Date: April 2013

UML Testing Profile (UTP)

Version 1.2 with change bars

OMG Document Number: formal/2013-04-04
Standard document URL: http://www.omg.org/spec/UTP/1.2/
Associated Files:
Normative: http://www.omg.org/spec/UTP/20120801/utp_1.2.xmi
http://www.omg.org/spec/UTP/20120801/utptypes_1.2.xmi

Copyright © 2002-2003, Ericsson
Copyright © 2002-2003, International Business Machines Corporation
Copyright © 2002-2003, FOKUS
Copyright © 2002-2003, Motorola, Inc.
Copyright © 2002-2013, Object Management Group
Copyright © 2002-2003, Rational
Copyright © 2002-2003, Softeam
Copyright © 2002-2003, Telelogic

USE OF SPECIFICATION - TERMS, CONDITIONS & NOTICES

The material in this document details an Object Management Group specification in accordance with the terms, conditions and notices set forth below. This document does not represent a commitment to implement any portion of this specification in any company's products. The information contained in this document is subject to change without notice.

LICENSES

The companies listed above have granted to the Object Management Group, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version. Each of the copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

Subject to all of the terms and conditions below, the owners of the copyright in this specification hereby grant you a fully-paid up, non-exclusive, nontransferable, perpetual, worldwide license (without the right to sublicense), to use this specification to create and distribute software and special purpose specifications that are based upon this specification, and to use, copy, and distribute this specification as provided under the Copyright Act; provided that: (1) both the copyright notice identified above and this permission notice appear on any copies of this specification; (2) the use of the specifications is for informational purposes and will not be copied or posted on any network computer or broadcast in any media and will not be otherwise resold or transferred for commercial purposes; and (3) no modifications are made to this specification. This limited permission automatically terminates without notice if you breach any of these terms or conditions. Upon termination, you will destroy immediately any copies of the specifications in your possession or control.

PATENTS

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

GENERAL USE RESTRICTIONS

Any unauthorized use of this specification may violate copyright laws, trademark laws, and communications regulations and statutes. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or used in any form or by any means--graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems--without permission of the copyright owner.

DISCLAIMER OF WARRANTY

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE.
IN NO EVENT SHALL THE OBJECT MANAGEMENT GROUP OR ANY OF THE COMPANIES LISTED ABOVE BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, RELIANCE OR COVER DAMAGES, INCLUDING LOSS OF PROFITS, REVENUE, DATA OR USE, INCURRED BY ANY USER OR ANY THIRD PARTY IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The entire risk as to the quality and performance of software developed using this specification is borne by you. This disclaimer of warranty constitutes an essential part of the license granted to you to use this specification.

RESTRICTED RIGHTS LEGEND

Use, duplication or disclosure by the U.S. Government is subject to the restrictions set forth in subparagraph (c) (1) (ii) of The Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013 or in subparagraph (c)(1) and (2) of the Commercial Computer Software - Restricted Rights clauses at 48 C.F.R. 52.227-19 or as specified in 48 C.F.R. 227-7202-2 of the DoD F.A.R. Supplement and its successors, or as specified in 48 C.F.R. 12.212 of the Federal Acquisition Regulations and its successors, as applicable. The specification copyright owners are as indicated above and may be contacted through the Object Management Group, 109 Highland Avenue, Needham, MA 02494, U.S.A.

TRADEMARKS

MDA®, Model Driven Architecture®, UML®, UML Cube logo®, OMG Logo®, CORBA® and XMI® are registered trademarks of the Object Management Group, Inc., and Object Management Group™, OMG™ , Unified Modeling Language™, Model Driven Architecture Logo™, Model Driven Architecture Diagram™, CORBA logos™, XMI Logo™, CWM™, CWM Logo™, IIOP™ , IMM™, MOF™ , OMG Interface Definition Language (IDL)™ , and OMG Systems Modeling Language (OMG SysML)™ are trademarks of the Object Management Group. All other products or company names mentioned are used for identification purposes only, and may be trademarks of their respective owners.

COMPLIANCE

The copyright holders listed above acknowledge that the Object Management Group (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

Software developed under the terms of this license may claim compliance or conformance with this specification if and only if the software compliance is of a nature fully matching the applicable compliance points as stated in the specification. Software developed only partially matching the applicable compliance points may claim only that the software was based on this specification, but may not claim compliance or conformance with this specification. In the event that testing suites are implemented or approved by Object Management Group, Inc., software developed using this specification may claim compliance or conformance with the specification only if the software satisfactorily completes the testing suites.

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/report_issue.htm).

Table of Contents

Preface v

1 Scope 1

2 Conformance 2

2.1 Conformance with UML 2

2.2 Conformance with UTP 3

3 References 3

3.1 Normative References 3

3.2 Informative References 3

4 Terms and Definitions 4

5 Symbols and Acronyms 7

6 Additional Information 7

6.1 Language Architecture 7

6.2 Dependencies to OMG Specifications 8

6.2.1 Normative Dependencies 8

6.2.1.1 Unified Modeling Language (UML) 8

6.2.1.2 Object Constraint Language (OCL) 8

6.2.2 Informative Dependencies 8

6.3 How to Read this Specification 9

6.4 Acknowledgments 9

7 Predefined Type Library 13

7.1 General 13

7.2 Abstract Syntax 13

7.3 Stereotype Description 13

7.3.1 Arbiter 13

7.3.2 Duration 14

7.3.3 Timepoint 15

7.3.4 Timer 15

7.3.5 Timezone 16

7.3.6 Verdict ... 17
8 Test Architecture ... 19
8.1 General ... 19
8.2 Foundation ... 19
8.2.1 Abstract Syntax ... 20
8.2.2 Stereotype Descriptions ... 20
8.2.2.1 SUT ... 20
8.2.2.2 TestComponent ... 21
8.2.2.3 TestContext ... 22
9 Test Behavior ... 25
9.1 General ... 25
9.2 Foundation ... 25
9.2.1 Abstract Syntax ... 25
9.2.2 Stereotype Descriptions ... 26
9.2.2.1 DetermAlt ... 26
9.2.2.2 FinishAction ... 28
9.2.2.3 LogAction ... 29
9.2.2.4 TestCase ... 30
9.2.2.5 ValidationAction ... 32
9.3 Defaults ... 34
9.3.1 Abstract Syntax ... 34
9.3.2 Stereotype Description ... 34
9.3.2.1 Default ... 34
9.3.2.2 DefaultApplication ... 37
9.4 Timer-related Concepts ... 38
9.4.1 Abstract Syntax ... 39
9.4.2 Stereotype Description ... 39
9.4.2.1 ReadTimerAction ... 39
9.4.2.2 StartTimerAction ... 40
9.4.2.3 StopTimerAction ... 41
9.4.2.4 TimeOut ... 41
9.4.2.5 TimeOutMessage ... 42
9.4.2.6 TimeOutAction ... 43
9.4.2.7 TimerRunningAction... 43
9.5 Timezone Co-ordination... 44
9.5.1 Abstract Syntax ... 44
9.5.2 Stereotype Descriptions ... 44
9.5.2.1 GetTimezoneAction ... 44
9.5.2.2 SetTimezoneAction ... 45
10 Test Data ... 47
10.1 General ... 47

10.2 Test Data Specification 47
10.2.1 Abstract Syntax 48
10.2.2 Stereotype Descriptions 48
10.2.2.1 DataPartition 48
10.2.2.2 DataPool 49
10.2.2.3 DataSelector 49
10.3 Test Data Values 50
10.3.1 Abstract Syntax 51
10.3.2 Stereotype Descriptions 51
10.3.2.1 CodingRule 51
10.3.2.2 LiteralAny 52
10.3.2.3 LiteralAnyOrNull 53
10.3.2.4 Modification 53
11 Test Management 59
11.1 General 59
11.2 Test Planning and Scheduling 59
11.2.1 Abstract Syntax 60
11.2.2 Stereotype Description 60
11.2.2.1 TestObjectiveSpecification 60
11.3 Test Monitoring and Control 63
11.4 Test Result Analysis 64
11.4.1 Abstract Syntax 64
11.4.2 Stereotype Description 64
11.4.2.1 TestLog 64
11.4.2.2 TestLogApplication 66
11.4.2.3 TestLogEntry 67
Annex A - Deprecated Elements 71
Annex B - Test Management Concepts 73
Annex C - Mappings 77
Annex D - Examples 89
Annex E - Stereotype and Type Overview 113
Annex F - MOF-based Metamodel 119
Annex G - Scheduling 131
Annex H - XMI Schema 139

Preface

About the Object Management Group

OMG

Founded in 1989, the Object Management Group, Inc. (OMG) is an open membership, not-for-profit computer industry standards consortium that produces and maintains computer industry specifications for interoperable, portable and reusable enterprise applications in distributed, heterogeneous environments. Membership includes Information Technology vendors, end users, government agencies and academia.

OMG member companies write, adopt, and maintain its specifications following a mature, open process. OMG's specifications implement the Model Driven Architecture® (MDA®), maximizing ROI through a full-lifecycle approach to enterprise integration that covers multiple operating systems, programming languages, middleware and networking infrastructures, and software development environments. OMG's specifications include: UML® (Unified Modeling Language™); CORBA® (Common Object Request Broker Architecture); CWM™ (Common Warehouse Metamodel); and industry-specific standards for dozens of vertical markets.

More information on the OMG is available at *http://www.omg.org/*.

OMG Specifications

As noted, OMG specifications address middleware, modeling, and vertical domain frameworks. All OMG specifications are available from the OMG website at:

http://www.omg.org/spec

Specifications are organized by the following categories:

Business Modeling Specifications

Middleware Specifications

- **CORBA/IIOP**
- **Data Distribution Services**
- **Specialized CORBA**

IDL/Language Mapping Specifications

Modeling and Metadata Specifications

- **UML, MOF, CWM, XMI**
- **UML Profile**

Modernization Specifications

Platform Independent Model (PIM), Platform Specific Model (PSM), Interface Specifications

- **CORBAServices**
- **CORBAFacilities**

OMG Domain Specifications

CORBA Embedded Intelligence Specifications

CORBA Security Specifications

All of OMG's formal specifications may be downloaded without charge from our website. (Products implementing OMG specifications are available from individual suppliers.) Copies of specifications, available in PostScript and PDF format, may be obtained from the Specifications Catalog cited above or by contacting the Object Management Group, Inc. at:

OMG Headquarters
109 Highland Avenue
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
Email: *pubs@omg.org*

Certain OMG specifications are also available as ISO standards. Please consult *http://www.iso.org*

Typographical Conventions

The type styles shown below are used in this document to distinguish programming statements from ordinary English. However, these conventions are not used in tables or section headings where no distinction is necessary.

Times/Times New Roman - 10 pt.: Standard body text

Helvetica/Arial - 10 pt. Bold: OMG Interface Definition Language (OMG IDL) and syntax elements.

Courier - 10 pt. Bold: Programming language elements.

Helvetica/Arial - 10 pt: Exceptions

Note – Terms that appear in *italics* are defined in the glossary. Italic text also represents the name of a document, specification, or other publication.

Issues

The reader is encouraged to report any technical or editing issues/problems with this specification to *http://www.omg.org/report_issue.htm*.

1 Scope

The UML Testing Profile 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. Model-based test specifications expressed with the UML Testing Profile are independent to any methodology, domain, or type of system.

The UML Testing Profile is a part of the UML ecosystem (see Figure 1.1), 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.

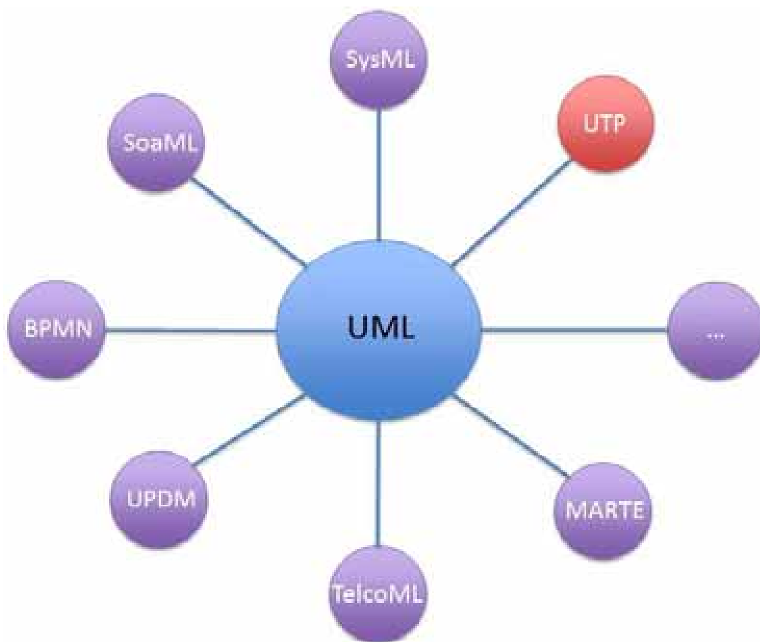


Figure 1.1 - The UML ecosystem

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 system under test (SUT) and its boundaries, the derivation of test components, and the identification of communication channels between the SUT and the test components over which data can be exchanged.
- Build the model-based test specification 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 SUT and to verify whether the SUT 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 test data: Modeling of test data includes the data values being used for stimuli into the SUT as well as for responses expected from the SUT (e.g., the test oracle).
- Provide necessary information pertinent to test scheduling optimization: Test scheduling optimization can be based on priorities, risk-related information, 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 like progress reports.

The intended audience being able to read model-based test specifications expressed with UML Testing Profile models includes, among others:

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

The intended audience of the 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.
- People who want to tailor the UML Testing Profile to satisfy (project-/domain-/process-)specific needs.

2 Conformance

2.1 Conformance with UML

The UML Testing Profile must be conformant with the UML compliance level L3. This is due to the fact that the upcoming and revised UML 2.5 will only define one single compliance level that is identical to what is called L3 [UMLs].

2.2 Conformance with UTP

The UML Testing Profile defines two compliance levels. Compliance level 1 deals with the most fundamental profile capabilities of a UML tool, whereas compliance level 2 provides optional enhancements upon compliance level 1. Meeting compliance level 1 is required for an implementation to be considered as standard compliant implementation. The optional enhancements of compliance level 2 can be implemented in combination or separately of each other.

Level 1-compliant implementations must meet the following requirements:

1. Abstract Syntax: Standard compliant implementations have to comprise the UML Testing Profile's abstract syntax, (i.e., all stereotypes that are defined in the normative clauses). Stereotypes or tag definitions in non-normative clauses of this specification are considered to be optional.

2. Type Library: This specification defines a predefined type library for the UML Testing Profile. Standard compliant implementations have to provide this predefined type library.

3. XMI exchange: Standard compliant implementations have to be exchangeable among UML-compliant tools XMI exchange format defined by the UML metamodel and the UML Testing Profile.

Level 2-compliant implementations must meet one or both of the following requirements in addition to level 1:

4. Notation: This specification defines only few notational extensions to UML's concrete syntax. Standard compliant implementations may incorporate the dedicated notational extensions, respectively concrete syntax of the UML Testing Profile, however, they are not required to actually provide those. In case a vendor strives to meet also this optional compliance points, the notations are not allowed to be proprietary but match exactly the definitions made in this specification.

5. Constraints: Standard compliant implementation may implement and check the constraints that have been defined for UML Testing Profile abstract syntax in an automated way.

3 References

3.1 Normative References

- [UMLi] Unified Modeling Language (UML) Specification: Infrastructure, version 2.41, formal/formal/2011-08-05, 2011
- [UMLs] Unified Modeling Language (UML) Specification: Superstructure, version 2.4.1, formal/2011-08-06, 2011
- [OCL] Object Constraint Language (OCL), version 2.3.1, formal/2012-01-01, 2012
- [MOF] Meta Object Facility (MOF) Core Specification, version 2.4.1, formal/2011-08-08, 2011

3.2 Informative References

- [UTP_RTF] OMG ADTF: *RFP on a UML Testing Profile*, ad/01-07-08, 2001.
- [SysML] OMG Systems Modeling Language (OMG SysML), Version 1.2, formal/2010-06-01, 2010.
- [SoaML] Service oriented architecture Modeling Language (SoaML) Specification, Version 1.0, formal/2012-03-01, 2012.

- [MARTE] Modeling and Analysis of Real-Time and Embedded Systems (MARTE), Version 1.1, formal/2011-06-02, 2011.
- [BMM] Business Motivation Model (BMM), Version 1.1, formal/2012-03-01, 2012.
- [QFTP] UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms Specification (QFTP), Version 1.1, formal/2008-04-05, 2008.
- [JUNIT] JUnit: http://www.junit.org.
- [TTCN3p1] ES 201 873-1: The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language. V4.2.1 (2010-07), 2010.
- [TTCN3p3] ETSI ES 201 873-3: The Testing and Test Control Notation version 3; Part 3: TTCN-3 Graphical Presentation Format (GFT). V3.2.1 (2007-02), 2010.
- [MSC] ITU-T Z.120: Message Sequence Charts (MSC), Nov. 1999.
- [ISTQB] ISTQB Standard glossary of terms used in Software Testing, Version 2.1 (dd. April 1st, 2010)
- [IEEE1012] 1012-1998 - IEEE Standard for Software Verification and Validation, 1998
- [IEEE829] 829-2008 - IEEE Standard for Software Test Documentation, 2008
- [ISO29119] ISO/IEC 29119 – Software Testing, Draft, 2012
- [ISO9646-1] ISO/IEC 9646-1 -- Information technology -- Open Systems Interconnection -- Conformance testing methodology and framework -- Part 1: General concepts, 1994
- [IEEE610] 610-1991 – IEEE Standard Computer Dictionary, 1991
- [ISO24765] ISO/IEC/IEEE 24765:2010 – Systems and software engineering – Vocabulary, 2010
- [ES202951]] ETSI ES 202 951 - Methods for Testing and Specification (MTS); Model-Based Testing (MBT); Requirements for Modelling Notations, v1.1.1, 2012.

4 Terms and Definitions

This clause provides the terms and concepts of the UML Testing Profile.

Arbiter	See description of interface «Arbiter» on page 13
Black box testing	A test conducted without knowledge of the internal structure of the system under test.
Coding Rule	See description of «CodingRule» on page 51
Component testing	The testing of individual software components (see [ISTQB]).
Coordination	Concurrent (and potentially distributed) test components have to be coordinated both functionally and in time in order to assure deterministic and repeatable test executions resulting in well-defined test verdicts. Coordination is done explicitly with normal message exchange between components or implicitly with general ordering mechanisms.
Data Partition	See description «DataPartition» on page 48
Data Pool	See description «DataPool» on page 49

Default	See description «Default» on page 34
FinishAction	See description «FinishAction» on page 28
Glass box testing	See White box testing
Implementation under test (IUT) - ([ISO9646-1])	see SUT
Integration testing	Testing performed to expose defects in the interfaces and in the interactions between integrated components or systems (see [ISQTB]).
Model-based testing	An umbrella of techniques that use (semi-)formal models as engineering artifacts in order to specify and/or generate test-relevant artifacts, such as test cases, test scripts, reports, etc. (changed from [ES202951]).
Observation	Test data reflecting the reactions from the SUT and used to assess the SUT reactions which are typically the result of a stimulus sent to the SUT.
Priority	The level of importance assigned to an item. [ISO24765]
Scheduler	See description of interface Scheduler in the Annex.
Stimulus	Test data sent to the SUT in order to control it and to make assessments about the SUT when receiving the SUT reactions to these stimuli.
SUT	See description «SUT» on page 20
System testing	The process of testing an integrated system to verify that it meets specified requirements (see [ISQTB]).
Test bed - ([IEEE610])	see test environment
Test behavior	Dynamic aspects of a test case or parts of the test components involved in the test case.
Test case	See description «TestCase» on page 30
Test Component	See description «TestComponent» on page 21
Test Configuration	The collection of test component objects and of connections between the test component objects and to the SUT. The test configuration defines both (1) test component objects and connections when a test case is started (the initial test configuration) and (2) the maximal number of test component objects and connections during the test execution.
Test Context	See description «TestContext» on page 22
Test Control	A test control is a specification for the invocation of test cases within a test context. It is a technical specification of how the SUT should be tested with the given test context.
Test environment	An environment containing hardware, instrumentation, simulators, software tools, and other support elements needed to conduct a test [ISTQB]
Test execution system	see test execution tool
Test execution tool	A type of test tool that is able to execute other software using an automated test script, e.g., capture/playback. [after ISTQB]
Test Invocation	A test case can be invoked with specific parameters and within a specific context. The test invocation leads to the execution of the test case. The test invocation is denoted in the test log.

