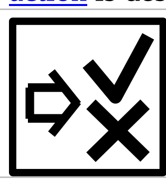
	soon as an element is both expected and forbidden, it is by definition forbidden. In situations where the set of expected elements is expressed by means of a generic UML term, i.e., AnyReceiveEvent or an «ExpectResponseAction» Message without a signature, the elements yield by the forbiddenElement tag definition are excluded from the set of expected responses. The default arbitration specification for the expect response action is described by expect response action AS.
Extension	Message, Trigger
Super Class	AtomicProceduralElement
Associations	expectedElement : NamedElement [*] A set of elements (MessageEvent, BehavioralFeature or Message) that are expected from the test item in addition to the expected element defined by the underlying UML behavioral element.
	arbitrationSpecification {redefines arbitrationSpecification} : ExpectResponseArbitrationSpecification [0..1] Refers to an expect response action arbitration specification that overrides the default and implicit arbitration specification if set. It redefines the Property *arbitrationSpecification* of test action.
	forbiddenElement : NamedElement [*] A set of elements (MessageEvent, BehavioralFeature or Message) that are not expected to be received from the test item. If left empty, no element is forbidden.

Constraints	Type of expected elements
	
	/* The tag definition expectedElement may only refer to instances of the metaclasses Message, Event, Signal or InstanceSpecification of Signal, BehavioralFeature and Trigger. */
	
	`context ExpectResponseAction inv:`
	`self.expectedElement.oclAsType(uml::Message) ->size() =1`
	`xor`
	`self.expectedElement.oclAsType(uml::Event)->size() =1`
	`xor`
	`self.expectedElement.oclAsType(uml::Signal)->size() =1`
	`xor`
	`self.expectedElement.oclAsType(uml::InstanceSpecification) ->size() =1`
	`xor`
	`self.expectedElement.oclAsType(uml::BehavioralFeature) ->size() =1`
	`xor`
	`self.expectedElement.oclAsType(uml::Trigger) ->size() =1`

	Type of forbidden elements
	
	/* The tag definition forbiddenElement may only yield instances of the metaclasses Message, Event, Signal or InstanceSpecification of Signal, BehavioralFeature and Trigger. */
	
	`context ExpectResponseAction inv:`
	`self.forbiddenElement.oclAsType(uml::Message) ->size() =1`
	`xor`
	`self.forbiddenElement.oclAsType(uml::Event)->size() =1`
	`xor`
	`self.forbiddenElement.oclAsType(uml::Signal)->size() =1`
	`xor`
	`self.forbiddenElement.oclAsType(uml::InstanceSpecifi`

	cation) ->size() =1 xor self.forbiddenElement.oclAsType(uml::BehavioralFeature) ->size() =1 xor self.forbiddenElement.oclAsType(uml::Trigger) ->size() =1
	At least one response /* DRTA02: It is necessary that a expect response action expects to receive at least one response. */
Change from UTP 1.2	«ExpectResponseAction» has been newly introduced by UTP 2.

8.5.3.4.5 SuggestVerdictAction

Description	SuggestVerdictAction: A test action that instructs the tester to suggest a particular procedural element verdict to the arbitration specification of the test case for being taken into account in the final test case verdict. Stereotype «SuggestVerdictAction» extends InvocationAction which allows for using a variety of metaclasses for application. However, there must be at least one argument InputPin defined for the InvocationAction of the predfefined type verdict or subclasses thereof. For example, a CallOperationAction could be used to invoke a (not standardized, yet proprietary) arbiter-specific interface operation. Another possibility is to use SendObjectAction without specifying the target Pin, which has the semantics of providing the Verdict instance to the arbitrating facility of a test execution system without needing a dedicated Interface. However, during test execution the suggest verdict action must be made executable. This may include manually writing the verdict instance into a paper-based document. If used in an Interaction, the InvocationAction that is stereotyped with «SuggestVerdictAction» must be referenced from an ActionExecutionSpecification that indirectly covers a Lifeline that represents a test component role in the underlying test configuration. Indirectly means that the corresponding start and end OccurenceSpecification of the ActionExecutionSpecification cover the test component lifeline. The default arbitration specification for the suggest verdict action is described by suggest verdict action AS.
Graphical syntax	
Extension	InvocationAction
Super Class	AtomicProceduralElement
Associations	arbitrationSpecification {redefines arbitrationSpecification} : SuggestVerdictArbitrationSpecification [0..1]

	Refers to a suggest verdict action arbitration specification that overrides the default and implicit arbitration specification if set. It redefines the Property *arbitrationSpecification* of test action.
Constraints	Type of Argument /* The type of the argument InputPin must be the predefined verdict type or a subtype thereof. */
Change from UTP 1.2	«SuggestVerdictAction» has been newly introduced by UTP 2.

8.6 Test Data

Testing is mainly about the exchange of data and the ability to compare actual responses and their payload received from the test item at test execution with the expected one stated in the test case. Therefore, testers usually have to take at least two data-related concepts into account. First, the specification of data, i.e., the known types and the constraints applied on these types for deriving data values that abide by these constraints. Second, a flexible mechanism to specify data values and their allowed matching mechanisms for test case execution.

Data specification-related concepts are provided and further described by the concepts of the Data Specifications chapter.

Data value-related concepts are provided and further described by the concepts of the Data Values chapter.

8.6.1 Data Specifications

This section specifies the stereotypes to implement the data specification concepts introduced in section Test Data of the Conceptual Model.

8.6.1.1 Data Specifications Overview

The diagram below shows abstract syntax of the data specifications package.

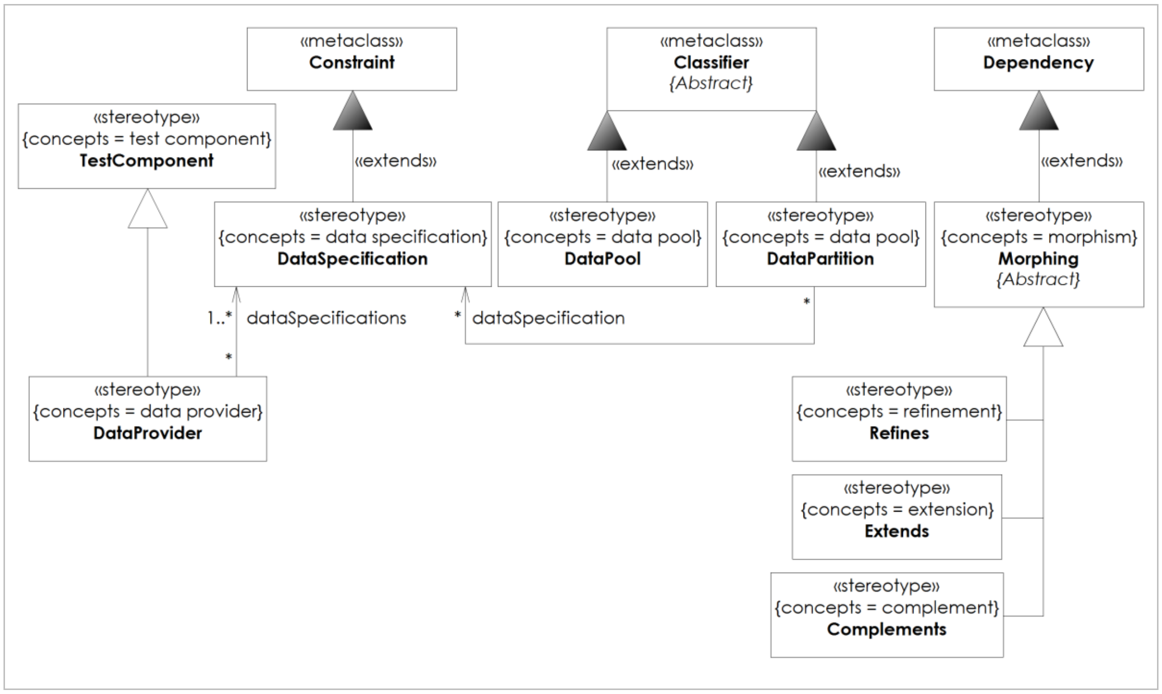


Figure 31 - Data Specifications Overview

8.6.1.2 Stereotype Specifications

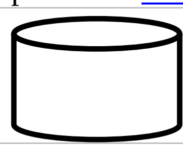
8.6.1.2.1 Complements

Description	Complements: A morphism that inverts data)i.e., that replaces the data items of a given set of data items by their opposites). The stereotype «Complements» specializes the abstract stereotype «Morphing» and logically negates the specification of the morphed data specifications within the morphing data specification. That means that complement morphism result in a complementing data specification that is the difference set of the complemented or morphed data specification.
Extension	Dependency
Super Class	Morphing
Change from UTP 1.2	«Complements» has been newly introduced by UTP 2.

8.6.1.2.2 DataPartition

Description	DataPartition: A role that some data plays with respect to some other data (usually being a subset of this other data) with respect to some data specification. The stereotype «DataPartition» extends a UML Classifier and represents a set of data that complies with one or more data specifications.
Extension	Classifier
Associations	dataSpecification : DataSpecification [*]
Change from UTP 1.2	«DataPartition» has been newly introduced by UTP 2.

8.6.1.2.3 DataPool

Description	DataPool: Some data that is an explicit or implicit composition of other data items. The stereotype «DataPool» extends a UML Classifier and represents a set of physical data without complying to any particular data specification.
Graphical syntax	
Extension	Classifier
Change from UTP 1.2	Changed from UTP 1.2. In UTP 1.2 «DataPool» extended both Classifier and Property.

8.6.1.2.4 DataProvider

Description	DataProvider: A test component that is able to deliver (i.e., either select and/or generate) data according to a data specification. The stereotype «DataProvider» is a specialization of stereotype «TestComponent». Such a test component is used to provide a data partition, represented as a Constraint extended by the stereotype «DataPartition», by generating some new data or by selecting some existing data from another data partition or a data pool according to some data specifications (represented as a Constraint extended by the stereotype «DataSpecification»).
Extension	Classifier, Property
Super Class	TestComponent
Associations	: TestDesignDirective
	dataSpecifications : DataSpecification [1..*]
Change from UTP 1.2	«DataProvider» has been newly introduced by UTP 2.

8.6.1.2.5 DataSpecification

Description	DataSpecification: A named boolean expression composed of a data type and a set of constraints applicable to some data in order to determine whether or not its data items conform to this data specification. The stereotype «DataSpecification» extends Constraint and is used to describe the constraints within the context of one or more types, instances of those types have to comply with. DataSpecifications are used to build and define DataPartitions. Since «DataSpecification» is an extension of Constraint the specification of the Constraint is defined by a ValueSpecification. This specification might be as simple as a LiteralString (e.g., natural language describing the constraint) or as complex as a formal language statement (e.g., Alf or OCL). UTP does not prescribe the notation used for describing the specification of a «DataSpecification» Constraint. In case a Constraint with «DataSpecification» is directly contained in Classifier, it is considered semantically equivalent to «DataSpecification» Constraint defined outside of this Classifier and with a «Refines» Dependency established between the «DataSpecification» Constraint and the Classifier.
Extension	Constraint
Associations	: DataProvider [*]
	: DataPartition [*]

Constraints	DataType in DataSpecification
	
	/* DRTD01: It is necessary that each data specification specifies at least one data type. */
	
	`context DataSpecification inv:`
	`self.base_Constraint.constrainedElement->size() >=1`

Change from UTP 1.2	«DataSpecification» has been newly introduced by UTP 2.

8.6.1.2.6 Extends

Description	Extends: A morphism that increases the amount of data (i.e., that adds more data items to a given set of data items). The stereotype «Extends» specialized the abstract stereotype «Morphing» and logically OR-combines the specification of the morphed data specifications within the morphing data specification. That means that extension morphism result in a data specification that is more general than the extended or morphed data specifications.
Extension	Dependency
Super Class	Morphing
Change from UTP 1.2	«Extends» has been newly introduced by UTP 2.

8.6.1.2.7 Morphing

Description	Morphing: A structure-preserving map from one mathematical structure to another. The abstract stereotype «Morphing» extends Dependency and is used to derive data specifications from other data specifications. This enables a high degree of reusability of existing data specifications. «Morphing» is intended to be subclassed and simply acts as a common superclass for shared semantics and constraints. A Dependency stereotyped with a subclass of «Morphing» always emanates from a Constraint with «DataSpecification» applied. It must point to a UML Classifier, to a UML Package containing some UML Classifiers, or to a Constraint with «DataSpecification» applied. If it targets a «DataSpecification» Constraint, it morphs the definitions of that data specification (called the morphed data specification) into a new data specification (called morphing data specification). If it targets a Classifier (or a set of Classifiers contained in a Package), all constraints applied on those Classifiers or their attributes are considered as an implicit morphed data specification attached to the Classifier which is eventually morphed into a morphing data specification. The exact effect of morphing a data specification into another data specification is defined by the concrete subclasses of the stereotype «Morphing».
Extension	Dependency
Sub Class	Complements, Extends, Refines

Constraints	Clients of a «Morphing» Dependency
	
	/* DRTD03: As clients of a Dependency stereotyped with a concrete substereotype of «Morphing» only the following elements are allowed: Constraint with «DataSpecification» applied. */
	
	`context Morphing inv:`
	`self.base_Dependency.client->forAll(c:NamedElement| not (c.oclAsType(uml::Constraint).oclIsInvalid()) and`

	`not c.getAppliedStereotype('UTP::Refines').oclIsUndefined() or c.getAppliedStereotype('UTP::Extends').oclIsUndefined() or c.getAppliedStereotype('UTP::Complements').oclIsUndefined() )`
	Suppliers of a «Morphing» Dependency / * DRTD04: As suppliers of a Dependency stereotyped with a concrete substereotype of «Morphing» only the following elements are allowed: Constraint with «DataSpecification» applied, UML Classifier, and UML Package. */ `context Morphing inv: self.base_Dependency.client->forAll(c:NamedElement| not (c.oclAsType(uml::Constraint).oclIsInvalid() and c.oclAsType(uml::Classifier).oclIsInvalid()) and not c.getAppliedStereotype('UTP::Refines').oclIsUndefined() or c.getAppliedStereotype('UTP::Extends').oclIsUndefined() or c.getAppliedStereotype('UTP::Complements').oclIsUndefined())`
Change from UTP 1.2	«Morphing» has been newly introduced by UTP 2.

8.6.1.2.8 Refines

Description	Refines: A morphism that decreases the amount of data (i.e., that removes data items from a given set of data items). The stereotype «Refines» specialized the abstract stereotype «Morphing» and logically AND-combines the specification of the morphed data specifications within the morphing data specification. That means that refinement morphism result in a data specification that is more specific than the refined or morphed data specifications.
Extension	Dependency
Super Class	Morphing

Change from UTP 1.2	«Refines» has been newly introduced by UTP 2.

8.6.2 Data Values

The payload of an expect response action is also called expected response argument value as opposed to the actual response argument value. During arbitration specification, usually a comparator evaluates whether the actual response matches with the expected ones in terms of event type and its payload. It is then the task of the arbitration specification to decide on the verdict that has to be assigned. In UTP data values are expressed by means of ValueSpecifications to specify both the payload for a stimulus and the payload of expected responses. In case of an expected response, the ValueSpecification does also implicitly define a matching mechanism used by a comparator during arbitration in order to evaluate whether the expected payload matches the actual payload.

The implicitly applied matching mechanism is determined by the ValueSpecification used to describe an expected payload argument in the context of an expected response. The prescribed matching mechanisms semantics, inherently bound to ValueSpecifications, are defined by UTP as follows:

- ValueSpecification (abstract metaclass): In general, any native UML ValueSpecification infers an *equality matching mechanism*, i.e., the actual payload, also known as response argument value, must be exactly the same as the expected payload. Any deviation will result in a mismatch.
- LiteralInteger: Checks for equality of the expected and actual response Integer-typed argument value.
- LiteralString: Checks for equality of the expected and actual response String-typed argument value.
- LiteralReal: Checks for equality of the expected and actual response Real-typed argument value.
- LiteralBoolean: Checks for equality of the expected and actual response Boolean-typed argument value.
- LiteralUnlimitedNatural: Checks for equality of the expected and actual response Integer-typed argument value including infinity.
- LiteralNull: Checks for absence of an actual response argument value of any type.
- InstanceValue: Checks for equality of the expected and actual response complex data type instance argument value.

All these equality matching mechanisms are natively given by UML, whereas UTP adds just a few more ValueSpecifications that provide matching mechanisms currently not given by UML. These kinds of ValueSpecifications are sometimes called Wildcards (TTCN-3) or Facets (XML Schema):

- AnyValue: Represents a set of all possible values for a given type and checks if actual response argument value is contained in this set. In case of optionality, the set of known values includes the absence of a value. This is implemented as stereotype «AnyValue».

- RegularExpression: Represents a set of values for a given type described by a regular expression and checks if the actual response argument value belongs to that set. This is implemented as stereotype «RegularExpression».

Both stimuli and expected responses yield data values for distinct signature elements. A signature element is defined as instance of either a Parameter or Property (i.e., this specification introduces a virtual metaclass SignatureElement that is the joint superclass of Property and Parameter and has at least the following attributes: type : UML::Type, lower : Integer, upper : UnlimitedNatural). Given by UML [UML25], a "... Type specifies a set of allowed values known as the instances of the Type." This specification denotes this set in the context of a SignatureElement expressed as *type(se)*, with *type(se)* as *SignatureElement.type*, and use *T* as abbreviation for *type(se)*.

We specify

$$T_{SE}(T) = \begin{cases} T & \text{if } lower(se) > 0 \\ T \cup \emptyset & \text{otherwise} \end{cases}$$

with *se* instance of SignatureElement and *lower(se)* as SignatureElement.lower and denote it by SE type.

A ValueSpecification *V* as an argument for a SignatureElement is specified as

$$V \in \mathcal{P}_{SE}(T_{SE})$$

These basic definitions are further used for the specific ValueSpecification matching mechanism extensions introduced by UTP.

8.6.2.1 Data Value Extensions

The diagram below shows the abstract syntax of the ValueSpecification extensions introduced by UTP.

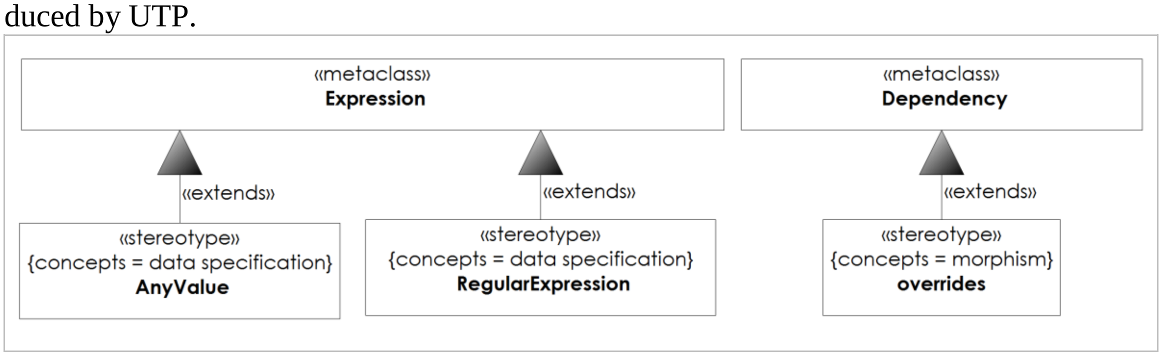


Figure 32 - Data Value Extensions

8.6.2.2 Stereotype Specifications

8.6.2.2.1 AnyValue

Description	The stereotype «AnyValue» extends ValueSpecification and represents an implicit set of known values for a given type. The expected response argument value matches with each actual response argument value, as long as type-compliance is given. In case of optionality, the set of known values includes the absence of a value. Fomally, AnyValue is defined as the power set of SE, i.e. T_{SE} The actual argument a matches the expected argument specified by AnyValue if $a \in T_{SE}$.
Extension	Expression
Change from UTP 1.2	Changed and renamed from UTP 1.2. In UTP 1.2, «AnyValue» was called «LiteralAny» and extended LiteralSpecification.

8.6.2.2.2 overrides

Description	Overrides is a relationship between at least two InstanceSpecifications, i.e., the modifying InstanceSpecification and the modified InstanceSpecification. Modifying InstanceSpecifications constitute the client elements of the underlying dependency, and consequently, modified InstanceSpecifications constitute the supplier elements of the underlying dependency. A modifying InstanceSpecification reuses all slot values of the modified InstanceSpecification in a way as if the slot values would have been copied into the modifying InstanceSpecification as its owned slots. Furthermore, the modifying InstanceSpecification is allowed to specify slots, which have not been declared by the modified InstanceSpecification at all. This enables user to gradually complete InstanceSpecifications and to reuse already or maybe partially defined InstanceSpecifications in order to create large sets of data by avoiding redundancy. Additionally, a modifying InstanceSpecification is able to overwrite slots with new values. A slot is considered to be overwritten if a modifying InstanceSpecification defines an owned slot that refers to the very same defining feature as the owned slot of the modified InstanceSpecification, or to a feature that redefines, directly or transitively, the slot's defining feature. An overwriting slot‘s value list entirely replaces the value list of the slot that is overwritten. Modification requires type compatibility between the modifying and modified InstanceSpecifications. Type compatibility is given if a modifying InstanceSpecification’s classifier list is compatible with the modified InstanceSpecification’s classifier list. Two classifier lists are compatible if the modifying InstanceSpecification’s classifier list is a proper subset of the modified InstanceSpecification’s classifier list. A proper subset is considered to be given if each classifier of the modifying InstanceSpecification’s classifier list is type compatible with at least one classifier of the modified InstanceSpecification classifier list. Type compatibility between classifiers is defined in the UML specifications. Cyclic modifications are not allowed. A cyclic modification describes a situation in which a modifying InstanceSpecifica-

	tion establishes a modification to a modified InstanceSpecification and the latter one already modifies, directly or transitively, the modifying InstanceSpecification.
Extension	Dependency
Constraints	Restriction of client and supplier /* As client and supplier of the underlying Dependency, only InstanceSpecification are allowed. */ context overrides inv: self.base_Dependency.client->forAll(c|c.oclIsKindOf(uml::InstanceSpecification)) and self.base_Dependency.supplier->forAll(c|c.oclIsKindOf(uml::InstanceSpecification))
	Cyclic modifications /* Cyclic override are not allowed. A cyclic override means that an overridden InstanceSpecification transitively overrides its overriding InstanceSpecification. */
Change from UTP 1.2	«overrides» was renamed by UTP 2. In UTP 1.2, it was named «modifies».

8.6.2.2.3 RegularExpression

Description	The stereotype «RegularExpression» extends Expression and represents an implicit set of values for a given type described by a regular expression. The expected response argument value matches with each actual response argument value if the actual one belongs to the set of values defined by the regular expression. A RegularExpression can be used for test data generation or to compare whether an actual response matches with expected response. The attribute *symbol* of the underlying Expression must contain the String that is evaluated as the regular expression. It might be omitted, in that case the *operands* of the underlying Expression must be used as abstract syntax tree for the regular expression.
Extension	Expression

Change from UTP 1.2	«RegularExpression» has been newly introduced by UTP 2.

8.7 Test Evaluation

The concepts for test evaluation are necessary to decide about the outcome of the dynamic test process activities. They implement in the specification of (proprietary) arbitration specifications on test set, test case and procedural element level, as well as in the ability to incorporate the test logs produced during the execution of a test-specific procedure and its procedural element in a platform-independent, but user-specific way.

8.7.1 Arbitration Specifications

In dynamic testing, the term *Arbitration* describes the application of a certain rule set on the outcome of a test execution activity, usually captured as test log for comprehensibility, in order to derive the final verdict of an execution test set or test case. Thus the arbitration of an executed test set or test case is the most important activity of the test evaluation activities with respect to requirements, test requirement or test objective coverage. Arbitration can both happen immediately during test execution (ad-hoc arbitration) and after test execution based on the captured test logs (post-execution arbitration). Due to whatever reason (organizational, technical etc.), one might be preferred over the other.

UTP provides a dedicated framework for declaring, setting and overriding arbitration specifications for arbitratable elements (i.e., test sets, test cases, procedural elements and test actions). It is left open, if the arbitration activities are carried out automatically or by a human. In the first case, UTP does not prescribe any implementation details of an arbiter component as part of an test execution system, nor how or when information from test sets, test cases and procedural elements are passed to an arbiter component. Even the degree of formalism of an arbitration specification is left open. An arbitration specification might be represented by something as simple as an identifier (referring to an implementation), by natural language describing the arbitration rules, by any kind of UML Behavior or by something formal as executable specifications or mathematical definitions.

UTP introduces three different kinds of verdicts that can be produced:

- procedural element verdicts: Verdicts produced by a procedural element arbitration specification;
- test case verdicts: Verdicts produced by a test case arbitration specification;
- test set verdicts: Verdicts produced by a test set arbitration specification.

The fundamental verdict calculation and provisioning schema is as follows:

- test set arbitration specifications: they derive the test set verdict from the test case verdicts that have been executed as part of the test set (i.e., the test case verdicts are passed to the arbitration specification of the surrounding test set);

- test case arbitration specifications: they derive the test case verdicts from the procedural element verdicts (first and foremost the test action verdicts) that have been executed as part of the test case (i.e., the procedural element verdicts are assembled and passed on to the test case arbitration specification);
- procedural element arbitration specifications: they derive procedural element verdicts from the information conveyed by the procedural element, or in case of a compound procedural element, the procedural element verdicts received from the arbitration specifications of the contained procedural elements.

UTP claims to offer the highest degree of flexibility to the test engineers by reusing and overriding arbitration specifications. The reuse of arbitratable elements with different arbitration specifications eases the reuse of those elements for different purposes. e.g., functional test cases could be reused for security testing by simply overriding their default arbitration specifications. Therefore, UTP introduces the notion of the *effective* arbitration specification that determines the arbitration specification that will be eventually used for the verdict calculation. Since arbitration specification can be overridden and redefined, it is important to understand the rules that apply for determining the eventual effective arbitration specification. The respective effective arbitration specification for an arbitratable element (i.e., test set, test case and procedural element) depends on the location of a potential binding between these two concepts:

- Default arbitration specification (implicit binding): UTP defines a default (i.e., implicitly set) arbitration specification for any arbitratable element by means of the UTP Arbitration Library. As long as the respective attributes for the arbitration specification are left empty (i.e., null set), the corresponding default arbitration specification is considered the effective arbitration specification of that element. Semantically, leaving the attribute for the corresponding arbitration specification empty is equivalent with setting it explicitly with the prescribed default arbitration specification contained in the UTP Arbitration Library.
- Static arbitration specification (static binding): If user-specific arbitration specifications are explicitly set for an arbitratable element, the default arbitration specification is considered as overridden. As a result, whenever the element is executed, the static arbitration specification assumes the role of the *effective* arbitration specification for any further executions of that element.
- Dynamic arbitration specification (dynamic binding): By means of procedure invocation, represented by the stereotype «ProcedureInvocation», it is possible to override the explicitly set static arbitration specification for the current execution. Dynamically set arbitration specifications override both the default and static arbitration specifications within the context of the current execution scope. Therefore, the stereotype «ProcedureInvocation» defines the attribute *dynamicArbitrationSpecification*.

The precedence rules for arbitration specification in conflicting situations are: dynamic binding > static binding > implicit binding.

8.7.1.1 Test Procedure Arbitration Specifications

The most important arbitratable element of UTP is of course the test case. Therefore, UTP offers a dedicated arbitration specification stereotype (i.e., «TestCaseArbitrationSpecification») to define proprietary test case arbitration specifications for static and dynamic binding. Arbitration specifications for test sets can be set either as part of the test set itself (i.e., set via the attribute *testSetAS* of the stereotype «TestSet») or as part of a corresponding test execution schedule (i.e., set via the attribute *testSetAS* of the stereotype «TestExecutionSchedule»).

The calculation of the effective test set arbitration specification can lead to conflicting situations, because both a test set and a corresponding test execution schedule of that test set may declare a static arbitration specification. In that case, the effective arbitration specification is determined in a cascading style (i.e., more inner arbitration specification overrides the more outer arbitration specification). That means that the static arbitration specification of a test execution schedule always has precedence over the static arbitration specification of the surrounding test set. However, since test execution schedule can be invoked by means of procedure invocation (represented by the stereotype «ProcedureInvocation») it is nonetheless possible to re-override the static arbitration specification of a test execution schedule with a dynamic arbitration specification when invoking the test execution schedule.

8.7.1.1.1 Arbitration Specifications Overview

The following figure shows the fundamental capabilities of the arbitration specification facility of UTP. In general, any executable concept in UTP can be arbitrated and may provide its specific verdict to the calculation of other verdicts. Further information of invocation schemes or the semantic of override is given by the semantics descriptions of the respective arbitration specifications and procedure invocation.

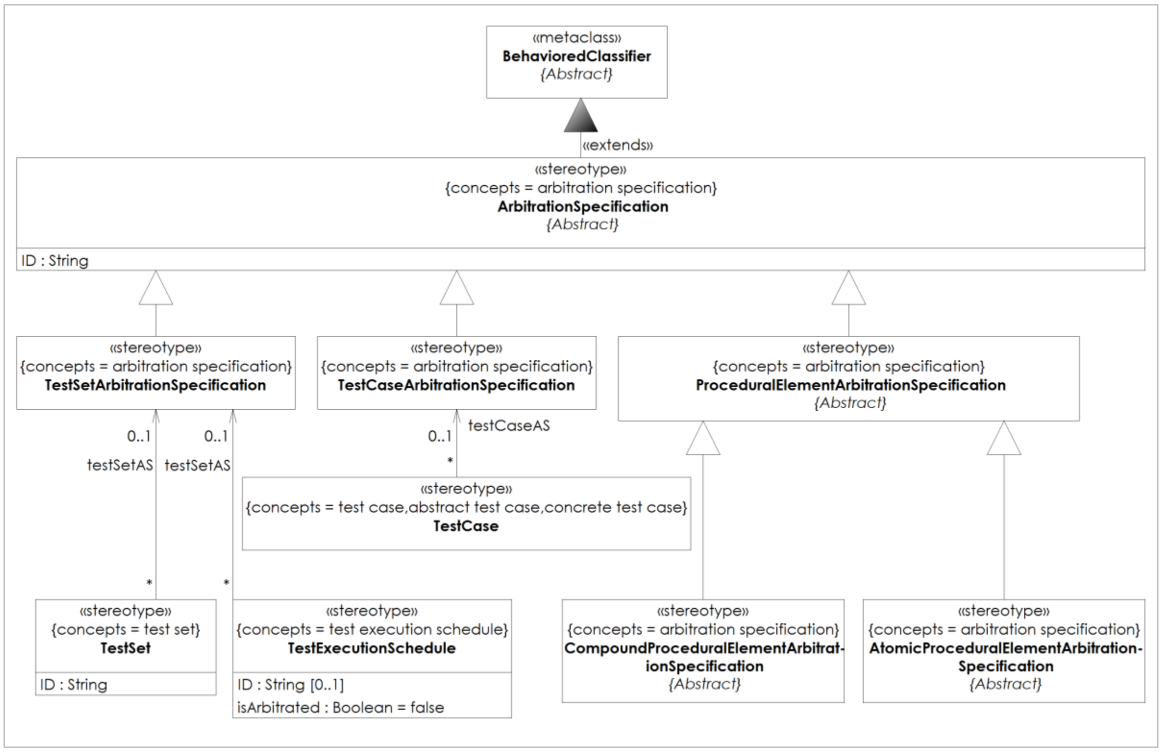


Figure 33 - Arbitration Specifications Overview

8.7.1.1.2 Stereotype Specifications

8.7.1.1.2.1 ArbitrationSpecification

Description	ArbitrationSpecification: A set of rules that calculates the eventual verdict of an executed test case, test set or procedural element. The stereotype «ArbitrationSpecification» extends BehavioredClasifier and is used to specify the decision process for verdicts. It is an abstract stereotype that is specialized by stereotypes that deal with the verdicts of test sets, test cases, and procedural elements (i.e. test set verdicts, test case verdicts, and procedural element verdicts).
Graphical syntax	
Detailed semantics	The concept of an arbitration specification allows for specifying user-defined algorithms for the calculation of the verdict based on the executed test cases or the captured test case logs.

<table>
<tr><td></td><td>
An arbitration specification might be associated with a test case, with a test set or with a test context. In case of the test case, the respective arbitration specification is only applied to the mentioned test case. In the latter two cases, the arbitration specification is applied to all test cases nested directly or indirectly in the test set or test context. In case of multiple arbitration specifications, the one that is the closest to the respective test case is used during execution.
There is a default arbitration specification defined in UTP 2. In case no other arbitration specification is given for one of the three above mentioned concepts, the default one is implicitly applied. Its calculation algorithm is specified as follows:

When a test case is not executed yet or just invoked, its verdict is 'None'. 'None' indicates that no communication between test components and the test item has been carried out yet. 'None' is the weakest verdict. It is supposed to be never set by the tester directly.
A 'Pass' indicates that the test case was successful and that the test item behaved according to expectations.
A 'Fail' indicates that the test item is not behaving according to what was expected.
An 'Inconclusive' means that it cannot be determined whether the test item performs according to what was expected or not.
The precedence rules for the predefined verdicts are: None < Pass< Inconclusive< Fail. This means verdict 'Fail' was calculated by the arbitration specification; the test case's verdict could not result in a Pass or Inconclusive.

An arbitration specification is usually implemented by test execution tool by means of an Arbiter.
</td></tr>
<tr><td>Extension</td><td>BehavioredClassifier</td></tr>
<tr><td>Sub Class</td><td>ProceduralElementArbitrationSpecification, TestCaseArbitrationSpecification, TestSetArbitrationSpecification</td></tr>
</table>

Attributes	ID : String [1] A unique identifier that unambiguously identifies the given arbitration specification.
Associations	/referencedBy : TestContext [*]
	: CompoundProceduralElement
Constraints	Verdict of ArbitrationSpecification /* DRAS01: It is necessary that an arbitration specification determines exactly one verdict. */
Change from UTP 1.2	«ArbitrationSpecification» has been newly introduced into UTP 2.