Test item - ([IEEE829])	see SUT
Test level	Specific instantiation of a test sub-process. [ISO29119]
Test Log	See description «TestLog» on page 64.
Test model	A model that specifies various testing aspects, such as test objectives, test plans, test architecture, test cases, test data etc.
Test object - ([ISTQB])	see SUT
Test Objective	See description «TestObjectiveSpecification» on page 60.
Test phase	see test level
Test step	The smallest atomic (i.e., indivisible) part of a test case specification that is executed by a test execution system during test case execution.
Test type	Group of testing activities that are focused on specific quality characteristics. [ISO29119]
Testing	The process consisting of all life cycle activities, both static and dynamic, concerned with planning, preparation and evaluation of software products and related work products to determine that they satisfy specified requirements, to demonstrate that they are fit for purpose and to detect defects. [ISTQB]. Any activity which provide information on the qualities of a system under test.
Timer	See description of «Timer» on page 15.
Timezone	See description of «Timezone» on page 16.
Unit testing	See Component testing.
Utility Part	A part of the test system representing miscellaneous components that help test components to realize their test behavior. Examples of utility parts are miscellaneous features of the test system.
Validation	The process of determining whether or not the software and/or its specification meet system requirements and user needs at both functional and performance levels (see [IEEE1012]). Answers: Did we build the right product.
Validation Action	See description of «ValidationAction» on page 32.
White box testing	Testing based on an analysis of the internal structure of the component or system.
Wildcard	Wildcards allow the user to explicitly specify whether the value is present or not, and/or whether it is of any value. Wildcards are special symbols to represent values or ranges of values. Wildcards are used instead of symbols within instance specifications. Three wildcards exist: a wildcard for any value, a wildcard for any value or no value at all (i.e. an omitted value), and a wildcard for an omitted value. See description of «LiteralAny» on page 52 and «LiteralAnyOrNull» on page 53

5 Symbols and Acronyms

MOF	Meta-Object Facility
OCL	Object Constraint Language
UML	Unified Modeling Language
SysML	Systems Modeling Language
TTCN-3	Testing and Test Control Notation, version 3
OMG	Object Management Group
CORBA	Common Object Request Broker Architecture
CWM	Common Warehouse Metamodel
MDA	Model-Driven Architecture
XMI	XML: Metadata Interchange
GIOP	General Inter-ORB Protocol
IIOP	Internet Inter-ORB Protocol
ASN.1	Abstract Syntax Notation One
PER	Packed Encoded Rules
IDL	Interface Definition Language
XML	eXtensible Markup Language
MSC	Message Sequence Charts
GFT	Graphical Presentation Format

6 Additional Information

6.1 Language Architecture

The UML Testing Profile consists of two normative packages, a model library providing predefined types and a profile that uses the predefined types for its definition. The UML Testing Profile has a flat hierarchy, i.e., it consists of exactly one single profile package that is not intended to be further sub-structured. Figure 6.1 depicts the overall architecture of the UML Testing Profile including its relation to UML itself.

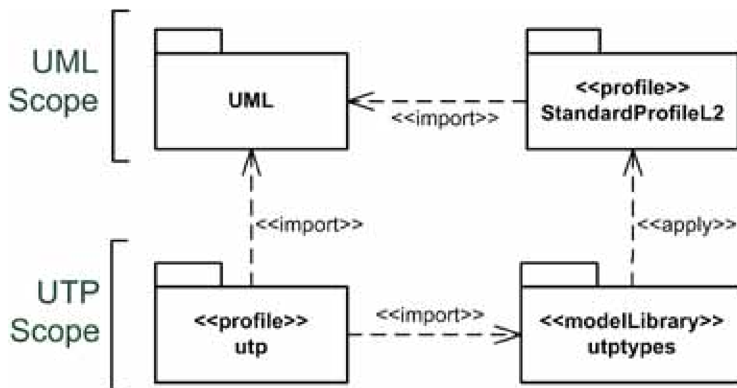


Figure 6.1 - UML Testing Profile Language Architecture

6.2 Dependencies to OMG Specifications

6.2.1 Normative Dependencies

6.2.1.1 Unified Modeling Language (UML)

The UML Testing Profile is based on the UML specification. The UML Testing Profile is defined by using the metamodeling and profiling mechanism of UML. It has been designed with the following principles in mind:

- UML integration: as a native UML profile, the UML Testing Profile is defined on the basis of the metamodel provided in the UML Superstructure [UMLs] and follows the principles of UML profiles as defined in the UML Infrastructure [UMLi].
- Reuse and minimality: wherever possible, the UML Testing Profile directly uses the UML concepts for extension. New concepts are only added where explicitly needed. Only those concepts are extended or added by UML Testing Profile, which have been demonstrated in the software, hardware, and protocol testing area to be of central relevance to the definition of test specifications and are not part of UML.

6.2.1.2 Object Constraint Language (OCL)

The UML Testing Profile uses OMG's Object Constraint Language (OCL) to precisely formalize the constraints stated in natural language for the abstract syntax. This is supposed to support tool vendors to implement the UML Testing Profile properly with regard to its constraints. As already said in sub clause 2.2, standard compliant implementations are neither enforced to actually implement these constraints with OCL, nor to implement these constraints at all. They are mainly used for a precise clarification of the constraints complementary to the description given in natural language.

6.2.2 Informative Dependencies

A UML Testing Profile model may have further, optional dependencies to other OMG specifications. The following list does not claim to be complete, but briefly describes purposeful combinations of UTP with other models taken from the experiences of the UML Testing Profile working group.

- SysML: User may want to combine the UML Testing Profile and SysML in order to leverage the requirements capabilities introduced by SysML to express requirements traceability. In addition, the UML Testing Profile can be used to specify tests for socio-technical systems defined in SysML.

- MARTE: In order to define test specifications for real-time and embedded systems, user may combine the UTP with (parts of) MARTE.
- SoaML: User may want to combine the UML Testing Profile with SoaML in order to be able to express test specifications for service-oriented architectures with UTP.
- BMM: The Business Motivation Model (BMM) may be used to specify testing goals and objectives, to systematically derive and document a testing strategy, as well to identify and assess potential risks.

6.3 How to Read this Specification

This specification is intended to be read by the audience listed in Clause 1 in order to learn, apply, implement, and support UTP.

To start with, all readers are encouraged to read Clause 1. In order to learn more about the conformance of UML and UTP as well as the compliance levels between the UTP specification and the UTP tool implementation, please read Clause 2. Some references to other standards are listed in Clause 3. Important definition of terms, acronyms, and additional information are listed in the Clauses 4-6.

The definition of the UML Testing Profile itself can be found in Clauses 7-11. Clause 7 starts with the definition of predefined types that are required for the specification of the UML Testing Profile. The concepts are grouped into four logical packages: test architecture, test behavior, test data, and test management. In the Annex you will find various additional information, e.g., UTP examples, Mapping rules to JUnit and TTCN-3, MOF-based Metamodel, XMI schema of UTP, etc.

Modeling tool vendors should read the whole document, including the annexes. Modelers and engineers shall read Annex D to understand how the language is applied to an example. This document may be read in both sequential and non-sequential manner.

6.4 Acknowledgments

The following companies submitted and/or supported parts of the original specification:

- Fraunhofer FOKUS
- Ericsson
- Motorola
- University of Lübeck
- Telelogic
- IBM
- Softeam
- iLogix
- IRISA
- Scapa Technologies

The following persons were members of the team that designed and wrote this specification: Ina Schieferdecker, Øystein Haugen, Paul Baker, Zhen Ru Dai, Jens Grabowski, Serge Lucio, Eric Samuelson, Clay Williams.

The following companies submitted and/or supported parts of the 1.1 RTF of this specification:

- Fraunhofer FOKUS
- KnowGravity, Inc.
- Lockheed Martin
- SINTEF
- sepp.med GmbH
- UFR Sciences et Techniques
- Softeam
- THALES
- IBM
- NoMagic, Inc.
- University of Applied Sciences Hamburg
- University of Pitesti

The following persons were members who submitted and/or supported parts of the 1.1 RTF of this specification: Marc-Florian Wendland, Markus Schacher, Jon D. Hagar, Ina Schieferdecker, Armin Metzger, Zhen Ru Dai, Fabien Perureux, Eldad Palachi, Laurent Rioux, J.D. Baker, Alin Stefanescu, Andreas Hoffmann, Max Bureck

The following companies submitted and/or supported parts of the 1.2 RTF of this specification:

- Fraunhofer FOKUS
- KnowGravity, Inc.
- Lockheed Martin
- University of Applied Sciences Hamburg
- sepp.med GmbH
- Softeam
- SELEX SI
- THALES
- IBM
- UFR Sciences et Techniques
- simVentions
- CEA List
- SINTEF

The following persons were members who submitted and/or supported parts of the 1.2 RTF of this specification: Marc-Florian Wendland, Markus Schacher, Jon D. Hagar, Ina Schieferdecker, Armin Metzger, Zhen Ru Dai, Cyril Ballagny, Fabien Perureux, Eldad Palachi, Gabriella Carrozza, Georg Götz, Matt Willson, Laurent Rioux, Arnaud Cuccuru, Oystein Haugen, Andreas Hoffmann.

7 Predefined Type Library

7.1 General

Some concepts which are introduced by the UML Testing Profile require certain types in addition to those already provided by UML. This clause describes the predefined UML Testing Profile type library that is mandatory in order to implement the UML Testing Profile.

7.2 Abstract Syntax

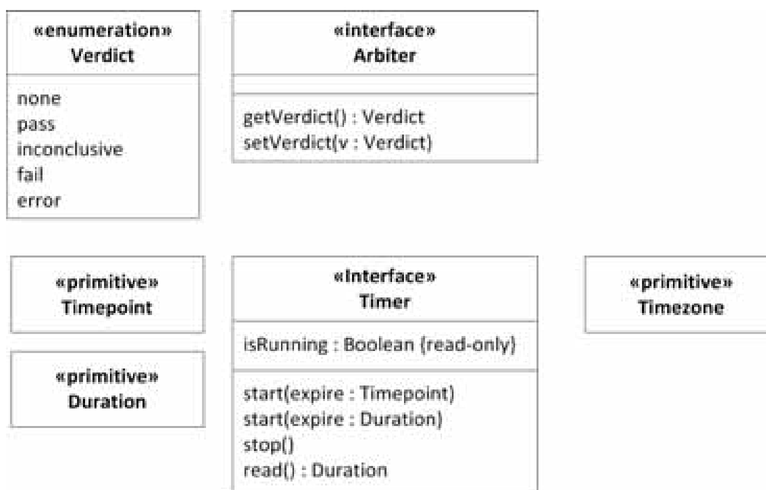


Figure 7.1 - Predefined Type Library

7.3 Stereotype Description

7.3.1 Arbiter

Description

Arbiter is a predefined interface defining operations used for arbitration of tests. Test cases, test contexts, and the runtime system can use realizations of this interface to assign verdicts of tests and to retrieve the current verdict of a test (verdicts are discussed in “Verdict” on page 17).

Generalizations

None.

Attributes

None.

Operations

- getVerdict() : Verdict
  Returns the current verdict.

- setVerdict(v : Verdict)
  Sets a new verdict value.

Semantics

Arbiter is a predefined interface provided with the UML Testing Profile. The purpose of an arbiter implementation is to determine the final verdict for a test case. This determination is done according to a particular arbitration strategy, which is provided in the implementation of the arbiter interface.

The arbiter interface provides two operations for use in verdict setting: *getVerdict* and *setVerdict*. The *setVerdict* operation is used to provide updated information to the arbiter by a test component regarding the current status of the test case in which it is participating. Every validation action causes the *setVerdict* operation on the arbiter implementation to be invoked (see Section 9.2.2.5, “ValidationAction,” on page 32 for more information on validation actions.)

A test case or a test context use the arbiter to evaluate test results and to assign the overall verdict of a test case or test context respectively. There is a default arbitration algorithm based on functional, conformance testing, which generates **Pass**, **Fail**, **Inconclusive**, and **Error** as verdict, where the precedence of the verdicts are defined as **Pass** < **Inconclusive** < **Fail** < **Error**.

The arbitration algorithm can be user-defined, if needed.

Constraints

None.

Notation

No additional notation for arbiter defined.

Examples

An example arbiter can be found in Figure D.29.

7.3.2 Duration

Description

Duration is a predefined primitive type used to specify a time range.

Semantics

A Duration denotes an arbitrary time range. The concrete time unit or the format of a how a Duration is expressed is not predefined and is left open to the user. Instances of Duration have to be expressed as uml::sDuration with its type set to utp::Duration.

The keyword ‘now’ is a short cut for expressing the current point in time.

Constraints

[1] Values of the primitive type Duration have to be expressed as uml::Duration with its type set to utp::Duration.

7.3.3 Timepoint

Description

Timepoint is a predefined primitive type used to specify concrete points in time a Timer is supposed to expire.

Semantics

A Timepoint represents a concrete point in time in the future, where a Timer is supposed to expire and to raise a time out event. Instances for that primitive type have to be expressed as TimeExpression with its type set to Timepoint. The concrete format of such a point in time is not predefined and is left open to the user.

The keyword 'now' is a short cut for expressing the current point in time.

Constraints

[1] Values of the primitive type Timepoint have to be expressed as TimeExpression with its type set to Timepoint.

7.3.4 Timer

Description

A predefined interface specifying features needed to specify a timer. A timeout is generated automatically when a timer expires and is sent to the timer owner. Timers provide means to observe and control test behavior. Also, a timer can be used to prevent from deadlocks, starvation, and instable system behavior during test execution.

Generalizations

None.

Attributes

- isRunning : Boolean [1]
  Returns true if the timer is currently active, false otherwise. isRunning is a read only property.

Operations

- start(expires : Timepoint)
  Starts the timer and sets the time of expiration.

- start(expires : Duration)
  Starts the timer and sets the duration of expiration.

- stop()
  Stops the timer.

- read() : Duration
  Reads the expiration time of the timer.

Semantics

A timer is owned by an active class, commonly participating in test cases, and is started with a predefined time of expiration. A timer can either be private or public. If private, the time can only be accessed by its owning active class. If public, anyone with sufficient visibility may access and manipulate the timer. When a timer expires after its predefined time, a timeout is generated automatically. It is sent immediately to the active class that owns the timer. What timeout representation is used for the timeout depends on the behavioral description in that the timeout is supposed to occur.

Timers can be started, stopped, and checked by a start timer action (calling start()), a stop timer action (calling stop()), a timer running action (reading property isRunning), and a read timer action (calling read()). By means of the start() operation, a timer may be started with a certain time value. The predefined time value of a timer is always positive. For example, "start Timer1(now+2.0)" means to start a timer and to stop it at latest in 2 time units, otherwise it expires. With the stop() operation, an active timer can be stopped. The expiration time of an active timer can be retrieved by the read() operation.

The timer attribute *isRunning* is a boolean value and indicates whether the timer is still active or not.

Constraints

[1] Only test components and test contexts may own properties realizing the timer interface.

[2] In case the timer is started with a Timepoint, the expiration time point must be a time point in the future.

Notation

A timer has no specific notation; however, the timer actions used to access the features of the timer do have specific notation. For details, look in the sub clauses for «StartTimerAction», «StopTimerAction», and «ReadTimerAction».

Examples

An example for a timer definition can be found in Figure D.9.

7.3.5 Timezone

Description

Timezone is a predefined primitive type representing a timezone. Timezones are used to group test components together. Test components belonging to the same timezone (i.e., having the same value on the zone attribute) are synchronized and can share time values.

Semantics

Timezones are used to group test components together. Test components with the same timezone value constitute a group and are considered to be synchronized. The semantics of synchronization is not specified. Timezone values can be compared for equality.

Comparing time-critical events within the same timezone is allowed. Comparing time-critical events of different timezones is a matter of semantic variation point and should be decided by the tool vendor. By default, comparison between events in two different timezones is illegal.

Semantic Variation Point

The comparison of time-critical events from different timezones is illegal by default.

7.3.6 Verdict

Description

A verdict is a predefined enumeration specifying the set of possible evaluations of a test case. Five enumeration literals are defined: none, pass, fail, inconclusive, error.

- none The test case has not been executed yet.
- pass The system under test adheres to the expectations.
- inconclusive The evaluation cannot be evaluated to be pass or fail.
- fail The system under test differs from the expectation.
- error An error has occurred within the testing environment.

The Verdict type may be extended by the users with more literals.

Semantics

The verdict is a predefined enumeration datatype that contains at least the values fail, inconclusive, pass, error indicating how this test case execution has performed.

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 SUT has been carried out yet. **None** is the weakest verdict. It is never set by the tester directly. A **pass** indicates that the test case is successful and that the SUT has behaved according to what should be expected. A **fail** on the other hand shows that the SUT is not behaving according to the specification. An **inconclusive** means that the test execution cannot determine whether the SUT performs well or not. An **error** tells that the test system itself and not the SUT fails.

The precedence rules for the predefined literals are: **none** < **pass** < **inconclusive** < **fail** < **error**. This means whenever a test component submits a local verdict fail, the final test case verdict will never result with a **pass**, nor **inconclusive**.

The final verdict of a test case is determined by an arbiter.

Constraints

None.

Notation

The predefined literals **none**, **pass**, **inconclusive**, **fail**, and **error** are shown as keywords (normally in bold face).

Examples

None.

8 Test Architecture

8.1 General

Concepts described in the test architecture section are essential to specify structural aspects of a test environment and a corresponding test configuration in order to embed and execute test cases against a system under test. The test environment comprises everything that is necessary to execute test cases, e.g., test components, hardware, simulators, etc. The test configuration describes how parts of the test environment, in particular test components, are connected with the system under test. However, the UML Testing Profile does not provide any dedicated concepts for designing the system under test, since these capabilities are already provided by and inherited from UML. The system under test is part of the test configuration and due to the stringent black-box nature of the UML Testing Profile it does not provide any further information to the test environment than its outermost public interfaces (see Figure 8.1).

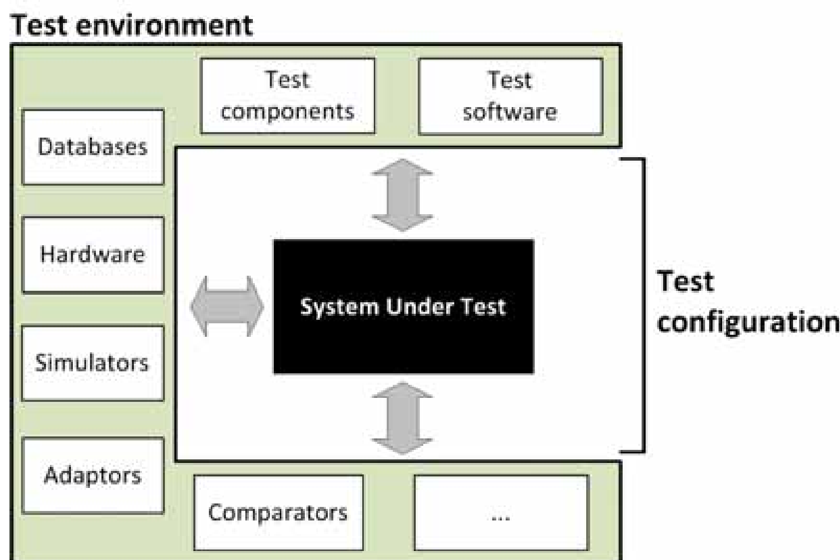


Figure 8.1 - Concepts of test environment and test configuration

Since the system under test represents just a specific role within a test configuration, it is not possible to determine what testing level is actually addressed. The system under test may represent a single component (component testing), a cluster of components (integration testing) or a complete system (system testing). This, in turn, keeps the complexity of the system under test always on the same level, i.e., a black box with well-defined communication channels and ports, regardless whether a single component or an entire system has to be tested.

Further concepts of a test environment like the actual test cases, which are executed by that environment, are not shown here, since they belong the test behavioral part of the UML Testing Profile.

8.2 Foundation

This sub clause provides the user with fundamental concepts to define the test environment and test configuration. The combination of test environment and test configuration is referred to as test context within the UML Testing Profile. A test context may be described on three different architectural levels, which are:

1. Type level: On type level, the test context simply represents the architectural design of the test environment and test configuration.
2. Instance (specification) level: A single test context might be instantiated to precisely specify the instances that take part in the test configuration. Additionally, a test context might be instantiated several times in order to express varying test configuration, different sets of data that should be used for test case execution, etc.
3. Deployment level: By using the UML deployment specification concepts in combination with the concepts being specified in the test architecture chapter, the user can define how the test environment and the test configuration is deployed on physical artifacts (e.g., machines, files, databases, etc.).

8.2.1 Abstract Syntax

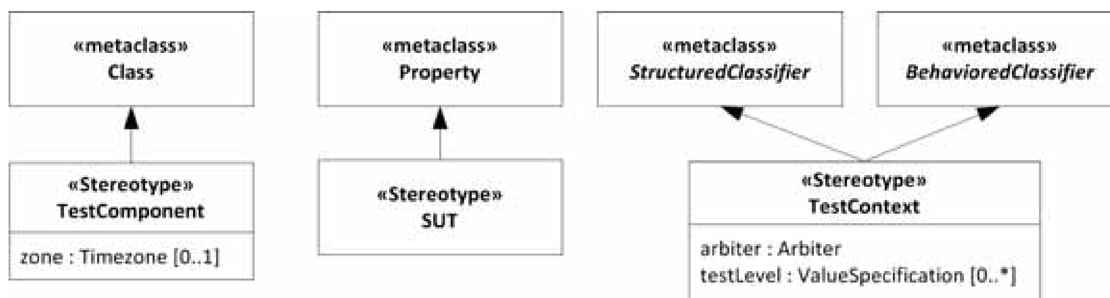


Figure 8.2 - Test Architecture Foundation

8.2.2 Stereotype Descriptions

8.2.2.1 SUT

Description

«SUT» is applied to one or more properties of a classifier to specify that they constitute the system under test. The features and behavior of the SUT is given entirely by the type of the property to which the stereotype is applied.

Extensions

- Property (from UML::CompositeStructures::InternalStructures)

Generalizations

None.

Attributes

None.

Semantics

The system under test (SUT) is a part of a test context. It refers to a system, subsystem, or component that is being tested. An SUT can consist of several objects. The SUT is stimulated via its public interface operations and signals by the test components. No internals of an SUT are known or accessible during test case execution, due to its black-box nature.

Constraints

None.

Notation

No additional notation for «SUT» defined.

Examples

Examples of «SUT» can be found in Figure D.10 and Figure D.24.

8.2.2.2 TestComponent

Description

Test components are part of the test environment and are used to communicate with the system under test (SUT) and other test components. The main function of test components is to drive a test case by stimulating the system under test through its provided interfaces and to evaluate whether the actual responses of the system under test comply with the expected ones.

Extensions

- Class (from UML::Kernel)

Generalizations

None.

Attributes

- zone : Timezone [0..1]
  Specifies the timezone to which a test component belongs (Timezones are discussed in "Timezone" on page 16.

Semantics

A test component is commonly an active class that provides or requires interfaces in order to communicate with the system under test (SUT) or other test components. Test components usually establish connectors between their dedicated (i.e., owned) ports and ports of other participants in a test case, e.g., either the system under test or other test components. Connectors are used as communication channels over which exchange of test data is actually carried out.

Test components participate in test cases mainly for stimulating the system under test (SUT) with test data and for evaluating whether the responses of the system under test adhere with the expected ones afterwards. In addition, test components can be used to provide auxiliary, user-defined functionality during the execution of a test case.

«TestComponent» can either be applied to a Class, a Component, or a Node. A test component as a Class or Component represents a part of the architectural design of the test environment. A test component as a Node is normally used as part of a deployment specification of the test environment.

In case the test system is distributed over different timezones (e.g., a test system that is deployed in the cloud) the involved test components may need to be kept in synch with regard to their actual timezones. This can be achieved by using the optional zone attribute, if required. The corresponding timezone actions (see SetTimezoneAction and GetTimezoneAction) are intended to set and retrieve the timezone of a test component.

Constraints

[1] «TestComponent» can only be applied to a Class or the following subclasses of Class: Component and Node.

context TestComponent

inv:
self.base_Class.oclIsKindOf(Class) or
self.base_Class.oclIsKindOf(Component) or
self.base_Class.oclIsKindOf(Node)

Notation

No additional notation for «TestComponent» defined.

Examples

Examples of «TestComponent» can be found in Figure D.9 and Figure D.22.

8.2.2.3 TestContext

Description

A test context acts as a grouping mechanism for a set of test cases. The composite structure of a test context is referred to as test configuration. The classifier behavior of a test context may be used for test control.

Extensions

- StructuredClassifier (from UML::CompositeStructures::InternalStructures)
- BehavioredClassifier (from UML::CommonBehaviors::BasicBehaviors, Communications)

Generalizations

None.

Attributes

- arbiter : Arbiter [1]
  Realizes the arbiter interface.
- testLevel : ValueSpecification [0..*]
  Indicates to what phase of a testing process this test context and its contained test cases belong.

Semantics

A test context is both a StructuredClassifier and BehavioredClassifier providing the foundation for a test configuration, a collection of test cases (operating on those test cases) and an optional test control that schedules the execution order of its test cases.

The test configuration is a collection of test component objects and of connections between the test component objects and to the SUT. A test configuration defines both (1) test component objects and connections when a test case is started (the initial test configuration) and (2) the maximal number of test component objects and connections during the test execution. It is established by using the composite structure concepts for StructuredClassifiers.

Test cases are either realized as operations or as owned behavior (both stereotyped with «TestCase») of a test context. If required, test control can be added to a test context by defining a classifier behavior for the test context, describing the order and under what conditions test cases are supposed to be executed.

The *test level* tag definition can be used to indicate to what phase of a testing process the test context, the test configuration and all the contained test artifacts in the test context belong. Common industry-related standards distinguish component level, integration level, system level, and acceptance level testing. The UML Testing Profile neither restricts the naming nor the number of test levels a user may want to use.

A test level inherently determines the actual boundaries of the system under test, since those normally differ at different test levels. However, each test context can belong to more than one test level (e.g., the same test environment might be used for system and acceptance testing). Conversely, two different test contexts may belong to the same test level, but can still target different system under tests (e.g., at component level).

This semantic adheres with the definitions given in other relevant standards, i.e., [ISO29119], [IEEE829], [ISTQB].

Constraints

[1] The classifier with «TestContext» applied must be both an instance of StructuredClassifier and BehavioredClassifier.

Notation

No additional notation for «TestContext» defined.

Examples

Figure 8.3 represents an example how a user may specify project-, process-, or domain-dependent test level values, as well as the corresponding object model for the relevant parts of the example.

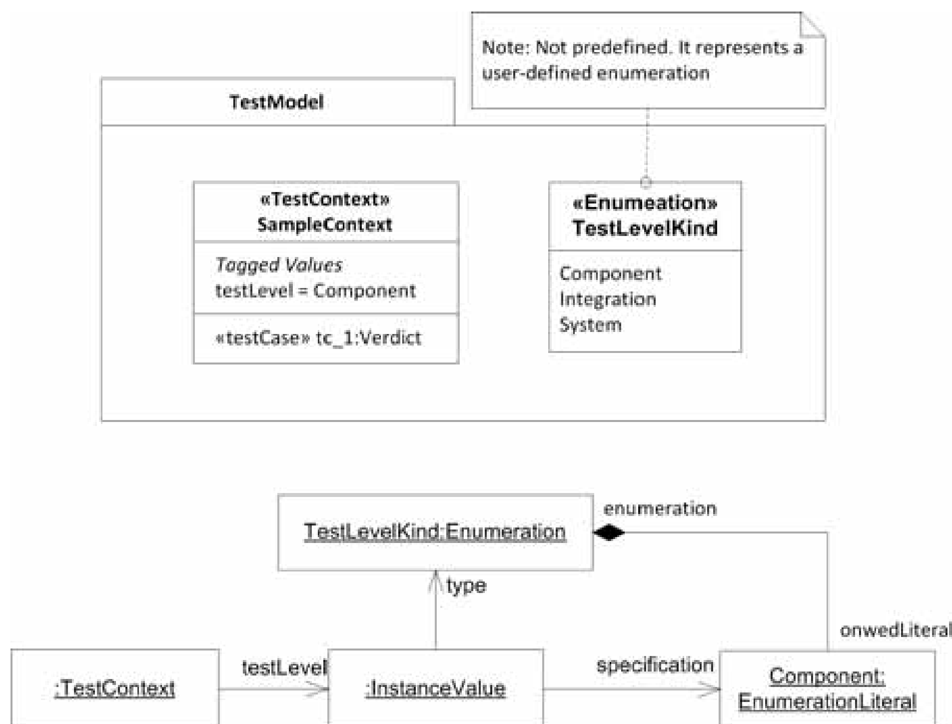


Figure 8.3 - User-defined test level values and corresponding object model

In the upper part a package called *TestModel* is shown. It contains a test context (Class or Component stereotyped with «TestContext») called *SampleContext*. The relevant part for this is the tagged value *testLevel*, depicted in the upmost compartment of *SampleContext*. It is set to value *Component*. Right beside the test context, a user-defined Enumeration *TestLevelKind* is shown with three literals: *Component, Integration, System*. The *Component* literal serves as the value for the *testLevel* tag definition of the test context *SampleContext*. Thus, within the scope of this test model, there is a clear and unambiguous understanding what values can actually be used for specifying the test level of a test context.

Using an Enumeration is just one example of how values for the *testLevel* tag definition might be specified. Other methodologies may provide different solutions, e.g., test levels expressed as LiteralStrings, OpaqueExpressions, or even complex types.

The lower part of Figure 8.3 shows the corresponding object model.
Examples of «TestContext» can be found in Figure D.3, Figure D.9, and Figure D.22.

9 Test Behavior

9.1 General

Test behavior is the concrete specification of the dynamic test behavior using e.g., sequences, alternatives, loops, and defaults to stimulate and observe the SUT. UTP introduces concepts to specify test behavior related to a test context with its predefined test configuration. For the definition of test behavior, any kind of UML behavior diagrams may be used, e.g., Sequence Diagram, State Machines, etc.

9.2 Foundation

In order to express the intention of a test, a *test objective* with a general description of what should be tested is given. Its concrete specification is a *test case*. A test case is a specification of one specific case to test the system, including the required test behavior with its test inputs, test conditions, and test result. Furthermore, a test case can also be specified as a set of single test cases, e.g., as a StateMachine.

A test case commonly consists of particular test-related actions like setting the verdict of a test case or logging of some relevant piece of information during a test case execution. These concepts are referred to as test actions.

9.2.1 Abstract Syntax

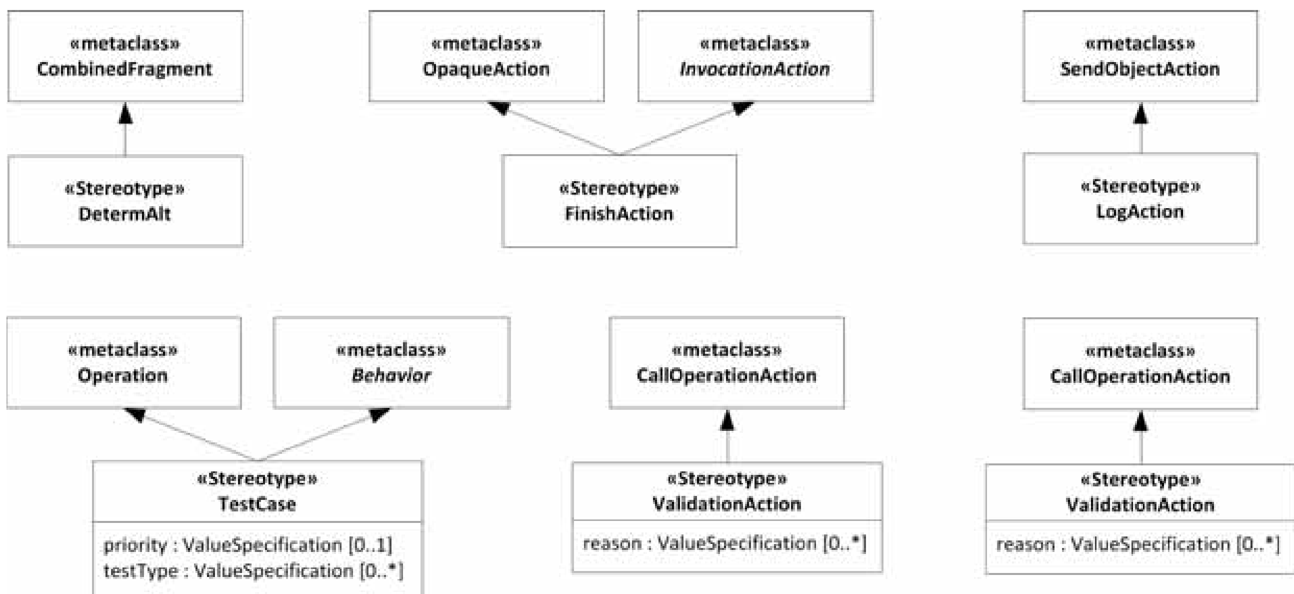


Figure 9.1 - Test Behavior Foundation

9.2.2 Stereotype Descriptions

9.2.2.1 DetermAlt

Description

A deterministic alternative is a CombinedFragment where the operands are evaluated in exact the same order as they appear in the model (respectively diagram) regardless of the fact that the guards of more than one InteractionOperand evaluate to true.

Extensions

- CombinedFragment (from UML::Interactions::Fragments)

Generalization

None.

Attributes

None.

Semantics

When a deterministic alternative is reached during the execution of a test case, the involved (i.e. covered) test components wait until they receive an event. An event can be either a message or a time out generated by a timer visible to the test component. After an event was recognized, the evaluation of the deterministic alternative is carried out. The evaluation mechanism of the guards of a deterministic alternative restricts the ordinary evaluation mechanism of combined fragments in UML in a way that guards only allowed to references values either local to the test component lifeline or global to the entire enclosing interaction. Furthermore, the evaluation of the entering condition for an interaction operand will be carried out in two-step-process during runtime:

- **Guard evaluation**: At first, the operand's guard condition will be evaluated. If it fails, the entire operand will be excluded for further investigation and not entered. If the guard condition evaluates to true, an additional evaluation step succeeds.
- **Occurrence specification evaluation**: After a successful guard evaluation, the very first occurrence specification on the test component lifeline within the interaction operand, which is supposed to be entered, will be checked. This may include a check whether a particular timer has expired, or whether a particular message is received.

  If the first covering OccurrenceSpecification in the InteractionOperand represents a check for a time out event, and the referenced timer has not expired during runtime, the InteractionOperand will not be entered.

  If the first covering OccurrenceSpecification represents the reception of a message, the InteractionOperand will be entered if, and only if, the expected message matches the actual message during runtime.

  An expected message matches with a actual messages, if all the messages arguments are equal to the expected message arguments. Wildcards may be used to leave certain arguments of the expecting message open. If the occurrence specification evaluation is also fulfilled, the interaction operand will be entered. Otherwise, the interaction operand will be skipped and the evaluation of the next interaction operand (if present) starts immediately.

It is neither specified when the test case will be concluded by the execution environment nor what test case verdict will be delivered if none of the interaction operands are allowed to be entered.

Textually in prefix notation the definition is as follows:

determAlt([guard1]op1) = **alt**([guard1]op1)

determAlt([guard1]op1, [guard2]op2) = **alt**([guard1]op1, [**else**] **determAlt**([guard2]op2))

In general

determAlt([guard1]op1, [guard2]op2, ..., [guardn]opn) =

alt([guard1]op1, [**else**]**determAlt**([guard2]op2,...,[guardn]opn))

Constraints

[2] The interaction operator of the deterministic alternative must be of InteractionOperatorKind::alt.

[3] The guards of a deterministic alternative's operands must contain only references to values local to the test component lifeline, or values global to the surrounding Interaction.

[4] The guards of a deterministic alternative must contain only references to values local to the lifeline, representing a classifier, stereotyped by «TestComponent», or values global to the whole Interaction.

Notation

Figure 9.2 shows three possible visualizations of a deterministic alternative.

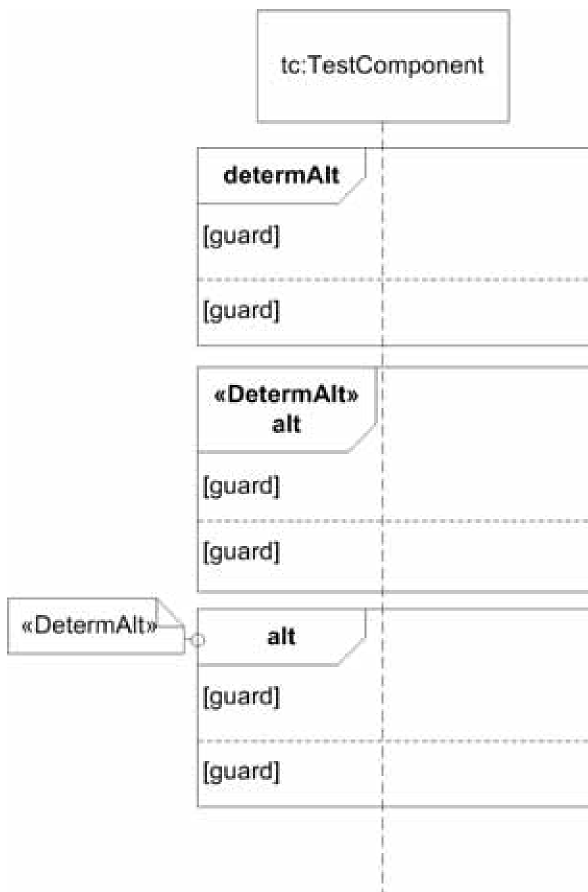


Figure 9.2- Visualization options for deterministic alternatives

The first option is to reuse the ordinary visualization of CombinedFragment from UML, but with 'determAlt' depicted in the upper left cornered rectangle of the CombinedFragment frame.

The second possibility is to put the stereotype («DetermAlt») above the label of the normal alt InteractionOperator. This is the closest analogy to UML's usual visualization of stereotypes.

The third possibility is to show a comment-like symbol with text «DetermAlt» beside the CombinedFragment that is attached with the CombinedFragment's frame.

Examples

An example of «DetermAt» can be found in Figure D.14.

9.2.2.2 FinishAction

Description

A finish action is an action that completes the test case for one test component immediately. The action has no implicit effect on other test components involved in the same test case, but it has recognized the need for other test components to be notified of the finish such that they may no longer expect messages from the finished test component. This must be specified explicitly.

Extensions

- OpaqueAction (from UML::Actions::BasicActions)
- InvocationAction (from UML::Actions::BasicActions)

Generalizations

None.

Attributes

None.

Semantics

A finish action moves a test component to a state where it awaits the conclusion of the test case that is now running. This may mean waiting for other test components to finish their behavior corresponding to that test case. In the traces of the test behavior for that test case, there will be no event occurrences on a test component after it was finished. Consequently, the local verdict of a finished test component cannot be altered.

- If the finish action is used within an Interaction, the test component to be finished is determined by the lifeline, representing a test component, which is covered by the ActionExecutionSpecification pointing to the finish action.
- If the finish action is used within an Activity and if there is at least one test component defined in the test configuration, than there must be a target input pin owned by the finish action. The target input pin is typed by the classifier of the test component. It represents the test component to be finished. The multiplicity of the target input pin must be 1.

Extending InvocationAction allows the invocation of a particular behavioral description on the finished test component, which is called when the test component is finished.

Constraints

[1] If «FinishAction» is applied to an OpaqueAction and if it is used within an activity, there is exactly one input pin with name target. Its type must be a StructuredClassifier with «TestComponent» applied, and its multiplicity must be set to 1.

[2] If «FinishAction» is applied to an InvocationAction, the multiplicity of the target input pin must be 1.

Notation

In Sequence diagrams, the «FinishAction» is shown as a black rectangle on the lifeline.

In StateMachine diagrams, the «FinishAction» is shown as a flow branch ending in a black quadrat.

In Activity diagrams, the «FinishAction» is shown as a black quadrat.

Examples

An example of «FinishAction» can be found in Figure D.14.

9.2.2.3 LogAction

Description

A log action is used to log entities during execution for further analysis. The logged entities can be simple strings, complete diagrams, instance values, or any other entity of interest.

Extensions

- SendObjectAction (from UML::Actions::IntermediateActions)

Generalizations

None.

Attributes

None.

Semantics

The target of a log action either refers implicitly to a logging facility in the run-time system, or explicitly to an element within the model acting as the receiver of the information to be logged. In the first case, the logging can be seen as a declaration of what should be logged rather than saying how or where the information is finally stored.

The request input pin of the underlying SendObjectAction determines what information shall be logged. The type of the request input pin is not determined and may represent simple strings, instance values, or any other information values of interest.

A log action must submit either one single information value or a set of information (at least one), which are supposed to be logged all at once. This is a shortcut for invoking the log action multiple times in a row.

Specifying the receiver of the log action is optional. Per default, omitting the receiver object means the execution environment is responsible to log the values transferred by the log action.

Constraints

[1] The multiplicity of the target input pin of a «LogAction» is set to 0..1.