8.7.1.1.2.2 TestCaseArbitrationSpecification

Description	TestCaseArbitrationSpecification: A set of rules that calculates the eventual verdict of an executed test case, test set or procedural element. A «TestCaseArbitrationSpecification» specifies the rules for the eventual calculation of a test case verdict based on the procedural element verdicts that have been executed in the context of the corresponding test case. If explicitly set, the implicit default «TestCaseArbitrationSpecification» is overridden for a certain execution or all executions of the given test case.
Extension	BehavioredClassifier
Super Class	ArbitrationSpecification
Associations	: TestCase [*]
Change from UTP 1.2	Newly introduced by UTP 2.

8.7.1.1.2.3 TestSetArbitrationSpecification

Description	TestSetArbitrationSpecification: A set of rules that calculates the eventual verdict of an executed test case, test set or procedural element. A «TestSetArbitrationSpecification» specifies the rules of how a test set verdict will be calculated based on the verdicts of the test sets that have been executed in the context of the corresponding test set. Since a test set is not required to be executed using a test execution schedule. In that case, the execution order of the test set's test cases is simply undefined and up to the executing entity. A test set arbitration specification is used by both «TestSet» and «TestExecutionSchedule».
Extension	BehavioredClassifier
Super Class	ArbitrationSpecification
Associations	: TestSet [*]
	: TestExecutionSchedule [*]
Change from UTP 1.2	Newly introduced by UTP 2.

8.7.1.2 Procedural Element Arbitration Specifications

The procedural element arbitration specification sections summarize the different type of arbitration specifications that can be used to define proprietary procedural element arbitration specifications.

8.7.1.2.1 Arbitration of AtomicProceduralElements

The diagram below shows the abstract syntax of arbitration specification elements for atomic procedural elements.

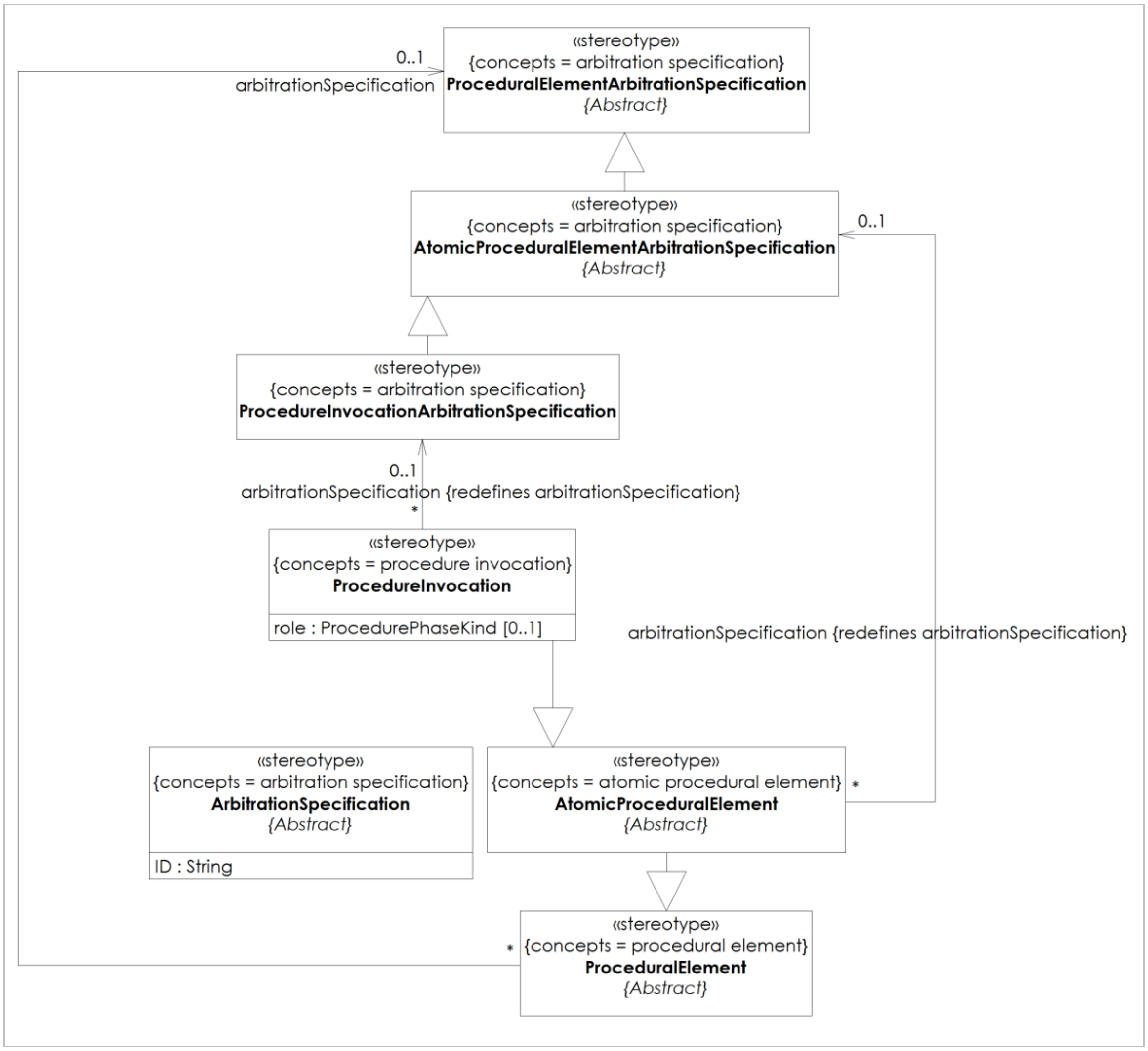


Figure 34 - Arbitration of AtomicProceduralElements

8.7.1.2.2 Arbitration of CompoundProceduralElements

The diagram below shows the abstract syntax of arbitration specification elements for compound procedural elements.

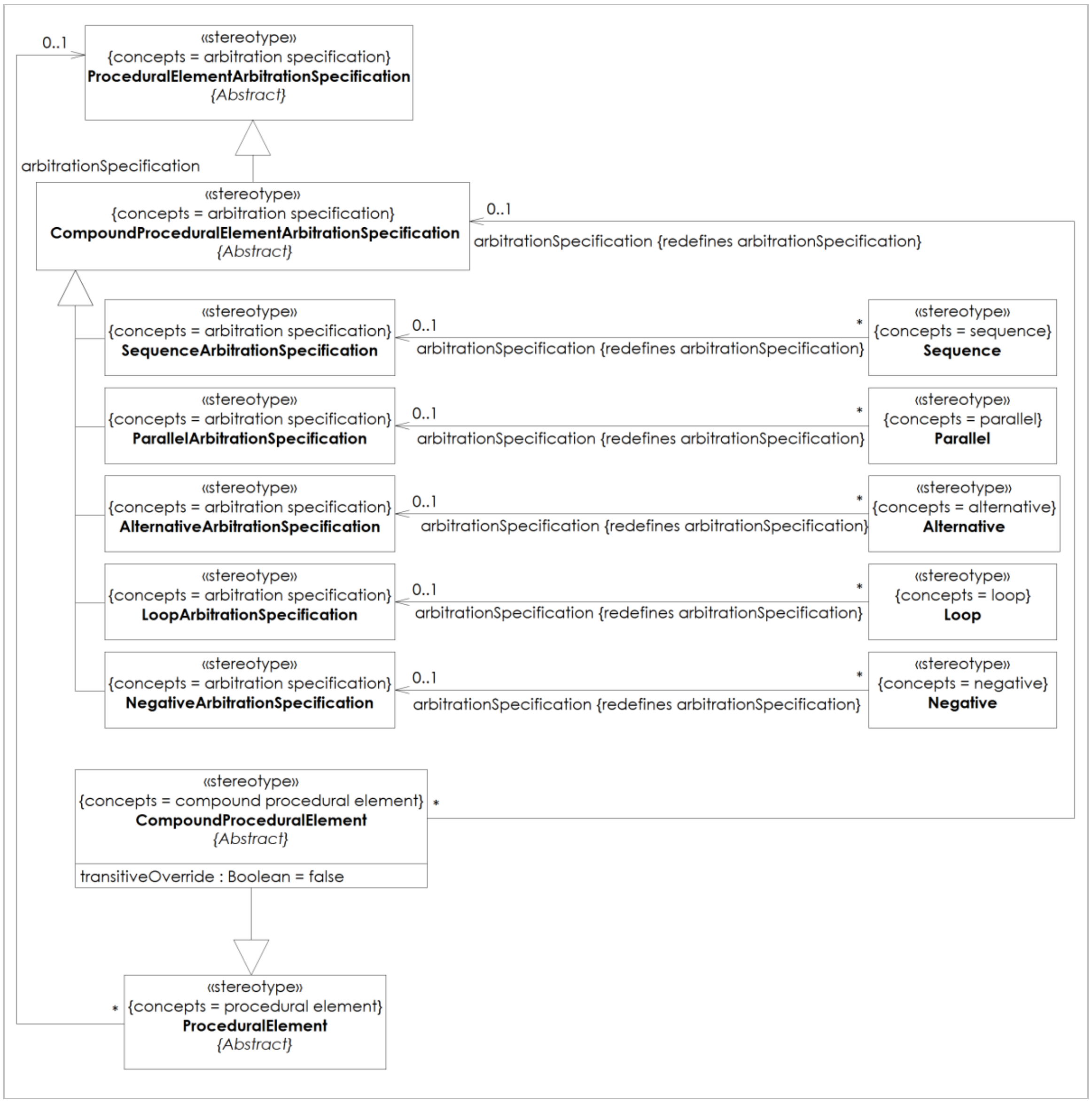


Figure 35 - Arbitration of CompoundProceduralElements

8.7.1.2.3 Stereotype Specifications

8.7.1.2.3.1 AlternativeArbitrationSpecification

Description	An «AlternativeArbitrationSpecification» calculates a verdict for a set of procedural elements that are executed in mutually exclusive branches.
Extension	BehavioredClassifier
Super Class	CompoundProceduralElementArbitrationSpecification

Associations	: Alternative [*]
Change from UTP 1.2	Newly introduced by UTP 2.

8.7.1.2.3.2 AtomicProceduralElementArbitrationSpecification

Description	An «AtomicProceduralElementArbitrationSpecification» calculates a verdict for a single atomic procedural element.
Extension	BehavioredClassifier
Super Class	ProceduralElementArbitrationSpecification
Sub Class	CheckPropertyArbitrationSpecification, CreateLogEntryArbitrationSpecification, CreateStimulusArbitrationSpecification, ExpectResponseArbitrationSpecification, ProcedureInvocationArbitrationSpecification, SuggestVerdictArbitrationSpecification
Associations	: AtomicProceduralElement [*]
Change from UTP 1.2	Newly introduced by UTP 2.

8.7.1.2.3.3 CompoundProceduralElementArbitrationSpecification

Description	A «CompoundProceduralElementArbitrationSpecification» calculates a verdict for a set of procedural elements that are executed together. The verdict is derived from all or parts of the verdicts calculated of their respective arbitration specifications.
Extension	BehavioredClassifier
Super Class	ProceduralElementArbitrationSpecification
Sub Class	AlternativeArbitrationSpecification, LoopArbitrationSpecification, NegativeArbitrationSpecification, ParallelArbitrationSpecification, SequenceArbitrationSpecification
Associations	: CompoundProceduralElement [*]
Change from UTP 1.2	Newly introduced by UTP 2.

8.7.1.2.3.4 LoopArbitrationSpecification

Description	A «LoopArbitrationSpecification» calculates a verdict for a set of procedural elements that are sequentially executed in a loop.
Extension	BehavioredClassifier
Super Class	CompoundProceduralElementArbitrationSpecification
Associations	: Loop [*]
Change from UTP 1.2	Newly introduced by UTP 2.

8.7.1.2.3.5 NegativeArbitrationSpecification

Description	A «NegativeArbitrationSpecification» calculates a verdict for set of procedural elements that are forbidden to be executed in this sequence.
Extension	BehavioredClassifier
Super Class	CompoundProceduralElementArbitrationSpecification
Associations	: Negative [*]
Change from UTP 1.2	Newly introduced by UTP 2.

8.7.1.2.3.6 ParallelArbitrationSpecification

Description	A «ParallelArbitrationSpecification» calculates a verdict for a set of procedural elements that were executed in parallel.
Extension	BehavioredClassifier
Super Class	CompoundProceduralElementArbitrationSpecification
Associations	: Parallel [*]
Change from UTP 1.2	Newly introduced by UTP 2.

8.7.1.2.3.7 ProceduralElementArbitrationSpecification

Description	A «ProceduralElementArbitrationSpecification» calculates a verdict for a single or a set of procedural elements.
Extension	BehavioredClassifier
Super Class	ArbitrationSpecification

Sub Class	AtomicProceduralElementArbitrationSpecification, CompoundProceduralElementArbitrationSpecification
Associations	: ProceduralElement [*]
Change from UTP 1.2	Newly introduced by UTP 2.

8.7.1.2.3.8 ProcedureInvocationArbitrationSpecification

Description	A «ProcedureInvocationArbitrationSpecification» calculates a verdict for an executed procedure invocation.
Extension	BehavioredClassifier
Super Class	AtomicProceduralElementArbitrationSpecification
Associations	: ProcedureInvocation [*]
Change from UTP 1.2	Newly introduced by UTP 2.

8.7.1.2.3.9 SequenceArbitrationSpecification

Description	A «SequenceArbitrationSpecification» calculates a verdict for a sequence of executed procedural elements.
Extension	BehavioredClassifier
Super Class	CompoundProceduralElementArbitrationSpecification
Associations	: Sequence [*]
Change from UTP 1.2	Newly introduced by UTP 2.

8.7.1.3 Test-specific Action Arbitration Specifications

The test action arbitration specification sections summarize the different types of arbitration specifications that can be used to define proprietary arbitration specifications for prescribing test action.

8.7.1.3.1 Arbitration of Test-specific Actions

The diagram below shows the abstract syntax of the arbitration specifications for dedicated test actions.

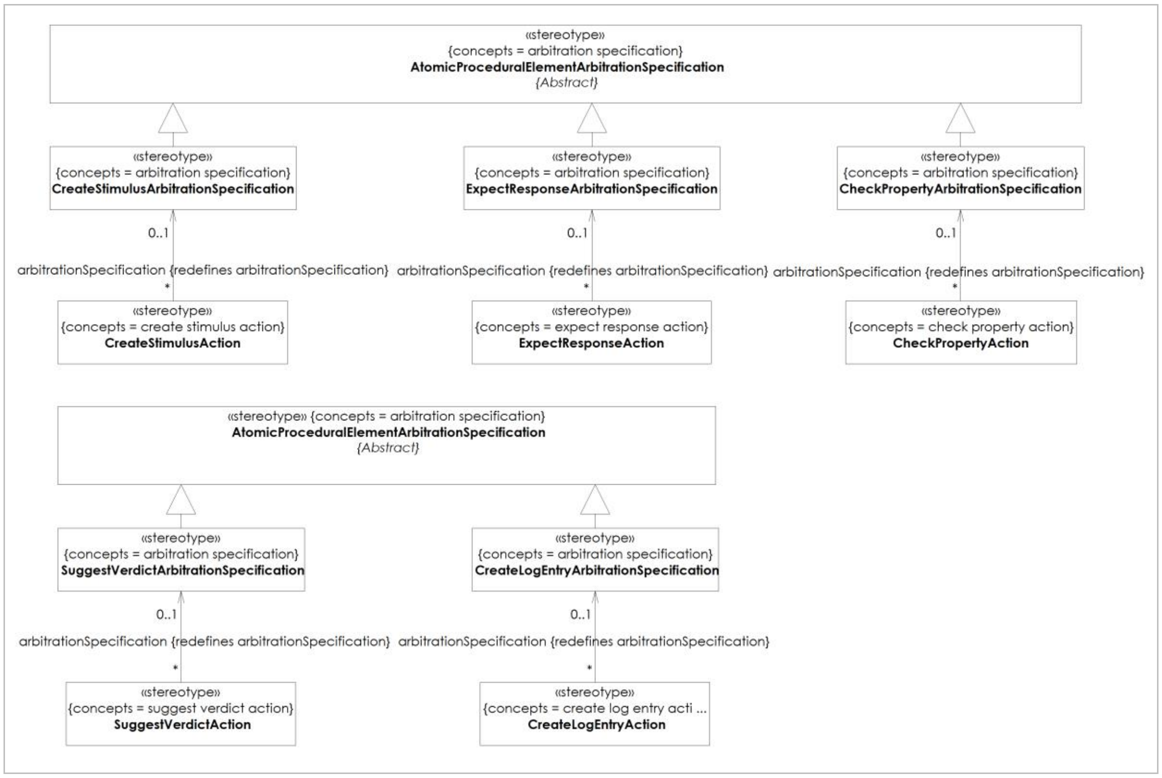


Figure 36 - Arbitration of Test-specific Actions

8.7.1.3.2 Stereotype Specifications

8.7.1.3.2.1 CreateStimulusArbitrationSpecification

Description	An «AtomicProceduralElementArbitrationSpecification» that specifies the verdict calculation rule for a create stimulus action.
Extension	BehavioredClassifier
Super Class	AtomicProceduralElementArbitrationSpecification
Associations	: CreateStimulusAction [*]
Change from UTP 1.2	Newly introduced by UTP 2.

8.7.1.3.2.2 ExpectResponseArbitrationSpecification

Description	An «AtomicProceduralElementArbitrationSpecification» that specifies the verdict calculation rule for an expect response action.
Extension	BehavioredClassifier

Super Class	AtomicProceduralElementArbitrationSpecification
Associations	: ExpectResponseAction [*]
Change from UTP 1.2	Newly introduced by UTP 2.

8.7.1.3.2.3 CheckPropertyArbitrationSpecification

Description	An «AtomicProceduralElementArbitrationSpecification» that specifies the verdict calculation rule for a check property action.
Extension	BehavioredClassifier
Super Class	AtomicProceduralElementArbitrationSpecification
Associations	: CheckPropertyAction [*]
Change from UTP 1.2	Newly introduced by UTP 2.

8.7.1.3.2.4 SuggestVerdictArbitrationSpecification

Description	An «AtomicProceduralElementArbitrationSpecification» that specifies the verdict calculation rule for a suggest verdict action.
Extension	BehavioredClassifier
Super Class	AtomicProceduralElementArbitrationSpecification
Associations	: SuggestVerdictAction [*]
Change from UTP 1.2	Newly introduced by UTP 2.

8.7.1.3.2.5 CreateLogEntryArbitrationSpecification

Description	An «AtomicProceduralElementArbitrationSpecification» specification that specifies the verdict calculation rule for a create log entry action.
Extension	BehavioredClassifier
Super Class	AtomicProceduralElementArbitrationSpecification
Associations	: CreateLogEntryAction [*]
Change from UTP 1.2	Newly introduced by UTP 2.

8.7.2 Test Logging

The test logging facility allows incorporating details about the execution of test-specific procedures, such as test execution schedules and test cases, but also of procedural elements. UTP prescribes certain information that are essential for any kind of test log, but ensures the required degree of flexibility in order to cope with the variety of existing (including proprietary) test log formats and contents of arbitrary test execution systems.

The test logging facility comprises the following concepts and their manifestations.

- test log, implemented as the abstract stereotype «TestLog»;
- test set log, implemented as stereotype «TestSetLog» that specializes «TestLog»
- test case log, implemented as stereotype «TestCaseLog» that specializes «TestLog»
- test log structure, implemented as stereotype «TestLogStructure»;
- test log structure binding, implemented as stereotype «TestLogStructureBinding».

8.7.2.1 Test Logging Overview

The following diagram shows the abstract syntax of the test logging facility.

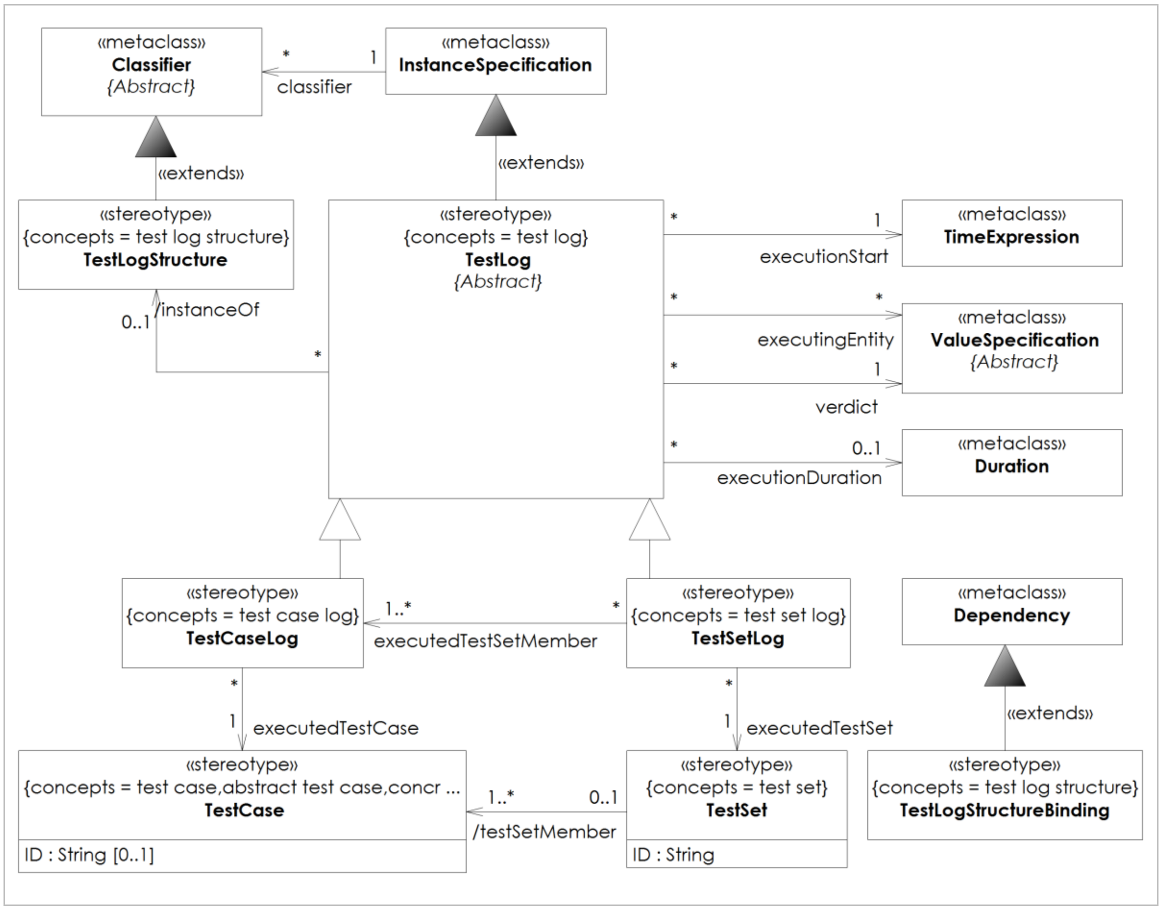


Figure 37 - Test Logging Overview

8.7.2.2 Stereotype Specifications

8.7.2.2.1 TestCaseLog

Description	TestCaseLog: A test log that captures relevant information on the execution of a test case. A test case log captures the least relevant information on the execution of a test case by an executing entity. The at least required information is defined by the corresponding and potentially implicit test log structure of the test case log.
Extension	InstanceSpecification
Super Class	TestLog
Associations	: TestSetLog [*]
	executedTestCase : TestCase

	Refers to the TestCase whose execution was captured by means of the given TestCaseLog.
Change from UTP 1.2	Newly introduced by UTP 2.

8.7.2.2.2 TestLog

Description	TestLog: A test log is the instance of a test log structure that captures relevant information from the execution of a test case or test set. The least required information to be logged is defined by the test log structure of the test log. A test log captures information on the execution of a test case or test set that actually happened according to the specification required by its test log structure. Each test log is, at least, an instance of the implicitly defined default test log structure. This is reflected by its tag definitions that comprise the required log information. If further information is not required for capturing by an executing entity, a test log may not refer to an explicit test log structure (i.e., the Classifier of the underlying InstanceSpecification remains empty). In addition to the information given by the implicit default test log structure, users may set an explicitly defined a test log structure of arbitrary complex internal structures. In that case, the underlying InstanceSpecification may capture the additional information by relying on the native UML InstanceSpecification mechanism, namely Slots.
Extension	InstanceSpecification
Sub Class	TestCaseLog, TestSetLog
Associations	executionStart : TimeExpression Denotes the point in time when the execution of the test case or test set began.
	executionDuration : Duration [0..1] Denotes how long the execution of the test case or test set lasted.
	executingEntity : ValueSpecification [*] Lists all the entities (human tester or test execution tool) that

	carried out the execution of a test case or test set.
	verdict : ValueSpecification ——— Captures the Verdict of the test case or test set as it was calculated by the respective arbitration specifications. The type of the ValueSpecification for verdict must be set to the predefined UTP type verdict.
	/instanceOf : TestLogStructure [0..1] ——— Refers to TestLogStructure that was instantiated by this TestLog. It is derived from the Classifier of the underlying InstanceSpecification, if present.
	/referencedBy : TestContext [*]

Constraints	Number of Classifier for InstanceSpecification
	———
	/* An InstanceSpecification with "TestLog" applied must have at most one Classifier with "TestLogStructure" applied. */
	
	`context TestLog inv:`
	`self.base_InstanceSpecification.classifier.getAppliedStereotype('UTP::TestLogStructure')->size() =0 xor self.base_InstanceSpecification.classifier.getAppliedStereotype('UTP::TestLogStructure')->size() =1`

	Allowed metaclass for extension
	———
	/* TestLog must only be applied to InstanceSpecification, but not to EnumerationLiterals. */
	
	`context TestLog inv:`
	`self.base_InstanceSpecification.oclIsTypeOf(InstanceSpecification)`

	Type of verdict ValueSpecification
	———
	/* The type of the ValueSpecification referenced by the attribute verdict must be of type UTP verdict as defined in the UTP 2 Predefined Types library. */
	
	`context TestLog inv:`
	`self.verdict.type = UTPLibrar-`

	ies::UTPAuxiliaryLibrary::TypesLibrary::Verdict
Change from UTP 1.2	Changed from UTP 1.2. In UTP 1.2 «TestLog» was used to capture the execution of a test case or a test set (called test content in UTP 1.2). In UTP 2, two dedicated concepts have been newly introduced therefore (i.e., «TestCaseLog» and «TestSetLog»).

8.7.2.2.3 TestLogStructure

Description	A test log structure enables the specification of user-defined structures that must be logged by an executing entity, such as human tester or a test execution tool, during the execution of test suites, test cases or test execution schedules. This information is also called the least required log information, because executing entities are not restricted to capturing only information mentioned in the test log structure. A test log structure may describe both the required information for the header part as well as the body part of a test log. There is an implicit default (undefined) test log structure available in UTP that every user-defined test log structure complies with. The default test log structure represents the least required log information for the header part. This information comprises • one or more of an executing entity; • a point in time where the execution of the test case, test suite or test execution schedule began; • the duration the execution of the test case, test suite or test schedule lasted; and • the final verdict that was calculated by the corresponding arbitration specification. Those pieces of information of the default (implicit) test log structure are represented as tag definitions of the stereotype test log solely because they are eventually instantiated when a test log is created.
Extension	Classifier
Associations	: TestLog [*]

Constraints	Allowed metaclasses for extension
	
	/* Only DataType and Class are allowed as base metaclass for the extension. */
	
	`context TestLogStructure inv:`
	`self.base_Classifier.oclIsTypeOf(DataType) or self.base_Classifier.oclIsTypeOf(Class)`

	Specialization of TestLogStructure Classifier
	
	/* Classifiers with <<TestLogStructure>> applied must only extend Classifier with <<TestLogStructure>> applied */
	
	`context TestLogStructure inv:`
	`self.base_Classifier.allParents()->forAll(c:Classifier|c.getAppliedStereotype('UTP::TestLogStructure') <> null)`

	Internal structure of TestLogStructure Classifier
	
	/* Classifiers with «TestLogStructure» applied must only own Properties. */
	
	`context TestLogStructure inv:`
	`self.base_Classifier.ownedMember->forAll(member : NamedElement| member.oclIsTypeOf(Property) or (member.oclIsTypeOf(Classifier) and not member.oclIsTypeOf(UseCase))`

	CollaborationUse not allowed
	
	/* A «TestLogStructure» Classifier must not participate in Collaborations. /*
	
	`context TestLogStructure inv:`
	`self.base_Classifier.oclIsTypeOf(Class) implies self.base_Classifier.oclAsType(Class).collaborationUse->isEmpty()`

Change from UTP 1.2	Newly introduced by UTP 2.

8.7.2.2.4 TestLogStructureBinding

Description	A test log structure binding is responsible to explicitly bind test log structures to test cases or test sets. It is possible to reuse the very same test log structure at different locations. Since there are different possibilities how to model this, UTP suggests three methods to achieve multiple binding of test log structures: • Single Dependency/many suppliers method: This method binds many test cases or test sets as suppliers of the «TestLogStructureBinding» Dependency to a single «TestLogStructure» Classifier client. • Multiple Dependencies/single suppliers method: This method binds a single test case or test set as supplier of the «TestLogStructureBinding» Dependency to a single «TestCase» BehavioredClassifier client. • Combined method: This method combines the first two methods. The sum of all bound test log structures for a test case or test set is calculated by merging all suppliers of all visible «TestLogStructureBinding» Dependencies in a certain logical or technical scope. Visibility of test log structure binding is not defined by this specification. Moreover, this specification neither prescribes how test log structure bindings are finally put into effect by an executing entity nor how to select them for later use by an executing entity. Since Dependency is a PackageableElement, a possible method could be to use the UML deployment capabilities in order to implement the desired «TestLogStructureBinding» Dependency to putting it into effect in the test execution system.
Extension	Dependency

Constraints	Specification of Dependency client
	
	/* A Dependency with «TestLogStructureBinding» must have exactly one client containing a Classifier with «TestLogStructure» applied. */
	
	`context TestLogStructureBinding inv:`
	`self.base_Dependency.client->size() = 1 and self.base_Dependency.client.oclAsType(uml::Classifier)->size() =1 and`

	`self.base_Dependency.client.oclAsType(uml::Classifier).getAppliedStereotype('UTP::TestLogStructure')->size() =1`

	Specification of Dependency supplier
	
	/* A Dependency with «TestLogStructureBinding» must have at least one but an unlimited number of suppliers containing a BehavioredClassifier with «TestCase» applied. */
	
	`context TestLogStructureBinding inv:`
	`self.base_Dependency.supplier->size() >= 1 and self.base_Dependency.supplier->forAll(n:NamedElement | n.oclIsKindOf(BehavioredClassifier) and not n.getAppliedStereotype('UTP::TestCase').oclIsUndefined())`

Change from UTP 1.2	Newly introduced by UTP 2.

8.7.2.2.5 TestSetLog

Description	A test set log captures the least required information on the execution of a test set by an executing entity. The least required information is defined by the corresponding (potentially implicit) test log structure of the test set log. A test set log consists mainly of the logs of the executed test cases that are members of the test set. Since not all test cases of a test set must necessarily be executed by an executing entity, a test set log may only refer to the test case logs of a subset of the test set's test cases.
Extension	InstanceSpecification
Super Class	TestLog
Associations	executedTestSetMember : TestCaseLog [1..*] Refers to the test cases that are the members of the test set log's corresponding test set and whose execution were captured as a result of the execution of the test set.
	executedTestSet : TestSet

	Refers to the test set whose execution was captured by means of the given test case log.

Constraints	Executed test cases and definition of test set members must be consistent
	
	A TestSetLog must only refer to TestCaseLogs of TestCases that are members of the executed TestSet. */
	
	`context TestSetLog inv:`
	`self.executedTestSet->includes(self.executedTestSetMember.executedTestCase)`

Change from UTP 1.2	Newly introduced by UTP 2.

9 Model Libraries

This section describes a set of type libraries relevant to UTP.

9.1 UTP Types Library

The following diagram shows the predefined types provided by UTP 2.

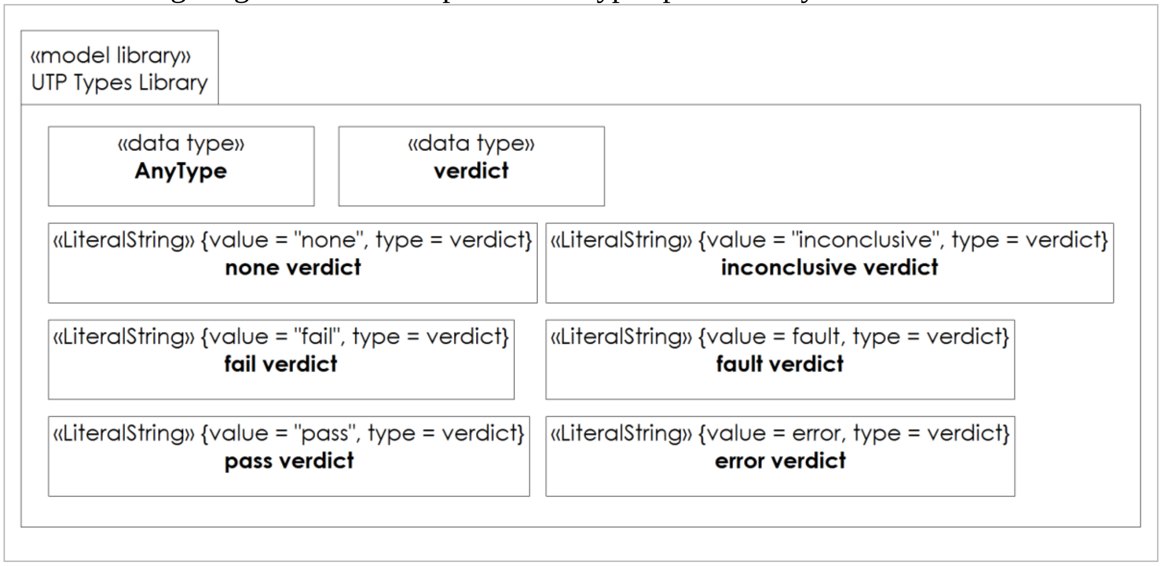


Figure 38 - Predefined Types Library

Name	Description
AnyType	The pre-defined type AnyType is the least common ancestor of any type known in the context of a certain type system. As a result, StructuralFeatures typed with AnyType can be assigned any value, regardless whether primitive or complex.
error verdict	The pre-defined error verdict instance.
fail verdict	The pre-defined fail verdict instance.
fault verdict	The pre-defined fault verdict instance.
inconclusive verdict	The pre-defined inconclusive verdict instance.
none verdict	The pre-defined none verdict instance.
pass verdict	The pre-defined pass verdict instance.
verdict	The pre-defined type verdict represents the basis for any verdict-related mechanisms. Tester may subclass the verdict type in order build specialized verdict types.