[2] The multiplicity of the request input pin of a «LogAction» is set to 1..*.

[3] There must be no further argument input pins attached to a «LogAction».

Notation

No additional notation for «LogAction» defined.

Examples

None.

9.2.2.4 TestCase

Description

A test case is a behavioral feature or behavior specifying tests. A test case specifies how a set of test components interact with an SUT to realize a test objective.

Test cases are owned by test contexts, and therefore have access to all parts of the test configuration, other global variables (e.g., data pools, etc.) or further behavioral features (e.g., auxiliary methods).

A test case always returns a verdict. The verdict may be arbitrated — calculated by the arbiter, or non-arbitrated (i.e., provided by the test behavior).

Extensions

- Behavior (from UML::CommonBehaviors::BasicBehaviors)
- Operation (from UML::Kernel,Interfaces)

Generalizations

None.

Attributes

- priority : ValueSpecification [0..1]
  A comparable value of importance of a test case in order to optimize test planning.
- testType : ValueSpecification [0..*]
  Specific quality criteria that is verified by that test case.

Semantics

The semantics of test cases are given by the semantics of the (test) behavior that realizes it.

The tag definition priority may be used to support test planning activities, e.g., in a way that higher prioritized test cases are scheduled to be executed before lower prioritized test cases. An often used prioritization schema is an enumeration that consists of the three literals (in order of descending precedence): high, medium, low.

A test type indicates what concrete quality criteria are going to be verified by the related test case. A test context commonly contains multiple test cases which are targeting different quality criteria of the same system under test. The verification of a specific quality criterion of the system under test may be performed in a single test level or scattered across a different test levels (e.g., performance testing at executed at component testing level and also executed at a system testing level).

Constraints

[1] The type of the return result parameter of a test case must be Verdict.

[2] «TestCase» cannot be applied both to a behavior and its specification.

[3] If «TestCase» is applied to an operation, the featuring classifier must have «TestContext» applied.

[4] If «TestCase» is applied to a behavior, the context of the behavior must have «TestContext» applied.

Notation

No additional notation for «TestCase» defined.

Examples

Figure 9.3 represents an example how a user may specify project-, process-, or domain-dependent test type values, as well as the corresponding object model for the relevant parts of the example.

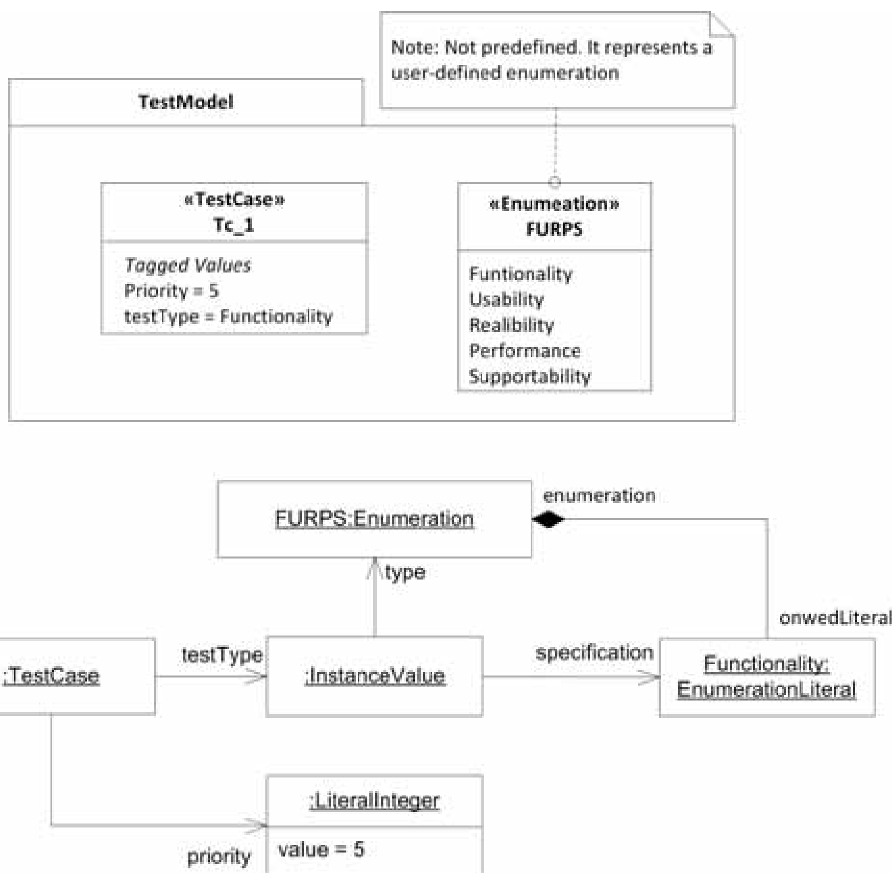


Figure 9.3 - Test type and priority example

In the upper part a package called *TestModel* is shown. It contains a test case called *TC_1* and a user-defined Enumeration called *FURPS* that represents the popular FURPS (Functionality, Usability, Reliability, Performance, and Supportability) quality model. The test type of *TC_1* refers to the *Functionality* literal of that Enumeration by using an InstanceValue (shown in the object model below). In addition, a LiteralInteger value has been used to specify the priority of the test case. The use of both the Enumeration and LiteralInteger is user-specific and might only make sense within the scope of this model. Other approaches may use LiteralString, OpaqueExpression, or even complex types instead of Enumeration and LiteralInteger.

The lower part of Figure 9.3 depicts the corresponding object model.

Examples of «TestCase» can be found in Figure D.5, Figure D.13, and Figure D.25.

9.2.2.5 ValidationAction

Description

Validation actions are used to set verdicts in test behaviors by calling the *setVerdict* operation from the arbiter interface.

Extensions

- CallOperationAction (from UML::Actions::BasicActions)

Generalizations

None.

Attributes

- reason : ValueSpecification [0..*]
  Any adequate information specifying the reason why a particular verdict has been assigned by the corresponding validation action.

Semantics

A validation action calls the *setVerdict* operation on the arbiter of the test context.

If a reason is given for a validation action, its concrete interpretation is not predefined by the UML Testing Profile. The simplest interpretation is a plain message stating why the particular verdict has been assigned, however, a tool vendor might decide to also incorporate complex values as well.

Constraints

[1] The operation of the action must be the *setVerdict* operation from the arbiter interface.

[2] The target of the action must refer to a classifier realizing the arbiter interface.

[3] The argument of the action must be an expression evaluating to a Verdict literal.

[4] Validation actions can only be used in test cases (i.e., a behavior where «TestCase» is applied to the behavior or its specification).

Notation

No additional notation for «ValidationAction» defined.

If the arbiter is depicted explicitly, a message may be shown describing the communication between the test component performing the validation action, and the arbiter.

The reason for a validation action is depicted as a Comment symbol that is attached to the validation action.

Examples

In Figure 9.4 two examples of «ValidationAction» are shown. The left most one contains the pass literal and will always set the verdict to pass. The right most one contains an expression, and will set the verdict to pass if x is greater than 10 and fail otherwise.

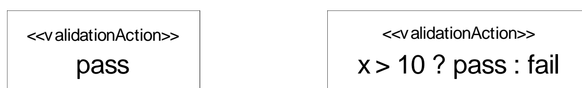


Figure 9.4 - Example validation actions

In Figure 9.5 an example is given where a validation action is complemented with a reason message.

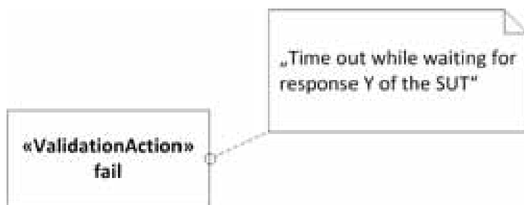


Figure 9.5- A validation action with a reason message

9.3 Defaults

Typically, a test case specification describes the normative or expected behavior for the SUT. To catch unexpected responses such as exceptions, default behavior is needed in order to keep the number of erroneous test case executions to a minimum. A default specification is a means to make partial behavior definitions of test components complete and robust against unexpected behavior of the SUT. Predefined default behavior provides certain convenience to the test developer.

In UTP, a default is activated by applying it to structural or behavioral structures. For instance, in a State Machine, it can be attached to a state machine, a state or a region; in an Activity, a default can be specified for an Action or the Activity itself; in an Interaction, defaults can be applied to its InteractionFragments. But it could also be applied to an entire test component, meaning that each instance of each component would apply this to any possible situation in a test case the test component is involved in.

9.3.1 Abstract Syntax

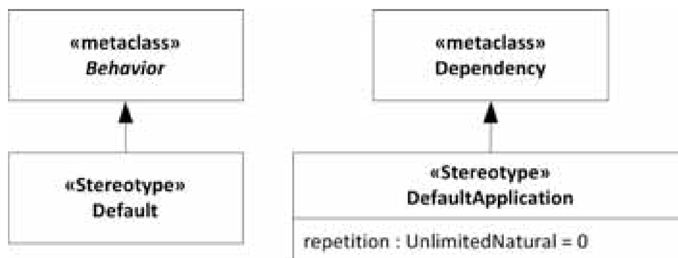


Figure 9.6 - Test Behavior Defaults

9.3.2 Stereotype Description

9.3.2.1 Default

Description

A system specification, expressed in UML, is not necessarily complete. To be complete in this sense means that it specifies every possible trace of execution. In particular if Interactions are used to specify the behavior, the normal situation is that the specification is partial only specifying in detail those scenarios that are of particular importance. In a testing context, however, there is a need to have complete definitions such that the number of erroneous test case executions can be kept to a minimum.

The default specifications are units of specification defined in the UML Testing Profile as a means to make partial definitions of test components complete in a compact, yet flexible way. The UML Testing Profile defines mechanisms for defaults on Interactions as well as State Machines.

The general idea about defaults is the following. A test behavior specification typically describes the normative or expected behaviors for the SUT. However, if during test execution an unexpected behavior is observed, then a default handler is applied. We have included default behavior definitions on several different levels. If the innermost default fail to recognize the observed behavior, the default of the next level is tried.

The reason for designing with defaults rather than making sure that the main description is complete, is to separate the most common and normal situations from the more esoteric and exceptional. The distinction between the main part and the default is up to the designer and the test strategies.

The UML Testing Profile has chosen to associate the default applications to static behavioral structures. In Interactions we may apply defaults to interaction fragments, in State Machines to StateMachines, States or Regions, and in Activities to Actions and Activities. Since each default in an Interaction applies only to one test component, we attach the defaults on interaction fragments to the intersection between the fragment and the test component.

We said above that default behavior is invoked when the main description cannot describe the observed behavior. More precisely the default mechanism is invoked when a trigger in a State Machine or message reception in an Interaction or an action in an Activity is not defined by the main description or an explicit runtime constraint is violated. The point of default invocation will be well-defined as an event occurrence in Interactions or a State (-stack) in State Machines or an accept event action in Activities.

Whenever a default behavior has executed, there is the question of where the main behavior should resume. There are several options, and the UML Testing Profile has chosen to distinguish between these different situations:

- The default execution is considered an interrupt, and the main behavior should resume exactly at the trigger that led to the default interruption. This situation is called *repeat*.
- The default execution is considered a limited disruption such that the resumption should be after the unit to which the executed default was attached. This situation is called *continue*.
- The default execution is considered to conclude the test case execution. The test component should not return to its main description for this test case. This situation can be considered a special case of the continue-situation provided that there exists an action called «Description», which ensures the completion of the test case execution locally for this test component.

We acknowledge that defaults are often described in the notation that the main description is made in. If the main specification is written with Interactions, the defaults will normally be Interactions. But it is possible with hybrid descriptions where, for example, some defaults to a main description written in Interactions are in fact State Machines.

For a more precise and detailed description of the semantics of defaults for Interactions and StateMachines the reader is referred to the sections on «Default» and «DefaultApplication».

Extensions

- Behavior (from UML::CommonBehaviors::BasicBehaviors)

Generalizations

None.

Attributes

None.

Semantics

We describe the semantics of defaults differently for Interaction, Activities, and State Machines since UML itself describes the semantics of these concepts in different terms.

For defaults that are described on Interactions, we define the semantics as an algorithm that combines the traces of the default behavior with the traces of the main description.

The combination algorithm is given here. Assume that there is a main description of an interaction fragment. Its semantics can be calculated to a set of traces. We project this set of traces onto the test component with the default by removing all event occurrences of other lifelines from the traces. The result is a set of traces only involving event occurrences of the test component. This is what we call the main description. Every trace in this set can be split in three portions: a head, a trigger, and a tail. The trigger is normally a receiving event occurrence. One particular trace can therefore be constructed in portions in several ways. The default is a behavior and is therefore also a set of traces, each of which can be divided in two portions: a trigger and a tail.

For every portioned trace in the main description, construct more traces, by concatenating main-head with every default trace provided that main-trigger is different from default-trigger. Retain the information on a trigger that it was originally a default-trigger by a flag. Finally filter out all traces starting with main-head+trigger-with-default-flag if there is another trace in the set starting with main-head+main-trigger and the main-trigger is equal to the trigger-with-default-flag. This makes up the set of traces for the interaction fragment with associated default.

These rules will ensure that main descriptions are considered before defaults, and inner defaults are considered before outer ones.[1]

The above rule applies when the default application repetition attribute is set to 0 (called continue). When the default application repetition attribute is greater than 0 (called repeat), the resulting set of traces is more elaborate since the default portions are repeated a number of times depending on the repetition count of the repetition attribute.

For defaults that are described on Activities, we consider the actions of the Activity together with the actions of the Default. The simple rule to combine the default with the main description is that the result is the union of all actions, but such that initial actions being part of the Default can only occur (and by doing so triggering the execution of that default) if there are no equal initial accept events in the Activity where the default is attached to.

For defaults that are described on State Machines, we consider the default as well as the State (or Region) applying the default to be transition matrices. The simple rule to combine the default with the main description is that the result is the union of all transitions, but such that transitions triggered by a default-trigger can only appear if there are no transitions from the same state with a main-trigger equal to that default-trigger. The start transition of the default is always removed from the resulting set of transitions.

---

1. When one package with an associated default imports another package with a default, the imported default is considered more outer than that of the importing package. Likewise if one class with a default is a specialization of another with a default, the default of the specialized class is considered more inner than that of the general class.

We may have hybrid defaults where a default may be applied within an Interaction, but defined as a State Machine. In such cases the default State Machine is considered equivalent to the set of traces that it can produce (or said differently, the strings that the automata may accept). We may have also hybrid defaults where a default may be applied within an Interaction or State Machine, but defined as an Activity. In such cases the default Activity is considered equivalent to the set of traces the Activity can produce (along the Petri-net like semantics of Activities).

In case a «Default» is applied to a package, the behavior of the default is applied to every test component being contained either directly or indirectly (as a descendant) in that package.

If there is no user-defined default that applies, what happens is a Semantic Variation Point.

The Semantic Variation Point may have the following interpretations or other interpretations decided by the tool vendors.

- Corresponding to UML 2.0 where the reaction to an unexpected trigger (in State Machines) is that the event is ignored if there is no matching trigger on a transition.
- The event can be deferred.
- The test component may perform a FinishAction (see "Description" on page 28).
- The activity may conclude.

Constraints

None.

Notation

No additional notation for «Default» defined.

Examples

Examples of «Default» can be found in Figure D.14, Figure D.15, and Figure D.16.

9.3.2.2 DefaultApplication

Description

A default application is a dependency used to apply a default behavior to a unit of testing on a test component.

Extensions

- Dependency (from UML::Kernel::Dependencies)

Generalizations

None.

Attributes

- repetition : UnlimitedNatural [1] = 0
  An integer value indicating how often the control flow will jump back to the unit of testing that has the default applied. Default value is 0 (i.e., to continue with the subsequent behavior).

Semantics

A default application relates a default to a unit of test behavior. A unit of test behavior can be one of Package, Classifier, Behavior, InteractionFragment, State or Region. The unit of testing must be related to a test component. Default behavior can be integrated into any situation where the test component expects a particular reaction of the SUT.

In case a default is applied to a Package, the default will be applied to each possible unit of testing of each test component either contained directly (child) or indirectly (descendant) in that package.

In case a default is applied to an InteractionFragment, the InteractionFragment must cover a lifeline representing a test component. Most likely a default is applied to a subclass of OccurrenceSpecification, representing the reception of a signal or operation call or a reply message (MessageOccurrenceSpecification) on a test component lifeline.

The *repetition* attribute indicates how often the default behavior will jump back to the unit of testing from where the default has been applied. ‘0’ indicates the default will not jump back to the unit of testing and continue.

Constraints

[1] The client of a default application must be one of Package, Classifier, Behavior, InteractionFragment, State, or Region.

[2] The multiplicity of client of a default application is restricted to [1].

[3] The supplier of a default application must be a Behavior with «Default» applied.

Notation

The notation for a «DefaultApplication» is identical to a Comment (i.e., a rectangle with a bent corner). The text in the comment symbol has the following syntax:

> **default** default-identifier [repetition]

If nothing is given following the default identifier, infinity is assumed.

For Interactions, the comment is attached to an intersection point between the interaction fragment and the lifeline of the test component.

For State Machines, the comment is attached to a state symbol. If the state is divided in regions, the attachment point designates on which region the default is associated.

Defaults may also be attached to class symbols.

Examples

Examples of «DefaultApplication» are found in Figure D.13 and Figure D.16.

9.4 Timer-related Concepts

Timer-related concepts are useful to specify how time impacts on the expected behavior of test cases, regarding e.g., the deadlines to meet during test cases execution, the time to spend sending/receiving messages or executing actions.

The UML Testing Profile enhances the declarative UML simple time concepts by adding an imperative timer concept using predefined types (such as the interface Timer and the primitive types Duration and Timepoint) and actions. The UTP predefined types Timepoint and Duration are especially meant to provide concrete types to the ValueSpecifications related to time (such as TimeExpression and Duration) given by UML. The Timer-related actions (e.g., starting or stopping a timer or checking whether a timer has been timed out) are meant to explicitly operate on these types and exploit concrete time values during the execution of test cases.

The imperative mechanism of timer handling enables the user to react on time constraint violations in a distinguished manner, instead of indicating that the entire behavioral description has failed, which is actually the semantics of UML simple time. Thus, when a timer has timed out, this must not necessarily result in a failed test case (in most cases it results in a failed test case, though). For example, if the logic of the test case expects that the SUT must not respond to a stimulus for a particular time slot, this could be handled by using the imperative timer concept.

9.4.1 Abstract Syntax

I

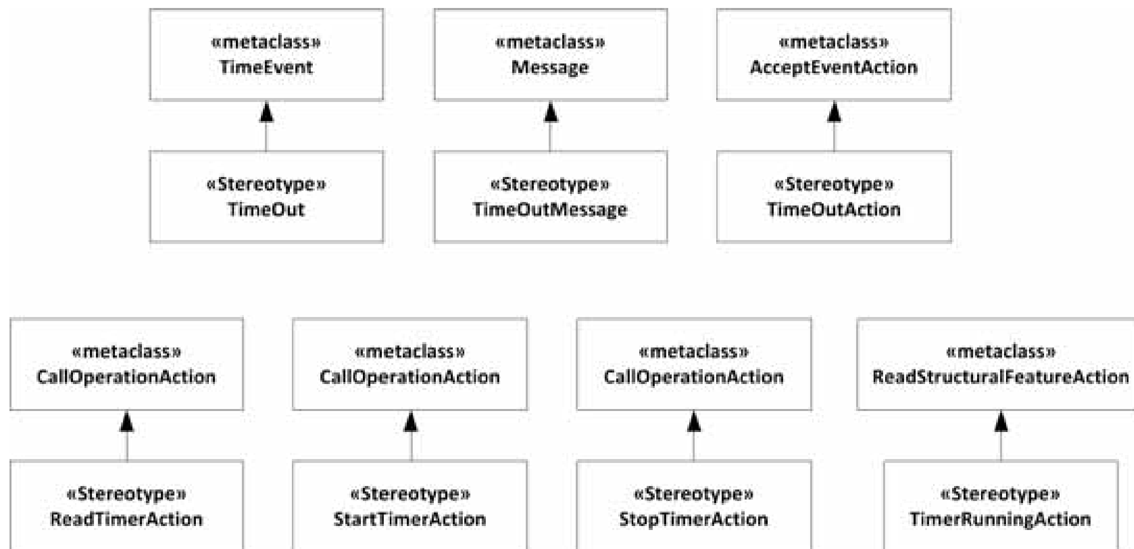


Figure 9.7 - Test Behavior Timer-related Concepts

9.4.2 Stereotype Description

9.4.2.1 ReadTimerAction

Description

An action used to read a timer to obtain the expiration time of a timer.

Extensions

- CallOperationAction (from UML::Actions::BasicActions)

Generalizations

None.

Attributes

None.

Semantics

The read timer action reads the expiration time of a timer. The read timer action returns null for timers that are not running.

Constraints

[1] The operation of the action must be the read operation from the Timer interface.

[2] The target of the action must refer to a classifier realizing the Timer interface.

[3] The type of the result must be Time.

Notation

No additional notation for «ReadTimerAction» defined.

Examples

None.

9.4.2.2 StartTimerAction

Description

An action used to start a timer.

Extensions

- CallOperationAction (from UML::Actions::BasicActions)

Generalizations

None.

Attributes

None.

Semantics

The start timer action starts a timer. The start timer action on a running timer restarts the timer.

Constraints

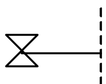
[1] The operation of the action must be the start operation of the Timer interface.

[2] The target of the action must refer to a classifier realizing the Timer interface.

[3] The argument of the action must be a Time value.

Notation

The notation for a «StartTimerAction» is an empty hourglass. A thin line connects the hourglass and the line where the timer is started.



Examples

A «StartTimerAction» example can be found in Figure D.11.

9.4.2.3 StopTimerAction

Description

An action used to stop a timer. Stops the timer if it is currently running, does nothing otherwise.

Extensions

- CallOperationAction (from UML::Actions::BasicActions)

Generalizations

None.

Attributes

None.

Semantics

The stop timer action stops a running timer. The stop timer action on a timer that is not running has no effect.

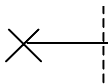
Constraints

[1] The operation of the action must be the stop operation of the Timer interface.

[2] The target of the action must refer to a classifier realizing the Timer interface.

Notation

The notation for a «StopTimerAction» is a cross. A thin line connects the cross and the line where the timer is stopped.



Examples

A «StopTimerAction» example can be found in Figure D.11.

9.4.2.4 TimeOut

Description

A timeout event represents the timeout mechanism for usage in StateMachines.

Extensions

- TimeEvent (from UML::CommonBehaviors::SimpleTime)

Generalizations

None.

Attributes

None.

Semantics

A timeout is generated by a timer when it expires and may trigger an associated behavior. The timeout event is placed in the input pool of the object owning the timer.

Constraints

None.

Notation

The notation for a «TimeOut» reuses the notation for TimeEvent.

Examples

None.

9.4.2.5 TimeOutMessage

Description

A timeout message represents the timeout mechanism for usage in Interactions.

Extensions

- Message (from UML::Interactions::BasicInteractions)

Generalizations

None.

Attributes

None.

Semantics

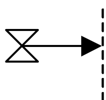
A timeout message is generated by a timer when it expires. The timeout message is sent to the active class that owns the timer.

Constraints

None.

Notation

The notation for the «TimeOutMessage» is an empty hourglass. An arrow with a filled head connects the hourglass and the line where the timeout occurs.



Examples

A «TimeOutMessage» example can be found in Figure D.11.

9.4.2.6 TimeOutAction

Description

A timeout action represents the timeout mechanism for usage in Activities.

Extensions

- AcceptEventAction (from UML::Actions::CompleteActions)

Generalizations

None.

Attributes

None.

Semantics

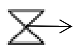
A timeout is enabled when the timer expires. It may trigger an associated activity. The timeout action occurs, when all input conditions for that activity are satisfied (including the timeout action).

Constraints

None.

Notation

The notation for the «TimeOutAction» is an empty hourglass (it reuses the syntax for the accept time event action in activities). An arrow with an unfilled head connects the hourglass and the activity to which the timeout is an input condition.



Examples

None.

9.4.2.7 TimerRunningAction

Description

An action used to check if a timer is currently running or not.

Extensions

- ReadStructuralFeatureAction (from UML::Actions::IntermediateActions)

Generalizations

None.

Attributes

None.

Semantics

The timer running action checks the running status of a timer. Returns a boolean value, true if the timer is running, false otherwise.

Constraints

[1] The structural feature of the action must be the isRunning attribute of the Timer interface.

[2] The type of the result must be Boolean.

[3] The target of the action must refer to a classifier realizing the Timer interface.

Notation

No additional notation for «TimerRunningAction» defined.

Examples

None.

9.5 Timezone Co-ordination

In case a distributed test environment is specified, test components may need co-ordination with regards to their actual timezones. This can be achieved during runtime with the actions that are described in this sub clause.

9.5.1 Abstract Syntax

Figure 9.8- Test Behavior Timezone Co-ordination

9.5.2 Stereotype Descriptions

9.5.2.1 GetTimezoneAction

Description

An action to dynamically retrieve the current timezone of a test component.

Extensions

- ReadStructuralFeatureAction (from UML::Actions::IntermediateActions)

Generalizations

None.

Attributes

None.

Semantics

The get timezone action can be invoked at run-time by a test component to retrieve its current time zone. The result is a Timezone value.

Constraints

[1] The type of the structural feature of the action must be Timezone.

[2] The type of the result must be Timezone.

Notation

No additional notation for «GetTimezoneAction» defined.

Examples

None.

9.5.2.2 SetTimezoneAction

Description

An action to dynamically set the timezone of a test component.

Extensions

- WriteStructuralFeatureAction (from UML::Actions::IntermediateActions)

Generalizations

None.

Attributes

None.

Semantics

The set timezone action can be invoked at run-time by a test component to set its current time zone. The value of a set timezone action refers to a Timezone value.

Constraints

[1] The type of the structural feature of the action must be Timezone.

[2] The value must be a Timezone value.

Notation

No additional notation for «SetTimezoneAction» defined.

Examples

None.

10 Test Data

10.1 General

Specifying test cases almost always requires specifying test data. Particularly in the context of information systems the state of the system to be tested is often represented by a quite substantial volume of data – the contents of its database. Similarly, stimuli and responses of information systems often carry medium-sized data of complex structure. This section describes the mechanisms provided by the UML Testing Profile to specify such test data in an efficient manner. Test data as part of model-based test specification serve different purposes:

- Specification of data values to be supplied with a stimulus, i.e., sent to the SUT within a test case, or to be retrieved as a response, i.e., data values returned by the SUT.
- Definition of both the initial state of the SUT needed to start a test case, i.e., the precondition of the test case, and expected state of the SUT after executing a test case, i.e., the postcondition of the test case.

Another application scenario for test data specification is test data generation. Even though the UML Testing Profile does not explicitly cope with generators or instructions for generators, it provides the essential information which can be used as input for an external test data generator to generate large set of instances for a particular data partition, for example.

10.2 Test Data Specification

Concepts provided in the test data specification sections aiming at the definition of structural aspects of test data. With structural aspects we refer to both the inner structure of complex data types and/or data pools, but also to constraints that are applied on that internal structure. Those internal structures of test data specifications might be a simplified view on the actual internal structure (e.g., the internal structure of a data pool need not match the actual database schema it specifies for the sake of simplicity). Therefore, the UML Testing Profile defines three test data specification concepts:

1. *DataPool* to specify physical containers of data.
2. *DataPartition* to specify subsets of concrete sets of instances.
3. *DataSelector* to allow user to specify which actual data values shall be retrieved; data values are typically stored in data pool.

A data partition represents a named set of instances (e.g., “VIP customers”) sharing some properties important to some test cases and is specified by means of an InstanceSpecification. In the testing domain, a data partition also is often known as equivalence class. Data partitions may be reused by other data partitions for refinement. The data partition “VIP customers living in New York” is a refinement of the data partition “VIP customers.” By means of their cardinality, data partitions may also be constrained on their size. For example, we may require exactly 200 VIP customers living in New York. Finally, data partition may also be associated with other data partitions. For example, each representative of the data partition “VIP customers” must be related to a representative of the data partition “open large orders.”

A data selector specifies the conditions that each instance belonging to a data partition must fulfill, for example “of type ‘customer’ and (VIP = true or revenue > $10'000).” Refining a data partition means extending the corresponding data selectors. For example, the data selector for the data partition “VIP customers living in New York” could be “of type ‘customer’ and (VIP = true or revenue > $10'000) and city = ‘New York.’”

Finally, a specification of a pool represents physical data value containers that may already contain instances of which members of data partitions may be drawn by invoking a certain data selector. As an example, the data pool "operational customer database" may be used to retrieve the members of the two data partitions "VIP customers" as well as "VIP customers living in New York."

10.2.1 Abstract Syntax

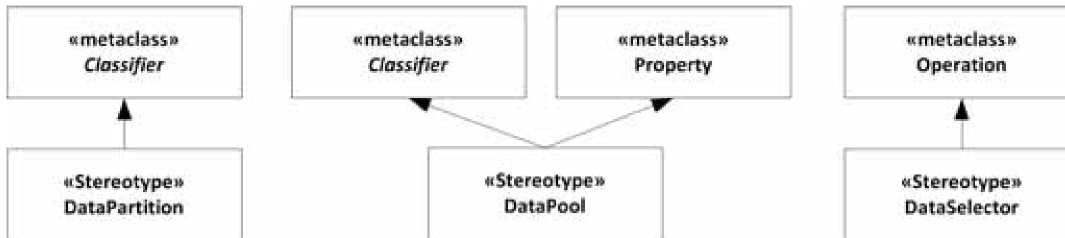


Figure 10.1 - Test Data Specification

10.2.2 Stereotype Descriptions

10.2.2.1 DataPartition

Description

A data partition specifies a container for a set of values. These data sets are used as abstract equivalence classes during test context and test case evaluation.

Extensions

- Classifier (from UML::Kernel, Dependencies, PowerTypes)

Generalizations

None.

Attributes

None.

Semantics

A data partition is used to define an equivalence class for a given type (e.g., "ValidUserNames," etc.). By denoting the partitioning of data explicitly we provide a more visible differentiation of data. A data partition is a stereotyped classifier that must be associated with a data pool.

Constraints

[1] A classifier with «DataPartition» applied can only be associated with a data pool or another data partition.

Notation

No additional notation for «DataPartition» defined.

Examples

A data partition example is given in Figure D.23.

10.2.2.2 DataPool

Description

A data pool specifies a container for explicit values or data partitions that are used by test contexts or test cases. A data pool provides an explicit means for associating data values for repeated tests (e.g., values from a database, etc.) and equivalence classes that can be used to define abstract data sets for test evaluation.

Extensions

- Classifier (from UML::Kernel, Dependencies, PowerTypes)
- Property (from UML::CompositeStructures::InternalStructures)

Generalizations

None.

Attributes

None.

Semantics

Test cases are often executed repeatedly with different data values to stimulate the SUT in various ways. Also when observing data, abstract equivalence classes are used to define sets of allowable values. Typically these values are taken from data partitions, or lists of explicit values. For this purpose a data pool provides a means for associating data sets with test contexts and test cases. A data pool is a classifier containing either data partitions (equivalence classes), or explicit values; and can only be associated with either a test context or test components.

Constraints

[1] A classifier with «DataPool» applied can only be referenced by a test context or test component.

[2] A property with «DataPool» applied can only be applied to a property associated connected with a test component within the context of a classifier with «TestContext» applied.

[3] A classifier with «DataPool» applied cannot be associated with both a test context and a test component.

Notation

No additional notation for «DataPool» applied.

Examples

A «DataPool» example is given in Figure D.23.

10.2.2.3 DataSelector

Description

A data selector allows the implementation of different data selection strategies.

Extensions

- Operation (from UML::Kernel, Interfaces)

Generalizations

None.

Attributes

None.

Semantics

To facilitate the different data selection strategies and data checking one or more data selectors can be associated with either a data pool or data partition. Typically, these data selectors are operations that operate over the contained values or value sets.

Constraints

[1] If «DataSelector» is applied to an operation, the featuring classifier must have either «DataPool» or «DataPartition» applied.

Notation

No additional notation for «DataSelector» defined.

Examples

A «DataSelector» example is given in Figure D.23.

10.3 Test Data Values

This sub clause collects concepts that are dedicated to the specification or mapping of values in a test specification. These concepts are logically divided into

- coding rules for test data values,
- wildcards for test data values, and
- reuse of test data values.

Coding rules indicate how the test data values, which are exchanged between the test environment and the system under test shall be realized, i.e., to what communication protocol they belong (e.g., HTTP, ASN.1, WSDL…).

Wildcards are typically used for a loose specification of concrete test data values to be expected from the SUT or provided to the SUT. The UML already provides extensive means to create ValueSpecifications; however, it is sometime helpful to not be forced to completely specify each possible test data value. Therefore, the UML Testing Profile introduces the concepts of wildcards for allowing any possible concrete data value as long as the data value is present, or explicitly allowing the absence of a data value in contrast. These wildcards are called *LiteralAny* and *LiteralAnyOrNull*. They complement the set of ValueSpecifications that are introduced by UML itself, such as LiteralString, LiteralNull, etc.

Reuse of already existing test data values to create new, maybe slightly different test data values, is an important requirement for large-scale test specification and complex test data value specifications. The UML does not provide a dedicated concept to reuse already existing InstanceSpecifications for modification. Therefore, the UML Testing Profile defines an additional concept, called *Modification*. A modification enables users to rely upon already existing, and maybe

incomplete, InstanceSpecifications in order to refine or modify them in new InstanceSpecifications. This allows the specification of modular InstanceSpecifications that are either refined (i.e., being made complete) or modified to be reused in different situations.

10.3.1 Abstract Syntax

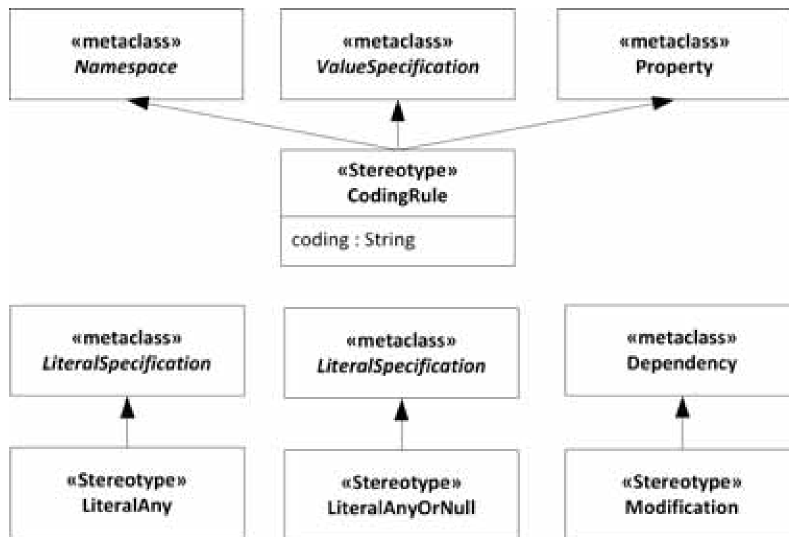


Figure 10.2 - Test Data Values

10.3.2 Stereotype Descriptions

10.3.2.1 CodingRule

Description

A coding rule specifies how values are encoded and/or decoded. Coding rules are defined outside the UML Testing Profile and referred to within the UML Testing Profile specifications. Referring to such coding rules allows the specification of both valid and invalid codings. A simple string is used to refer to a coding scheme(s); for example, "PER" (Packed Encoded Rules).

Extensions

- Property (from UML::CompositeStructures::InternalStructures)
- ValueSpecification (from UML::Kernel)
- Namespace (from UML::Kernel)

Generalizations

None.

Attributes

- coding : String [1]
  A string defining the selected coding scheme.

Semantics

If a coding rule is applied to a value specification, it specifies how the value(s) are coded. If it is applied to a namespace, it specifies the coding for all values contained within the namespace. If it is applied to a property, it specifies how the values of this property are encoded. If coding rules are applied to several places in a hierarchy, rules defined at a lower level have precedence over rules defined at a higher level.

Notation

The notation for a «CodingRule» is identical to a comment in UML (i.e., a rectangle with a bent corner). The text in the comment symbol begins with the keyword **coding** followed by the string defining the coding scheme. A dashed line is used to connect the comment symbol to the namespace, property, and value specification.

Examples

A «CodingRule» example is given in Figure D.10.

10.3.2.2 LiteralAny

Description

Literal any is a wildcard specification representing any value out of a set of possible values.

Extensions

- LiteralSpecification (from UML::Kernel)

Generalizations

None.

Attributes

None.

Semantics

If a literal any is used as a literal specification, it denotes any possible value out of a set of possible values. If it is used (e.g., for the reception of a message in an interaction), it specifies that the specified message with any value at the place of the literal any is to be received. If it is used (e.g., for the sending of a message in an interaction), it specifies that this message with a selected value at the place of the literal any is to be sent. The selection of this value can be done along different selection schemes such as default values or random values.

Constraints

None.

Notation

The notation for «LiteralAny» is the question mark character, ‘?’.

Examples

None.

10.3.2.3 LiteralAnyOrNull

Description

Literal any or null is a wildcard specification representing any value out of a set of possible values, or the lack of a value.

Extensions

- LiteralSpecification (from UML::Kernel)

Generalizations

None.

Attributes

None.

Semantics

If a literal any or null is used as a literal specification, it denotes any possible value out of a set of possible values or the lack of the value. If it is used (e.g., for the reception of a message in an interaction), it specifies that the specified message with any value at the place of the literal any or null or without that value is to be received. If it is used (e.g., for the sending of a message in an interaction), it specifies that this message with a selected value or without a value at the place of the literal any or null is to be sent. The selection of this value or the selection of the absence can be done along different selection schemes such as default values or random values.

Notation

The notation for «LiteralAnyOrNull» is the star character, '*'.

Examples

None.

10.3.2.4 Modification

Description

A modification is a specialized Dependency that allows InstanceSpecifications to be modified (and therefore reused) by other InstanceSpecifications.

Extensions

- Dependency (from UML::Kernel::Dependencies)

Generalizations

None.

Attributes

None.

Semantics

A modification 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, 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 (see subsections Semantics and Constraints (Constraint 6) of section 7.3.8 Classifier of UML 2.4.1 specification).

Cyclic modifications are not allowed. A cyclic modification describes a situation in which a modifying InstanceSpecification establishes a modification to a modified InstanceSpecification and the latter one already modifies (directly or transitively) the modifying InstanceSpecification.

Constraints

[1] Only InstanceSpecifications are allowed to be part of the underlying dependency's client elements.

```
context Modification

inv:
self.base_Dependency.client->forAll
(client : NamedElement | client.oclIsKindOf(InstanceSpecification))
```

[2] Only InstanceSpecifications are allowed to be part of the underlying dependency's supplier elements.

```
context Modification

inv:
self.base_Dependency.supplier->forAll(supplier : NamedElement |
supplier.oclIsKindOf(InstanceSpecification))
```

[3] Cyclic modifications are prohibited.

Notation

A modification resembles the notation of a Dependency in UML, but with the keyword «modifies» set as the dependency's name.

Figure 10.3 - Notation for Modification

Examples

As an example for the application of «Modification», an analogy of the well-known triangle example is used. Figure 10.4 depicts three classes that describe the different kind of triangles distinguished by the relative length of their sides. The base class *Triangle* represents a scalene triangle, i.e., all sides are unequal; the *IsoscaleTriangle* represents a triangle where two sides are equally long (in this scenario the sides denoted by the attributes a and b); finally, the *EquilateralTriangle* represents a triangle in which all sides are equally long.

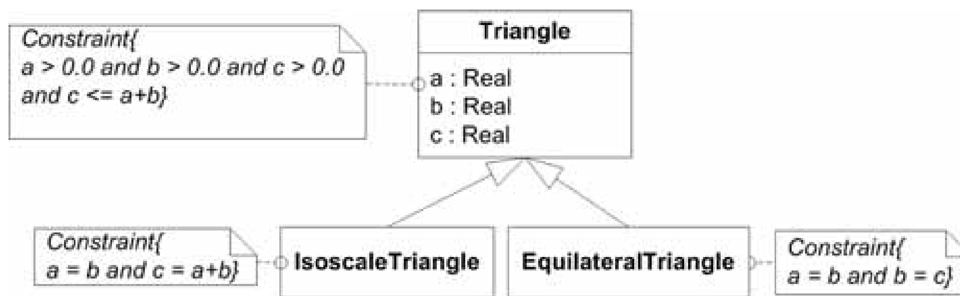


Figure 10.4 - Triangle classes

In order to create InstanceSpecifications for triangles a user may decide to specify a set of fundamental values for particular sides of the triangle, which can be reused for creating multiple but varying InstanceSpecifications for triangle in order to avoid redundancy. This results in the situation depicted in Figure 10.5.