9.2 UTP Auxiliary Library

9.2.1 UTP Auxiliary Library

The UTP auxiliary library collects well-established and commonly accepted information whose use is optional. The purpose of the auxiliary library is to provide users with a set of useful and predefined types and values to foster reusability across modeling tools and approaches. For example, the ISO 25010 quality model is supposed to be used by multiple organizational units within the test process. Instead of building proprietary and potentially technically conflicting representations of the very same quality model, users may reuse the ISO 25010 [ISO25010] quality model that comes along with UTP itself. Of course, such types and values are often tailored to specific needs (e.g., Robustness testing is a frequently used testing type which is actually given in ISO 9216 or ISO 25010), but still needs to be specified. However, the existence of the UTP auxiliary model does not prevent such an approach.

9.2.1.1 The UTP auxiliary library

Overview of the UTP auxiliary library.

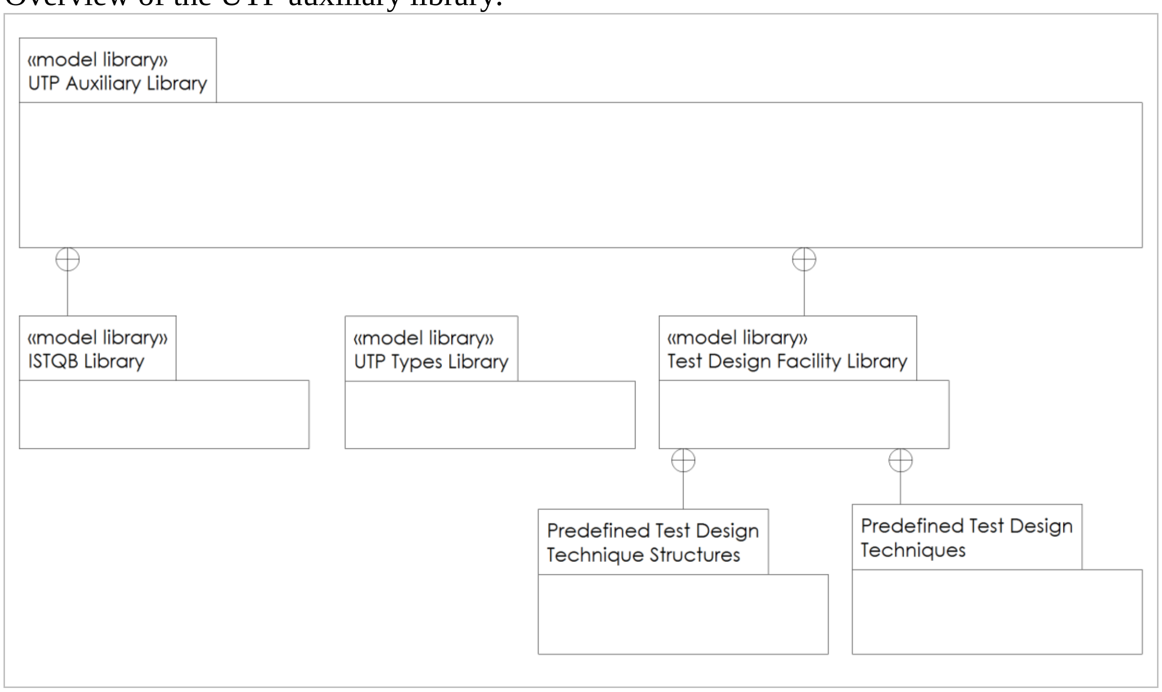


Figure 39 - The UTP auxiliary library

9.2.1.2 ISTQB Library

The ISTQB library offers concepts that can be used to organize some aspects of the test process, if required. In particular, the ISTQB library offers a commonly used set of test levels and test set purposes.

9.2.1.2.1 Overview of the ISTQB library

The following diagram shows the predefined test process library provided by UTP to be used for the specification of test contexts and test sets.

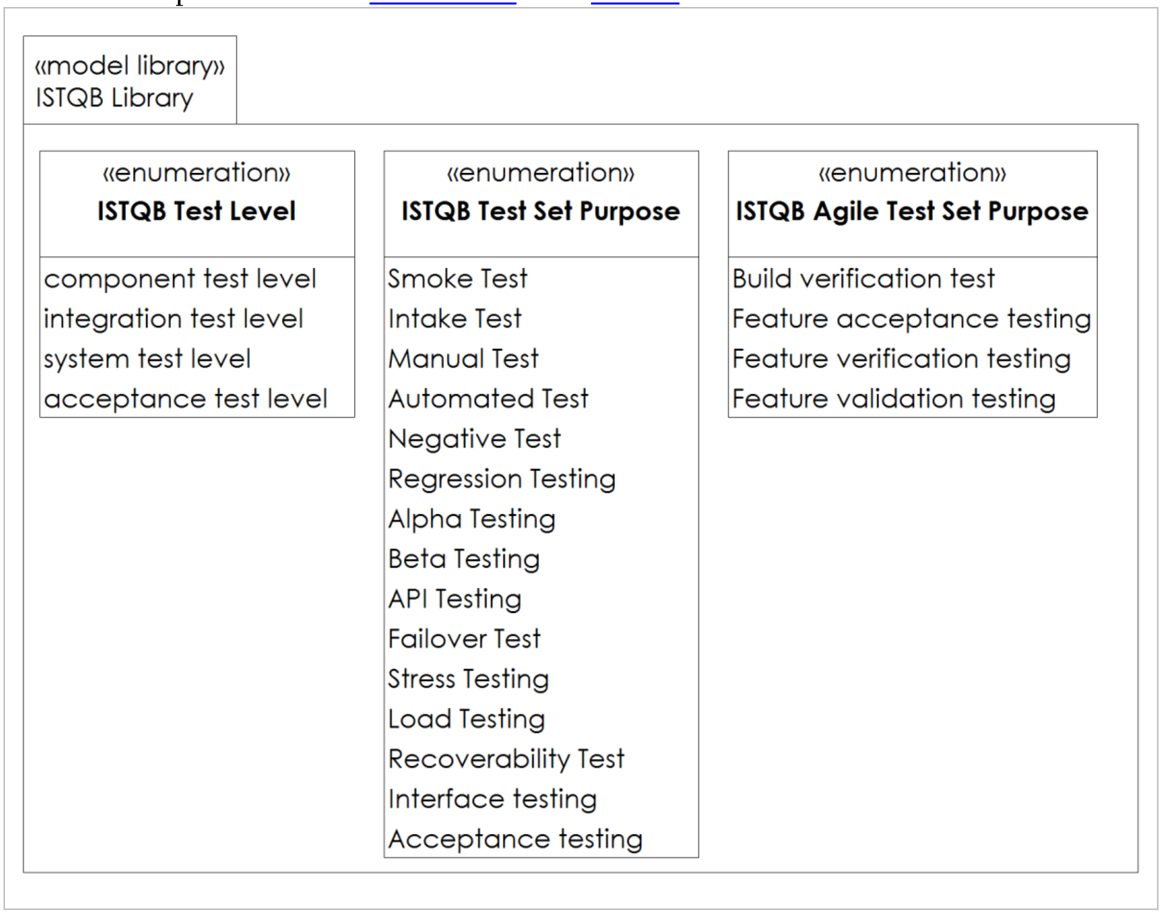


Figure 40 - Overview of the ISTQB library

Name	Description	Enumeration literals
ISTQB Agile Test Set Purpose		Build verification test "A set of automated tests which validates the integrity of each new build and verifies its key/core functionality, stability and testability. It is an industry practice when a high frequency of build releases occurs (e.g., Agile projects) and it is run on every new build before the build is released for further testing." [ISTQB]
		Feature acceptance testing Acceptance testing of a feature, often broken down into Feature verification testing and Feature validation testing.
		Feature verification testing Usually carried out automatically may be done by developers or testers, and involves testing against the user story's acceptance criteria.
		Feature validation testing Usually carried out manually and can involve developers, testers, and business stakeholders working collaboratively to determine whether the feature is fit for use, to improve visibility of the progress made, and to receive real feedback from the business stakeholders.
ISTQB Test Level	A common set of test levels. A test level is considered as a set of testing activities related to the outermost boundaries of the test items.	component test level A test designed to provide information about the quality of the component.
		integration test level A test designed to provide information about the direct interface between two integrated components for example in the form of a parameter list.

Name	Description	Enumeration literals
		system test level A test designed to assess the quality of the complete system after integration.
		acceptance test level A test designed to demonstrate to the customer the acceptability of the final system in terms of their specified requirements.
ISTQB Test Set Purpose	A set of reasons why test sets might have been assembled.	Smoke Test "A subset of all defined/planned test cases that cover the main functionality of a component or system, to ascertaining that the most crucial functions of a program work, but not bothering with finer details." [ISTQB]
		Intake Test "A special instance of a smoke test to decide if the component or system is ready for detailed and further testing. An intake test is typically carried out at the start of the test execution phase." [ISTQB]
		Manual Test A test set whose test cases will be executed manually.
		Automated Test A test set whose test cases will be executed automatically.
		Negative Test "Tests aimed at showing that a component or system does not work." [ISTQB]

Name	Description	Enumeration literals
		Regression Testing "Testing of a previously tested program following modification to ensure that defects have not been introduced or uncovered in unchanged areas of the software, as a result of the changes made." [ISTQB]
		Alpha Testing "Simulated or actual operational testing by potential customers/users or an independent test team at the software developers' site, but outside the development organization. Alpha testing is employed for off-the-shelf software as a form of internal acceptance testing." [ISTQB]
		Beta Testing "Operational testing by potential and/or existing customers/users at an external site not otherwise involved with the developers, to determine whether or not a component of system satisfies the user needs and fits within the business processes. Note: Beta testing is often employed as a form of external acceptance testing in order to acquire feedback from the market." [ISTQB]
		API Testing "Testing the code which enables communication between different processes, programs and/or systems. API testing often involves negative testing, e.g., to validate the robustness of error handling." [ISTQB]

Name	Description	Enumeration literals
		Failover Test "Testing by simulating failure modes or actually causing failures in a controlled environment. Following a failure, the fail-over mechanism is tested to ensure that data is not lost or corrupted and that any agreed service levels are maintained (e.g., function availability or response times)." [ISTQB]
		Stress Testing "A type of performance testing conducted to evaluate a system or component at or beyond the limits of its anticipated or specified workloads, or with reduced availability of resources such as access to memory or servers. [After IEEE 610]" [ISTQB]
		Load Testing "A type of performance testing conducted to evaluate the behavior of a component or system with increasing load, e.g. number of parallel users and/or numbers of transactions to determine what load can be handled by the component or system." [ISTQB]
		Recoverability Test "The process of testing to determine the recoverability of a software product." [ISTQB]
		Interface testing "An integration test type that is concerned with testing the interfaces between components or systems." [ISTQB]
		Acceptance testing "Formal testing with respect to user needs,

Name	Description	Enumeration literals
		requirements, and business processes conducted to determine whether or not a system satisfies the acceptance criteria and to enable the user, customers or other authorized entity to determine whether or not to accept the system." [ISTQB]

9.2.1.3 Test Design Facility Library

The test design facility library provides a set of test design techniques as well as some default test design technique structures that can be used out of the box for the specification of the test design activities. Since these test design techniques are by definition not dependent upon the test design input element, they are called context-free test design techniques.

9.2.1.3.1 The UTP test design facility library

The following diagram shows the predefined test design techniques provided by UTP 2 to be used for the specification of test directives.

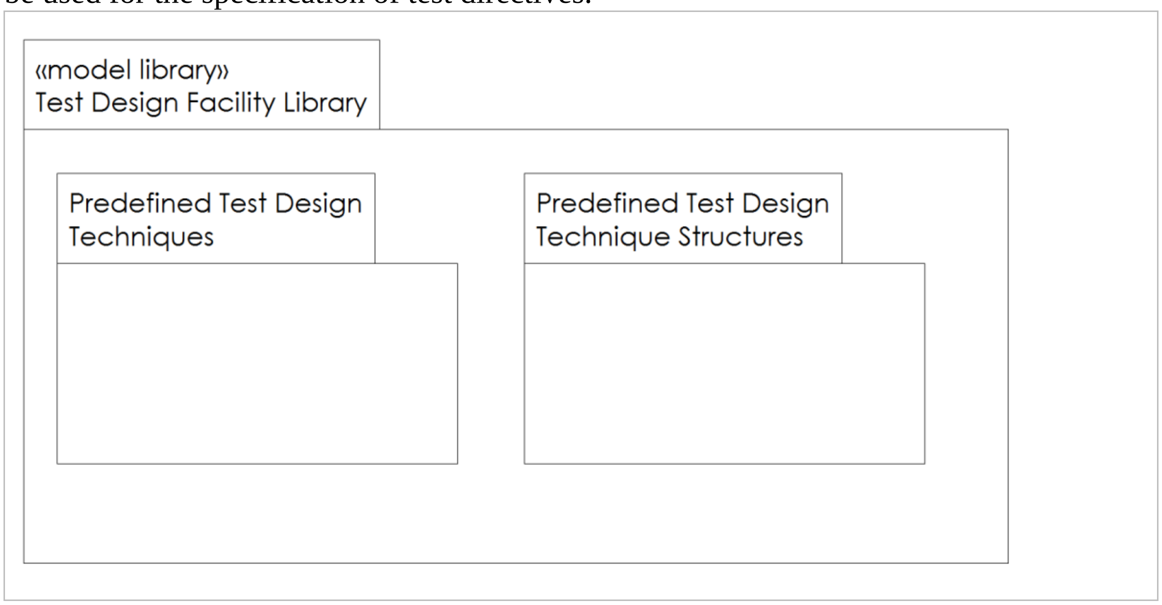


Figure 41 - The UTP test design facility library

9.2.1.3.2 Predefined Test Design Techniques

UTP offers a set of context-free test design techniques, meaning that these test design techniques do not require any further information from the test design input of the assembling test design directive. They can be immediately used by the generic test design directive or any other predefined or specialized test design technique or test design directive.

9.2.1.3.2.1 Predefined context-free test design techniques

The following diagram depicts the predefined and ready-to-use test design technique provided by UTP 2.

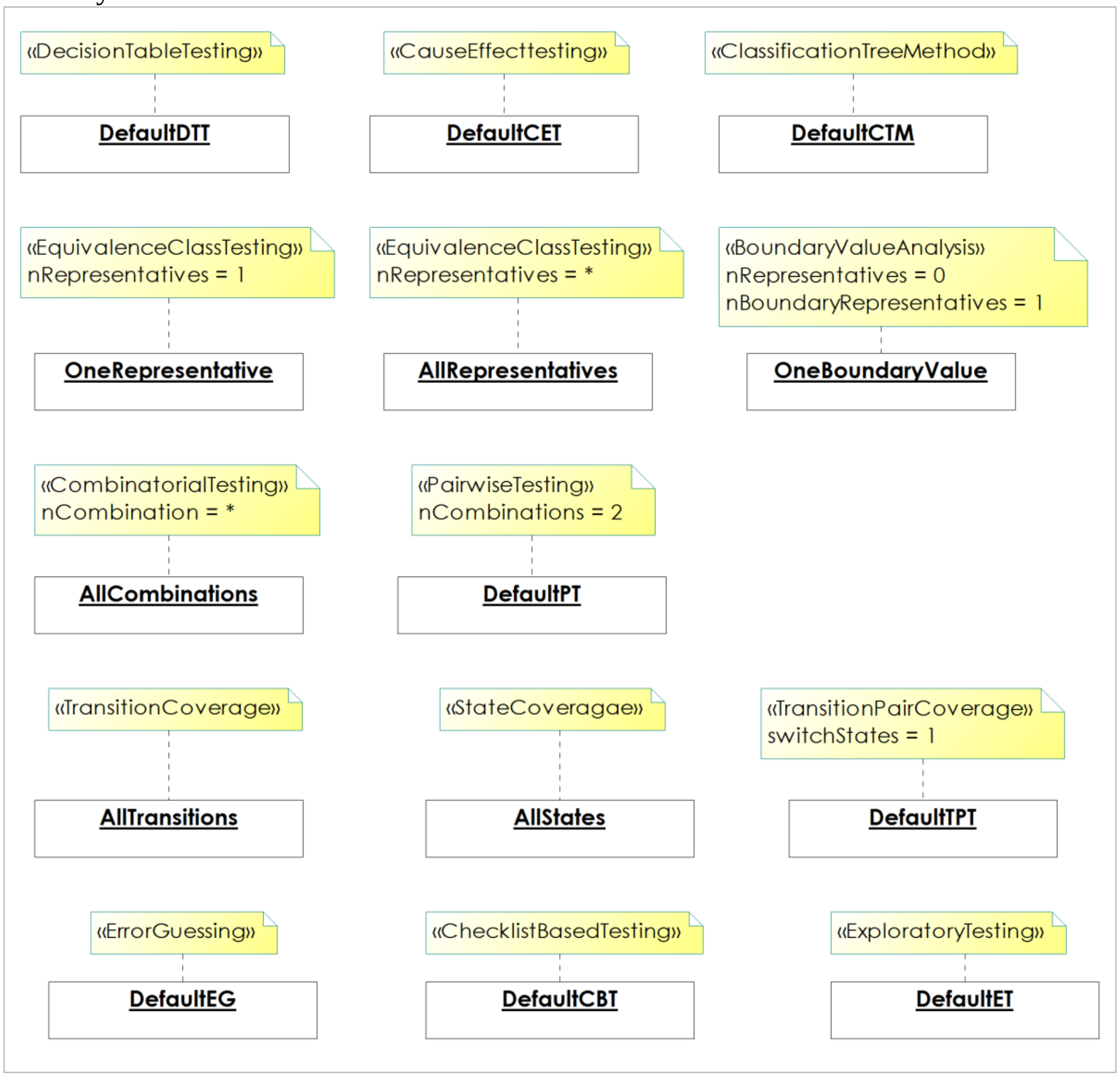


Figure 42 - Predefined context-free test design techniques

Name	Description
AllCombinations	A predefined instance of the CombinatorialTesting TestDesignTechnique ready for being assembled by TestDesignDirectives. The semantics is that all possible

Name	Description
	combinations of input parameters must be covered by the resulting test cases.
AllRepresentatives	A predefined instance of the EquivalenceClassPartitioning TestDesignTechnique ready for being assembled by TestDesignDirectives. All representatives of the equivalence classes must be selected.
AllStates	The predefined instance of the StateCoverage TestDesign-Technique ready for being assembled by TestDesignDirec-tives. The default semantics is that all States of the corre-sponding State Machine(s) must be covered by the resulting test cases.
AllTransitions	The predefined instance of the TransitionCoverage TestDesignTechnique ready for being assembled by TestDesignDirectives. The default semantics is that all Transitions of the corresponding State Machine(s) must be covered by the resulting test cases.
DefaultCBT	The predefined instance of the ChecklistBasedTesting TestDesignTechnique ready for being assembled by TestDesignDirectives.
DefaultCET	The predefined instance of the CauseEffectAnalysis TestDesignTechnique ready for being assembled by TestDesignDirectives.
DefaultCTM	The predefined instance of the ClassificationTreeMethod TestDesignTechnique ready for being assembled by TestDesignDirectives.
DefaultDTT	The predefined instance of the DecisionTableTesting TestDesignTechnique ready for being assembled by TestDesignDirectives.
DefaultEG	The predefined instance of the ErrorGuessing TestDesign-Technique ready for being assembled by TestDesignDirec-tives.
DefaultET	The predefined instance of the ExploratoryTesting TestDesignTechnique ready for being assembled by TestDesignDirectives.
DefaultPT	The predefined instance of the PairwiseTesting TestDesign-Technique ready for being assembled by TestDesignDirec-tives.
DefaultTPT	The predefined instance of the TransitionPairTesting TestDesignTechnique ready for being assembled by TestDesignDirectives. The default semantics is that at least all pairs of subsequent Transitions must be covered by the

Name	Description
	resulting test cases.
OneBoundaryValue	The predefined instance of the BoundaryValueAnalysis TestDesignTechnique ready for being assembled by TestDesignDirectives. The default semantics is that a single value at the boundaries of the equivalence class must be selected.
OneRepresentative	A predefined instance of the EquivalenceClassPartitioning TestDesignTechnique ready for being assembled by TestDesignDirectives. Exactly one representative of each equivalence class must be selected.

9.2.1.3.3 Predefined Test Design Technique Structures

The predefined test design technique structures offer some structural information to enrich test design techniques, if required.

9.2.1.3.3.1 Overview of the predefined test design technique structures

The following diagram depicts the predefined and ready-to-use test design technique structures provided by UTP. They can be used to build proprietary generic test design techniques or to augment the predefined test design techniques.

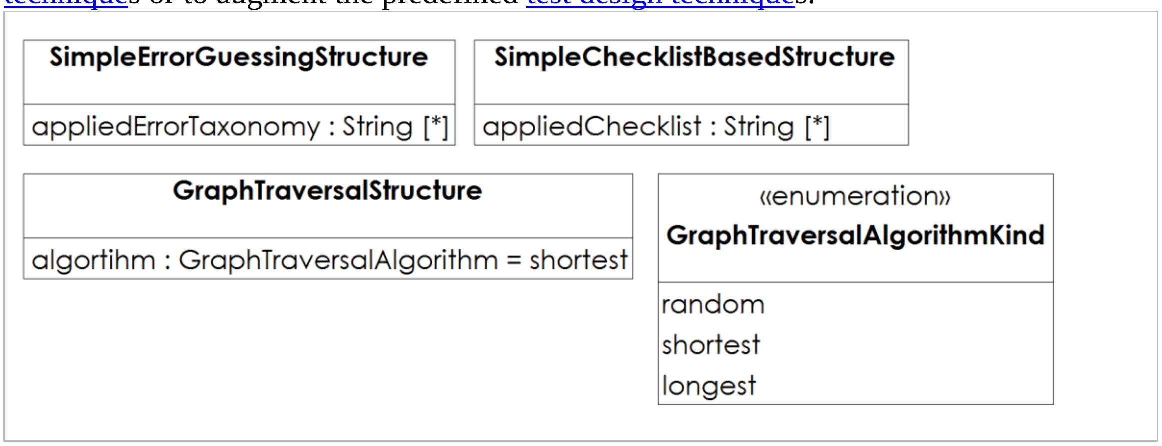


Figure 43 - Overview of the predefined test design technique structures

Name	Description
GraphTraversalStructure	A test design technique structure that enables testers to specify the traversal algorithm a test designing entity must apply.
SimpleChecklist-	A checklist-based test design technique that enables test

Name	Description
BasedStructure	engineers to refer to some checklists that should be used for test design.
SimpleErrorGuessing-Structure	An error guessing test design technique that enables test engineers to refer to some error taxonomies that should be used for test design.

Name	Description	Enumeration literals
Graph-TraversalAlgorithmKind	A set of graph traversal strategies.	random A test designing entity must take a random walk through the graph in order to achieve a certain coverage criterion of the test design input element.
		shortest A test designing entity must take the shortest path possible in order to achieve a certain coverage criterion of the test design input element.
		longest A test designing entity must take the longest path possible to achieve a certain coverage criterion of the test design input element.

9.3 UTP Arbitration Library

This section defines the predefined arbitration specifications provided by the UML Testing Profile (UTP). The role of an arbitration specification is to determine the verdict of the corresponding test set, test case, or procedural element. These verdicts have the following priorities (the former overrides the latter): Error > Fault > Fail > Pass > Inconclusive > None.

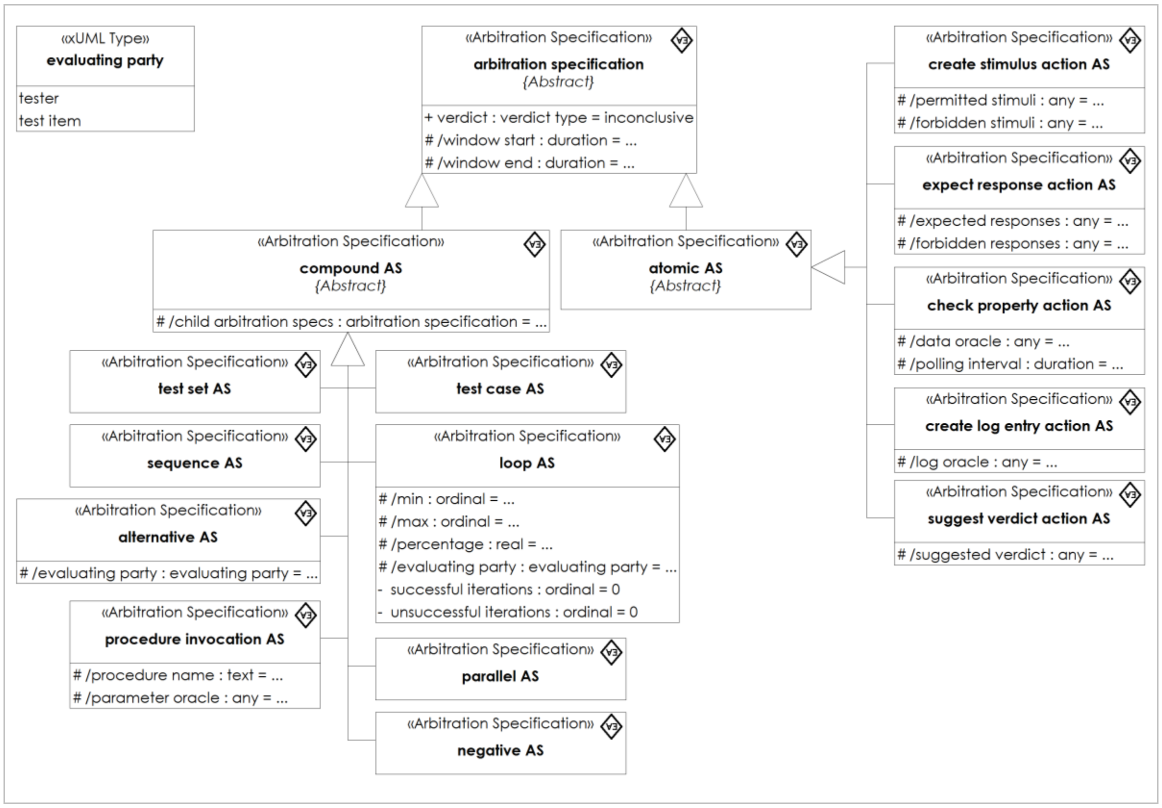


Figure 44 - Default Arbitration Specifications

The class diagram above shows the generalization/specialization hierarchy that reflects the hierarchy of standardized actions to specify test behavior as defined in section Test Behavior. By default, each arbitration specification acts as a private arbiter behind the corresponding action that decides on the verdict (represented as the public attribute verdict) of the corresponding action. Each arbitration specification operates on a number of properties (represented as protected derived attributes in this diagram) that are derived from the corresponding actions. An "oracle" in this context is a boolean function that may decide on the correctness of supplied data. In contrast to atomic AS, compound AS decide their verdict based on the verdict decided by the arbitration specification of one or more child actions that are enclosed by this action. This is reflected by the derived attribute /child arbitration specs. Some arbitration specifications rely their decision on the fact of who (i.e., the tester or the test item) is deciding on a condition evaluated within a child action (e.g., the condition that decides to continue a loop or to decide on one of a set of alternatives). This is reflected by derived attributes of the enum type evaluating party.

Arbitration Specification "alternative AS"

Arbitration Specification alternative AS	

Description	**Tester instructions:** within the time window, perform exactly one of the child actions.
	
	Arbitration specification:
	• None if not yet started
	• Inconclusive while running
	• Error if technical error occurred
	• Fault if /evaluating party is tester and
	    • more than one /child arbitration specs performed or
	    • /child arbitration specs completed before time window or
	    • /child arbitration specs completed after time window or
	    • /child arbitration specs reports Fault
	• Fail if /evaluating party is test item and
	    • more than one /child arbitration specs performed or
	    • /child arbitration specs completed before time window or
	    • /child arbitration specs completed after time window or
	    • /child arbitration specs reports Fail
	• Pass if exactly one /child arbitration specs performed and returned Pass

Attributes	/evaluating party : evaluating party [1] = action.evaluating party Defines which side (tester or test item) is deciding on the selection of an alternative.

Arbitration Specification "check property action AS"

Arbitration Specification check property action AS	

Description	**Tester instructions:** measure/monitor value of a property of the test item within the time window with minimal polling interval.
	
	Arbitration specification:
	• None if not yet started
	• Inconclusive while running
	• Error if technical error occurred
	• Fault if
	    • not polled at least once or
	    • polling interval exceeded
	• Fail if any value of property within the time window outside value limits
	• Pass if all values of property within the time window inside value limits

Attributes	/data oracle : any [1] = action.oracle Defines a boolean function that, when supplied with the actual value to be checked, determines whether it matches the expectations.
	/polling interval : duration [1] = action.polling interval Defines the minimum polling interval that is required to measure the property during the time window.

Arbitration Specification "create log entry action AS"

Arbitration Specification create log entry action AS	

Description	**Tester instructions:** within the time window, write the specified information to the current test log.
	
	Arbitration specification:
	• None if not yet started
	• Inconclusive while running
	• Error if technical error occurred
	• Fault if
	    • no wrong log entry created within the time window or
	    • wrong log entry created within the time window
	• Fail never
	• Pass if correct log entry created within the time window

Attributes	/log oracle : any [1] = action.oracle Defines a boolean function that, when supplied with the actual log entry to be checked, determines whether it matches the expectations.

Arbitration Specification "create stimulus action AS"

<table>
<tr><th colspan="2">Arbitration Specification create stimulus action AS</th></tr>
<tr><td>Description</td><td>Tester instructions: Within the time window, send exactly one of a set of permitted stimuli with specified payload, but none of a set of forbidden stimuli to the test item.

Arbitration specification:
None if not yet started
Inconclusive while running
Error if technical error occurred
Fault if
stimulus outside the time window or
stimulus not occurred at all or
stimulus carried wrong payload
Fail never
Pass if action completed</td></tr>
<tr><td rowspan="2">Attributes</td><td>/permitted stimuli : any [1] =
action.expected messages

Defines the set of stimuli that are permitted within the time window of this action.</td></tr>
<tr><td>/forbidden stimuli : any [1] =
action.forbidden messages

Defines the set of stimuli that are forbidden within the time window of this action.</td></tr>
</table>

Arbitration Specification "expect response action AS"

Arbitration Specification expect response action AS	

Description	**Tester instructions:** within the time window, receive one of a set of permitted responses with specified payload, but none of a set of forbidden responses from the test item.
	
	Arbitration specification:
	• None if not yet started
	• Inconclusive while running
	• Error if technical error occurred
	• Fault never
	• Fail if
	  • response outside the time window or
	  • response not occurred at all or
	  • response carried wrong payload
	• Pass if
	  • one of the permitted responses occurred within the time window and
	  • response carried correct payload and
	  • none of the forbidden responses occurred within the time window

Attributes	/expected responses : any [1] = action.expected messages Defines the set of responses that are permitted within the time window of this action.
	/forbidden responses : any [1] = action.forbidden messages Defines the set of responses that are forbidden within the time window of this action.

Arbitration Specification "loop AS"

Arbitration Specification loop AS	

Description	**Tester instructions:** within the time window, perform the child action between min and max times (min=0 and max=1 represents an optional child action).
	
	Arbitration specification:
	• None if not yet started
	• Inconclusive while running
	• Error if
	    • technical error occurred or
	    • the /child arbitration specs reports Error
	• Fault if /evaluating party is tester and
	    • the number of /child arbitration specs iterations after the time window is not within /min .. /max or
	    • any of the /child arbitration specs reports Fault
	• Fail if /evaluating party is test item and
	    • the number of /child arbitration specs iterations after the time window is not within /min .. /max or
	    • more than /percentage of the /child arbitration specs iterations report Fail
	• Pass if the number of the /child arbitration specs iterations that reported Pass after the time window is within /min .. /max

Attributes	/min : ordinal [1] = action.min Defines the minimum number of iterations that must be executed within this loop.
	/max : ordinal [1] = action.max Defines the maximum number of iterations that must be executed within this loop.
	/percentage : real [1] = action.percentage Defines the percentage of the minimum number of iterations

	that must be successfully executed within this loop in order to get a "pass" verdict for this action.
	/evaluating party : evaluating party [1] = action.evaluating party Defines which side (tester or test item) is deciding on the continuation of a loop.
	successful iterations : ordinal [1] = 0 Contains the actual number of successful iterations executed so far within this loop.
	unsuccessful iterations : ordinal [1] = 0 Contains the actual number of unsuccessful iterations executed so far within this loop.

Arbitration Specification "negative AS"

Arbitration Specification negative AS	
Description	**Tester instructions:** within the time window, the child action may not be performed. **Arbitration specification:** • None if not yet started • Inconclusive while running • Error if technical error occurred • Fault if /child arbitration specs reports Fault • Fail if action completed and /child arbitration specs reported Pass • Pass if action completed and /child arbitration specs reported Fail

Arbitration Specification "parallel AS"

Arbitration Specification parallel AS	
Description	**Tester instructions:** within the time window, perform all child actions concurrently (i.e., in any order including inter-leaving). **Arbitration specification:** • None if not yet started • Inconclusive while running • Error if technical error occurred • Fault any of the completed reports Fault • Fail any of the completed reports Fail • Pass all completed reported Pass

Arbitration Specification "procedure invocation AS"

Arbitration Specification procedure invocation AS	

Description	**Tester instructions:** within the time window, invoke the specified procedure with the specified parameters and wait for its completion within the timeout period.
	
	Arbitration specification:
	• None if not yet started
	• Inconclusive while running
	• Error if
	    • technical error occurred or
	    • /child arbitration specs not completed before timeout
	• Fault if not invoked within time window
	• Fail if action completed and /child arbitration specs reported Fail
	• Pass if action completed and /child arbitration specs reported Pass

Attributes	/procedure name : text [1] = action.procedure name Defines the name of the procedure to be invoked by this action.
	/parameter oracle : any [1] = action.oracle Defines a boolean function that, when supplied with the actual procedure parameters to be checked, determines whether they match expectations.

Arbitration Specification "sequence AS"

Arbitration Specification sequence AS	

Description	**Tester instructions:** within the time window, perform all child actions in specified sequence.
	
	Arbitration specification:
	• None if not yet started
	• Inconclusive while running
	• Error if
	    • technical error occurred or
	    • the /child arbitration specs reports Error
	• Fault if any child action reports Fault
	• Fail if
	    • no /child arbitration specs reported Fault and
	    • any of the /child arbitration specs reported Fail
	• Pass if action completed and all /child arbitration specs reported Pass

Arbitration Specification "suggest verdict action AS"

Arbitration Specification suggest verdict action AS	

Description	**Tester instructions:** within the time window, suggest the specified verdict to the current Arbitration specification.
	
	Arbitration specification:
	• None if not yet started
	• Inconclusive while running
	• Error if
	    • technical error occurred or
	    • suggested verdict was error
	• Fault if
	    • no verdict suggested within the time window or
	    • suggested verdict was Fault
	• Fail suggested verdict was Fail
	• <suggested verdict> if correct verdict specified within the time window

Attributes	/suggested verdict : any [1] = action.oracle Defines the verdict that is suggested by this action to its the parent action.

Arbitration Specification "test case AS"

<table>
<tr><th colspan="2">Arbitration Specification test case AS</th></tr>
<tr><td>Description</td><td>

Tester instructions: perform the setup procedure (if defined) then, the main procedure, and finally the teardown procedure (if defined).

Arbitration specification:

- None if not yet started
- Inconclusive while running
- Error if technical error occurred
- Fault if
  - the setup procedure (if defined) has not been invoked or
  - the setup procedure (if defined) returned Fault or
  - the main procedure has not been Fault or
  - the main procedure returned Fault or
  - the teardown procedure (if defined) has not been invoked or
  - the teardown procedure (if defined) returned Fault
- Fail if the main procedure returned Fail
- Pass if the main procedure returned Pass

</td></tr>
</table>

Arbitration Specification "test set AS"

Arbitration Specification test set AS	
Description	**Tester instructions:** perform all test cases within the test set in the sequence specified by the test execution schedule (if defined). **Arbitration specification:** • None if not yet started • Inconclusive while running • Error if technical error occurred • Fault if any of the /child arbitration specs returned Fault • Fail if any of the /child arbitration specs returned Fail • Pass if all of the /child arbitration specs returned Pass

10 Annex A (Informative): Examples

This section illustrates some concepts of the UML Testing Profile by means of different examples. These examples were provided by different companies reflecting different approaches to MBT, different interpretations of MBT with UTP and finally different methodologies for applying UTP. It underlines the flexibility and open-endedness of UTP.

10.1 Croissants Example

10.1.1 The Test Item

This example illustrates some of the major concepts of UTP 2 on the "not so serious" test item (French) "Croissants". This is a particularly interesting example since the test item is not a software system (at least not in the classical sense ;-), but a rather common physical system (i.e., croissants).

Figure 45 - The Croissants Example

10.1.1.1 Given Requirements on the Test Item

Id	Type	Description	Req. on
RQ-0001	functional	Each croissant shall have a chocolate core	Croissant
RQ-0002	functional	Each croissant shall have a consistency of greater than 3	Croissant
RQ-0003	functional	Each croissant shall be considered as "good tasting" by more than 80% of ordinary people	Croissant

10.1.2 Test Requirements

The following diagram shows the hierarchy of test objectives as well as the constraints on this test series expressed as test requirements.

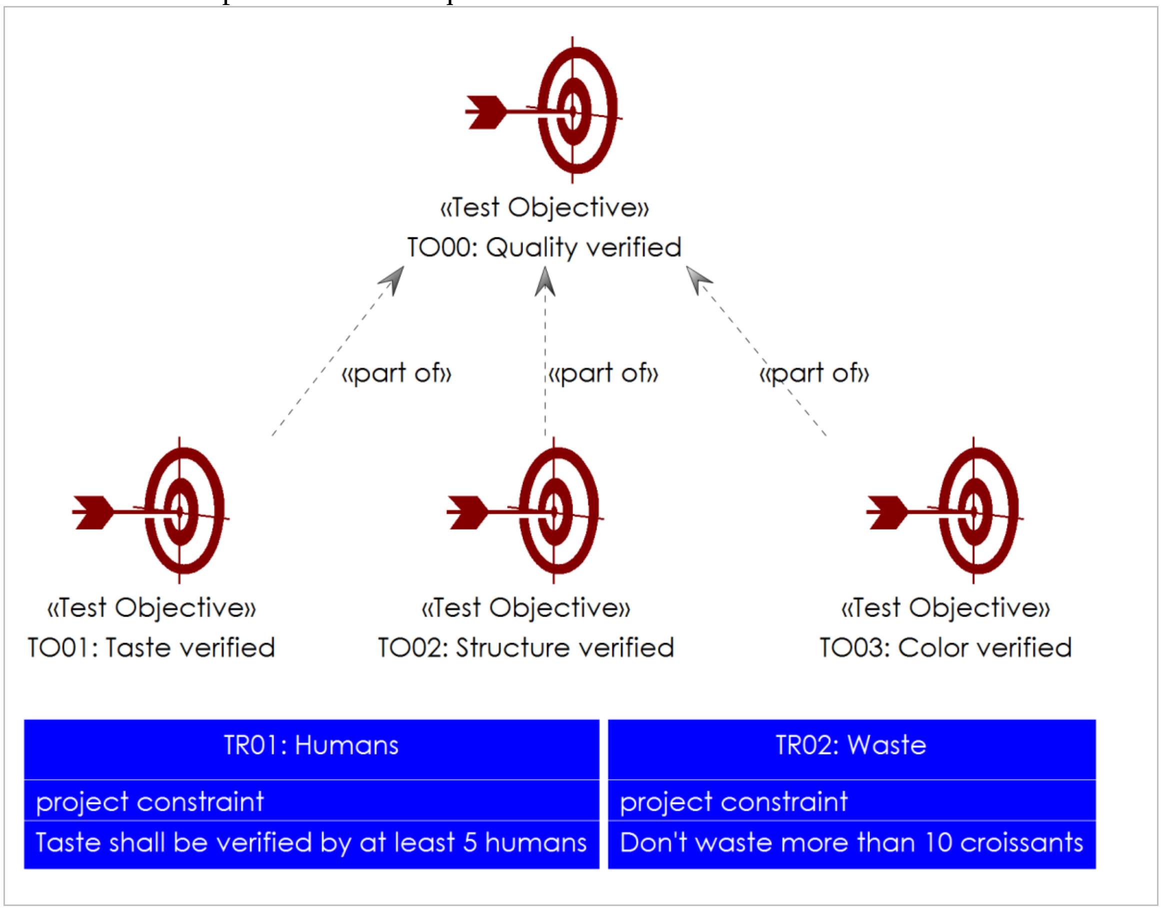


Figure 46 - Test Objectives

10.1.2.1 Given Test Objectives

Name	Description	Priority
TO00: Quality verified	The high quality of the croissants we enjoy during our working meetings is ensured.	n/a
TO01: Taste verified	The quality of the flavor of the croissants we enjoy during our working meetings is ensured.	high
TO02: Structure verified	The physical composition of the croissants we enjoy during our working meetings is ensured.	medium
TO03: Color verified	The tasteful look of the croissants we enjoy during our working meetings is ensured.	high

10.1.2.2 Given Requirements

TR01: Humans	
Description	Taste shall be verified by at least 5 humans
Requirement type	project constraint
Requirement kind	Quality

TR02: Waste	
Description	Don't waste more than 10 croissants
Requirement type	project constraint
Requirement kind	Resource Consumption

10.1.3 Test Design

The following diagram shows the applied test design strategy as well as the test directives derived from that test design strategy.

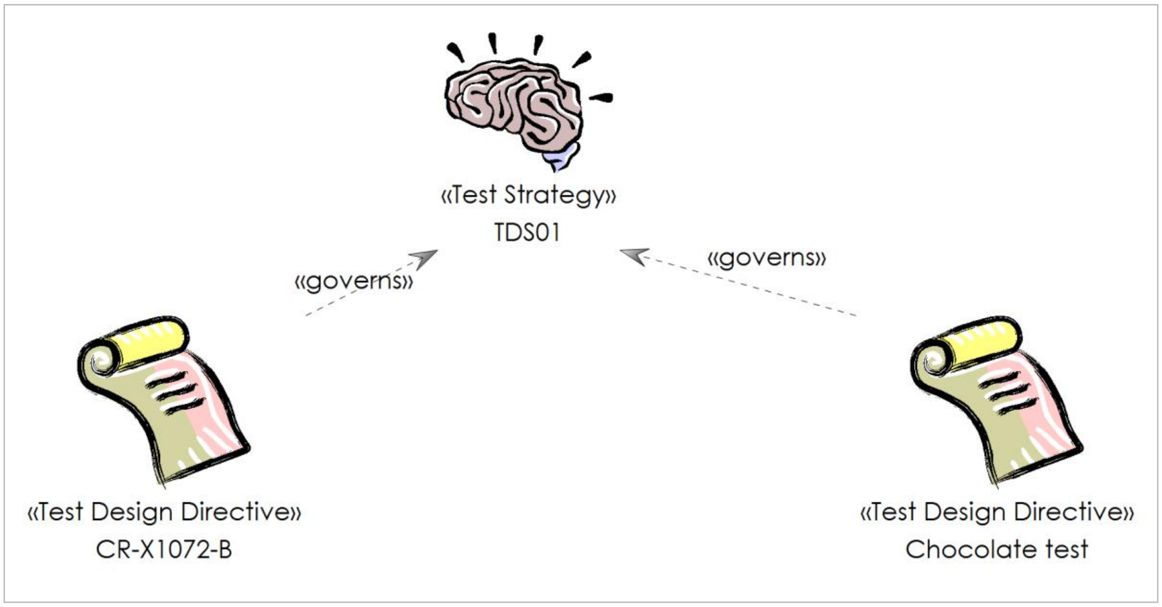


Figure 47 - Test Strategy

10.1.3.1 Test Design Strategies shown on "Test Strategy"

TDS01	
Description	At least 5 members of the UTP 2 WG will take a bite of a croissant.

10.1.3.2 Test Directives shown on "Test Strategy"

Chocolate test	
Description	Keep every piece of chocolate at least 10 seconds on your tongue.
Applies to	Chocolate Portion
Requires capability	Gustaoceptionary Proficiency

CR-X1072-B	
Description	Apply Croissant-Standard CR-X1072-B to test them.
Applies to	Croissant
Requires capability	Knowledge of CR-X1072-B

10.1.4 Test Configuration

The figure below shows the Test Configuration of the Croissants abstracted as a UML class diagram.

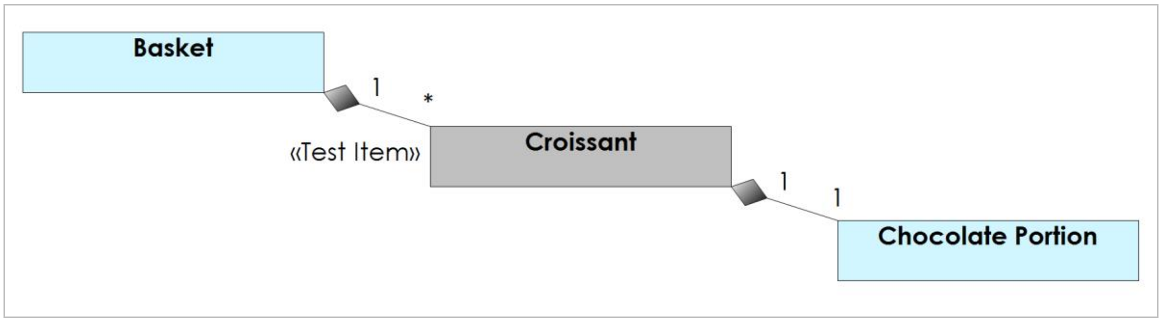


Figure 48 - Objects

Based on this description, the following figure shows the concrete test configuration instantiated as a composite structure diagram.

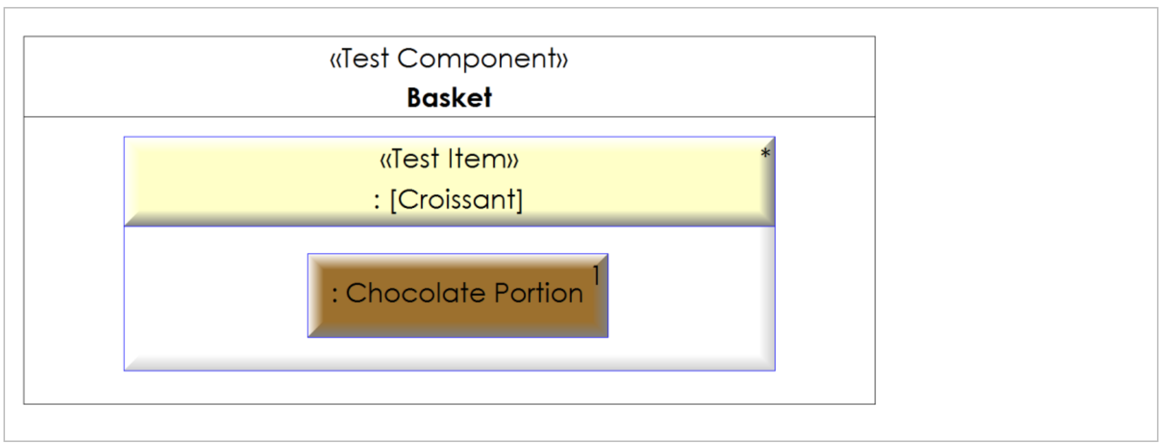


Figure 49 - Test Configuration

10.1.5 Test Cases

The test cases (particularly the test procedures) in this test set are not specified fully and formally but rather in a structured informal way. This is to show that test cases in UTPs don't always have to be fully formalized.

10.1.5.1 Test Set "Manual croissants test"

The following diagram shows the Test Set "Manual croissants test" containing the relevant test cases and how they relate to the stated test objectives. Further, the test requirements constraining this test set also are shown.

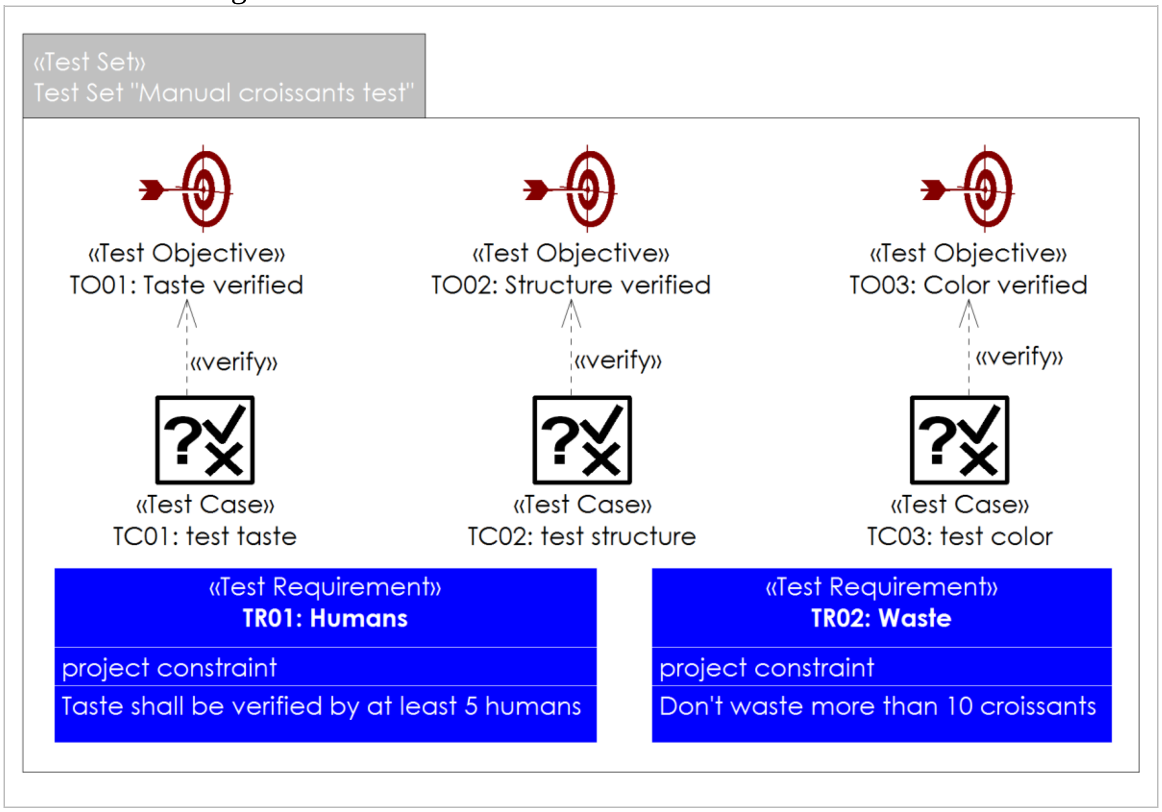


Figure 50 - Test Map

Test Cases shown on "Test Map"

TC01: test taste	
Test objectives	TO01: Taste verified
Priority	high
Precondition	• There must be a Croissant available
Test procedure	Apply the following steps: • Break the Croissant in its middle • Check whether there is chocolate in it • Bite into the Croissant • Evaluate its taste • Eat the remains or throw them into the waste basket
Postcondition	• The Croissant is eaten
Verifies	TO01: Taste verified

Estimated effort	10 seconds
Is abstract	FALSE

TC02: test structure	
Test objectives	TO02: Structure verified
Priority	low
Precondition	• There must be a Croissant available • The Croissant must not be broken
Test procedure	Apply the following steps: • Press the Croissant with two fingers • Check the elasticity of the Croissant • Bend the Croissant until it breaks • Check the breaking angle • Eat the remains or throw them into the waste basket
Postcondition	• The Croissant is broken
Verifies	TO02: Structure verified
Estimated effort	20 seconds
Is abstract	FALSE

TC03: test color	
Test objectives	TO03: Color verified
Priority	medium
Precondition	• There must be a Croissant available
Test procedure	Apply the following steps: • Look at the Croissant • Evaluate its color
Postcondition	• There is still a Croissant available
Verifies	TO03: Color verified
Estimated effort	5 seconds
Is abstract	FALSE

10.2 LoginServer Example

The LoginServer example represents a simplified version of a real case study taken from the EU FP7 research project REMICS. It was optimized for the initial submission section to demonstrate the core concepts of UTP 2 that are stable enough and unlikely to be substantially changed in the revised submissions. The LoginServer offers functionality to log into a system (in the mentioned REMICS project, the login functionality was integrated into a Cloud-based system for managing travel excursions). In this example section, the following capabilities of UTP 2 are demonstrated:

- Defining the structure of a test plan using test contexts as well as test level and test types
- Specification of test requirements as a result of the test analysis activities
- Modeling of the logical interfaces of the test item (also known as test item - test item) optimized for deriving logical test cases
- Modeling of the test type system and data specifications required for deriving appropriate data
- Specification of structural aspects of the test environment, in particular the required test components, test configuration and connection between the test environment and the test item
- Modeling of logical test cases using sequence diagrams (i.e., Interactions)
- Informal and rough description of a mapping from UTP 2 test cases expressed as sequence diagrams (i.e. Interactions) to semantically equivalent TTCN-3 test scripts

This example demonstrates the Test Model-only approach to model-based testing. There are no further (e.g., design or requirements) models available for reuse. In addition, the methodology follows the so called test requirement/requirements analysis , since the test design activities are guided by test requirements which, in turn, are derived from the test basis. Both the applied MBT approach and the test approach (which is called test practice in ISO 29119) of the LoginServer example are just a single interpretation how UTP 2 could be used and embedded into a methodology. The described test process and its distinct phases (e.g., test planning, test analysis, etc.) are inspired by the ISTQB fundamental test process.

10.2.1 Requirements Specification

The following table shows a simplified excerpt of the requirements for the LoginServer example. These few requirements suffice to demonstrate most of the core concepts of UTP 2.

Id	Name	Description
F1	User login	The user shall be able to log into the system using a valid ID/password combination.
F2	Failed user login	The system shall reject the login request and answer with an appropriate error message, if the user tries to log into the system with a known ID but invalid password.
F3	Unknown user login	The system shall reject the login request and answer with an appropriate error message, if an unknown user (i.e., a non-registered ID) requests a login.
F4	User banishing	The system shall banish an ID and answer with an appropriate message, if a user tries to log into system three times in a row with an invalid ID/password combination.
F5	Mail address modification	A user who is logged into the system shall be able to update his mail address. A valid mail address complies to the following regular expression: [a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,4}
F6	User logout	A user who is logged in shall be able to log out from the system.
F7	Login response time	The system shall respond to login request within 5 seconds.

Table 11 - LoginServer Requirements

10.2.2 Test Planning

In the test planning phase, the test manager usually starts specifying the test plan. This means that the resources for testing are estimated, requested and allocated. Furthermore, the test process is broken down into so called test sub-processes, each strives to fulfil the test objectives of this test sub-process. These test sub-processes are called test context in UTP 2.

Based on the knowledge about the system to be tested (also known as test item or test item), the test manager decides on the number of test sub-processes, their objectives and the strategies used to fulfil those test objectives. The diagram below shows the corresponding structure of the test specification for the LoginServer test item.

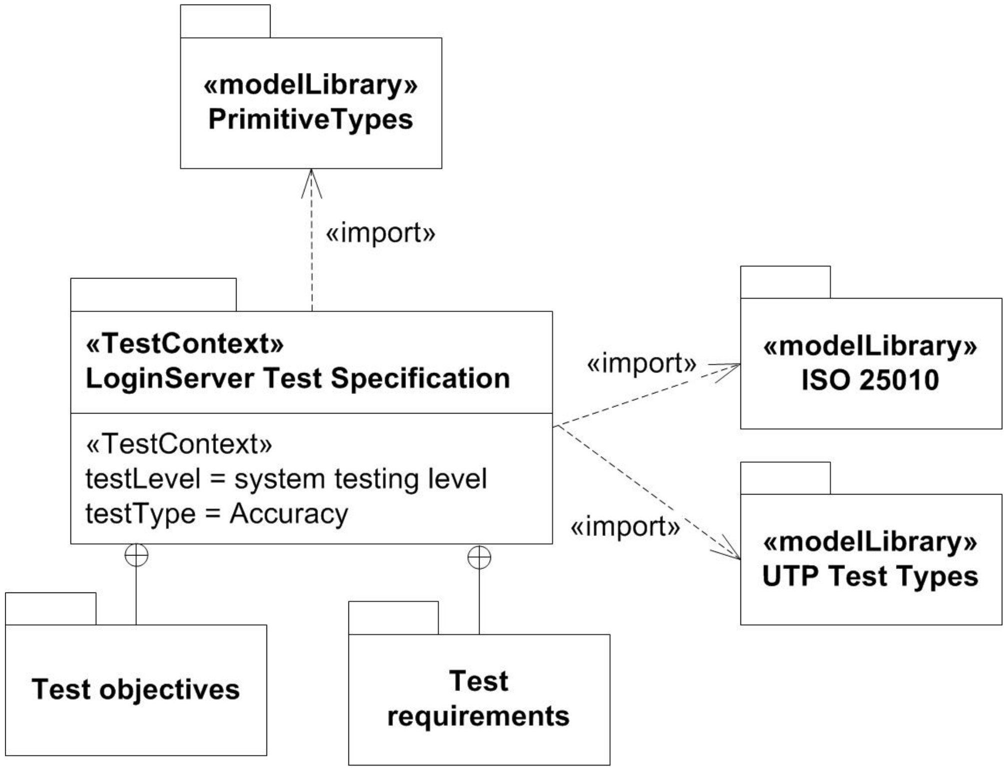


Figure 51 - The LoginServer Test Context

Due to the simplicity of the LoginServer, the entire test plan only consists of a single test context. In more sophisticated test processes, the test plan is usually sub-structured into multiple test (sub-)plans, so called master and level test plans. The test context copes with this need, since it allows for sub-structure test contexts. The diagram above also demonstrate the use of two model libraries provided by the UTP Auxiliary library in order to specify the test level and test type that are addressed by the given test context. In this example, the test context LoginServer Test Specification targets functional system testing. Each test case accessible to the test context is considered to be designed for the mentioned test level and test type. This enables tester to apply the very same test case to different test types and test levels (if needed). For example, it is a good practice to reuse functional test cases with different data sets or a different, yet compatible test configuration for security or performance testing.

The LoginServer Test Specification contains two ordinary packages for storing the test objectives and test requirements. Whereas the specification of test objectives is not shown in this example, the derivation of test requirements as one of the most important outcomes of the test analysis phase will be shown in the next section.

10.2.3 Test Analysis

The activities in the test analysis phase are, first and foremost, dedicated to analyze the test basis in order to comprehend both the test item and what is expected from the test item. Test basis is an abstract concept that comprises any information that helps deriving test cases for a certain test item with respect to the test objectives of the given test sub-process (i.e., test context). The requirements specification usually represents an important part of the test basis for functional system testing.

10.2.3.1 Derivation and Modeling of Test Requirements

In UTP, test requirements specify which features of a requirement should be verified by corresponding test cases. test requirements are an important means to establish traceability between test cases and the test basis, in particular the requirements. The degree of detail of test requirements varies between test processes and depends on different aspects like the applied test methodology, details of the test basis, sufficient time available to actually specify, review and validate those test requirements etc.

For the given example, only a subset of all possible test requirements is derived from the functional system requirements. For later submission, this specification will provide a more elaborated and complete example.

Id	Description	Covers	Comments
TR-F1-1	Ensure that a user successfully logs into the system, if the login request is performed using a valid ID and corresponding password.	User login	No information about response of the definition of valid ID yet. Req. change request submitted (RCR-ID: 0015)
TR-F1-2	Ensure that the system responses with an error message "Invalid ID" if an invalid ID was provided with the login request.	User login	Invalid ID behavior discussed with system architect. An according req. change request was submitted (RCR-ID: 0016)
TR-F5-1	Ensure that the system responses with a message "Mail address updated" if the modification request was successful. This requires a valid mail address. Valid mail addresses shall comply with the following regular expression: [a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,4}	Mail address modification	No information about response of the system available yet. Req. change request submitted (CR-ID: 0064). The current expected result is very likely to change in future.
TR-F5-2	Ensure that the system issues an error message "Invalid Format" if the mail address	Mail address mod-	No information about response of the system available yet. Req. change request

	the user submitted for modification does not comply with the regular expression given in F5.	ification	submitted (CR-ID: 0065). The current expected result is very likely to change in future.
TR-F5-3	Ensure that the system rejects the modification request if the user is not logged into the system with the error message "Login required".	Mail address modification	No information about response of the system available yet. Req. change request submitted (CR-ID: 0065). The current expected result is very likely to change in future.
TR-F6-1	Ensure that a user, who is currently logged into the system and requests a logout from the system, is actually logged out. The system shall responds with a message "User logged out"	User logout	
TR-F6-2	Ensure that the system responds with an error message "Logout requires to be logged in" if a user who is not logged into the system sends a logout request.	User logout	
TR-F7-1	Ensure that the system responds to login requests within 5 seconds.	Login response time	

Table 12 - Test Requirements

The diagram below depicts the content of the corresponding test requirement package. To keep the diagram clean, only unique identifier of the test requirements are shown. In this methodology, test requirements do not have a name, so the name is automatically (by virtue of a UTP 2 tool) kept in synch with the unique identifier. Unfortunately and deliberately for this example, the targeted requirements are not available as model artifacts, but stored somewhere else (e.g., a dedicated requirements management tool like DOORS or even Excel). Traceability from test requirements to requirements (i.e., from the test specification to the test basis) by means of UTP 2 can at most be established informally.

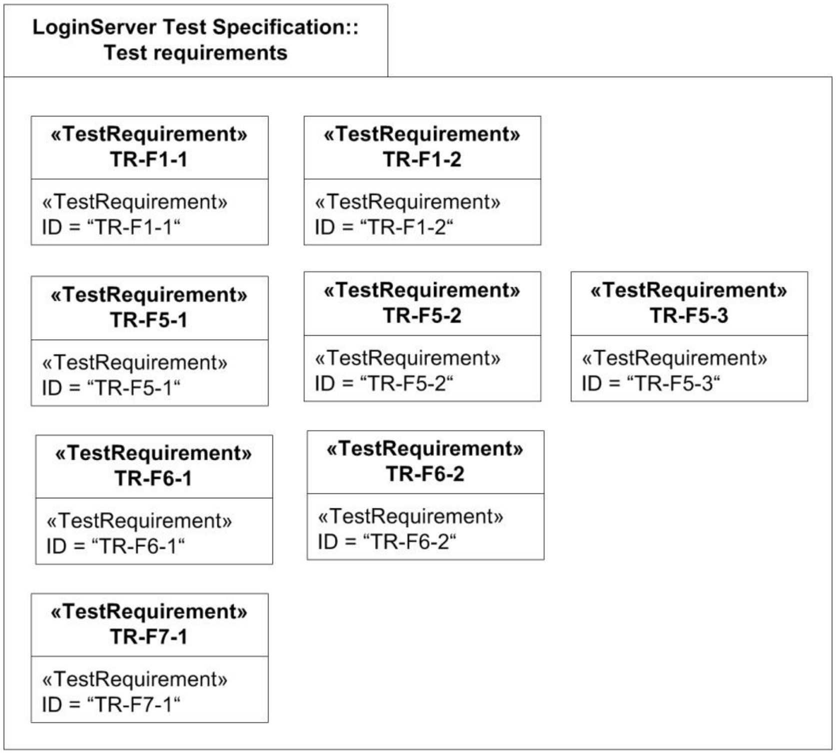


Figure 52 - Test Requirements

10.2.3.2 Modeling the Type System and Logical Interfaces

Since the test model is designed in a standalone manner, it is in the responsibilities of the test analysts to identify and specify the means for interacting with the test item. test requirements usually provide further information for the design of the logical interfaces of the test item and the test type system used for information exchange. For example, the phrase "a user … logs into the system if the login request is performed using a valid ID and corresponding Password." indicates that has to be an operation that allows providing an ID and a Password for a login request. Of course, the same holds true, of course, for the specification of constraints on data in order to build data specifications. The test requirements TR-F1-1 and TR-F5-1 are examples in which constraints on data are specified. These data constraints could be exploited for data-based test design strategies like equivalence class partitioning or boundary value analysis. Whatever test design technique will be applied, UTP 2 offers the required capabilities to capture such data constraints and explicitly specify data specifications as means of equivalence classes or even classification trees.

The diagram below shows the logical interface operations and test type systems derived from the test requirements TR-F1-1, TR-F2-1, TR-F6-1 and TR-F6-2.

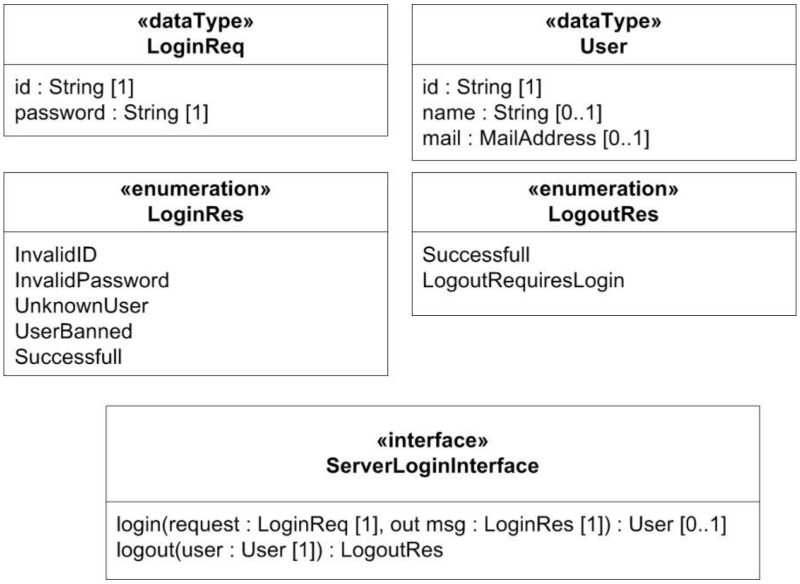


Figure 53 - Logical Interface of LoginServer (1)

The diagram below depicts the logical interface operations and test type systems derived from the test requirements TR-F5-1, TR-F52, and TR-F5-3.