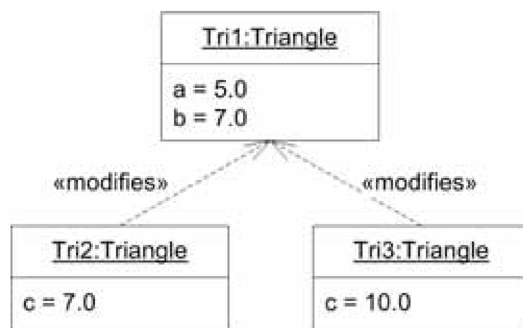


Figure 10.5 - InstanceSpecifications of Triangle classes using Modification

In Figure 10.5 the InstanceSpecification *Tri1* defines slots for the attributes a and b as well as values for those slots. From a tester's point of view, the InstanceSpecification *Tri1* is considered to be incomplete, since it does not define a slot or value for the required attribute c. Although InstanceSpecifications are by definition allowed to be incomplete (see section 7.3.22 InstanceSpecification of UML 2.4.1 specification), test execution systems usually require test data values that are used within test cases to be complete in the sense of fully specified. *Tri1* can be considered as the fundament for creating complete InstanceSpecifications by modifying *Tri1*, i.e., by establishing a «Modification» between Tri1 and the type

compliant InstanceSpecifications *Tri2* and *Tri3* as shown in Figure 10.5. In this example, *Tri1* represents the modified InstanceSpecification whereas *Tri2* and *Tri3* represent the modifying InstanceSpecifications. Slot overwriting is not shown in this example. From a logical point of view, *Tri2* and *Tri3* are semantically identical to the InstanceSpecifications mentioned in Figure 10.6.

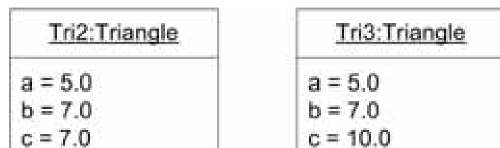


Figure 10.6 - Semantically equivalent InstanceSpecifications of Figure 10.5

The resulting InstanceSpecifications *Tri2* and *Tri3* can be further modified to define InstanceSpecifications for isoscale triangles and equilateral triangles (see Figure 10.7).

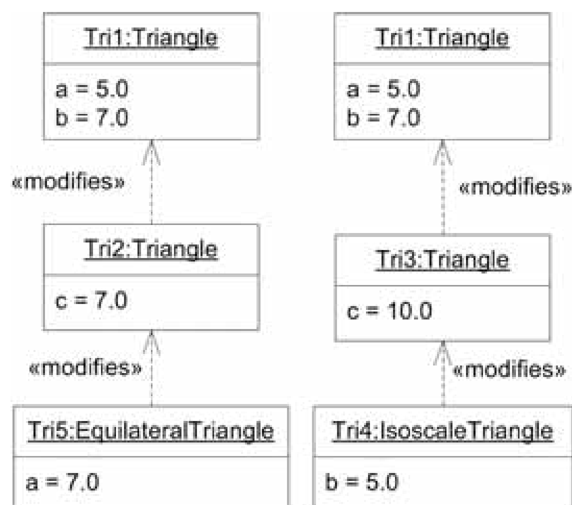


Figure 10.7 - Overwriting slots using Modification

The effective value (i.e., all values taken over from modified InstanceSpecifications) of slot a of *Tri2* is transitively overwritten with the value of slot *a* of *Tri5*. Transitively means that slot a is not physically owned by *Tri2*, but by *Tri1*. The same approach has been applied to the InstanceSpecification *Tri4*. Logically, the gradually refined and completed InstanceSpecifications *Tri4* and *Tri5* are equivalent to the InstanceSpecifications shown in Figure 10.7.

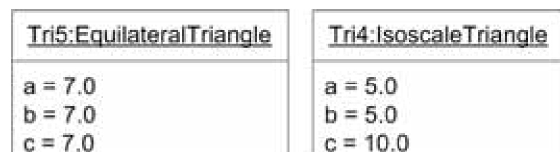


Figure 10.8 - Semantically equivalent InstanceSpecifications Figure 10.7

Figure 10.9 shows two violations of «Modification». On the left hand side a modification was established between incompatible InstanceSpecifications, i.e., AnotherTriangle is not type compliant to Triangle, because AnotherTriangle is not a subtype of Triangle. As a result, such a modification causes the underlying model to be ill-formed. On the right hand side a cyclic modification was defined, what is prohibited by definition.

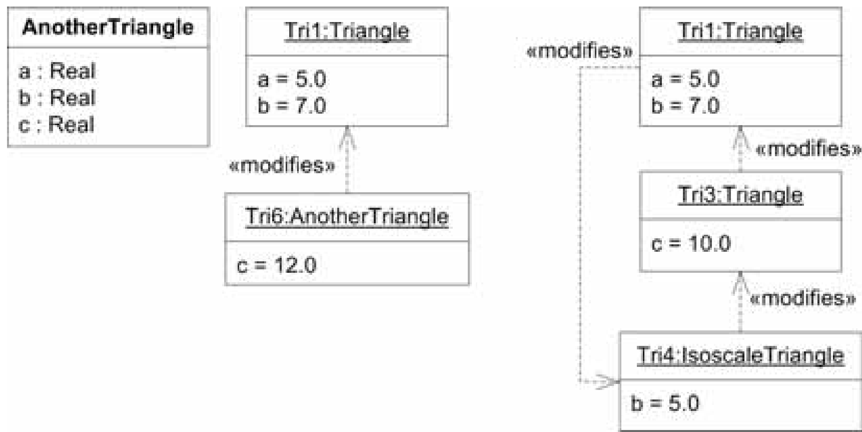


Figure 10.9 - Erroneous use of Modification

11 Test Management

11.1 General

This clause presents modeling concepts and elements that can be used to model aspects of test management, within the narrow scope of managing individual test activities and tests (i.e., not the full project/test lifecycle).

Test management is needed since it is impossible to test a software system exhaustively, thus testing becomes a sampling activity, which must be managed within cost, schedule, qualities, resources (human and facilities), and risk aspects. A variety of testing concepts (e.g., test approaches, techniques and types) exist to aid in choosing an appropriate sample set to test. Such concepts are defined in other standards (e.g., IEEE 1012 and ISO 29119). Users of the UML Testing Profile standard should refer to and be familiar with other test standards as all their concepts are not defined here. A key premise of this standard is the use of models and the UML Testing Profile to assist in the test planning, implement, control, support, and document. These are addressed here and Annex A presents additional (non-normative) test management concepts.

There are three general test management activities:

1. Test Planning and Scheduling
2. Test Monitoring and Control (including test execution)
3. Test Results Analysis

11.2 Test Planning and Scheduling

Test Planning is used to develop the test plan. Depending on where in the project this activity is implemented, it may produce a Project Test Plan (top level) or a test plan for a specific phase, such as a System Test Plan, or a sub test plan for a specific type of testing, such as a Performance Test Plan. Modeling can support aspects of such documents.

The UML Testing Profile contributes a dedicated concept called test objective to the test planning and scheduling phase. A test objective represents an early, mostly informal (textual) specification of later to be realized test cases. The benefit of test objectives is two-fold:

1. Testing activities can start at a very early point in time, even before a system specification or single line of code has been produced by the development team.
2. Traceability between requirements and the testing artifacts can be established at that early point in time, which is used subsequently to establish a coherent requirements traceability network.

Furthermore, the UML Testing Profile provides means that allow prioritization of test objectives that can be further leveraged for optimizing test schedules (i.e., the order in which test cases should be actually executed) of a test project or a single test phase.

11.2.1 Abstract Syntax

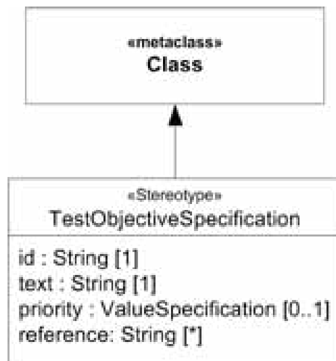


Figure 11.1 - Test Planning and Scheduling

11.2.2 Stereotype Description

11.2.2.1 TestObjectiveSpecification

Description

A test objective is represented by the stereotype «TestObjectiveSpecification» and specifies the reason or purpose for designing and executing tests. Test objectives must be measurable, thus, they must include sufficiently precise information how to assess, respectively evaluate the behavior or reaction of the system under test.

Extensions

- Class (from UML::Kernel)

Generalizations

None.

Attributes

- id : String [1]
  The identifier of a test objective.
- specification : String [1]
  A textual specification of the objective, test cases or test contexts must seek to fulfill.
- priority : ValueSpecification [0..1]
  Assigns a particular, process-/product-/ or domain-specific priority value to the test objective. This allows test objectives to be ordered and optimized for later test case design and/or their execution.
- reference : String [0..*]
  Textual references to additional material that is considered to be relevant for the test objective. Such a reference may point to elements outside the scope of the model, for example to a section of a normative document.

Semantics

A test objective is a textual specification of a well-defined target of testing, focusing on a single requirement or a set of related requirements as specified in the specification of the system under test (SUT).

A test objective merely describes what (logical) needs to be tested or how the system under test is expected to react to particular stimuli. It does not prescribe at all how (technical) that objective will be achieved by the realization of subsequently realized test cases.

Test objectives are usually realized or implemented by test cases. In addition, test objectives can establish traces to other artifacts that are addressed by a test objective, i.e., target of testing. Such targets might be expressed as use cases or concepts from other profiles, e.g., requirements from SysML. This allows users to establish trace links between a test objective and a target by using UML's native dependencies concepts, for example. If the target of testing is not formalized or not available for the test specification, they can be linked with the test objective nevertheless by the means of the test objective's generic references list.

A test objective can be prioritized. Prioritization enables test objectives to be ordered (e.g., by severity, urgency …) and optimized regarding project-, process-, and /or domain-specific needs. The priority of a test objective is typed by a ValueSpecification, hence, adequate values for a specific priority have to be provided by the model where the test objective is include in. The UML Testing Profile does not predefine any priority schema, but rather copes with the existing variance of potentially applicable priority schemas (e.g., qualified or quantified priority schemes) and terminology (e.g., low, medium, high or …) of different domains, methodologies, and/or processes.

Constraints

[1] «TestObjectiveSpecification» can only be applied to instances of Class. Further subclasses of Class (such as Component etc.) are not allowed to be stereotyped as «TestObjectiveSpecification».

```
context TestObjective

inv:
self.base_Class.oclIsTypeOf(Class)
```

Notation

No additional notation for «TestObjectiveSpecification» defined.

Examples

The following examples simply outline different scenarios how the test objective concept can be used. In Figure 11.2 two test objectives have been defined and linked with two test cases. Commonly, the definition of test objectives and their realization by corresponding test cases will be done with a certain delay in time. Test objectives often represent the earliest point in time, where testing activities may take place, whereas the specification of test cases is considered to be part of later testing process phases. However, we do not want to emphasize a certain process at this point.

In the figure, several different kinds of dependencies are used in order to connect test cases with test objective. Concretely, it is an Abstraction with «trace» applied on the left hand side and a Realization on the right hand side. By doing so, we want to deliberately illustrate that there are multiple possibilities to bring test objectives and test cases or test contexts together.

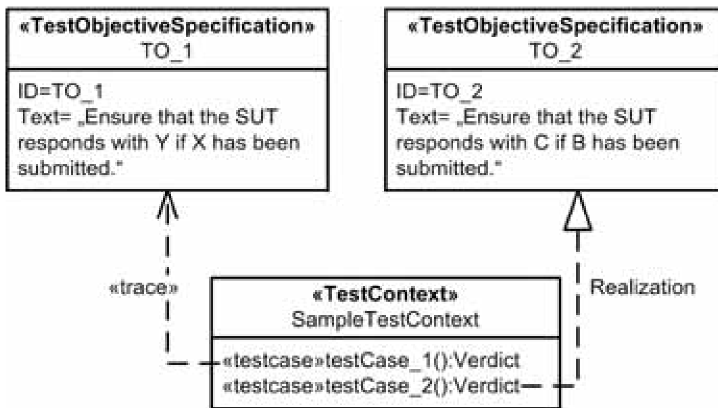


Figure 11.2- Linking between test cases and test objectives

Figure 11.3 extends the examples with regard to the target of testing. The test objective on the left hand side uses a UML «trace» dependency to establish a link between a test objective and its target of testing, in this case a use case. The right hand side represents a purposeful combination of the UML Testing Profile and SysML. SysML provides the user with dedicated concepts to deal with requirements, thus, users may use those concepts in combination with the UML Testing Profile to conduct requirements-driven testing, for example.

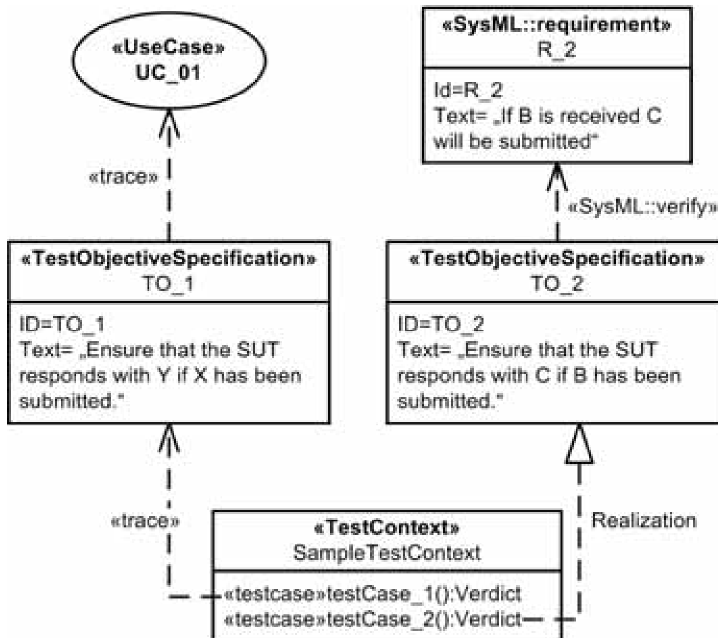


Figure 11.3- Establishing traceability between test objectives and further elements

In this example, the SysML concepts «requirement» and «verify» are used to establish requirements traceability within the test model. The traceability network on the right hand side can be interpreted as follows: Requirement 'R_2' is considered to be verified by the test objective with *id* 'TO_2.' The test objective 'TO_2' in turn is realized by test case with name 'testCase_2,' so transitively 'testCase_2' is considered to verify the requirement 'R_2.'

The example shown above is just one possibility how to establish traces from test cases to requirements. It does not claim to be the only valid way to have SysML and the UML Testing Profile interacting with each other. The following example shows a different possibility of how to integrate SysML concepts in a UML Testing Profile-based test specification.

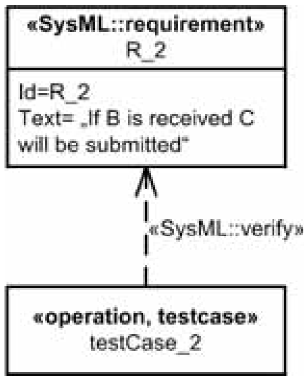


Figure 11.4- Direct verification of a requirement by test case

Figure 11.4 shows another, quite similar example to Figure 11.3. The main and significant difference in this scenario is that the test case 'testCase_2' (for illustration reasons depicted as a single element) directly verifies the requirement with id 'R_2.'

Both a link trace to an Operation of a Classifier within a compartment (see Figure 11.2 and Figure 11.3) and the standalone representation of an Operation as shown in Figure 11.4 is not defined by UML.

The combination of SysML and UML Testing Profile does not mean that the UML Testing Profile redefines the concepts already defined by SysML. In the examples shown above, the SysML concepts are rather reused in the test model. To do so, an implementation of the SysML profile must be available. It is not the intention of the authors to tightly couple the UML Testing Profile to SysML, thus, the usage of SysML concepts to establish requirements traceability in a test model is not prescribed nor mandatory. It is rather a recommendation to rely on already established and proven concepts, in case a SysML profile implementation is available. More information on the SysML requirements capabilities can be found in the SysML specification [SysML].

11.3 Test Monitoring and Control

Test monitoring and control ensures that the testing is performed in line with the test plan, schedule, and/or the organizational test factors, e.g., regulations, policy, etc. This activity can be applied to managing a whole test project (normally made up of a number of test phases and test types) and/or a single test phase or test. The purpose of the Test monitoring and control is to ensure that the testing is performed (design and execution until the test result are obtained) as defined with the test plans and/or models.

Use of modeling in monitor and control can include, but is not limited to:

1. Definition of test behavior as defined in Clause 9, 'Test Behavior'
2. Integration of the outcome of risk analysis (as outline in Annex C.5)
3. Observation and evaluation of test results of already executed test cases

Even though there are no explicit concepts defined in the UML Testing Profile to support the activities in test monitoring and control, this sub clause has been added to provide a coherent view on the test management discipline. However, the concepts of other conceptual parts of the UML Testing Profile can be used to leverage the process of test monitoring and control, as stated above.

11.4 Test Result Analysis

The analysis of test results is the final stages of test management. The UML Testing Profile defines means to capture test results from the execution of test cases. These are called test logs in the UML Testing Profile. When the testing activities for a particular test phase are completed, the test results will usually be fed into test project completion determination, but this is out of scope of the UML Testing Profile.

11.4.1 Abstract Syntax

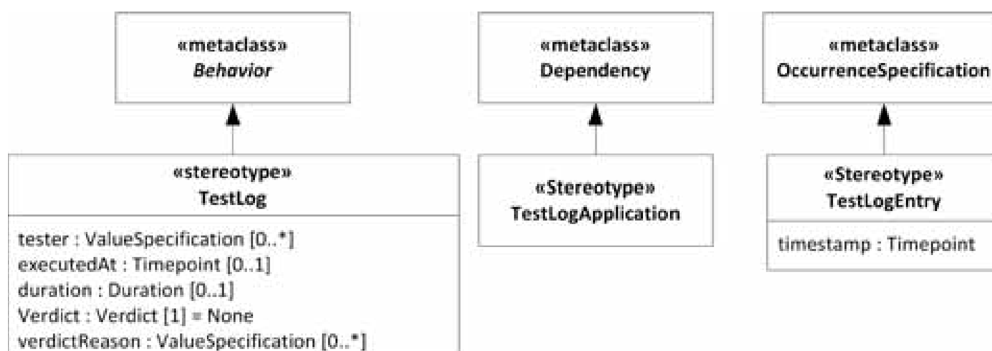


Figure 11.5- Test Result Analysis

11.4.2 Stereotype Description

11.4.2.1 TestLog

Description

A test log represents the behavior resulting from the execution of a test case or a test context. Also, it helps to understand potential actions and validations performed by the test context behavior that might impact the test case verdict. A test case or a test context may have any number of test logs.

Extensions

- Behavior (from UML::CommonBehaviors::BasicBehaviors)

Generalizations

None.

Attributes

- tester : ValueSpecification [0..*]
  The activator responsible for producing the test log.

- executedAt : Timepoint [0..1]
  The point in time when the test case execution was started.
- duration: Duration [0..1]
  The overall execution duration of the test case.
- verdict : Verdict [1] = None
  The overall verdict the test case execution has concluded with. The default verdict created by a test case is *None*.
- verdictReason : ValueSpecification [0..*]
  Any information that might be helpful to communicate why the verdict has been produced.

Semantics

A test log is a fixed (immutable) behavioral description resulting from the execution of a test case. It represents any event of interest being executed during the test case execution. A test log is associated with a verdict representing the adherence of the SUT to the test objective of the associated test case.

Each test log is either related to a test case or a test context (via test log application) representing the logged information of the execution of these elements.

It is neither defined what information will be logged, nor the granularity of logged events. An execution tool vendor may decide to just log validation actions and log actions, while another execution tool may also incorporate the creation and/or termination of test components into the log. The granularity of the information that shall be logged is not predefined.

There are several possibilities to specify a *tester* for a test log (i.e., the person(s) or tools that actually executed the corresponding) test case like an InstanceSpecification of an Actor or a simple String identifying a person, test execution system, etc. The UML Testing Profile does not prescribe how testers have to be identified.

The point in time when the test case has started its execution can be stored in the *executedAt* tag definition. The format for the point in time adheres to the semantics of Timepoint.

The duration of a test case execution can be stored in the *duration* tag definition. The format for the test case's execution duration adheres to the semantics of Duration.

The *verdict* of a test log is a mandatory information, since each test case has to produce a final verdict the test case concluded with. The default verdict of a test case is *None*. This is in synch with the semantics of Verdict itself.

If a validation action (or multiple) defines informative reasons why a final verdict has been assigned, those reasons can be summarized in the *verdictReason* tag definition. Commonly, if there is not a reason defined in a test case, the *verdictReason* will remain empty. There is no algorithm predefined how various reasons of validation actions are gathered along the test case execution and ultimately presented to the user.

Constraints

None.