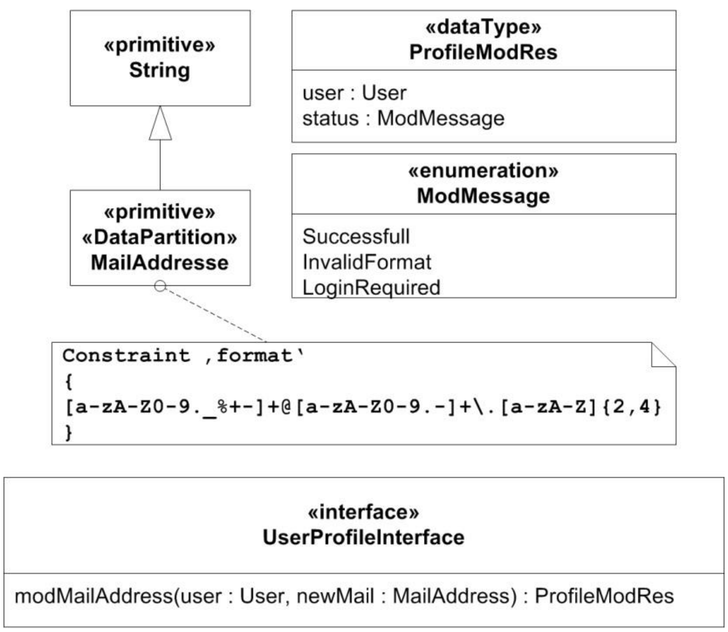


Figure 54 - Logical Interface of LoginServer (2)

10.2.3.3 Modeling Test Data

The data specification MailAddress specialized the primitive type String (provided by the UML PrimitiveTypes package imported by the surrounding test context) and restricts the values for this type according to requirement F5 and test requirements TR-F5-1. The actual specification of the Constraint 'format' is represented by a LiteralString (this cannot be inferred by the means of the diagram). The diagram below shows the corresponding object diagram of the relevant parts of the diagram above.

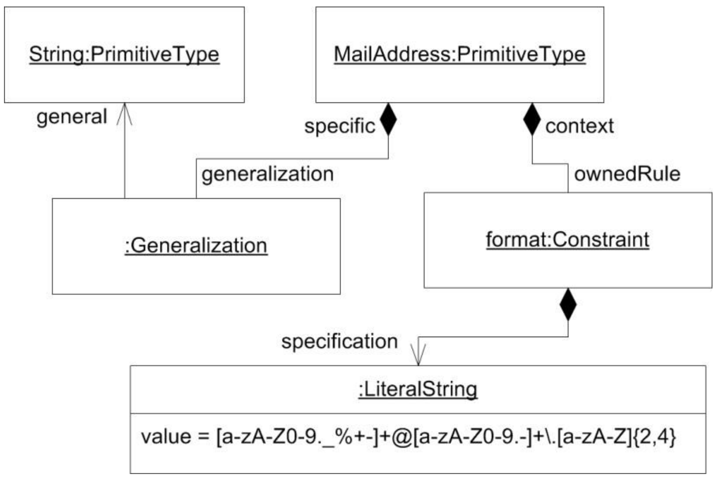


Figure 55 - Object Diagram specifying data

Both names and representation of derived artifacts are just examples how UTP 2 could be applied to support test analysis and depend on the respective methodology.

10.2.4 Test Design

The main target of the test design activity is to derive test cases by following either systematic test design techniques or in an ad-hoc manner. However performed, the test design activity is responsible for

- Deriving according test data based on the test type system
- Deriving the test architecture and test configuration including the communication channels between the test components and the test item
- Designing test cases based on the findings of the test analysis activities
- Link test cases to test objectives and/or test requirements

10.2.4.1 Test Architecture and Test Configuration

The test architecture comprises of the test item and the corresponding test components required driving the execution of test cases against the test item. The diagram below depicts the specification of two components within the LoginServer Test Specification. The decision made to go for two distinct interfaces for the LoginServer instead of a single interface results in a bigger modeling efforts, since an interface component (see BasicPortConfiguration) is required in order to offer multi-offering Ports. This diagram does not make use of any UTP 2 stereotypes but relies completely on the class modeling capabilities of UML. The Port ~basicPort of type Client is a conjugated Port typed by BasicPortConfiguration.

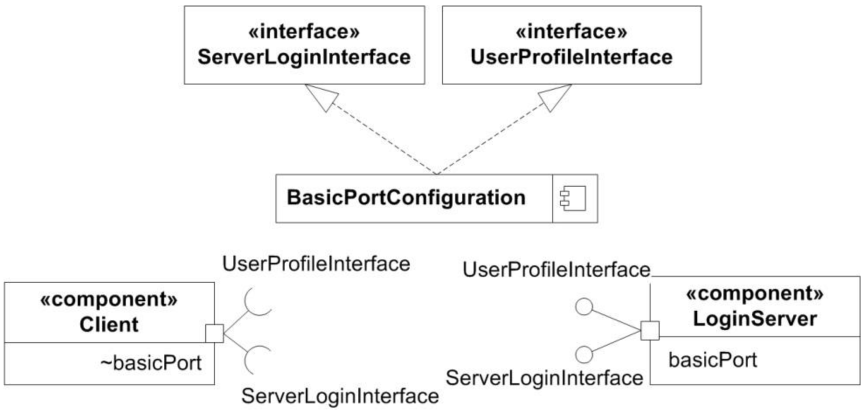


Figure 56 - LoginServer Component Specification

The role each of those components will play in the given test context is not prescribed. Binding of roles for types is accomplished by the test configuration. The test configuration also describes the communication channels over which information exchange among the test component(s) and the test item will be established later. UTP 2 allows for at least two ways to specify the test configuration:

- Shared test configuration: The shared test configuration mechanism enables the test analyst to bind test cases to a previously defined test configuration. By doing so, the test configuration might be reused by different test cases. One means to shared test configuration is by utilizing Collaborations. This is not shown in this example.
- Isolated test configuration: In contrast to shared test configuration, the isolated test configuration builds the test configuration every time from scratch. This option is only possible, if «TestCase» is applied on (a subclass of) Behavior directly. Since Behavior is a StructuredClassifier it is possible to directly make use of the stereotypes «TestItem» and «TestComponent» within the composite structure of the respective Behavior. However, this prevents the advantages of reuse.

The diagram below denotes the very simple test configuration contained in the test case TC1_F1. The test case could be seen as a test case declaration which can be created and fostered very early in the test process. The test configuration comprises two parts, one being stereotyped as «TestComponent» and the other stereotyped as «TestItem», whose compatible Ports are connected by Connector c1. The Connector is an important means for specifying over which communication channel the information exchange between test component(s) and test items are supposed to take place during the execution of the test case.

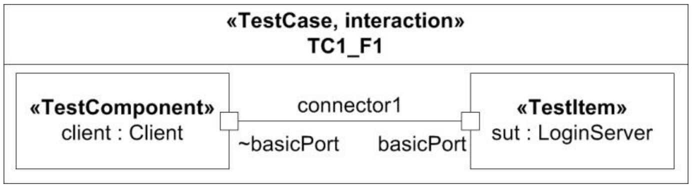


Figure 57 - LoginServer Test Configuration

UTP 2 does not prescribe nor emphasize which variant to be used. Often, this depends on the applied test modeling methodology, the applied tooling, and the acceptance of the test analysts. For example, if generative approaches to test design are applied, then it might not be important to reuse test configurations throughout several test cases for the test configurations would be automatically derived from the boundary descriptions of the «TestItem».

10.2.4.2 Specification of Complex Test Data

The test type system specifies which data types are supposed to be exchanged within test cases among the test components and the test item. For the actual specification of test cases, values or instances for the test type systems need to be defined. This is in particular necessary for complex data types (e.g., DataType, Class, Signal etc.). The diagram below shows the InstanceSpecifications for the data types LoginReq and User required for the realization of test cases.

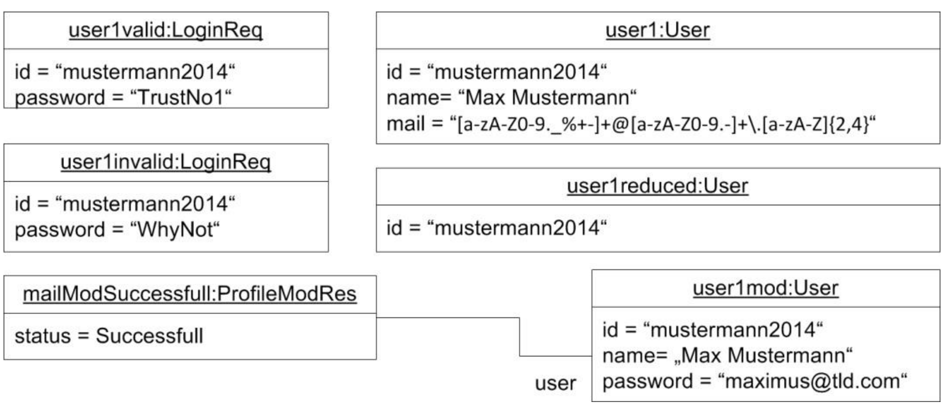


Figure 58 - Test Data Specification

The interesting aspect in the data specification is the difference in dealing with the mail address attribute in the User-type InstanceSpecification. In the first case (user1), the Slot value is set to the regular expression, which was taken over from the type definition of MailAddress. It will later on be used to define expected results of the login operation. The semantics of such a concept is that as long as the actual response for a user’s mail address complies with the stated regular expression, the actual response matches the action and will not cause the test case to Fail.

The InstanceSpecification user1reduced omits all slots that are not required for a user object. This will later on be used for the modification of a user's mail address. In the last case (user1mod) a concrete and very precise mail address was stated for the very same user. This InstanceSpecification is used as part of the profile modification response (i.e., data type ProfileModRes) after an update of the mail address was requested. This is necessary, since it is important to see that the modification of was actually successful. All other data values are defined directly within the test cases as ordinary ValueSpecifications.

10.2.4.3 Test Requirements Realization

The actual design of test cases is the most important part of the test design phase. According to the applied methodology for the given example, test requirements are supposed to be realized by test cases, and thus, test case transitively verify or falsify the requirements that are covered by test requirements. The assignment of test requirements to test cases is part of the test design phase and results in our case in the following (partially shown) assignments (see diagram below).

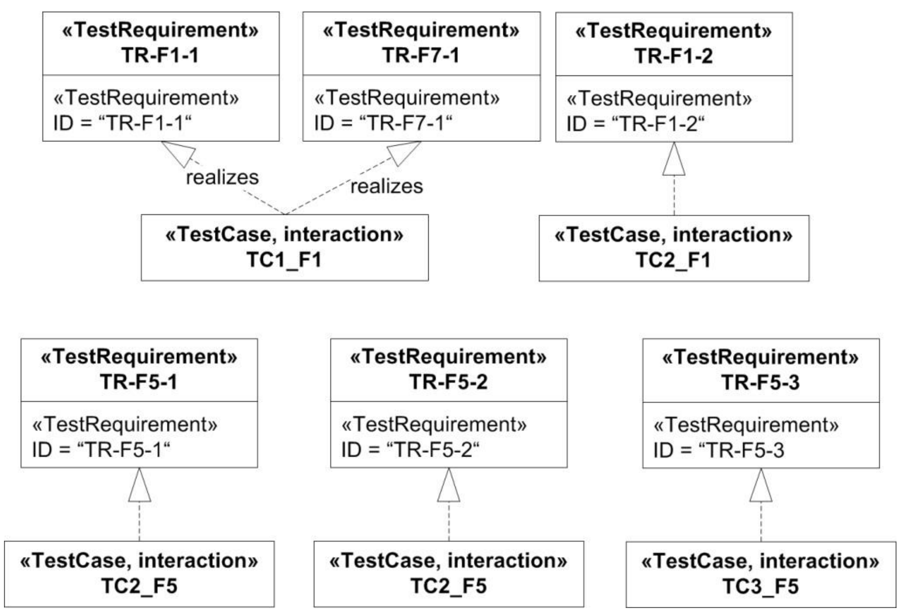


Figure 59 - Realization of Test Requirements

The respective test configuration for each test case is not shown in the diagram for the sake of comprehensibility, but is present nevertheless for each test case and identical to the test configuration shown above.

10.2.4.4 Design of Test Case Procedures

Based on both the specification of the test requirements what to test and the precise specification of the test configuration in order to realize how to test what has to be tested, the test case procedures can be derived. As already shown, in this example sequence diagrams (i.e., Interactions) are going to be used as a test procedure. The semantics of these test case interactions is that any deviation from the described interactions and message arguments will cause the test case to Fail. However, if the actual response matches the expected ones during test execution, the test case will Pass.

The two diagrams below show the test procedures of two test cases for the test requirements TR-F1-1, TR-F1-2 and TR-F7-1. This specification deliberately neglected the parameterization of test cases due to an unresolved issue filed against UML Interactions.

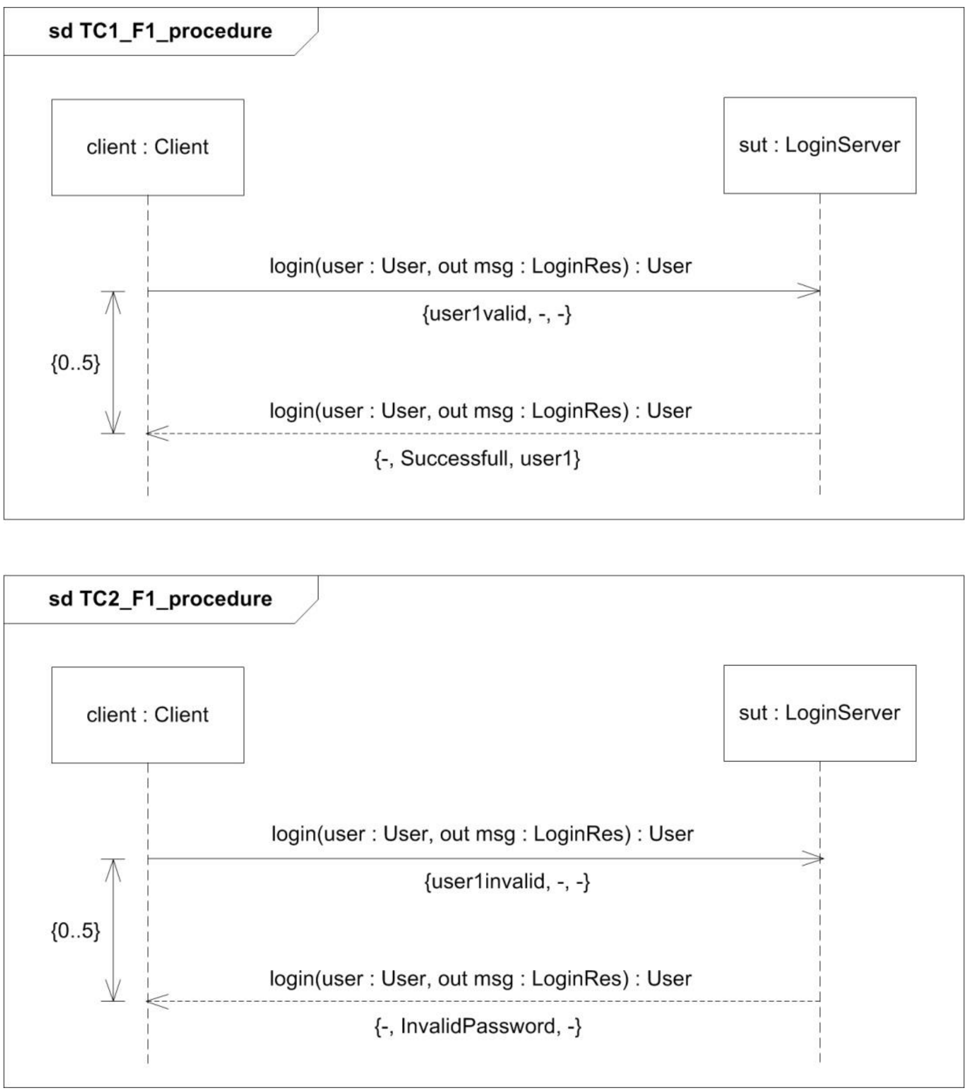


Figure 60 - Two Test Procedures

The DurationConstraints ensure that any response to the login request that is recognized later than 5 time units (in this case seconds) after the actual request will violate the DurationConstraint, and thus, cause the test case to Fail. The message arguments used in these test cases are represented by InstanceValues that have the same name as the InstanceSpecifications they refer to. Successful and InvalidPassword are EnumerationLiterals of the Enumeration LoginRes. The messages are sent via the Connector connector1 of the corresponding test configuration. This enables a precise definition of the Ports that should be used for sending stimuli and receiving expect response actions.

The diagram below depicts a test case for the successful modification of a logged in user's mail address. It reuses (actually reimplements for no explicit reuse - by means of InteractionUse of the test procedure of test case TC1_F1) the behavioral description for a successful user login request. The is usually called the preamble of the test case (although the current version of UTP 2 has no means to explicitly denote parts of the behavioral description as preamble or postamble; this is intended for revised submission).

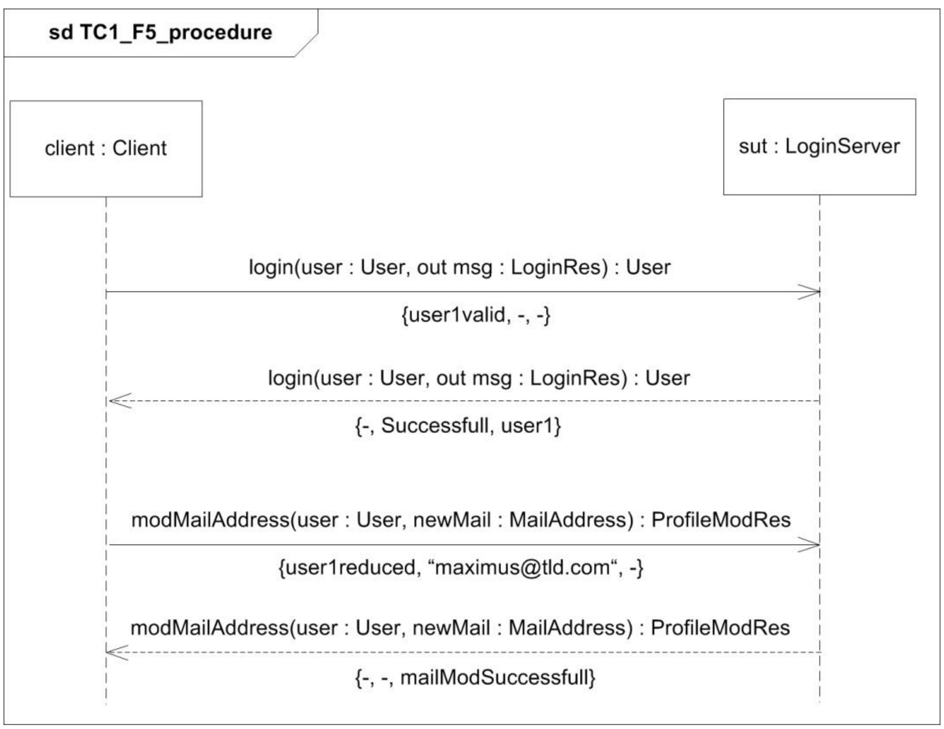


Figure 61 - Successful Test Case

The only technical deviation from the previously shown test cases is that the mailModAddress request message uses a LiteralString with value "maximus@tld.com" as message argument. Otherwise, no further peculiarities need to be discussed.

Note: The use of arguments of a message represented in curly brackets below the message arrow is not UML-compliant, but was chosen for the sake of clarity.

10.2.5 Mapping to TTCN-3

The Testing and Test Control Notation version 3 (TTCN-3) standardized by the European Telecommunications Standardization Institute (ETSI) prescribes a dedicated test language and test system framework that abide by the keyword-driven testing principle. Since its final adoption is has been heavily used within the telecommunications and automotive domain, but is in general, like UTP, independent of any domain. As TTCN-3 similarly to OMG standards is not restricted to certain methodology, the following described mapping represents just one possible way to translate UTP 2 test cases to TTCN-3. For example, it is restricted to Interactions for test case procedures, whereas in principle each of the UML behavior kinds could be used for specifying test procedures.

10.2.5.1 Mapping the Test Type System

TTCN-3 comes along with a fine-grained and powerful type system that resembles the one provided by UML, which was taken over by UTP. The following snippet shows the corresponding TTCN-3 code for the LoginServer test type system starting with primitive types, over enumerations to complex types.

```
type charstring MailAddress
        (pattern "\[a-zA-Z0-9._%+-\]+@\[a-zA-Z0-9.- \]+\.\
                   [a-zA-Z\]\{2,4\}");
type enumerated LoginRes
       {InvalidID, InvalidPassword, UnknownUser, UserBanned, Successfull};
type enumerated LogoutRes
       {Successfull, LogoutRequiresLogin};
type enumerated ModMessage
       {Successfull, InvalidFormat, LoginRequired};
type record LoginReq
{
    charstring id,
    charstring password
}
type record User
{
    charstring id,
    charstring name optional,
    charstring mail optional
}
type record ProfileModRes
{
    User user,
    ModMessage status
}
```

10.2.5.2 Mapping Interface Descriptions

In TTCN-3, interface operations are represented by so called signature types. A signature is a type that can be instantiated and resembles the invocation of an operation. The concept of an Interface as grouping namespace for Operations has no correspondent concept in TTCN-3. In case of ambiguous signature names (i.e., two Operation with the same name contained in different Interfaces) the qualified name of the Operation could be used as name of the signature since TTCN-3 does not offer type overloading. The mapping presented in this example utilizes the TTCN-3 group concept to logically cluster the signature types according to their containing UTP Interfaces; however, one has to be aware of the fact that a TTCN-3 group has no further semantics than to group elements. A TTCN-3 group is neither comparable to a UML Package nor any other Namespace for it does not have scoping semantics. The suggested mapping of the LoginServer interface descriptions is shown in the following snippet:

```
group ServerLoginInterface
{
   signature login (LoginReq request, out LoginRes msg) return User;
   signature logout (User user) return LogoutRes;
}
group UserProfileInterface
{
   signature modMailAddress (User user, MailAddress newMail) return ProfileModRes;
}
```

10.2.5.3 Mapping the Test Architecture

TTCN-3 relies on a component- and port-based architecture. That fits quite well with the offered concepts by UML, and thus, UTP. The following snippet demonstrates the mapping of the LoginServer test architecture to TTCN-3:

```
type port BasicPortConfiguration procedure
{
     inout login, logout, modMailInterface;
}
type component LoginSever
{
    port BasicPortConfiguration basicPort;
}
type component Client
{
    port BasicPortConfiguration basicPortConjugated;
}
```

10.2.5.4 Mapping the Test Data Specification

Data values utilized in message exchanges are called templates in TTCN-3. A template resembles an InstanceSpecification or dedicated ValueSpecification in UTP (actually UML). Templates can be either defined explicitly outside of a test case (called global templates), and thus, being reused by multiple test cases, or directly within in a message (called inline). At first this specification is going to show the mapping of global templates:

```
template LoginReq user1valid() :=
{
  id := "mustermann2014",
  password := "TustNo1"
};

template LoginReq user1invalid() :=
{
   id := "mustermann2014",
   password := "WhyNot"
};
template User user1() :=
{
   id := "mustermann2014",
   name := "Max Mustermann",
   mail :=  (pattern "\[a-zA-Z0-9._%+-\]+@\[a-zA-Z0-9.- \]+\.\
                     [a-zA-Z\]\{2,4\}")
};
template User user1reduced() :=
{
   id := "mustermann2014",
   name := omit,
   mail := omit
};
template User user1mod() :=
{
   id := "mustermann2014",
   name := "Max Mustermann",
   mail :=  "maximus@tld.com"
};
template ProfileModRes mailModSuccessfull() :=
{
  user := user1mod,
  status := Successful
};
```

10.2.5.5 Mapping Test Cases and Test Configuration

In TTCN-3 a test configuration is inherently bound to a test case, whereas in UTP a test configuration could be potentially shared across multiple test cases (even though this feature is not shown in the given example). The following snippet shows the mapping of the test case TC1_F1:

```
//determines the roles for Client and LoginSever
//runs on declares Client as TestComponent
//system declares LoginServer as TestItem
testcase TC1_F1() runs on Client system LoginServer
 {
       //establishes the Connector connector1
       map(self:basicPortConjugated, system:basicPort);

       //invokes the login operation by sending an instance of the
       //signature type login and starts an implicit timer with the
       //duration of 5 seconds
       basicPortConjugated.call(login:{user1valid,-}, 5000.0)
       {
           //continually checks whether the expected response is received
           //by the test system
           []basicPortConjugated.getreply(login:{-,Successfull}
                                                                   value user1)
           {
               //indicates that the test case has passed
               setverdict(pass);
           }
           //continually checks whether any other response is received
           []basicPortConjugated.getreply
           {
               //indicates that the test case has failed due to mismatch
               //between actual and expected response
               setverdict(fail)p;
           }
           //continually checks whether the implicit timer expired
           []basicPortConjugated.catch(timeout)
           {
               //indicates that the test case has failed due to timout
               setverdict(fail);
           }
       }
 }
```

10.3 Videoconferencing Example

This example is a small subset of the detailed models of a videoconferencing system (VCS) that was published in [1]. The original case study consists of modelling 20 subsystems of the VCS, where several of the subsystems work in parallel to each other. The key functionality of a VCS includes establishing a videoconference call with a set of other videoconferencing systems either directly or via specialized infrastructure comprising of specialized hardware such as gateways and call servers. However, in the original case study test related information wasn't explicitly specified on the models and all the details were embedded in test generators. This allows hiding test related details inside the test generators and models are independent of such information, but on the other hand makes the implementation of the test generator complicated. In addition, such an approach makes it difficult to maintain its implementation. The second approach that is demonstrated in this section is explicitly modelling the test related information using the UTP on the models. This approach provides more flexibility as compared to the first approach since test related information is explicitly specified in the models and test generators implementing the standard can understand and process the models.

This example illustrates some of the major concepts of UTP 2 on the software system of the VCS such as test item, test item configuration, and test component configuration. To demonstrate the application of UTP to this case study, this specification has tailored the original case study and used one small subset of the case study. The three key features of the VCS are modeled in this section, one focuses on the establishing the videoconference, the second one related to sending presentations in addition to the videoconference, and third one focuses on modeling behavior of VCS in the presence of packet loss.

The rest of this section is organized as follows. Section Given Requirements on the Test Item lists the key requirements that are focused for modelling in this section, Section Modeling the Structure of the System demonstrates how this specification models structure of the VCS using the UML class diagrams with UTP, Section Modeling the Behavior of the System demonstrates how this specification modeled the three key requirements as UML State Machines and UTP, Section The TRUST Test Generator shows our test generator that generates executable test cases from UML Class Diagrams and UML State Machines, and Section Mapping to Python Code shows a examples of mapping from the models to the Python script.

10.3.1 Given Requirements on the Test Item

In this section, this specification will demonstrate modelling the four key functionalities of a VCS that must be tested. These functionalities are listed in the table below:

Id	Type	Description

R-0001	functional	A VCS should be able to connect to maximum n number of VCSs at the same time.
R-0002	functional	A VCS should be able to start presentation even it is not in the videoconference. In this case, the presentation will be only shown to the VCS itself.
R-0003	functional	A VCS should be able to start presentation when it is in a videoconference. In this case, the presentation will be transmitted to all the connected VCSs (referred as end points).
R-0004	non-functional	A VCS should be able to handle packet loss. If the VCS cannot handle packet loss of greater than 2% for 2 minutes, it disconnects the current active call.

Table 13 - Videoconferencing Requirements

10.3.2 Modeling the Structure of the System

In this section, this specification models the structure of VCS that is modeled as a UML class diagram. A VCS can establish calls with 1 to * number of endpoints, i.e., other VCSs. The VCS is stereotyped as «TestItem» and «TestDesignInput» to label the system being tested, whereas other endpoints (i.e., Endpoint) is stereotyped as «TestComponent». The VCS has three attributes, NumberOfActiveCalls, MaximumNumberOfCalls, and PresentationMode representing the current number of endpoints in a videoconference, maximum number of calls supported by the VCS, and if the VCS is in presentation or not. The VCS class has four operations to support dialing to an endpoint (dial()), disconnecting a participant from a videoconference (callDisconnect()), starting presentation (presentationStart()), and stopping presentation (presentationStop()). In addition, this specification defines a constraint in OCL on VCS to model configuration for testing:

```
context VCS inv:
self.networkServices.h323.mode = VCS::Model::On
```

This constraint demonstrates that the VCS must be configured to support a videoconference with h323 conferencing protocol. The constraint is stereotyped as «TestItemConfiguration» to signify that the constraint is a configuration for VCS and is handled accordingly by test generator. In addition, «TestItem» has an attribute configuration {subsets roleConfiguration} , which is linked to this OCL constraint with «TTestItemConfiguration» (not shown in the figure). A similar constraint for Endpoint is also specified in the figure below and is stereotyped as «TestComponentConfiguration».

In addition, this specification defines an attribute in NetworkServices called packetLoss of type NFP_Percentage from the MARTE profile. The attribute is also stereotyped as

«Nfp» from MARTE to signify that this specification is capturing a non-functional property.

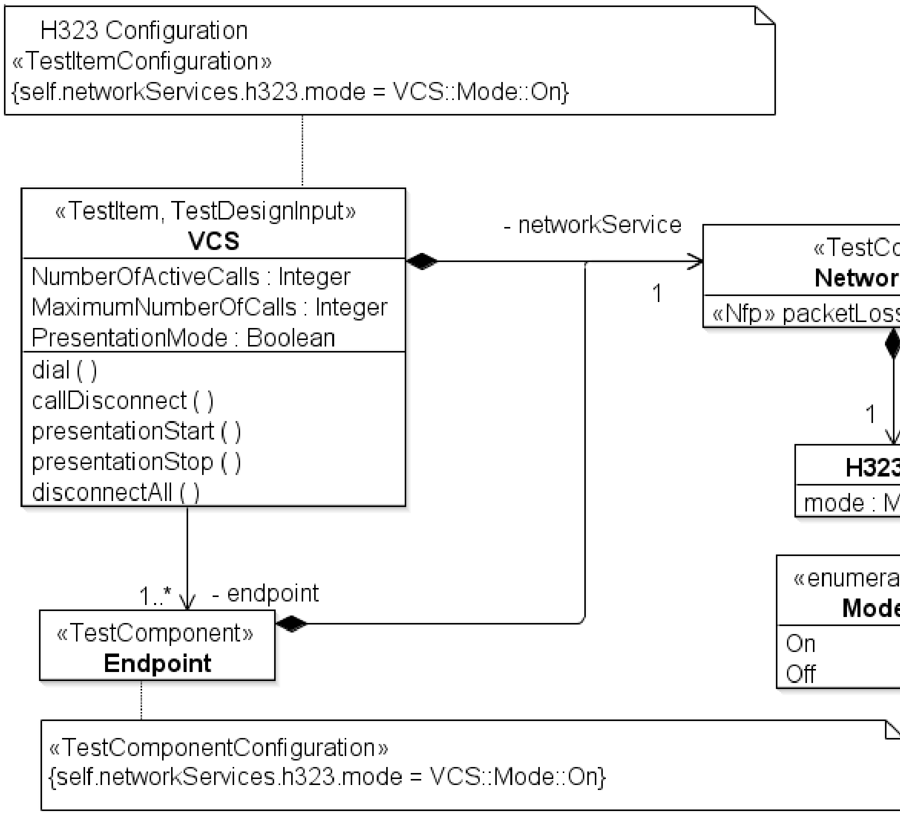


Figure 62 - UML Class Diagram

10.3.3 Modeling the Behavior of the System

The figure below shows the behavior of the VCS modeled as a UML state machine stereotyped as «TestDesignInput» to instruct test generator that the state machine should be used for generation of test cases. In our context it is important to stereotype a state machine that must be used for generation of test cases since not all the state machines are used for generation of test cases. The state machine has three regions: 1) The first region models first requirement for testing, i.e., establishing videoconference, 2) The second region models the second two requirements related to presenting while in a videoconference and presenting without a conference, and 3) The third region models the fourth requirement.

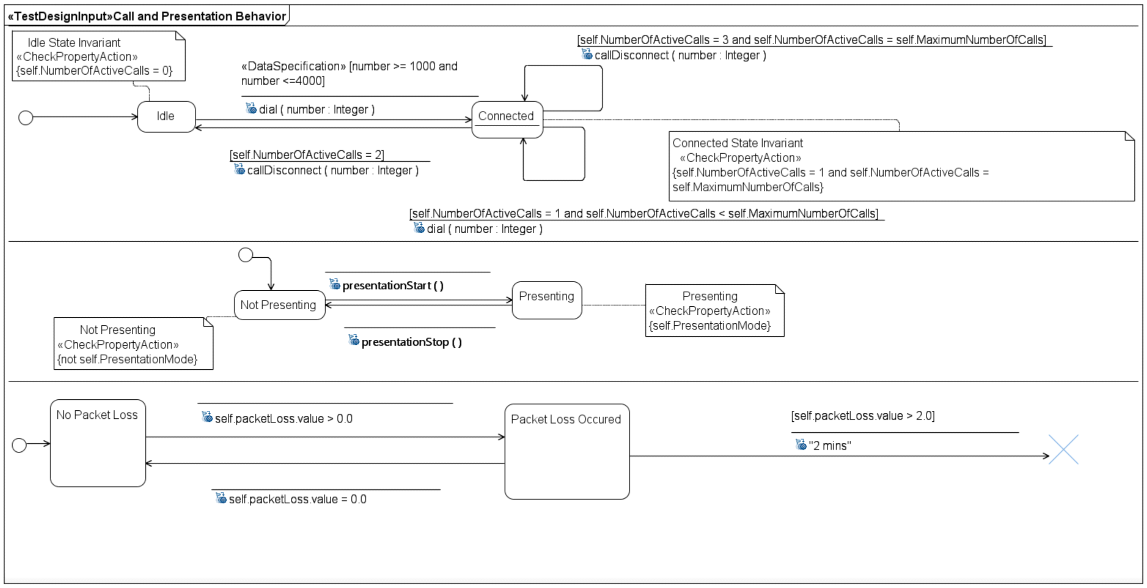


Figure 63 - UML State Machine Diagram

In the first region, this specification models the behavior of a VCS related to establishing a videoconference. The first region has two states, i.e., Idle and Connected demonstrating that the VCS is Idle state and the VCS is in a videoconference respectively. Each state has a state invariant defined as an OCL constraint based on the attributes defined in the VCS class diagram. For example, the Idle state has the following state invariant specified as an OCL constraint:

```
context VCS inv:
self.NumberOfActiveCalls = 0
```

The state invariant is stereotyped as «CheckPropertyAction» to instruct the test generator to use the constraint to generate code in Python that compares the actual state of VCS at the runtime (e.g., value of NumberOfActiveCalls in this example) with the one specified as CheckPropertyAction. If the state matches then it means everything is fine, however, if the state doesn't match it means there is a fault. The attributes of «CheckPropertyAction» are shown below in the figure. For example, the checkedProperty attribute is linked to the NumberOfActiveCalls in the VCS class (only shown as Entries:1). The value of expected is set to true meaning that the expected evaluation value of this constraint is true.

Property	Value
CheckPropertyAction	
checkedProperty	Entries: 1
expected	true
UTP::CheckPropertyAction::arbitrationSpecification	null

Figure 64 - Attribute values of «CheckPropertyAction»

Transitions in the state machine are modeled with Call Events corresponding to the operations defined in the VCS class. For example, from the Idle state, the transition with dial() trigger will lead to Connected if the call is established successfully. In addition, some of the transitions have guard conditions with the stereotype «DataSpecification». Recall that DataSpecification is "A named boolean expression composed of a data type and a set of constraints applicable to some data in order to determine whether or not its data items are conformant to this data specification" as defined in the conceptual model. A DataSpecification (e.g., guard condition in this example) signifies that the transition from the Idle state to the Connected state with a guard condition number>=100 and number <=4000, (i.e., an OCL constraint) can only be triggered by calling the dial(number:Integer) Call Event with a number between the range of values specified by the guard condition. In our context, this guard condition, i.e., an OCL constraint is used by the test generator to generate valid values within the range to trigger a transition, for example, the dial() operation in this case.

The second region of the state machine models the behavior of VCS related to starting the presentation in parallel to the videoconference. The region has two states (i.e., Not Presenting and Presenting) showing the states that the VCS is not presenting and presenting respectively. As with the first region, each state has a state invariant modeled as an OCL constraint. Similarly, the third region models the behavior of VCS in presence of packet loss.

10.3.4 The TRUST Test Generator

The figure below shows a very high level architecture of test case generator. The full details of the test generator can be found in [3]. At a high level, the test generator called as TRUST takes UML State Machines and UML Class Diagrams with stereotypes from UTP as input and generates executable test cases in Python based on various coverage criteria such as All State coverage and All Transition coverage (e.g., ts:TestStrategy with «StateTransitionTechnique») [3]. According to [ISTQB] StateTransitionTechnique is "A black box test design technique in which Test Cases are designed to execute valid and invalid state transitions". In addition, TRUST has a built in algorithm that flattens the state machines with hierarchy and concurrency before generating test cases. The details of the algorithm can also be found in [3]. TRUST also invokes a test data generation tool called EsOCL that takes input an OCL constraint (specified in class diagrams and state machines) and provides a set of data that satisfy the constraint based on a test data generation strategy (e.g., td:TestDataGenerationStrategy with the«BoundaryValueAnalysis» stereotype). According to [ISTQB], BoundaryValueAnalysis is "A black box test design technique in which Test Cases are designed based on boundary values". The details of EsOCL can be found in [4].

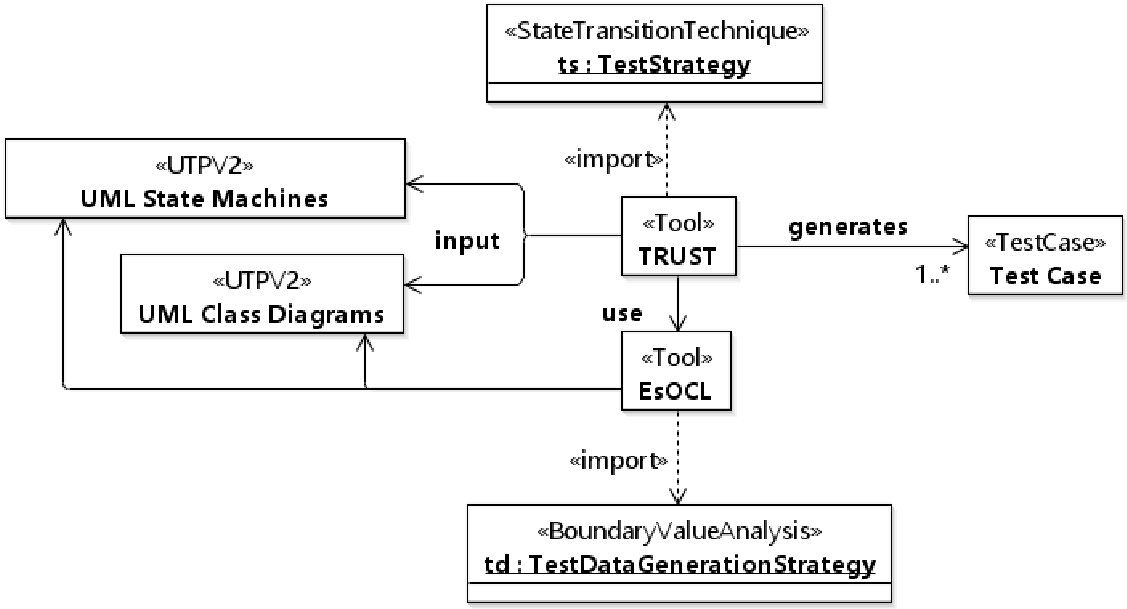


Figure 65 - Test Generator

The figure below shows a high level architecture of our Test Driver. The test driver takes input a test case in Python and executes it on the VCS that communicates with the n number of endpoints. The test driver also sends commands to configure endpoints based on test configurations specified in the test case. In our current example, the test driver executes only test cases on one VCS; however, in reality it can execute test cases on multiple VCSs in a videoconference. During the execution, test driver invokes an OCL Evaluator called DresdenOCL (www.dresden-ocl.org/) to evaluate OCL constraints that were stereotyped as «CheckPropertyAction» against the actual state of the VCS that eventually determines the success or failure of the execution of test cases.

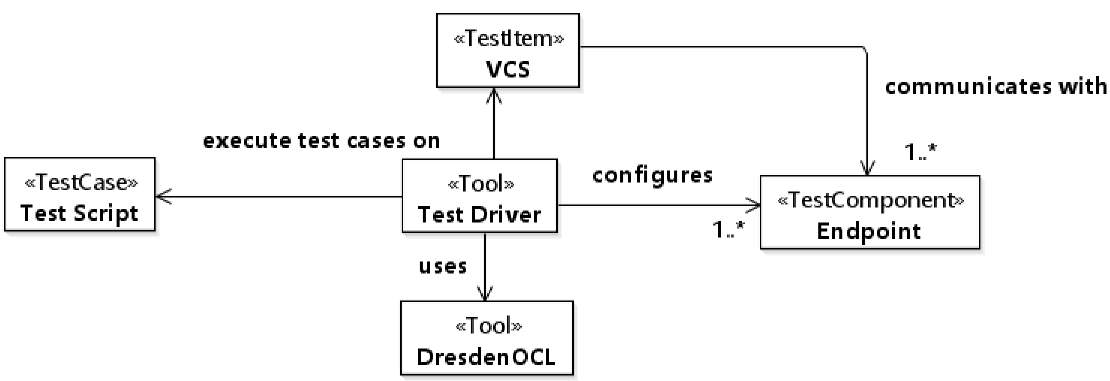


Figure 66 - Test Driver

10.3.5 Mapping to Python Code

Below, this specification shows a sample code corresponding to test item configuration and test component configuration. Line 1 and Line 2 reserves VCS (A) and Endpoint (B) for the execution of test cases, whereas Line 3 enables H323 mode on test item based on the constraints with stereotype in «TestItemConfiguration».

Line 1: self.A=tng.api.get_system('epa')

Line 2: self.B=tng.api.get_system('epb')

Line 3: self.A.cuil.config.NetworkServices.H323.Mode = "On"

Below, this specification shows the Python code corresponding to the start and stop presentation behavior and also the code that checks state of the VCS. Line 1 executes presentation start command on the VCS and Line 2 checks whether the VCS is in correct state by checking the value for the Presentation Mode attribute of the VCS, which should be equal to "On". Line 3 and Line 4 stop the presentation and check the state of the VCS respectively.

Line 1: Translate.TranslateA("Command.Presentation.Start()", self.A.cuil).translate

Line 2: self.assertFalse(self.A.cuil.status.Conference.Presentation.Model.text == "On")

Line 3: Translate.TranslateA("Command.Presentation.Stop()", self.A.cuil).translate

Line 4: self.assertFalse(self.A.cuil.status.Conference.Presentation.Model.text == "Off")

10.3.6 References

[1] Ali, Shaukat, Lionel Claude Briand, and Hadi Hemmati. "Modeling Robustness Behavior Using Aspect-Oriented Modeling to Support Robustness Testing of Industrial Systems." Software and Systems Modeling 11 (2012): 633-670.

[2] Ali, Shaukat, Lionel Claude Briand, Andrea Arcuri, and Suneth Walawege. An Industrial Application of Robustness Testing Using Aspect-Oriented Modeling, UML/MARTE, and Search Algorithms In ACM/IEEE 14th International Conference on Model Driven Engineering Languages and Systems (Models 2011), Edited by Jon Whittle, Tony Clark and Thomas Kühne. .: ACM/IEEE, 2011.

[3] Ali, Shaukat, Hadi Hemmati, Nina Elisabeth Holt, Erik Arisholm, and Lionel Briand. Model Transformations As a Strategy to Automate Model-Based Testing - a Tool and Industrial Case Studies. Simula Research Laboratory, 2010.

[4] Ali, Shaukat, Muhammad Zohaib Iqbal, Andrea Arcuri, and Lionel Claude Briand. "Generating Test Data From OCL Constraints With Search Techniques." IEEE Transactions on Software Engineering 39 (2013).

10.4 Subsea Production System Example

10.4.1 Description of Case Study

A subsea production system is a cyber-physical system that produces oil and gas from subsea. Typically such subsea production systems are highly configurable in the sense that their hardware topologies and software parameters can be configured based on requirements customer such as the size of a subsea field and its natural environment (e.g., depth of sea). A subsea production system is composed of two sets of systems: topside and subsea systems. Umbilical connections (e.g., cables or hoses which supply air, power, electrical power, fiber optics to subsea equipment) are established to connect topside and subsea. Commands (e.g., opening valves) are sent by operators via topside systems to subsea systems, which control different kinds of subsea actuators (e.g., choke and valve) and monitor various sensors (e.g., pressure and temperature).

Please note that the case study is designed to demonstrate that the UTP 2 stereotypes can be used for developing domain specific language based MBT methodologies such as RTCM [3].

10.4.2 Functionality to Test

To demonstrate the application of UTP 2 to this case study, this specification specifies one of the key functionalities of Subsea Electronic Module (SEM), which has configurable software deployed to control subsea instruments. This functionality OpenValve is specified using the Restricted Use Case Modeling methodology (RUCM) [1][2] and the RUCM Editor, as shown in the figure below. Notice that the use case model (i.e., UCModel) is indicated as a TestRequirement using <<TestRequirement>>, which is a UTP 2 stereotype.

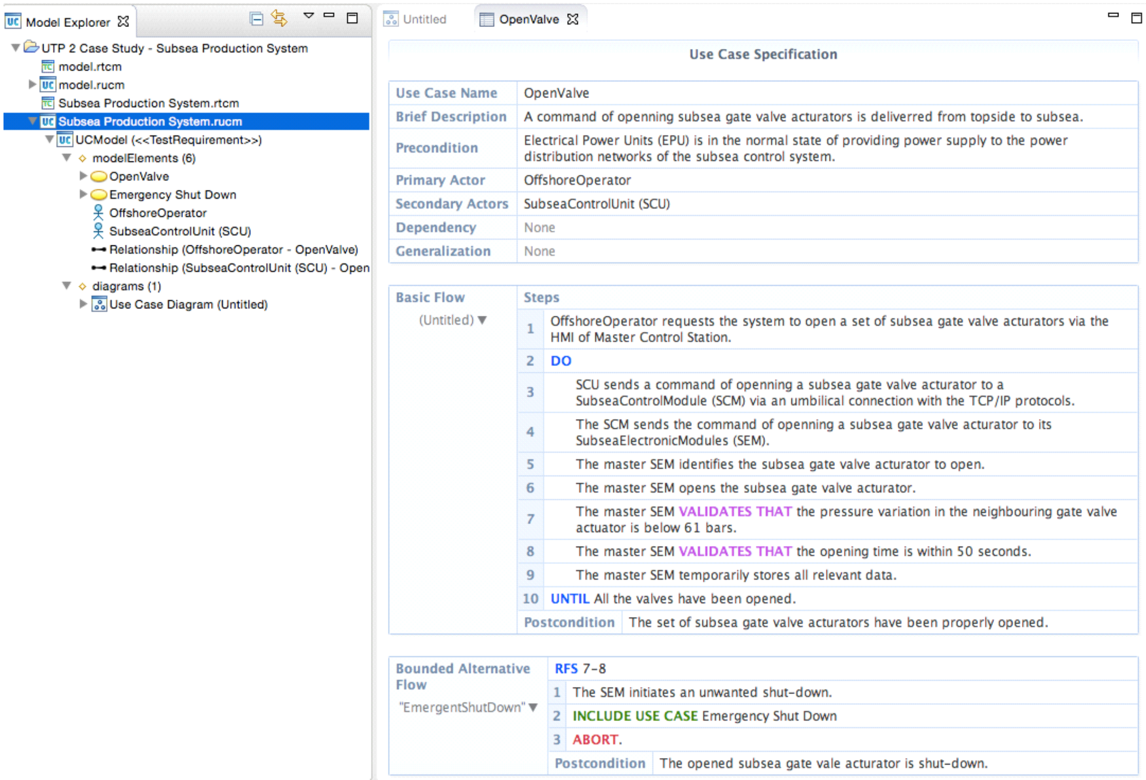


Figure 67 - Use Case OpenValve (Specified in RUCM)

10.4.3 Test Design Inputs

To test the *OpenValve* functionality presented in the figure above, this specification defines four test design inputs, as shown in the figure below. Notice that this specification aims to test the functionality of *OpenValve* of SEM using a simulator that is particularly designed for testing SEM.

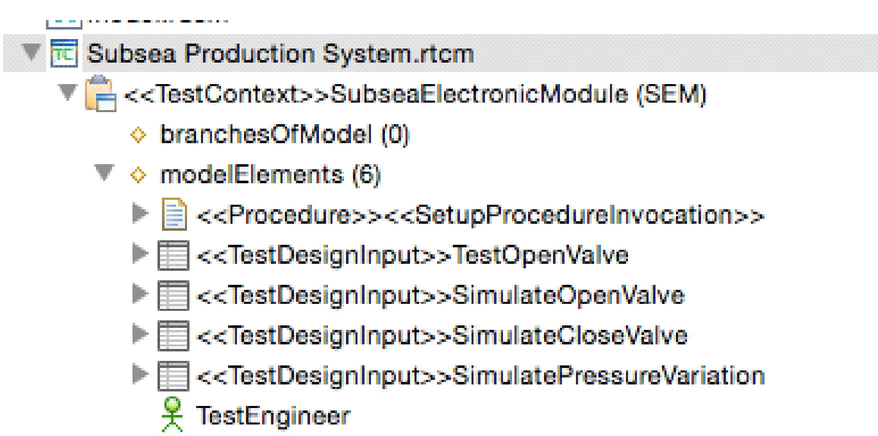


Figure 68 - The Four TestDesignInput and one procedure

The test objective of the test context SubseaElectronicModule (SEM) is defined as the description of the test context: "<<TestObjective>> The goal of these tests is for system testing of the functionalities of <<TestItem>> SEM."

In the figure below, this specification presents the test design input of *TestOpenValve*, which is specified/modeled using the Restricted Test Case Specification methodology (RTCM) [3]. Notice that the test case specification is annotated with UTP 2 stereotypes using stereotype notations. For example, steps 3, 4 and 10 of the basic flow (i.e., <<Sequence>>Pass) are annotated as <<ExpectResponseAction>>. Step 1 is annotated with <<CreateStimulusAction>> and steps 2, 6, 8 and 9 are annotated with <<ProcedureInvocation>> as these four steps invoke other test case specifications with keywords INCLUDE TC SPEC. Steps with keyword VERIFIES THAT are annotated with either <<ExpectResponseAction>> or <<CheckPropertyAction>>. TestSetup is annotated with <<TestConfiguration>> and can be reused across test case specifications.

Test Case Specification

Name	<<TestDesignInput>>TestOpenValve
Brief Description	This test case specification tests <<TestRequirement>><<UseCase>>OpenValve.
Precondition (Test Data Specification)	None
Tester	None
Dependency	INCLUDE TC SPEC <<TestDesignInput>>SimulateOpenValve, INCLUDE TC SPEC <<TestDesignInput>>SimulateCloseValve, INCLUDE TC SPEC <<TestDesignInput>>SimulatePressureVariation

Test Setup ▼	Name	<<TestConfiguration>>Test Setup
	Description	TestEngineer makes sure that <<TestItem>>SEM and <<TestComponent>>Simulator are connected and powerred on.

Basic Flow (Test Setup) "<<Sequence>>setup" ▼	Steps	
	1	<<CreateStimulusAction>>TestEngineer turns on the power of SEM.
	2	TestEngineer turns on the power of Simulator.
	3	<<CreateLogEntryAction>>TestEngineer records the initial state of SEM.
	4	<<CreateLogEntryAction>>TestEngineer records the initial state of Simulator.
	Postcondition (Test Oracle)	SEM is turned on. Simulator is turned on.

Basic Flow (Test Sequence) "<<Sequence>>pass" ▼	Steps	
	1	<<CreateStimulusAction>>TestEngineer uses Simulator to simulate the OpenValve command.
	2	<<ProcedureInvocation>>INCLUDE TC SPEC <<TestDesignInput>>SimulateOpenValve.
	3	<<ExpectResponseAction>>TestEngineer VERIFIES THAT the OpenValve signal received by Simulator is correct.
	4	<<CheckPropertyAction>>TestEngineer VERIFIES THAT the openning time is within the allowed threshold.
	5	<<CreateStimulusAction>>TestEngineer uses Simulator to simulate the CloseValve command.
	6	<<ProcedureInvocation>>INCLUDE TC SPEC <<TestDesignInput>>SimulateCloseValve
	7	<<Parallel>><<CreateStimulusAction>>TestEngineer uses Simulator to simulate the OpenValve command MEANWHILE <<CreateStimulusAction>>TestEngineer uses Simulator to simulate the pressure variation in the neighbouring gate valve actuator is above 61 bars.
	8	<<ProcedureInvocation>>INCLUDE TC SPEC <<TestDesignInput>>SimulateOpenValve
	9	<<ProcedureInvocation>>INCLUDE TC SPEC <<TestDesignInput>>SimulatePressureVariation
	10	<<CheckPropertyAction>>TestEngineer VERIFIES THAT SEM initiates emergent shut-down.
	Postcondition (Test Oracle)	<<TestSetArbitrationSpecification>>The test is passed.

Bounded Alt. Flow (Test Sequence) "<<Alternative>>fail" ▼	RFS <<Sequence>>pass 3-4,10	
	1	<<CreateLogEntryAction>>TestEngineer reports a failure.
	2	ABORT
	Postcondition (Test Oracle)	<<TestSetArbitrationSpecification>>The test is failed.

Figure 69 - test design input *TestOpenValve*

10.4.4 Generation of Test Sets and Abstract Test Cases

By taking the test design inputs as the input, the test generator of RTCM [3] automatically generates abstract test cases, as shown in the figure below. Based on different coverage criteria, from the test design input of *TestOpenValve*, the generator can generate three test sets, which contain various numbers of abstract test cases.

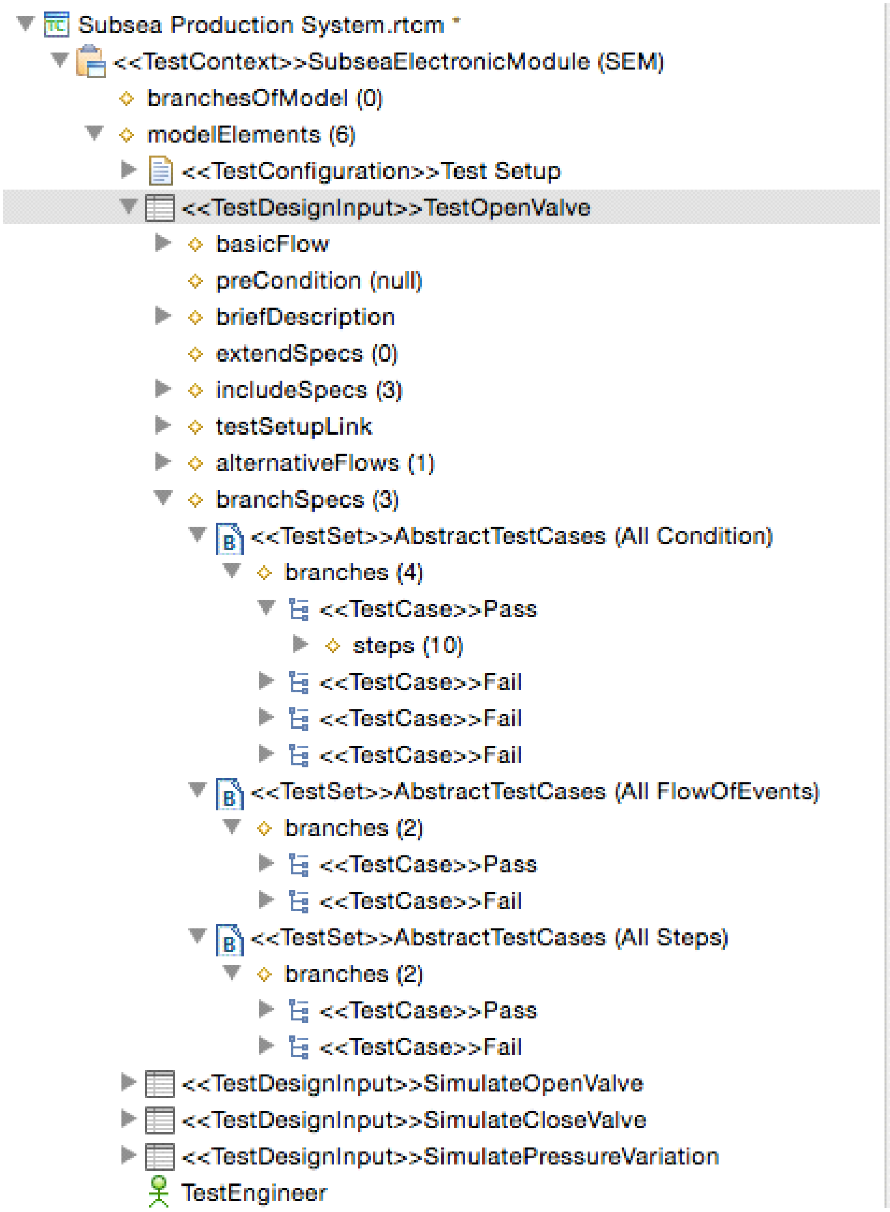


Figure 70 - Generated test sets

The automated generation is possible due to the fact that use case specifications in RUCM and test case specifications in RTCM can all be formalized as instances of the UCMeta [2] and TCMeta [3][4] metamodels respectively. Paths can then be automatically generated from formalized specifications/paths by following various coverage strategies (e.g., *All Sentence Coverage* and *All FlowOfEvents Coverage*).

One example of the abstract test cases generated from the test design input of *TestOpenValve* is provided in the figure below for reference. The step marked with the red color means the step failed. The step marked with the Green color means the step passes.

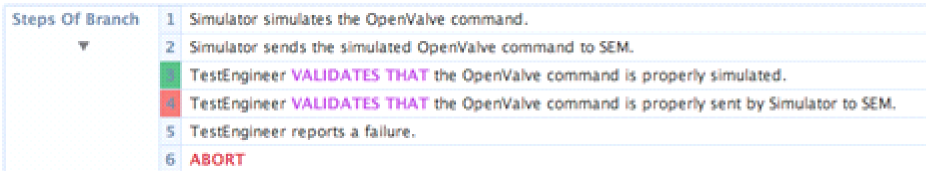


Figure 71 - An Example of a generated abstract test case

10.4.5 References

[1] Tao Yue, Lionel Briand, and Yvan Labiche, "Facilitating the Transition from Use Case Models to Analysis Models: Approach and Experiments", in Transactions on Software Engineering and Methodology (TOSEM), Volume 22, Issue 1, 2013.

[2] Tao Yue, Lionel Briand, and Yvan Labiche. "Toucan: an Automated Framework to Derive UML Analysis Models From Use Case Models.", in ACM Transactions on Software Engineering and Methodology (TOSEM) 24, no. 3 (2015).

[3] Tao Yue, Shaukat Ali, and Man Zhang. Applying A Restricted Natural Language Based Test Case Generation Approach in An Industrial Context, in International Symposium on Software Testing and Analysis (ISSTA)., 2015.

[4] Man Zhang, Tao Yue, Shaukat Ali, Huihui Zhang and Ji Wu. "A Systematic Approach to Automatically Derive Test Cases From Use Cases Specified in Restricted Natural Lan-guages", 8th System Analysis and Modelling Conference (SAM), 2014

10.5 ATM Example

10.5.1 General

This annex contains the Banking example introduced in the earlier version of UTP [UTP1.2]. The following model has been updated for the current UTP standard. It shows how to utilize UTP, version 2, to specify test models for unit level tests, component level tests and system tests.

The given example is motivated by an interbank exchange scenario in which a customer with an EU Bank account wishes to deposit money into that account from an Automated Teller Machine (ATM) in the United States. The figure below provides an overview of the architecture of the system. The ATM used by this customer interconnects to the EU Bank, through the SWIFT Network[1], which plays the role of a gateway between the logical networks of the US Bank and the EU Bank.

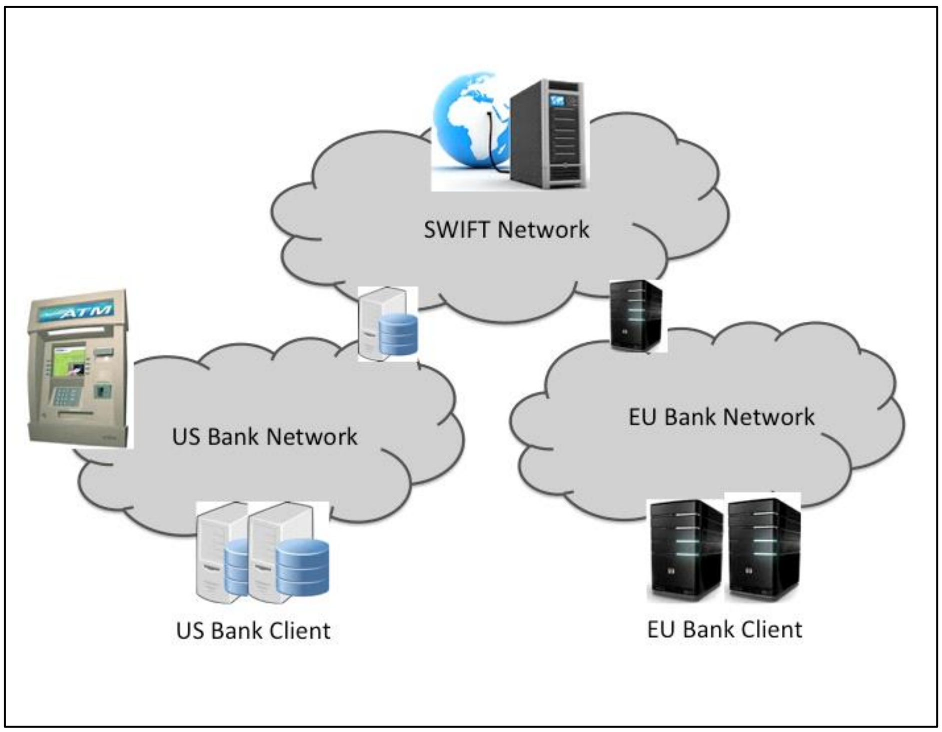


Figure 72 - Overview on the InterBank Exchange Network (IBEN)

The figure below shows the UML system model[2] of the InterBank Exchange Network. In the model, five UML packages called *ATM*, *Bank*, *SWIFTNetwork*, *HWControl* and *Mon-*

[1] SWIFT = Society for Worldwide Interbank Financial Telecommunication
[2] The diagrams of this example are modelled in Papyrus.

ey are provided. The dashed arrows between the packages show their import dependencies.

The following sub-sections demonstrate the use of UTP 2 for:

- unit test modeling on Money classes (Subsection 2),
- integration test modeling of the components *ATM*, *HWControl* and *Bank* (Subsection 3), and
- system test modeling of IBEN system (Subsection 4).

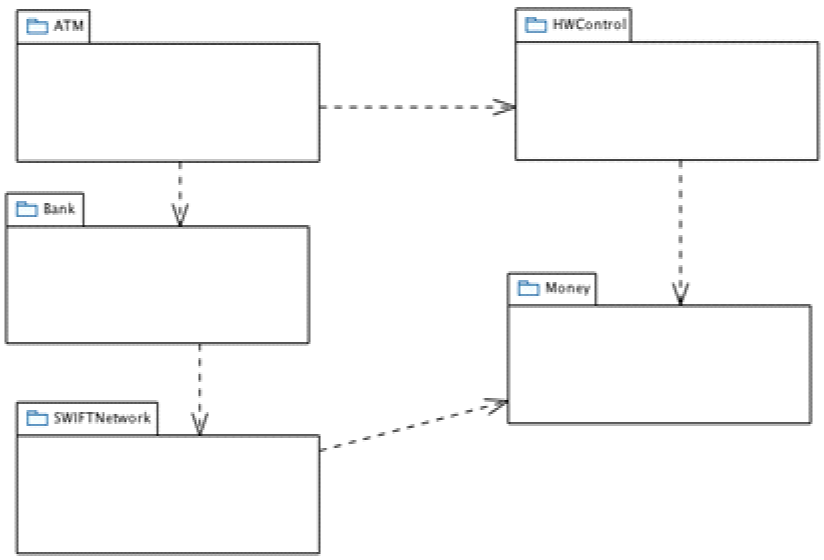


Figure 73 - Packages of the InterBank Exchange Network (IBEN) System Model

10.5.2 Unit Test Example

This sub-section illustrates the use of UTP version 2 in order to define unit test level test cases. It reuses and extends the *Money* and *MoneyBag* classes provided as examples of the well-known JUnit test framework ([JUnit_web], [JUnit_Example]).

Before starting modeling tests, the test item is first explained. The figure below shows the package *Money* (blue color) which will be tested.

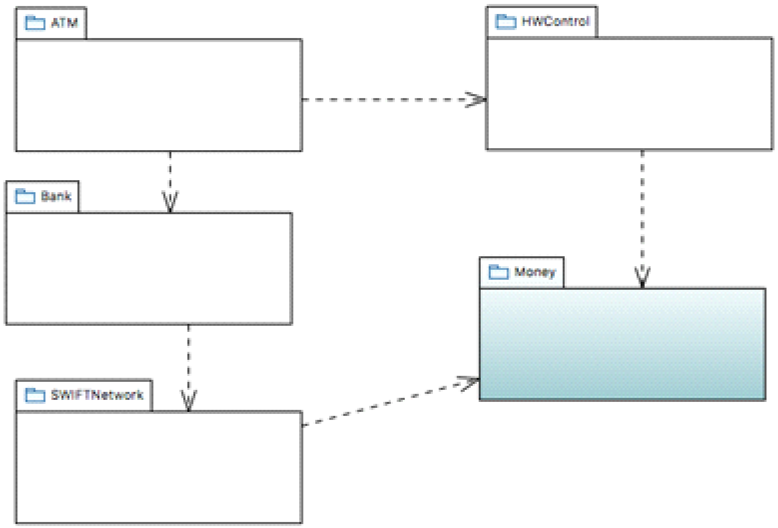


Figure 74 - Package *Money* with Test Items for Unit Test of IBEN

The figure below shows the classes defined in the package *Money*[3]. It shows an interface class called *IMoney*, which is realized by the class *Money*, and class *MoneyBag*.