Notation

No additional notation for «TestLog» defined.

Examples

An example of a test log is shown in Figure 11.7.

11.4.2.2 TestLogApplication

Description

A dependency to a test case or a test context.

Extensions

- Dependency (from UML::Kernel::Dependencies)

Generalizations

None.

Attributes

None.

Semantics

A test log application binds a concrete test log to the element, whose execution is described by the bounded test log. The number of targets for a test log is restricted to one, meaning a test log refers at any point in time to either a test case or test context. The multiplicity of the client and supplier properties of the underlying Dependency is restricted to 1.

Constraints

[1] The client of a test log application must be a named element with «TestLog» applied.

[2] The supplier of a test log application must be a named element with «TestCase» or «TestContext» applied.

Notation

The notation for a «TestLogApplication» is identical to a comment (i.e., a rectangle with a bent corner) with the keyword **testlog**.

Examples

See the example in Figure 11.6. *ATMSuite_log* is a test log of the test context *ATMSuite*. *invalidPIN_log* is a test log of the test case *invalidPIN*.

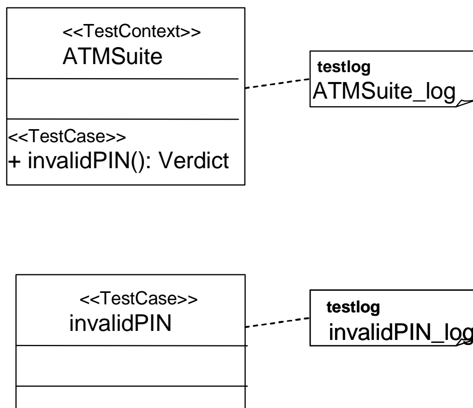


Figure 11.6- Example for test Logs

11.4.2.3 TestLogEntry

Description

A TestLogEntry represents the execution of a single test step. It therefore provides means to analyze test logs more precisely and to compare the executed test steps (i.e., test log entries in a test log) with the specified test steps (i.e., the test steps in a test case).

Extensions

- OccurrenceSpecification (from UML::BasicInteractions)

Generalizations

None.

Attributes

- timestamp : Timepoint [1]
  The timestamp represents a point in time when the corresponding test step of this test log entry has been executed by a test execution system during the execution of a test case.

Semantics

A test log entry represents a single test step in a test case that has been executed and therefore logged by the test execution systems during the execution of a test case. Test log entries have to be contained in «TestLog» Behavior solely.

The attribute *timestamp* specifies at which point in time a corresponding test step has been executed by the test execution system. Neither the format of that timestamp, nor how it is presented to the user is prescribed. Anyway, since timestamp is of type Timepoint, values for the tag definition must adhere to the semantics described in Timepoint.

«TestLogEntry» provide cross-cutting information all executed test steps have in common, without adding new semantics to the test steps itself. The concrete semantics of the logged test step depends on the actual semantics of the element «TestLogEntry» is applied to. For example, «TestLogEntry» applied to a receiving MessageOccurrenceSpecification that covers a test component lifeline represents the logging of a message being received by the test execution system.

Constraints

[1] «TestLogEntry» can only be applied to instances of OccurrenceSpecifications that are contained (maybe transitively) in «TestLog» Behavior.

[2] «TestLogEntry» can only be applied to instances of OccurrenceSpecifications that are recognizable by the test environment during the execution of a test case. Instances of OccurrenceSpecifications that cover a lifeline that represents a system under test (i.e., a part of the test context that has «SUT» applied) are not all allowed to have «TestLogEntry» applied.

Notation

The timestamp of a «TestLogEntry» shall be displayed at the left hand side of the sequence diagram, close to the diagram frame. The timestamp shall be at the same vertical position as the corresponding OccurrenceSpecification it belongs to. See Figure 11.7 in Examples below.

Examples

Figure 11.7 shows an example how test log entries might be presented to the user. Five different formats for a test log entries' timestamp are given at the left hand side of the «TestLog» sequence diagram. In order of execution time, the test log entries represent a logging of

1. A message that has been sent out to the system under test at timestamp *2012-06-12.11:34:55*.
2. A reply message that has been received by the test component at timestamp *1339500898*.
3. An asynchronous message that has been sent out to the system under test at timestamp *1339500899*.
4. The reception of an asynchronous message that has been received by the test component at timestamp *12. June 2012 11:35:04.615*.
5. The execution of a validation action at timestamp *00:00:11.217*. This timestamp represents an absolute point in time of the current duration of the test case execution. This test log entry has been logged after 11 seconds and 217 milliseconds during the execution of the test case.

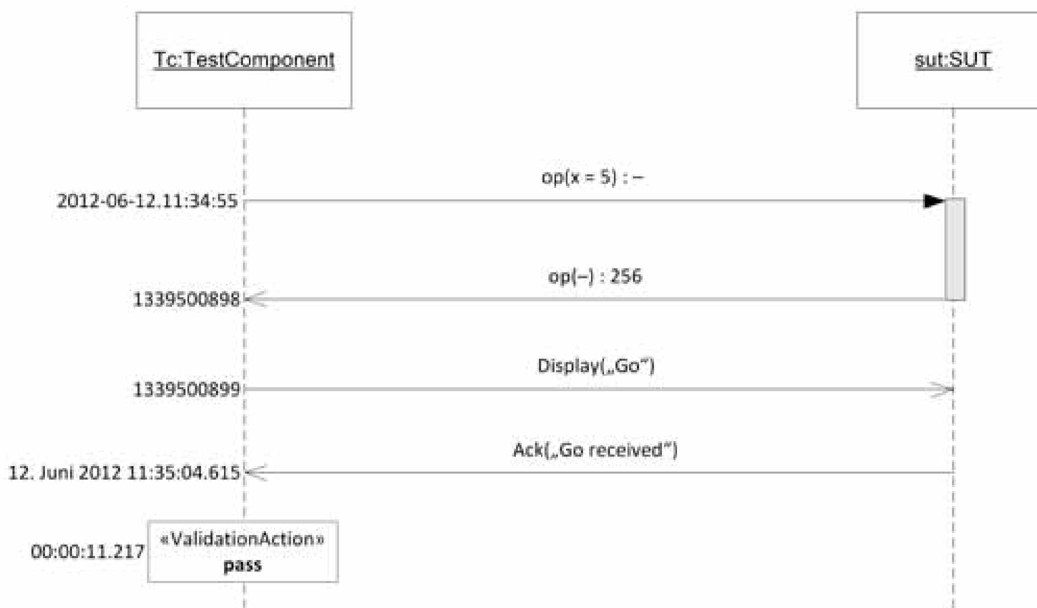


Figure 11.7- Example for test Log entries

This demonstrates effectively that the UML Testing Profile does not prescribe how the presentation of the timestamp value of a «TestLogEntry» has to look. It is left open to the vendor how to present the timestamp to the user.

Figure 11.8 and Figure 11.9 represents a small excerpt of the «TestLog» shown in Figure 11.7. The main focus is set to the object model (Figure 11.9) that precisely describes how «TestLogEntry» is intended to be used by implementations.

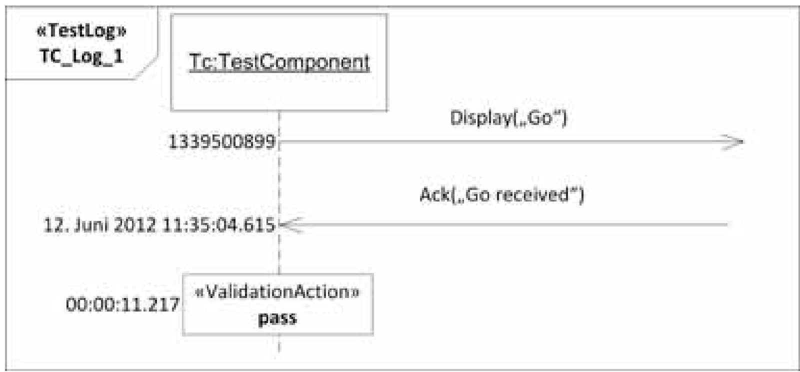


Figure 11.8- Simplified «TestLog» example

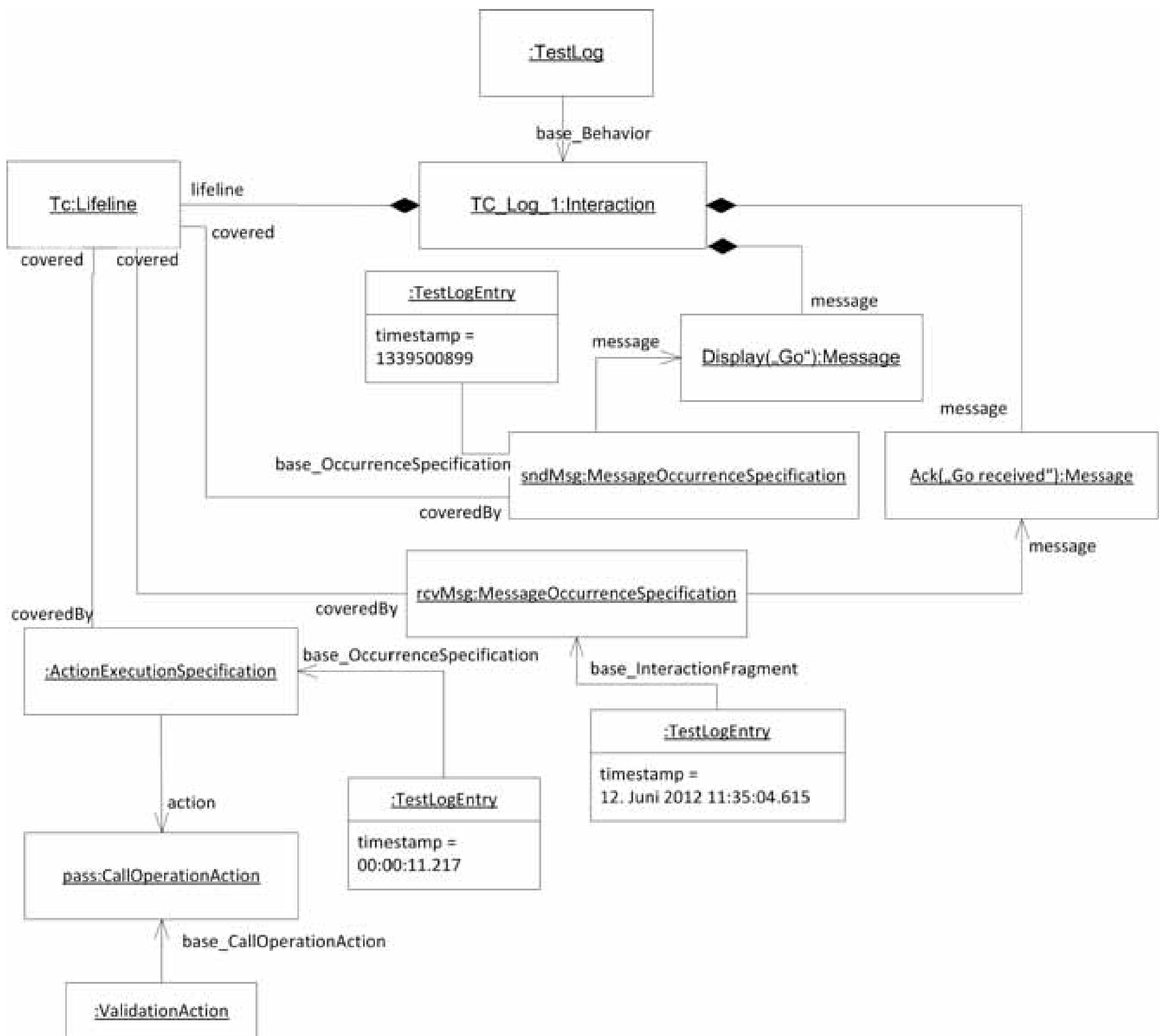


Figure 11.9- Corresponding object model of Figure 11.8

Annex A - Deprecated Elements

(normative)

A.1 TestObjective (extends Dependency)

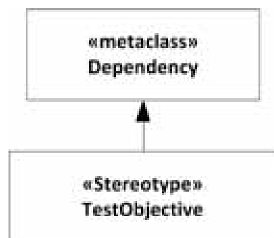


Figure A.1 - Abstract Syntax TestObjective

A.1.1 TestObjective

Description

A dependency used to specify the objectives of a test case or test context. A test case or test context can have any number of objectives and an objective can be realized by any number of test cases or test contexts.

Extensions

- Dependency (from UML::Kernel::Dependencies)

Generalizations

None.

Attributes

None.

Semantics

A test objective is a reason or purpose for designing and execution of a test case [ISTQB]. The underlying Dependency points from a test case or test context to anything that may represent such a reason or purpose. This includes (but is not restricted to) use cases, comments, or even elements from different profiles, such as requirements from [SysML].

Constraints

[1] The client of a test objective must be a named element with «TestCase» or «TestContext» applied.

Notation

A «TestObjective» is shown using a dashed line with a stick arrowhead pointing from a test case or test context to the element(s) that represent the objective. The keyword «objective» is shown near the dashed line.

Examples

The test objective in Figure A.2 specifies that the objective of the ValidWithdrawal test case is WithdrawMoney use case.

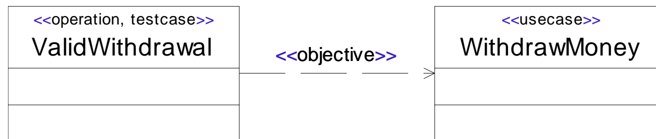


Figure A.2 - Test objective

A.1.2 Transformation Rule Sets for UTP 1.2 (informative)

A.1.2.1 Transformation Rule Set 1

This transformation rule set can be applied for every kind of «TestObjective» Dependency, no matter what source or target elements it is linked to.

1. For each «TestObjective» Dependency, create a test objective specification, i.e., a Class with «TestObjectiveSpecification» applied.

2. Add a predefined string value to TestObjectiveSpecification::text.

3. Link each source element of the «TestObjective» Dependency via an arbitrary Dependency to the newly created «TestObjectiveSpecification» Class. The «TestObjectiveSpecification» Class constitutes the target element for that Dependency.

   The kind of Dependency is not predefined, thus, any appropriate Dependency can be selected. We recommend using either Realization or «trace» Abstraction.

4. Link each target element of the «TestObjective» Dependency via an arbitrary Dependency to the newly created «TestObjectiveSpecification» Class. The «TestObjectiveSpecification» Class constitutes the source element for that Dependency.

   The kind of Dependency is not predefined, thus, any appropriate Dependency can be selected. We recommend using «trace» Abstraction. In case the SysML requirements capabilities are used and the «TestObjective» Dependency led to a «requirement» Class, we recommend using a «verify» Abstraction that emanates from the «TestObjectiveSpecification» Class and leads to the «requirement» Class.

A.1.2.2 Transformation Rule Set 2

This transformation rule set can be applied for «TestObjective» Dependencies that lead directly to a SysML «requirement» Class. Thus, this transformation rule set requires an implementation of the SysML profile being available.

1. Replace the «TestObjective» Dependency with a SysML «verify» Abstraction.

Annex B - Test Management Concepts

(non-normative)

B.1 General

This annex provides optional extensions to the UML Testing Profile to cope with test management requirements. The extensions for the stereotypes of the normative profile are optional. Only the concepts additional to the already defined concepts (in Clause 7) are listed in this Annex.

B.2 Abstract Syntax

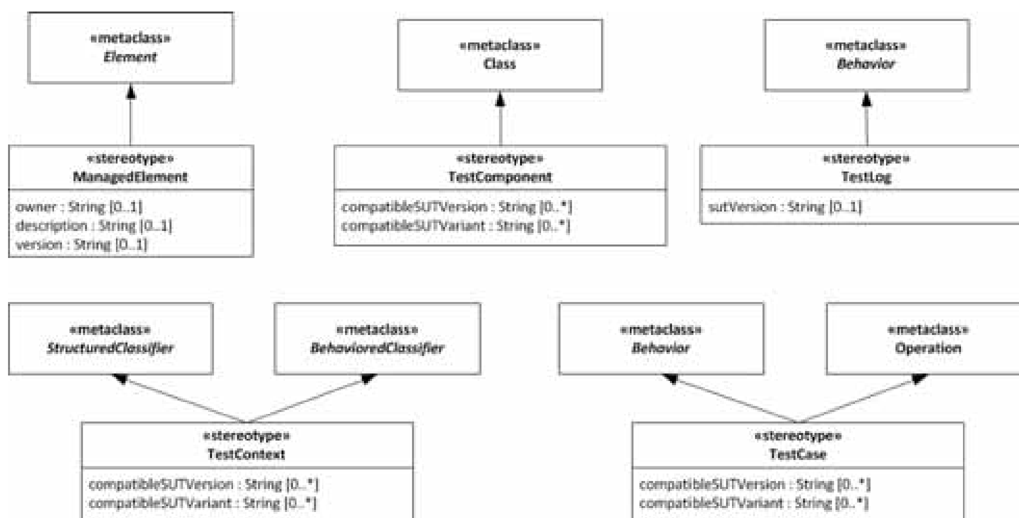


Figure B.1- Additional test management attributes

B.3 Stereotype Descriptions

B.3.1 ManagedElement

Description

A managed element provides additional project-, process- or any other meta-information to the element it is applied to.

Extensions

- Element (from UML::Kernel)

Generalizations

None.

Attributes

- owner : String [0..1]
  Identifies the owner which is responsible for this managed element.
- description : String [0..1]
  Provides the capability to describe the managed element in a greater detail.
- version : String [0..1]
  Information on the version of a managed element.

Semantics

Managed elements are often used in combination with other stereotypes, especially of the UML Testing Profile, to indicate the version or the owner of a particular test context, for example.

Constraints

None.

Notation

No additional notation for «ManagedElement» defined.

Examples

None.

B.3.2 TestComponent

Additional Attributes

- compatibleSUTVersion : String [0..*]
  Information on which SUT versions the test component applies to / is compatible with.
- compatibleSUTVariant : String [0..*]
  Information on which SUT variants the test component applies to / is compatible with. Used for test management regarding product variants.

B.3.3 TestContext

Additional Attributes

- compatibleSUTVersion : String [0..*]
  Information on which SUT versions the test context applies to / is compatible with.
- compatibleSUTVariant : String [0..*]
  Information on which SUT variants the test context applies to / is compatible with. Used for test management regarding product variants.

B.3.4 TestCase

Additional Attributes

- compatibleSUTVersion : String [0..*]
  Information on which SUT versions the test case applies to / is compatible with.

- compatibleSUTVariant : String [0..*]
  Information on which SUT variants the test case applies to / is compatible with. Used for test management regarding product variants.

B.3.5 TestLog

Additional Attributes

- sutVersion : String [0..1]
  Information against what version of the SUT the test case that produced this test log was executed at.

Annex C - Mappings

(normative)

C.1 Mapping to JUnit

JUnit is an open source regression testing framework written by Erich Gamma and Kent Beck (see www.junit.org). It has been made popular by the eXtreme Programming community and is widely used by developers who implement unit tests in Java. Over the past few years, it has become the de-facto standard for unit testing. JUnit has been translated to a variety of programming languages. Furthermore JUnit has been extended in various ways to support data driven testing, stubbing, etc.

This sub clause provides a mapping from the UML Testing Profile to JUnit. This mapping considers primarily the JUnit framework: When no trivial mapping exists to the JUnit framework, existing extensions to the framework are mentioned as examples of how the framework has been extended to support some of the concepts included in the UML Testing Profile.