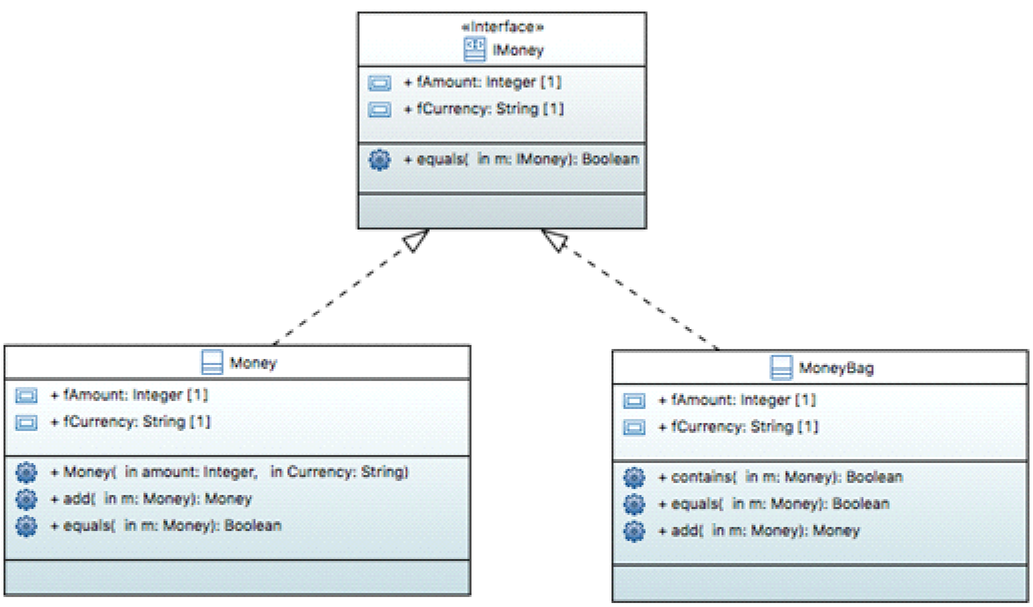


Figure 75 - Classes in Package *Money* in IBEN Modell

[3] Even though the naming of the package Money and of the class Money may lead to misunderstanding, the definition of the example provided by www.junit.org. is still used

The ATM uses these classes in order to count the bills entered by a user when making a deposit in cash. Two test requirements are defined:

- Verify that the Money class is appropriately counting the bills added by the user, when bills from the **same** currency are entered;
- Verify that the Money and MoneyBag classes are appropriately recognizing the bills added by the user when bills from **different** currencies are entered.

The figure below shows the test configuration between the test component named *unit-TestComponent* and the test items called *myMoney1* and *myMoney2* of class *Money* and *myMoneyBag* of class *MoneyBag*. The test configuration is modeled as UML Collaboration in order to be able to apply as CollaborationUse to the test cases.

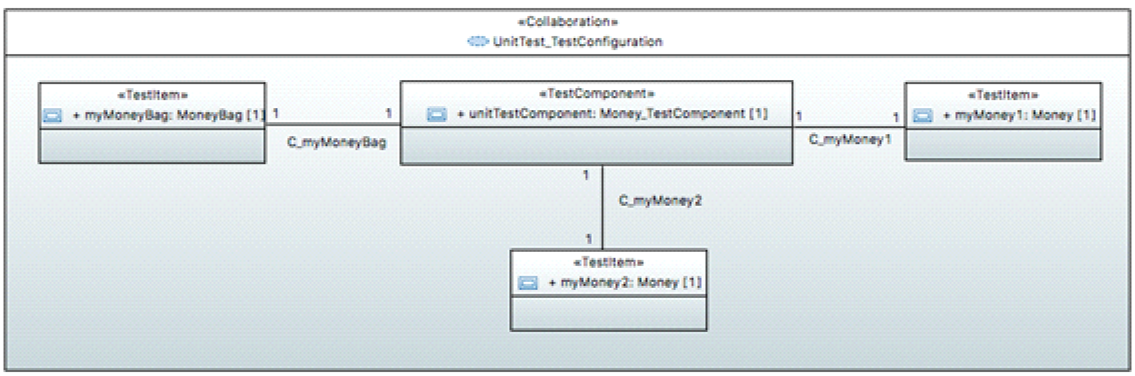


Figure 76 - Unit Test Configuration

The figure below shows the application of the unit test configuration to the test case *addSameMoney_TC*. By using the UML CollaborationUse the binding between the test configuration and the test case is guaranteed.

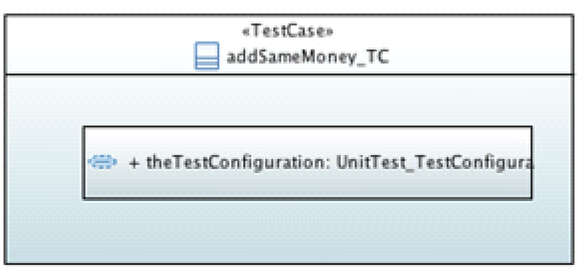


Figure 77 - Use of Test Configuration for Test Case *AddSameMoney_TC*

The figure below shows the test context of the unit test *UnitTest_Banking_Example*. Class *Money* is the item to be tested. It is defined in package *Money* which is imported from the system model. The package must be imported in order to get access during test execution. The test requirements *approveAddSameMoney* and *approveAddDifferentMoney* should approve that the addition of two money objects returns an object of class *Money* with the correct amount and currency. In the former requirement, money of the same currency will be added. In the latter, money of different currencies are to be added. The test cases called *addSameMoney* and *addDifferentMoney* verify the test test requirements.

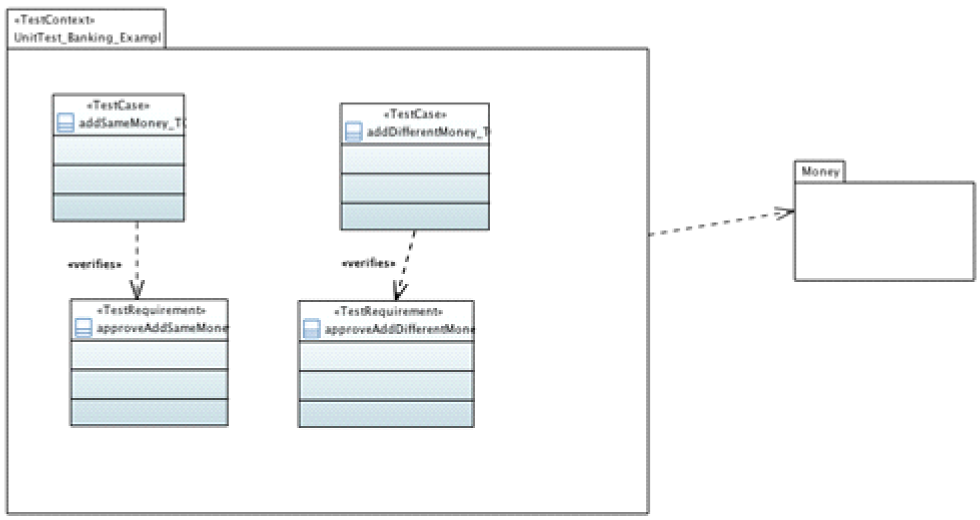


Figure 78 - Test Context for the Unit Test

The figure below specifies the behavior of the test case called *addSameMoney* verifying the test requirement *approveAddSameMoney*. In this test scenario, two objects of class *Money* are created, namely *myMoney1* with 20 USD and *myMoney2* with 50 USD. Afterward, *myMoney2* is added to *myMoney1*. The result is sent back to the test component for approval.

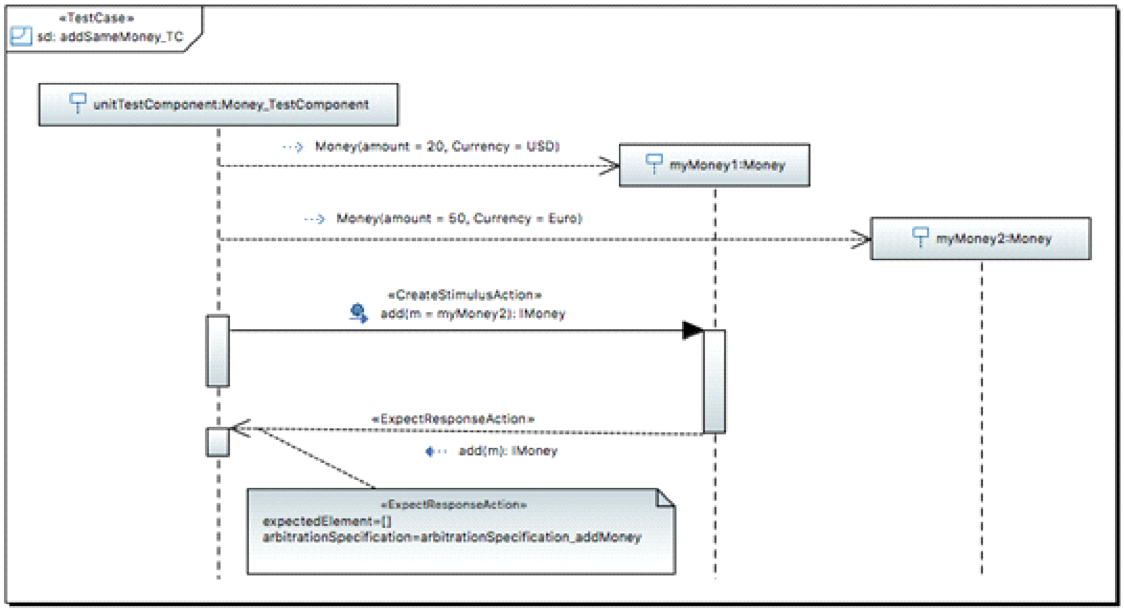


Figure 79 - Test case *addSameMoney_TC*

The correctness of the response is checked in either the default arbitration specification[4], or as in this case, by the user-defined arbitration specification called *arbitrationSpecification_addMoney*. Finally, the figure shows that in case the result of add() is 70 USD, the arbitration specification sets the test verdict to Pass, otherwise to Fail.

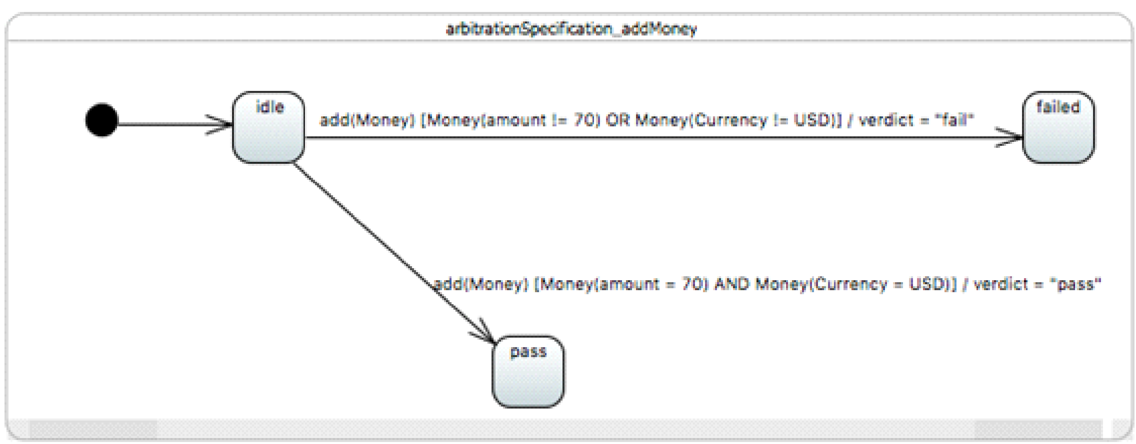


Figure 80 - User-Defined Arbitration Specification

The second test requirement *approveAddDifferentMoney* is verified by test case *addDifferentMoney* (see figure below). For this test case, a third test item of class *MoneyBag* is needed in order to be able to distinguish money of different currencies. This test case uses

[4] The default arbitration is provided by the tool vendor.

the default arbitration specifications that should be provided by the tool vendor.

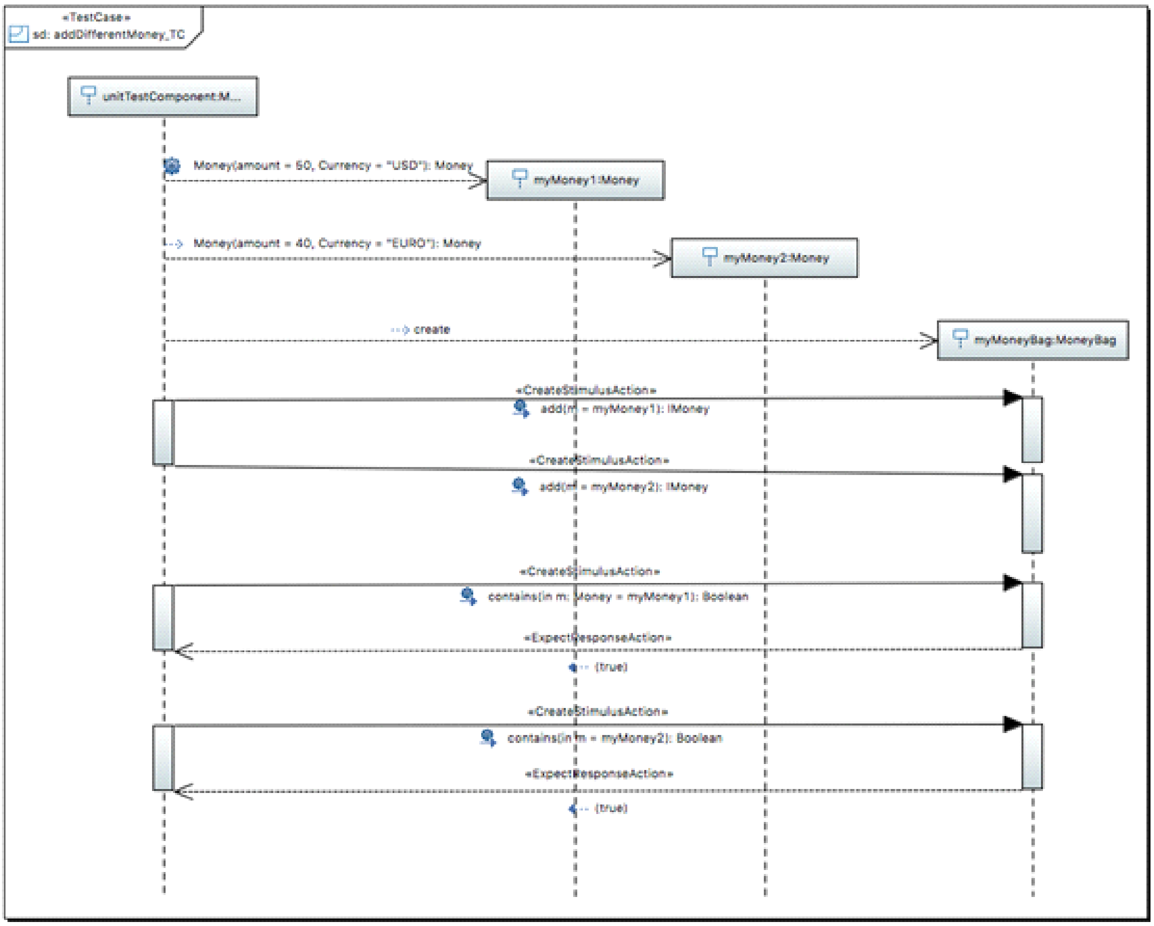


Figure 81 - Test Case *AddDifferentMoney*

10.5.3 Integration Testing Example

This section illustrates how UTP 2 can be used for specifying tests at integration test level. The main focus of integration testing is the communication of the test item and its test components.

The test requirements are to verify the logic of the ATM machine when a user initiates a money deposit transaction to an account in another part of the world. Thus, the test requirements include:

- The hardware terminal (*HWControl*) provides user's card and user's pin-code. The ATM shall authorize this card and its pin-code.
- After a successful authorization of user's data, money shall be deposited into the bank. The ATM shall assure a correct transaction communication with the *Bank*.

Since the logic of ATM itself is being tested, the rest of the IBEN (i.e. *HWControl*, *Bank*, and *SWIFTNetwork*) shall be emulated. The figure below shows the test items of blue color.

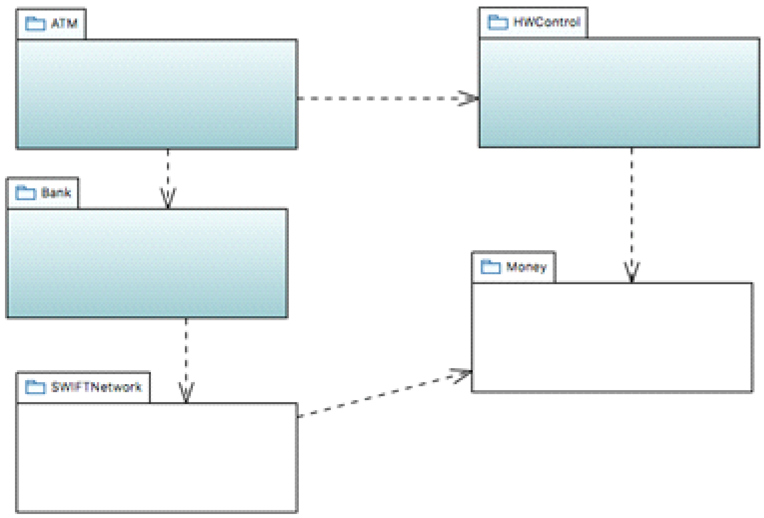


Figure 82 - Test Items for Integration Test of IBEN

The logic of the *ATM* is specified in the figure below. It imports both the *HWControl* and the *Bank* packages where only the interfaces to the hardware and the bank are needed. Component *ATM* controls the logic of ATM and is the test item for our integration test. It provides the *IATM* interface for the control logic and communicates with the hardware and the bank via interface. Since the hardware and the bank are emulated in the test, only the interface classes of the *HWControl* and *Bank* packages are needed (see the following three figures).

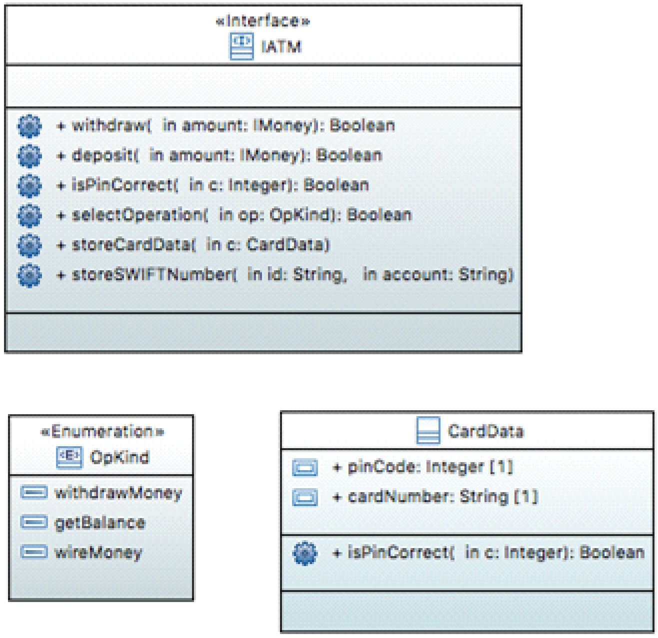


Figure 83 - Classes and Interface in Package *ATM*

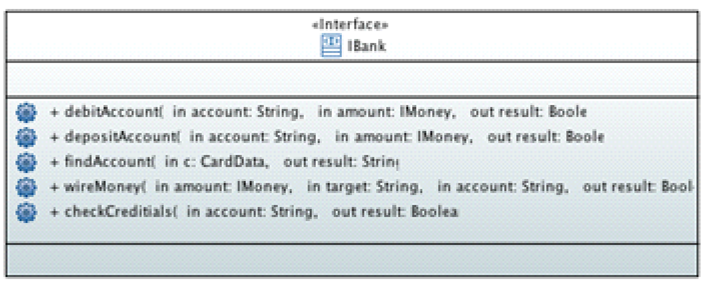


Figure 84 - Interface Class in Package *Bank*

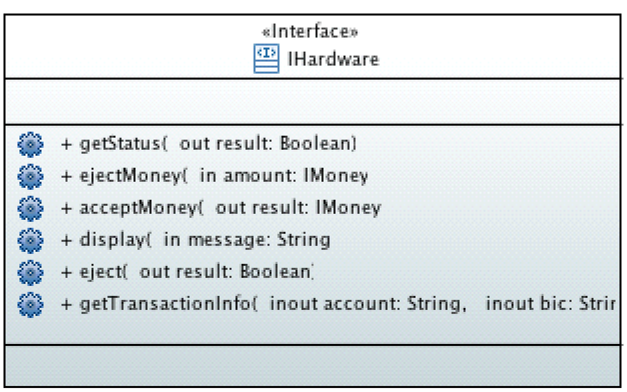


Figure 85 - Interface Class in Package *HWControl*

The figure below shows the test configuration of the test. It specifies the relationship between the test item, the emulated test components for the hardware and bank (*hw* and *be*), and a card data management component (*card*).

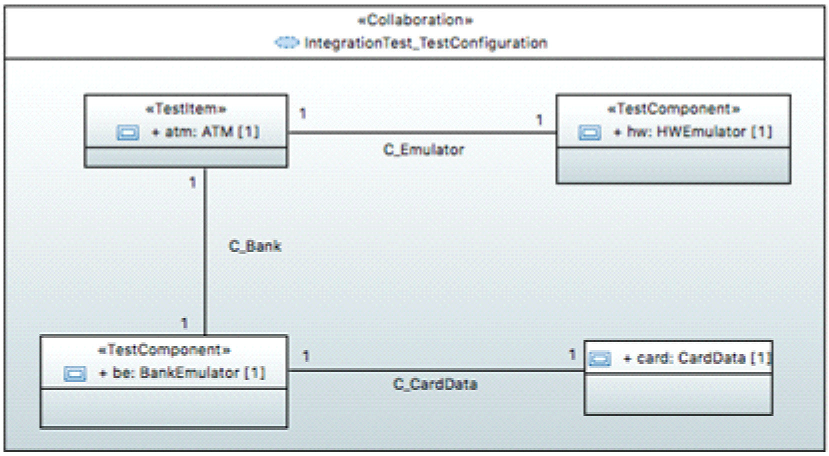


Figure 86 - Integration Test Configuration

The figure below shows the binding of the test configuration to test case *invalidPIN_TCI*.

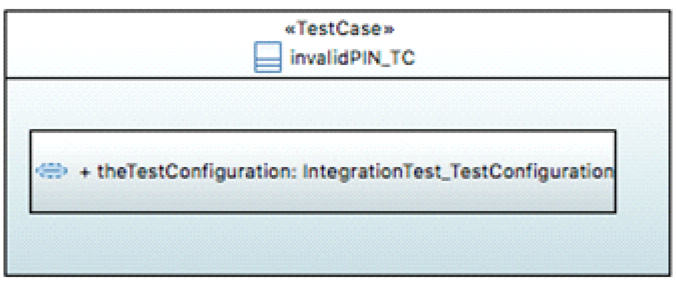


Figure 87 - Binding of Test Configuration to Test Case *invalidPIN_TC*

The ATM integration test package (see figure below) shows the model elements necessary to specify integration tests. It imports the ATM package of the system model in order to get access to the elements to be tested. The package contains two test components: *BankEmulator* and *HWEmulator* and three testcases: *validWiring*, *invalidPIN*, and *authorizeCard*. The test components *BankEmulator* and *HWEmulator* realize the interfaces of the *HWControl* and *Bank* packages and serve as emulators in order to communicate with the *ATM*.

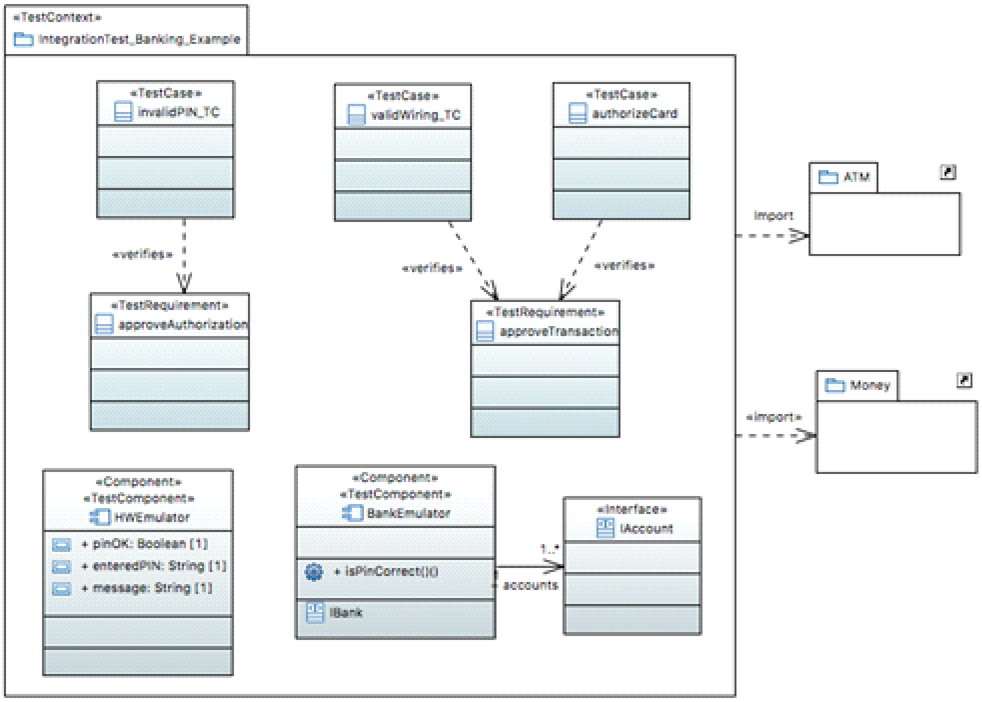


Figure 88 - Test Context for Integration Test

The following section only concentrates on the modeling of the test case *invalidPIN*, which approves the requirement of a correct authorization mentioned on earlier. The objective of this test is:

- Verify that if a valid card is inserted, and an invalid pin-code is entered, the user is prompted to re-enter the pin-code.

Behaviors of a test case can be specified using any UML behavior Diagrams (e.g. Interaction Diagram, State Machine, Sequence Diagram etc.). In this case, UML Sequence Diagram has been chosen (see figure below).

The signals between the test components are all stereotyped by UTP 2 actions (e.g. <<CreateStimulus-Action>>). By doing so, the default arbitration specifications are activated and it is assured that unexpected behavior is caught within the arbitration specifications. In parallel, the setting of test case verdicts is also done in the arbitration specifications. The response time of *isPinCorrect* should last no more than 3 seconds, otherwise the arbitration specification <<ExpectResponseAction>> will be carried out.

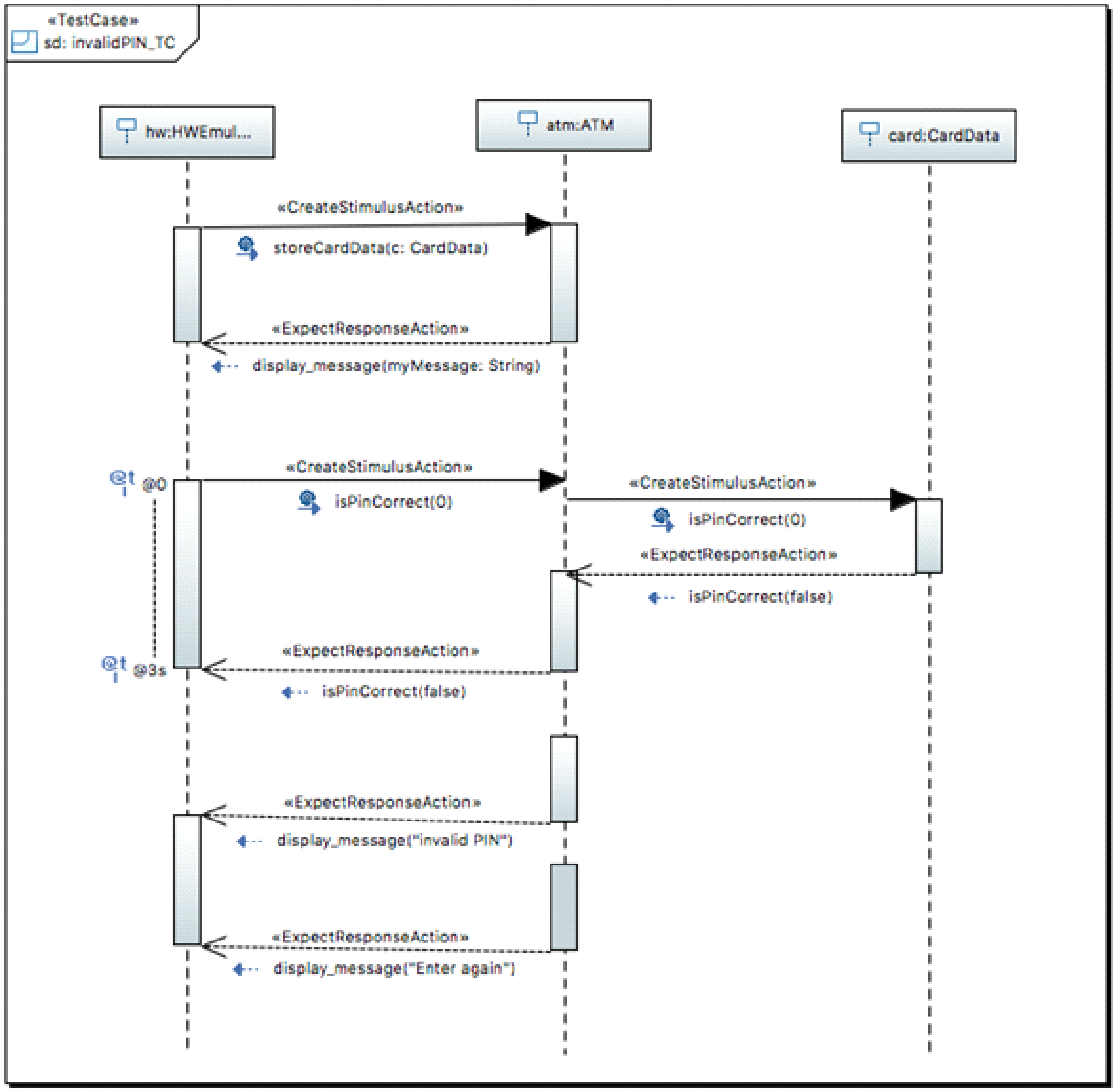


Figure 89 - Test Case *invalidPIN_TC*

In many cases, there's a need to specify the detailed behavior of individual test components (e.g., for test generation purposes). Therefore, state machines provide good means. The figure below shows an excerpt of test behavior for the *HWEmulator* test component which corresponds to test case *invalidPIN_TC*. The validation action <<ExpectResponseAction>> evaluates the test result and sets the test case verdict.

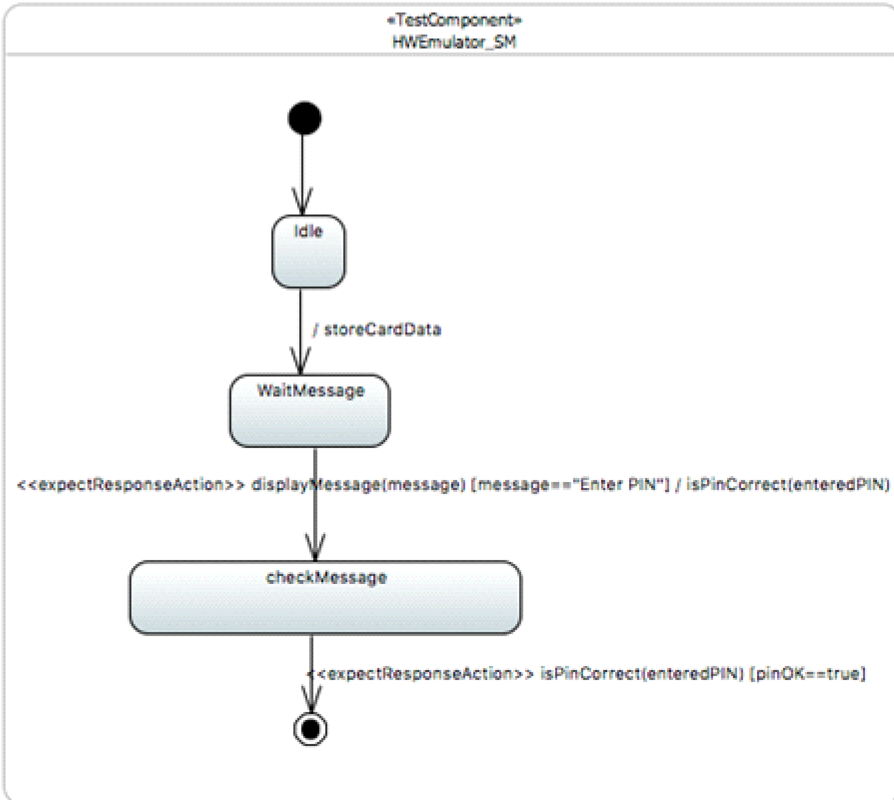


Figure 90 - Statemachine for the Hardware Emulator

10.5.4 System Test Example

This chapter shows the UTP2 model for system level tests. The test model shows an interbank exchange scenario where a customer with an EU bank account deposits money into his/her account from an ATM in the United States.

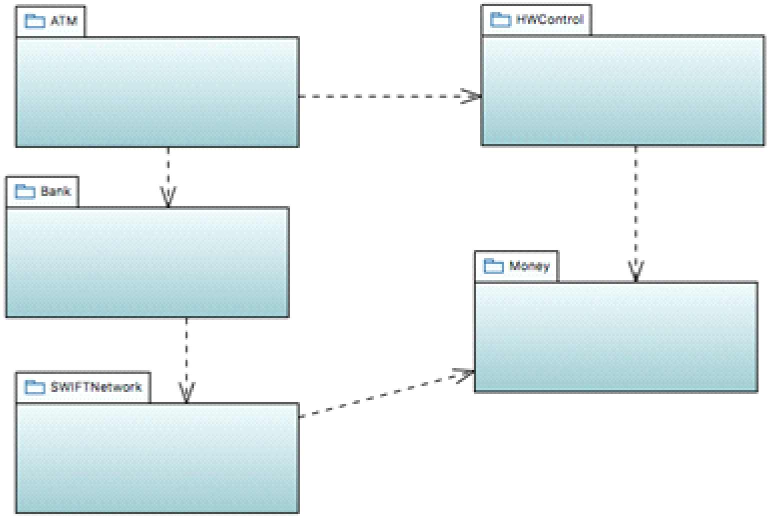


Figure 91 - Packages with Test Items for System Test of IBEN

In order to perform the system testing of IBEN, all the five packages in the system model are needed. The packages *ATM*, *Money*, and *HWControl* are known from the previous examples. The figure below illustrates the contents of the *Bank* package. The *IBank* interface provides methods to find, credit, and debit accounts. It checks credentials and wires money from one account to another. The *IAccount* interface also provides operations to credit and debit accounts, in addition to checking the balance of an account.

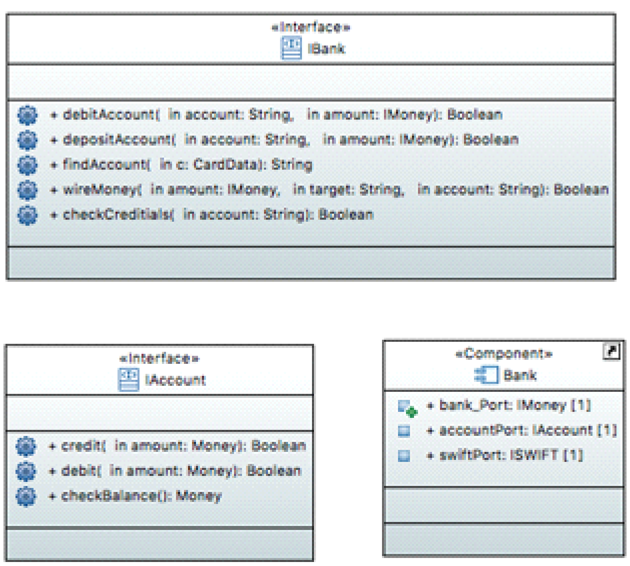


Figure 92 - Classes and Components in *Bank* Package

The figure below shows the content of the *SWIFTNetwork* package. The *ISWIFT* interface provides an operation to transfer a given amount from a source account to a target account. Since system testing is a black-box test strategy, only the communication between the interfaces is of interest.

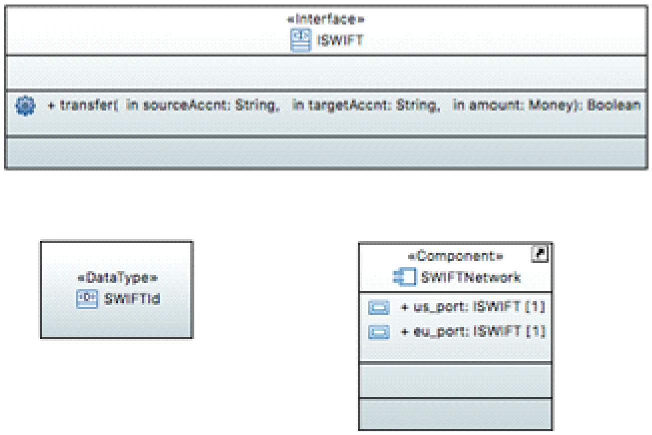


Figure 93 - Classes and Components in the *SWIFTNetwork* Package

For the system testing, the following test requirements are defined:

1. EU and US initiated transactions must behave correctly.

2. Money can be transferred rom an US account to an EU account, and vice-versa.

3. An invalid transfer should be identified and canceled.

4. The system should handle up to 1000000 transactions in parallel without system failure.

The figure below shows the system test context. The test items are the SWIFTNetwork, the US and EU Banks, and the ATM systems. Three test cases called *runUSTrxn*, *runEUTrxn* and *loadTest* are specified in this test context. The test cases *runUSTrxn* or *runEUTrxn* approve that a transaction that is initiated from the US ATM will be transferred to the EU Bank, or vice versa. The test case *loadTest* verifies a non-functional test requirement. It shall approve that IBEN behaves correctly even by high transaction requests. Two additional test components called *TransactionController* and *LoadManager* provide the capability to execute and verify that the money is transferred correctly.

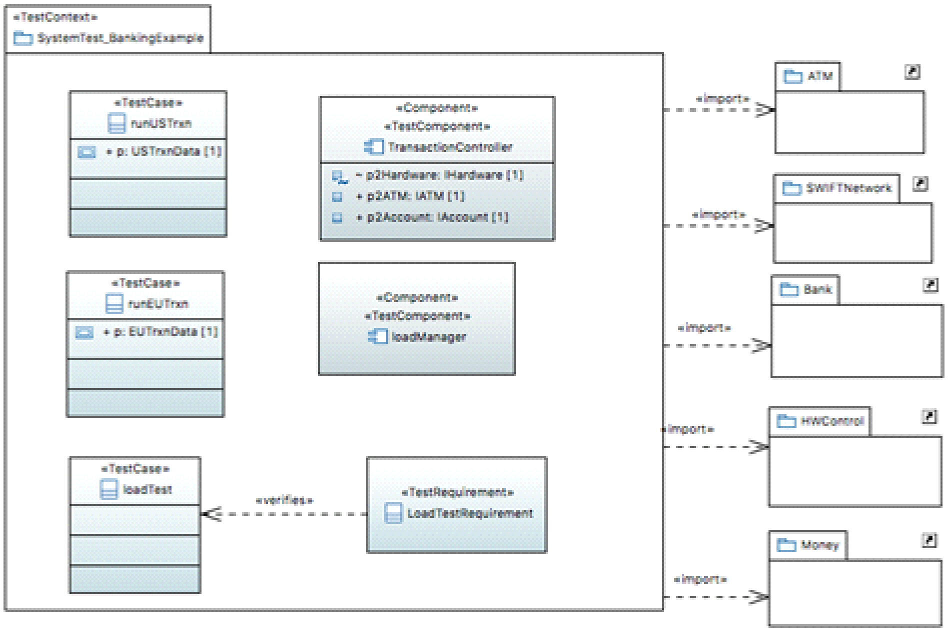


Figure 94 - System Test Context

The [test configuration]() is illustrated in the figure below. The *TransactionController* drives both ATMs on the European and US sides and is used to represent the accounts for both the US and EU banks. The *LoadManager* provides and controls the workload of the load test. It has access to the test data in the *SystemTestDataPool*.

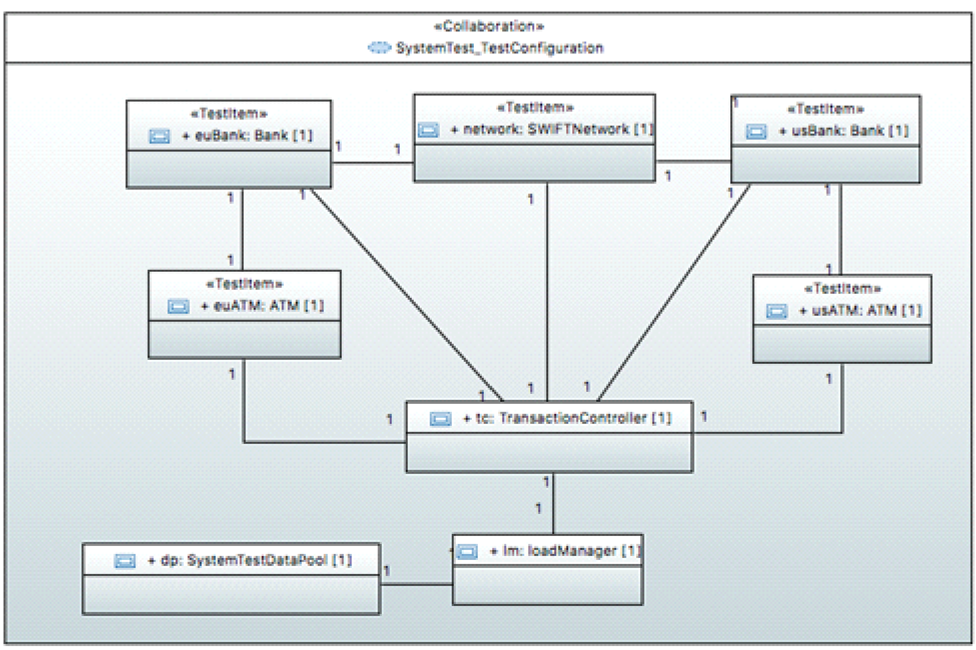


Figure 95 - System Test Configuration

The figure below shows data used for the system test. *TrxnData* defines the transaction data.

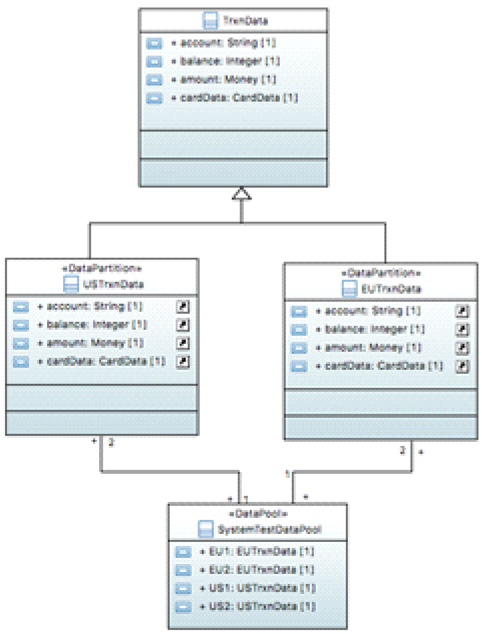


Figure 96 - Test Data and its Variations

The data pool *SystemTestDataPool* contains instances of *TrxnData* called *EU1*, *EU2*, *US1* and *US2* (see figure below. Two data partitions are defined in order to distinguish the EU transactions from the US transactions. These data partitions are chosen from the data pool and have two data samples each. Data instance *EU1* is shown in the diagram explicitly by all its attribute values[5]. Another data instance called *Fred* defines a modification of *EU1*, where 500 override the balance of 10000.

[5] This diagram only shows the data values of EU1. Those of EU2, US1 and US2 are equivalently defined.

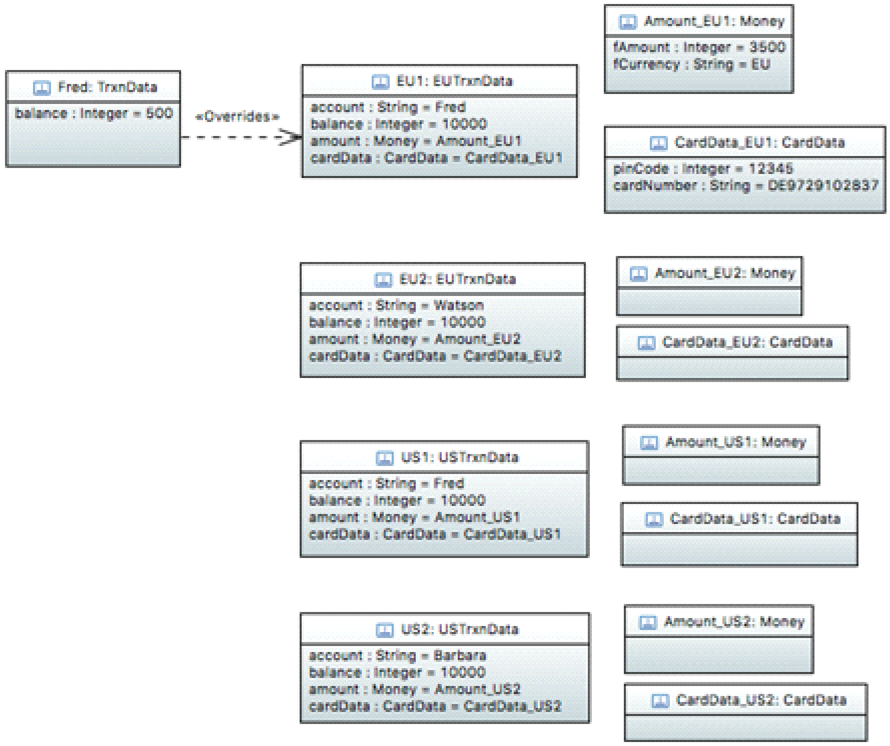


Figure 97 - Data Instances and its Modification

The figure below illustrates the behavior of test case *loadTest* which shall verify the test requirement 4 listed above. This test case shall approve that minimum 100 and maximum 1000000 transactions can be successfully handled in parallel. The *LoadArbitrationSpecification* will assure that whenever a transaction fails, the whole test will fail.

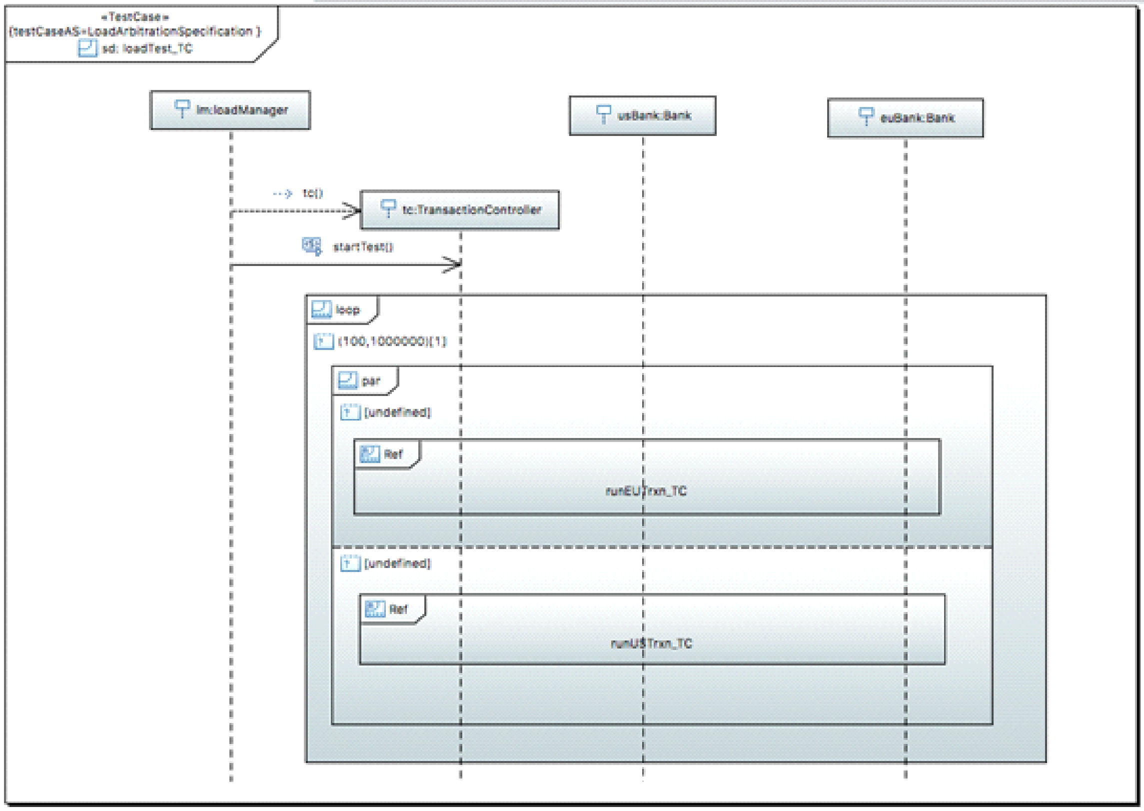


Figure 98 - Test Case *loadTest*

10.5.5 References

[UTP1.2] Object Management Group: "UML Testing Profile, version 1.2", OMG Document Number: formal/2013-04-03

[JUnit_Example] http://junit.sourceforge.net/doc/cookbook/cookbook.htm

[JUnit_web] www.junit.org

11 Annex B (Informative): Mappings

11.1 Mapping between UTP 1 and UTP 2

The following table summarizes the changes on stereotypes of UTP 2 compared with UTP 1.2:

Name	Change from UTP 1.2
Alternative	«Alternative» has been newly introduced by UTP 2.
AlternativeArbitration-Specification	Newly introduced by UTP 2.
AnyValue	Changed and renamed from UTP 1.2. In UTP 1.2, «AnyValue» was called «LiteralAny» and extended Literal-Specification.
ArbitrationSpecification	«ArbitrationSpecification» has been newly introduced into UTP 2.
AtomicProceduralEle-ment	«AtomicProceduralElement» has been newly introduced by UTP 2.
AtomicProceduralEle-mentArbitrationSpecifi-cation	Newly introduced by UTP 2.
BoundaryValueAnalysis	«BoundaryValueAnalysis» has been newly introduced by UTP 2.
CauseEffectAnalysis	«CauseEffectAnalysis» has been newly introduced by UTP 2.
ChecklistBasedTesting	«ChecklistBasedTesting» has been newly introduced by UTP 2.
CheckPropertyAction	«CheckPropertyAction» has been newly introduced by UTP 2.
CheckPropertyArbitra-tionSpecification	Newly introduced by UTP 2.
ChoiceOfValues	«ChoiceOfValues» has been newly introduced by UTP 2.
Classification-TreeMethod	«ClassificationTreeMethod» has been newly introduced by UTP 2.
CollectionExpression	
CombinatorialTesting	«CombinatorialTesting» has been newly introduced by UTP 2.
ComplementedValue	«ComplementedValue» has been newly introduced by UTP

Name	Change from UTP 1.2
	2.
Complements	«Complements» has been newly introduced by UTP 2.
CompoundProceduralElement	«CompoundProceduralElement» has been newly introduced by UTP 2.
CompoundProceduralElementArbitrationSpecification	Newly introduced by UTP 2.
CreateLogEntryAction	«CreateLogEntryAction» has been newly introduced by UTP 2.
CreateLogEntryArbitrationSpecification	Newly introduced by UTP 2.
CreateStimulusAction	«CreateStimulusAction» has been newly introduced by UTP 2.
CreateStimulusArbitrationSpecification	Newly introduced by UTP 2.
DataPartition	«DataPartition» has been newly introduced by UTP 2.
DataPool	Changed from UTP 1.2. In UTP 1.2 «DataPool» extended both Classifier and Property.
DataProvider	«DataProvider» has been newly introduced by UTP 2.
DataSpecification	«DataSpecification» has been newly introduced by UTP 2.
DecisionTableTesting	«DecisionTableTesting» has been newly introduced by UTP 2.
EquivalenceClassPartitioning	«EquivalenceClassPartitioning» has been newly introduced by UTP 2.
ErrorGuessing	«ErrorGuessing» has been newly introduced by UTP 2.
ExpectResponseAction	«ExpectResponseAction» has been newly introduced by UTP 2.
ExpectResponseArbitrationSpecification	Newly introduced by UTP 2.
ExperienceBasedTechnique	«ExperienceBasedTechnique» has been newly introduced by UTP 2.
ExploratoryTesting	«ExploratoryTesting» has been newly introduced by UTP 2.
Extends	«Extends» has been newly introduced by UTP 2.

Name	Change from UTP 1.2
GenericT-estDesignDirective	«GenericTestDesignDirective» has been newly introduced by UTP 2.
GenericTestDesign-Technique	«GenericTestDesignTechnique» has been newly introduced by UTP 2.
Loop	«Loop» has been newly introduced by UTP 2.
LoopArbitrationSpecification	Newly introduced by UTP 2.
MatchingCollectionExpression	«CollectionExpression» has been newly introduced by UTP 2.
Morphing	«Morphing» has been newly introduced by UTP 2.
Negative	«Negative» has been newly introduced by UTP 2.
NegativeArbitration-Specification	Newly introduced by UTP 2.
NSwitchCoverage	«NSwitchCoverage» has been newly introduced by UTP 2.
OpaqueProceduralElement	«OpaqueProceduralElement» has been newly introduced by UTP 2.
overrides	«overrides» was renamed by UTP 2. In UTP 1.2, it was named «modifies».
PairwiseTesting	«PairwiseTesting» has been newly introduced by UTP 2.
Parallel	«Parallel» has been newly introduced by UTP 2.
ParallelArbitrationSpecification	Newly introduced by UTP 2.
ProceduralElement	«ProceduralElement» has been newly introduced by UTP 2.
ProceduralElementArbitrationSpecification	Newly introduced by UTP 2.
ProcedureInvocation	«ProcedureInvocation» has been newly introduced by UTP 2.
ProcedureInvocationArbitrationSpecification	Newly introduced by UTP 2.
RangeValue	«RangeValue» has been newly introduced by UTP 2.

Name	Change from UTP 1.2
Refines	«Refines» has been newly introduced by UTP 2.
RegularExpression	«RegularExpression» has been newly introduced by UTP 2.
RoleConfiguration	«RoleConfiguration» is newly introduced in UTP 2.
Sequence	«Sequence» has been newly introduced by UTP 2.
SequenceArbitration-Specification	Newly introduced by UTP 2.
StateCoverage	«StateCoverage» has been newly introduced by UTP 2.
StateTransitionTech-nique	«StateTransitionTechnique» has been newly introduced by UTP 2.
SuggestVerdictAction	«SuggestVerdictAction» has been newly introduced by UTP 2.
SuggestVerdictArbitra-tionSpecification	Newly introduced by UTP 2.
TestCase	Changed from UTP 1.2. «TestCase» extended Behavior and Operation in UTP 1.2.
TestCaseArbitration-Specification	Newly introduced by UTP 2.
TestCaseLog	Newly introduced by UTP 2.
TestComponent	Changed from UTP 1.2. In UTP 1.2., «TestComponent» only extended Class.
TestComponentConfig-uration	«TestComponentConfiguration» has been newly introduced into UTP 2.
TestConfiguration	«TestConfiguration» has been newly introduced into UTP 2. It was conceptually represented by the composite structure of a «TestContext» in UTP 1.2.
TestConfigurationRole	«TestConfigurationRole» is newly introduced in UTP 2.
TestContext	Changed from UTP 1.2. In UTP 1.2 «TestContext» extend-ed StructuredClassifier and BehavioredClassifier as well as incorporated the concepts TestSet, TestExecutionSchedule and TestConfiguration into a single concept.
TestDesignDirective	«TestDesignDirective» has been newly introduced by UTP 2.
TestDesignDirec-tiveStructure	«TestDesignDirectiveStructure» has been newly introduced by UTP 2.

Name	Change from UTP 1.2
TestDesignInput	«TestDesignInput» has been newly introduced by UTP 2.
TestDesignTechnique	«TestDesignTechnique» has been newly introduced by UTP 2.
TestDesignTechniqueStructure	«TestDesignTechniqueStructure» has been newly introduced by UTP 2.
TestExecutionSchedule	«TestExecutionSchedule» has been newly introduced by UTP 2. It was conceptually represented as the classifier behavior of a «TestContext» in UTP 1.2.
TestItem	«TestItem» has been newly introduced into UTP 2 and supersedes the «SUT» stereotype in UTP 1.
TestItemConfiguration	«TestItemConfiguration» has been newly introduced into UTP 2.
TestLog	Changed from UTP 1.2. In UTP 1.2 «TestLog» was used to capture the execution of a test case or a test set (called test content in UTP 1.2). In UTP 2, two dedicated concepts have been newly introduced therefore (i.e., «TestCaseLog» and «TestSetLog»).
TestLogStructure	Newly introduced by UTP 2.
TestLogStructureBinding	Newly introduced by UTP 2.
TestObjective	Changed from UTP 1.2. In UTP 1.2, «TestObjective» was called «TestObjectiveSpecification».
TestProcedure	«TestProcedure» has been newly introduced by UTP 2.
TestRequirement	«TestRequirement» has been newly introduced into UTP 2.
TestSet	«TestSet» has been newly introduced by UTP 2. It was part of the TestContext in UTP 1.2.
TestSetArbitrationSpecification	Newly introduced by UTP 2.
TestSetLog	Newly introduced by UTP 2.
TransitionCoverage	«TransitionCoverage» has been newly introduced by UTP 2.
TransitionPairCoverage	«TransitionPairCoverage» has been newly introduced by UTP 2.
UseCaseTesting	«UseCaseTesting» has been newly introduced by UTP 2.
verifies	«verifies» has been newly introduced into UTP 2. In UTP 1.2 the «verify» stereotype from SysML was recommended.

The three primitive data types including Timepoint, Duration, and Timezone are also removed from UTP 2.

The following stereotypes are also removed from UTP 2: «GetTimeZoneAction», «SetTimeZoneAction», «DataSelector», «CodingRule», «LiteralAnyOrNull», and «TestLogEntry».

12 Annex C (Informative): Value Specification Extensions

12.1 Profile Summary

The following table gives a brief summary on the stereotypes introduced by the UML Testing Profile 2 (listed in the second column of the table). The first column specifies the mapping to the conceptual model shown in the previous section and the third column specifies the UML 2.5 metaclasses that are extended by the stereotypes.

Stereotype	UML 2.5 Metaclasses	Concepts
ChoiceOfValues	Expression	data
CollectionExpression	Expression	data
ComplementedValue	ValueSpecification	data
MatchingCollectionExpression	Expression	• data • data specification
RangeValue	Expression	data specification

12.2 Non-normative data value extensions

In addition to the normative ValueSpecification extensions of UTP, for sake of simplicity, UTP provides also some more extensions as part of this non-normative annex. These kinds of ValueSpecifications are:

- Complemented: Represents a set of expected response argument values for a known type described by a the complemented set of values described the underlying ValueSpecfciation and checks if actual response argument value belongs to that set.
- RangeValue: Represents a set of ordered expected response argument values for a known type described by its upper and lower boundaries. The Actual response argument value matches with each expected one if the actual one belongs to the set defined by its boundaries.
- ChoiceOfValues: Represents a set of expected response argument values for a known type described by an enumeration of values. The actual response argument value matches with expected one if the actual one belongs to the set defined by the enumeration.
- MatchingCollectionExpression: Represents a set of expected response argument collection values for a known type described by the members of the expected collection and the matching kind operator. The actual response argument collection

value match with the expected ones if the actual one belongs to the set of collections values defined by members and the collection matching kind.

- CollectionExpression: Represents a collection value used for defining argument collection values for stimuli or expected response values. If used as expected response argument collection value the actual response argument collection value matches with the expected one if their respective members match with each other. In case ordering is important, the members should also occur in the exact same order.

Implementations of the profile are free to decide how to incorporate and offer the non-normative extensions to the users.

12.2.1 Overview of non-normative ValueSpecification Extensions

The diagram below shows some additional, non-normative extensions to the UML ValueSpecifications metamodel. These UTP ValueSpecification extensions are deemed helpful for testers in order to be express data values used to specify the payload for stimuli and expected responses. It is treated as non-normative extension nonetheless, because all the given extensions could also be expressed by means of the OCL, which is considered as integral part of UML. However, OCL imposes additional knowledge on the test engineers which may result in a reduced acceptance by the industrial testing community. Therefore, this non-normative extension to the UTP provides dedicated concepts as special ValueSpecifications which can be immediately used by the testers without knowing anything about OCL at all. All these extended ValueSpecifications have been taken over from [TTCN-3] where they have been proven beneficial for the design of executable test cases in the industry since many years.

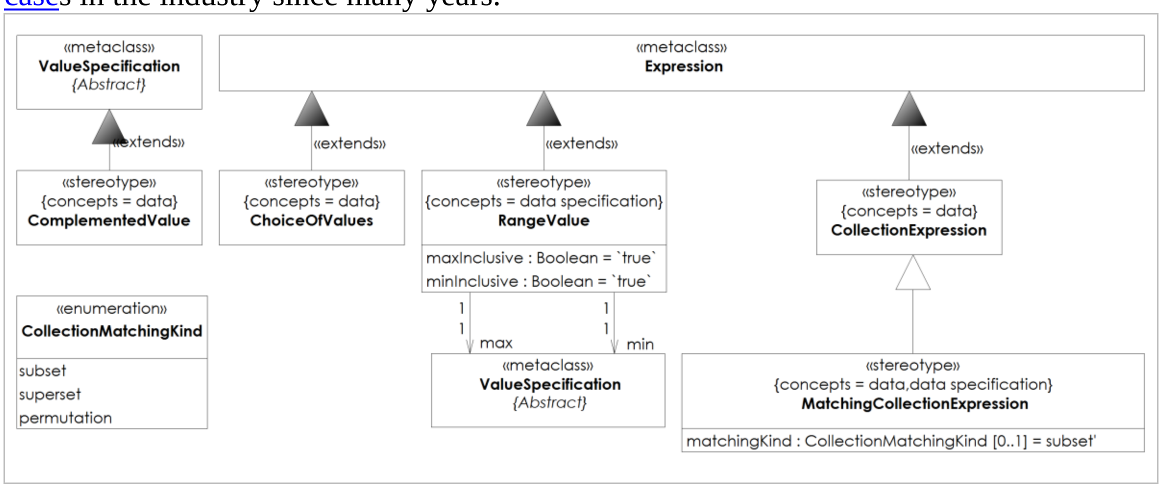


Figure 99 - Overview of non-normative ValueSpecification Extensions

12.2.2 Stereotype Specifications

12.2.2.1 ChoiceOfValues

Description	ChoiceOfValues represents an enumeration of possible values defined for the payload of an expected response, out of which at least one entry must match with the payload of the actual response. If a choice of possible values is used in a check response data action, then the enumerated values denote several possible check response data actions out of which one possible value must match with the actually received response data. The list of possible values is expressed as the list of ValueSpecifications composed by the underlying Expression's operand attribute. As defined above, any available ValueSpecification can be enumerated as choice of possible values. As a recommendation, ChoiceOfValues must either be only in check response data actions in test cases or for test generation. It is highly recommended to not use ChoiceofValues as payload for stimulus for it may negatively affect the repeatability of test case executions. Formally, a choice of values *COV* is specified by a subset of the SignatureElement's type: $COV \in \mathcal{P}(T_{SE})$ The actual argument a matches the expected argument specified by ChoiceOfValues if $a \in COV$
Extension	Expression
Change from UTP 1.2	«ChoiceOfValues» has been newly introduced by UTP 2.

12.2.2.2 CollectionExpression

Description	A CollectionExpression enables the modelling of collections based on the ValueSpecification metaclass Expression. Using collections values is essential when specifying stimuli and expected responses of a test case. By means of the stereotype «CollectionExpression» it is possible to describe inline values for a given ConnectableElement (i.e., Property or Parameter) and use those collections values as payload for a stimulus or an expected response as required. The kind (i.e., order and uniqueness) of the CollectionExpression is prescribed by the related MultiplicityElement (i.e., Property or Parameter) of this CollectionExpression. «CollectionExpression» might be used as payload for both stimulus and expected responses. If it represents the payload of an expected response, the payload of the actual responses must match with the expected CollectionExpression with respect to both, items listed in the collection and their respective index in the actual payload collection, if the corresponding ConnectableElement (i.e., Property or Parameter) is ordered. Any deviation is supposed to result in a mismatch.
Extension	Expression
Sub Class	MatchingCollectionExpression

12.2.2.3 ComplementedValue

Description	A ComplementedValue specifies a set of values that are not contained in the set specified by the genuine ValueSpecification. Formally, the ComplementedValue V_c is then defined as the complement of the ValueSpecification of SE, i.e. $V_c = T \setminus V \quad \text{with } V \in \mathcal{P}_{SE}(T)$
Extension	ValueSpecification
Change from UTP 1.2	«ComplementedValue» has been newly introduced by UTP 2.

12.2.2.4 MatchingCollectionExpression

Description	A MatchingCollectionExpression is a CollectionExpression that enables the tester to define matching criteria when used as the payload of an expected response. Thus, it is not allowed to use a MatchingCollectionExpression as payload for a stimulus, but only as payload for expected responses. The CollectionMatchingKind attribute of the CollectionExpression determines the matching mechanism that must be applied on the actual payload when received in order to calculate a match or mismatch of actual and expected responses. These matching kinds are the following: • subset (default) • superset • permutation If the corresponding MultiplicityElement (i.e., Property or Parameter) has is ordered (i.e., isOrdered = true), the collection items in the payload of the actual response have to occur in the exact same order as the elements in the expected response. Whether nested CollectionExpressions are considered to be flattened for the comparison of expected and actual responses is not defined in UTP 2.
Extension	Expression
Super Class	CollectionExpression
Attributes	matchingKind : CollectionMatchingKind [0..1] = subset'
Constraints	Must be used as payload for an expected responses /* A MatchingCollectionExpression must only specify the payload of an expected response. */ Use of permutation matching kind /* The matchingKind permutation must only be applied if the corresponding ConnectableElement (i.e., Property or Parameter) of the expected response has set isOrdered to false. */
Change from UTP 1.2	«CollectionExpression» has been newly introduced by UTP 2.

12.2.2.5 RangeValue

Description	A RangeValue represents a range between two naturally ordered boundaries, the upper and the lower bound. A RangeValue can be used as wildcard value (i.e. qualified) instead of a concrete value (i.e. quantified). Conceptually, a range represents an enumeration of the values between the min and max values; however, it does not represent a set or collection of values. In that sense, RangeValue is semantically equivalent to a ChoiceOfValue: ValueSpecification would explicitly enumerate all value between the min and max boundary. The eventual min value must always be less or equal than the eventual max value. In case that the min and max evaluate to the very same value, the range spans only a single value. If minInclusive is set to true, the lower boundary (represented by the min value) is included in the range, otherwise it is excluded. Default is true (i.e., the min value is included). If maxInclusive is set to true, the upper boundary (represented by the max value) is included in the range, otherwise it is excluded. Default is true, i.e., the max value is included. For example, if the min value evaluates to 10 and minInclusive is set to false, the actual lowerBoundary is 11. If a RangeValue is used in combination with an Integer- or Real-typed element, the lower and upper bounds describes the lowest and highest number of that numeric instance. If a RangeValue used in combination with a String-typed element (or subclasses thereof), the lower and upper bounds determine the minimal and maximal length of that String's instance. Users are allowed to define other proprietary natural orderings (e.g., complex types and re-use RangeValue to denote upper and lower boundaries for these types). The semantics how the ordering is defined; however, is out of scope of the RangeValue concept. If applied to an expected response, a RangeValue matches with the actual received value from the test item, and if the actual value is within the boundaries of the expected RangeValue.	
Extension	Expression	
Attributes	maxInclusive : Boolean [1] = \`true\`	
	minInclusive : Boolean [1] = \`true\`	

Associations	min : ValueSpecification
	max : ValueSpecification
Constraints	Operands shall be empty /* The attribute *operand* of the underlying Expression must be empty. */ context RangeValue inv: self.base_Expression.operand->notEmpty()
Change from UTP 1.2	«RangeValue» has been newly introduced by UTP 2.

13 Index

E

F

G

H

I

K

L

M

P

S

T

U

V