The following diagram gives an overview of the JUnit framework:

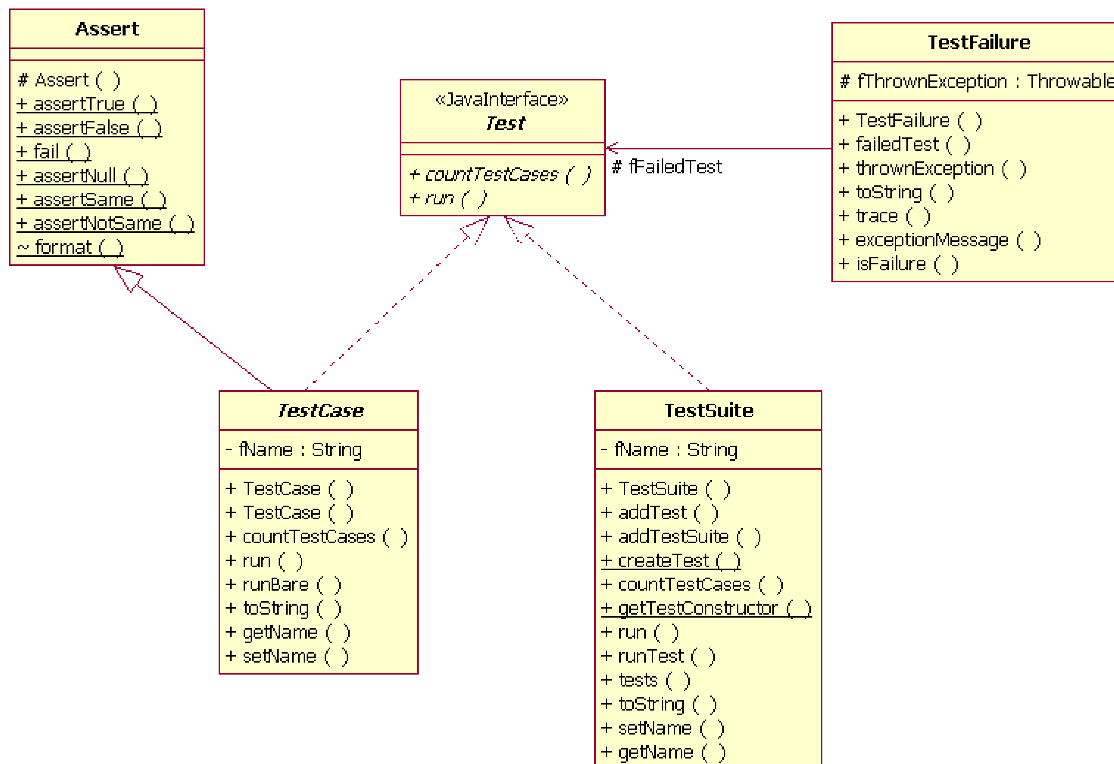


Figure C.1 - JUnit framework overview

When you create a test with JUnit, you generally go through the following steps:

1. Create a subclass of TestCase: The TestCase fixture.
2. Create a constructor that accepts a String as a parameter and passes it to the superclass.
3. Add an instance variable for each part of the TestCase fixture.
4. Override setUp() to initialize the variables.
5. Override tearDown() to release any permanent resources allocated in setUp.
6. Implement a test method with a name starting with the “test” string, and using assert methods.
7. Implement a “runTest” method to define the logic to run the different tests.

The following source code presents an example of a JUnit TestCase fixture.

```
public class MoneyTest extends TestCase {
    private Money f12CHF;
    private Money f24CHF;

    protected void setUp() {
        f12CHF= new Money(12, "CHF");
        f24CHF= new Money(24, "CHF");
    }

    public void testAdd() {
        Money f36CHF = f12CHF.add(f24CHF);
        assertTrue(f36CHF.equals(new Money(36, "CHF"));
    }

    public void testMultiply() {
        Money f24CHFa = f12CHF.multiply(new Money(2, "CHF"));
        assertTrue(f24CHF.equals(f24CHF));
    }

    protected void runTest throws Throwable() {
        testAdd();
        testMultiply();
    }
}
```

Table C.1 gives a mapping between the UML Testing Profile concepts and the JUnit concepts. Because of overlapping terminology JUnit is systematically used to prefix a term whenever appropriate.

Table C.1 - Comparison between the UML Testing Profile and JUnit concepts

UML Testing Profile	JUnit
Test Behavior:	
Test Control	A Test Control is realized by overloading the “runTest” operation of the JUnit TestCase fixture. It specifies the sequence in which several Test Cases have to be run.
Test Case	A Test Case is realized in JUnit as an operation. This operation belongs to the Test Context class, realized in JUnit as class inheriting from the JUnit TestCase class. The convention is that the name of this operation should start with the “test” string, and have no arguments so that the JUnit test runner can execute all the tests of the test context without requiring a Test Control.
Test Invocation	A Test Case is an operation that can be invoked from another Test Case operation or from the Test Control.
Test Objective	This concept can be realized in Junit using a call to the “setName” operation of the testing framework.
Stimulus	There is no such concept. Stimuli are not formalized in JUnit tests. They are directly part of the Test Cases implementations (body of the test methods).
Observation	There is no such concept. Observations are not formalized in JUnit tests. They are directly part of the Test Case implementations (body of the test methods).
Coordination	A coordination can be realized using any available synchronization mechanism available to the Test Components such as semaphores.
Default	Defaults are not supported by JUnit. One needs to implement Defaults directly by adding complexity in the behavior of the Test Case. Java’s exception mechanism can be used to realize the default hierarchy of the UML Testing Profile.
Verdict	In JUnit, predefined verdict values are *pass*, *fail*, and *error*. *Pass* indicates that the test behavior gives evidence for correctness of the SUT for that specific Test Case. *Fail* describes that the purpose of the Test Case has been violated. An *Error* verdict shall be used to indicate errors (exceptions) within the test system itself. There is no such thing as an *Inconclusive* verdict in JUnit. Therefore, the *Inconclusive* verdict will be generally mapped into *Fail*.
Validation Action	A Validation Action can be mapped to calls to the JUnit Assert library.
Log Action	There is no general purpose logging mechanism offered by the JUnit framework.
Test Log	There is no formal log provided by the JUnit framework.
Test Architecture:	

Table C.1 - Comparison between the UML Testing Profile and JUnit concepts

UML Testing Profile	JUnit
Test Context	A test context is realized in JUnit as a class inheriting from the JUnit TestCase class. To be noticed that the concept of Test Context exists in the JUnit framework but is different from the one defined in the UML Testing Profile.
Test Configuration	There is no such notion in JUnit. Generally speaking, there is rarely Test Components used in JUnit. The Test Behavior is most of the time implemented by the Test Context classifier.
Test Component	There is no Test Components per se in JUnit. Extensions to JUnit such as Mock Objects support specific forms of Test Components aimed at replacing existing classes. Nevertheless those components do not include the ability to return a verdict.
System Under Test (SUT)	The system under test doesn't need to be identified explicitly in JUnit. Any class in the classpath can be considered as a utility class or an SUT class.
Arbiter	The arbiter can be realized as a property of Test Context of a type TestResult. There is a default arbitration algorithm that generates *Pass*, *Fail*, and *Error* as verdict, where these verdicts are ordered as *Pass* < *Fail* < *Error*. The arbitration algorithm can be user-defined.
Utility Part	Any class available in the Java classpath can be considered as a utility class or an SUT part.
Test Data:	
Wildcards	There is no direct mapping to the JUnit framework. One could use pre-defined libraries to do such comparisons.
Data pool	A class together with operations to get access to the data pool.
Data partition	A class (inheriting from a data pool) together with operations to get access to the data partition.
Data selector	An operation of a data pool or a data partition.
Coding rules	There is no direct mapping to the JUnit framework. One could use pre-defined libraries to do such comparisons.
Time Concepts:	
Timezone	The time concepts are not supported by JUnit, but might be realized using standard APIs available to manipulate time.
Timer	
Test Deployment:	
Test artifact	Deployment is outside the scope of JUnit.
Test node	

Following is an example mapping the test of the Money class described in the previous sub clause - for the Test Cases given in Figure D.5 and Figure D.6.

```
public class MoneyTest extends TestCase {

    public void addSameMoney() {
        Money money1 = new Money(20, "USD"));
        Money money2 = new Money(50, "USD"));
        money1.add(money2);
        assertTrue(money1.equals(new Money(70,"USD")));
    }

    public void addDifferentMoney() {
        Money money1 = new Money(20, "USD"));
        Money money2 = new Money(50, "USD"));
        Money bag1 = money1.add(money2);
        assertTrue(bag1.contains(money1));
        assertTrue(bag1.contains(money2));
    }

    protected void runTest throws Throwable() {
        addSameMoney();
        addDifferentMoney();
    }
}
```

C.2 Mapping to TTCN-3

TTCN-3 - Testing and Test Control Notation (3rd edition) - is widely accepted as a standard for test system development in the telecommunication and data communication area. TTCN-3 comprises concepts suitable to all types of distributed system testing.

TTCN-3 is a test specification and implementation language to define test procedures for black-box testing of distributed systems. Stimuli are given to the system under test (SUT); its reactions are observed and compared with the expected ones. Based on this comparison, the subsequent test behavior is determined or the test verdict is assigned. If expected and observed responses differ, then a fault has been discovered that is indicated by a test verdict fail. A successful test is indicated by a test verdict pass.

TTCN-3 allows the description of complex distributed test behavior in terms of sequences, alternatives, loops, and parallel stimuli and responses. Stimuli and responses are exchanged at the interfaces of the system under test, which are defined as a collection of ports being either message-based for asynchronous communication or signature-based for synchronous communication. The test system can use any number of test components to perform test procedures in parallel. Likewise to the interfaces of the system under test, the interfaces of the test components are described as ports.

TTCN-3 is a modular language and has a similar look and feel to a typical programming language. In addition to the typical programming constructs, it contains all the important features necessary to specify test procedures and campaigns for functional, conformance, interoperability, load, and scalability tests like test verdicts, matching mechanisms to compare the reactions of the SUT with the expected range of values, timer handling, distributed test components, ability to specify encoding information, synchronous and asynchronous communication, and monitoring.

A TTCN-3 test specification consists of four main parts:

1. type definitions for test data structures
2. template definitions for concrete test data

3. function and test case definitions for test behavior

4. control definitions for the execution of test cases

TTCN-3 was one basis for the development of the UML Testing Profile. Still, they differ in several respects. The UML Testing Profile is targeted at UML providing selected extensions to the features of TTCN-3 as well as restricting/omitting other TTCN-3 features. A mapping from the UML Testing Profile to TTCN-3 is possible but not the other way around. The principal approach towards the mapping to TTCN-3 consists of two major steps:

1. Take the UML Testing Profile stereotypes and associations and assign them to TTCN-3 concepts.

2. Define procedures how to collect required information for the generated TTCN-3 modules.

Table C.2 compares the UML Testing Profile concepts with existing TTCN-3 testing concepts. All UML Testing Profile concepts have direct correspondence or can be mapped to TTCN-3 testing concepts.

Table C.2 - Comparison between the UML Testing Profile and TTCN-3 concepts

UML Testing Profile	TTCN-3
Test Behavior:	
Test Control	The control part of a TTCN-3 module.
Test Case	A TTCN-3 testcase. The behavior of a testcase is defined by functions that are generated via mapping functions applied to the behavioral features of a test context. The main test component (MTC) is used like a “controller” that creates test components and starts their behavior. The MTC controls also the arbiter.
Test Invocation	The execution of a TTCN-3 testcase.
Test Objective	Not part of TTCN-3, just a comment to a test case definition.
Stimulus	Sending messages, calling operations, and replying to operation invocations.
Observation	Receiving messages, operation invocations, and operation replies.
Coordination	Message exchange between test components.
Default	Altstep and activation/deactivation of the altsteps along the default hierarchy.
Verdict	The default arbiter and its verdict handling is an integral part of TTCN-3. For user-defined, a special verdict type and updating the arbiter with set verdicts is needed.
Validation Action	External function or data functions resulting in a value of the specific verdict type.
Log Action	Log operation.
Test Log	Not part of TTCN-3, but could be mapped just as a strict sequential behavioral function.
Test Architecture:	
Test Context	TTCN-3 module definition part covering all test cases and related definitions of a test context, having a specific TSI component type (to access the SUT) and a specific behavioral function to set up the initial test configuration for this test context.

Table C.2 - Comparison between the UML Testing Profile and TTCN-3 concepts

UML Testing Profile	TTCN-3
Test Configuration	Configuration operations create, start, connect, disconnect, map, unmap, running, and done for dynamic test configurations. Behavioral function to set up the initial test configuration.
Test Component	TTCN-3 component type[1], used for the creation of test components and their connection to the SUT and to other test components.
System Under Test (SUT)	The test system accesses the SUT via the abstract test system interfaces (TSI). The SUT interfaces result in port types used by TSI. One additional port is needed to communicate with a user-defined arbiter. Potentially additional ports are needed to coordinate/synchronize test components.
Arbiter	The UML Testing Profile default arbiter is a TTCN-3 built-in. User-defined arbiters are realized by the MTC.
Utility Part	External constants and/or external functions to refer and make use of the utility part, which is outside the TTCN-3 module.
Test Data:	
Wildcards	TTCN-3 matching mechanisms.
Data pool	An external constant (referring to the data in the data pool) or external functions to get access to the data pool.
Data partition	TTCN-3 matching mechanisms can be used to handle data partitions for observations. For stimuli however, user defined functions are needed to realize the test case execution with different data to be sent to the SUT.
Data selector	An external function to get access to the data of a data pool or data partition.
Coding rules	TTCN-3 encode and encode variant attributes.
Time Concepts:	
Timezone	Cannot be represented in TTCN-3.
Timer	TTCN-3 timer.

1. Please note that due to performance aspects (for example) there might be not only a one-to-one mapping between the UML Testing Profile and TTCN-3 components. Instead, a different test configuration might be used.

In the following, an example mapping is provided for the Bank ATM case study described in the previous sub clause - for the test case **`invalidPIN`** given in Figure D.11.

Two TTCN-3 modules are generated: one for ATM being the SUT (and being defined in a separate UML package) and another module for the ATM test defining the tests for the Bank ATM also in a separate UML package. The module **`ATM`** provides all the signatures available at the SUT interfaces, which are used during testing.

```
module ATM {
    //withdraw(amount : Integer): Boolean
    signature withdraw(integer amount) return boolean;
    //isPinCorrect(c : Integer) : Boolean
    signature isPinCorrect(integer c) return boolean;
    //selectOperation(op : OpKind) : Boolean
```

```
    signature selectOperation(OpKind op) return boolean;
    … // and so on
}
```

The module for the ATM test **ATMTest**

- imports all the definitions from the ATM module,
- defines the group for the ATM test context,
- provides within this group port and component type definitions, the function to set up the initial test configuration and finally the test cases.

To make this mapping more compelling, a user-defined arbiter is assumed in addition and the default handling is made explicit.

```
module ATMTest {
    import from ATM all;
    // utility IAccount
    type record IAccount {
        integer balance,
        charstring number
    }
    external const IAccount accounts[0..infinity];
    group ATMSuite {
      ... // all the definitions constituting the tests for ATM
    } // group ATMSuite
} // module ATMTest
```

The required and provided interfaces are reflected in corresponding port definitions **atmPort_PType** and **netCom_PType**, which are then used in the component type definitions **BankEmulator_CType** and **HWEmulator_CType** to constitute the component types for the PTCs:

```
   //required interfaces: IHardware
   //provided interface: IATM
   type port atmPort_PType procedure {
       in display_, ejectCard, ejectMoney, acceptMoney, getStatus; //IHardware
       out withdraw, isPinCorrect, selectOperation,
                 storeCardData, storeSWIFTnumber; //IBank
   }

   //required interface: IBank
   //no provided interface
   type port netCom_PType procedure {
       in debitAccount, depositAccount, findAccount,
       wireMoney, checkCredentials //IBank
   }
   // test component type BankEmulator
   type component BankEmulator_CType {
     port netCom_PType bePort;
     port Arbiter_PType arbiter; // user defined arbiter
   }
```

```
// test component type HWEmulator
type component HWEmulator_CType {
   port atmPort_PType hwCom;
   var boolean pinOk;
   var charstring enteredPIN;
   var charstring message_;
   timer t1;
}
```

The following shows the mapping for a user-defined arbiter. A specific type **`MyVerdict_Type`** together with an arbitration function **`Arbitration`** is used to calculate the overall verdict during test case execution. The final assessment is given by mapping the user-defined verdicts to the TTCN-3 verdict at the end. This enables the use of statistical verdicts where, for example, 5% failures lead to fail but less failures to pass. The arbiter is realized by the **`MTC`**. It receives verdict update information via a separate port arbiter. The arbitrated verdict is stored in a local variable **`mv`**.

```
//the arbitration
type enumerated MyVerdict_Type {
    pass_, fail_, inconc_, none_
}
type port Arbiter_PType message {
    inout MyVerdict_Type
}
// the MTC is just a controller
type component MTC_CType {
    port Arbiter_PType arbiter; // user defined arbiter
    var MyVerdict_Type mv:= none_;
}
function Arbitration(BankEmulator_CType be, HWEmulator_CType hwe)
runs on MTC_CType {
    while (be.running or hwe.running) {
      alt {
            [] arbiter.receive(none_) {...}
   [] ...}
            }
    }
    if (mv == pass_) { setverdict(pass) }
    else ...
}
```

The defaults in the defaults hierarchy are mapped to several altsteps, which will be invoked later along that hierarchy. In this example, an altstep for every component type is defined (i.e., **`HWEmulator_classifierdefault`** and **`BankEmulator_classifierdefault`**). The package level default **`ATMTestDefault`** does not need to be mapped - it is automatically realized by the TTCN-3 semantics.

```
altstep HWEmulator_classifierdefault()
runs on HWEmulator_CType {
    var charstring s;
  [] t1.timeout {arbiter.send(fail_);}
  [] hwCom.getcall(ejectCard:{}) {arbiter.send(fail_);}
  [] hwCom.getcall(display_:{?}) -> param (s) {
      if (s == "Connection lost") { arbiter.send(inconc_) }
```

```
      else {arbiter.send(fail_)} }
        }
   }
   altstep BankEmulator_classifierdefault()
   runs on BankEmulator_CType {
       //is empty
   }
```

The component type for the test system interface **SUT_CType** is constituted by the ports **netCom** and **atmPort** used during testing in the specific test context. A configuration function **ATMSuite_Configuration** sets up the initial test configuration and is invoked at first by every test case of that test context.

```
   // SUT
   type component SUT_CType {
     port netCom_PType netCom;
     port atmPort_PType atmPort;
   }
   // setup the configuration
   function ATMSuite_Configuration
   ( in SUT_CType theSUT, in MTC_CType theMTC, inout
         BankEmulator_CType be, inout HWEmulator_CType hwe)
   {
         be:=BankEmulator_CType.create;
         map(theSUT:netCom,be:bePort); //map to the SUT
         hwe:=HWEmulator_CType.create;
         map(theSUT:atmPort,hwe:hwCom); //map to the SUT


         connect(theMTC:arbiter,be:arbiter); // arbitration
         connect(theMTC:arbiter,hwe:arbiter); // arbitration
   }
```

The **invalidPIN** test case uses two PTCs **hwe** and **be** each having its own test behavior, which is defined by behavioral functions **invalidPIN_hwe** and **invalidPIN_be** as shown below.

```
function invalidPIN_hwe(integer invalidPIN) runs on HWEmulator_CType {
      activate(HWEmulator_classifierdefault());
      // here we need test derivation
      // just for that example straightforward definition along the lifeline
      var boolean enterPin_reply;
      hwCom.call(storeCardData:{current},nowait);
      t1.start(2.0);
      hwCom.getreply(display_:{"Enter PIN"});
      t1.stop;
      hwCom.call(isPinCorrect:{invalidPIN},3.0) {
      [] hwCom.getreply(isPinCorrect:{?} value false) {}
         }
         hwCom.getreply(display_:{"Invalid PIN"});
         hwCom.getreply(display_:{"Enter PIN again"});
      arbiter.send(pass_); // local verdict to the arbiter
}
```

```
function invalidPIN_be() runs on BankEmulator_CType {
      activate(BankEmulator_classifierdefault());
      // nothing more
}
```

Finally, the test case can be provided. According to the initial test configuration, two PTCs **hwe** and **be** are used. The configuration is set up with **ATMSuite_Configuration**. After accessing the value for the data partition **giveInvalidPIN(current)**, the test behavior on the PTCs is started with **invalidPIN_hwe** and **invalidPIN_be**. The arbiter **Arbitration(be,hwe)** controls the correct termination of the test case. This completes the mapping.

```
//+invalidPIN() : Verdict
  testcase invalidPIN_test()
  runs on MTC_CType system SUT_CType {
      var HWEmulator_CType hwe;
      var BankEmulator_CType be; // initial configuration
      ATMSuite_Configuration(system,mtc,be,hwe);
      const integer invalidPIN:= giveInvalidPIN(current);
      hwe.start(invalidPIN_hwe(invalidPIN));
      be.start(invalidPIN_be());
      Arbitration(be,hwe);
   }
```

The following table summarizes the mapping of test model elements for the **invalidPIN** test case to TTCN-3.

Table C.3 - Mapping of invalidPIN to TTCN-3

UML Testing Profile	TTCN-3
package ATM	module ATM
package ATMTest	module ATMTest
testContext ATMSuite	group ATMSuite
testComponent BankEmulator	type component BankEmulator
interface of BankEmulator	type port netCom_PType
testComponent HWEmulator	type component HWEmulator
interface of HWEmulator	type port atmPort_PType
timer of HWEmulator	timer of componentHWEmulator
operation storeCardData	signature storeCardData
operation display	signature display_
operation isPinCorrect	signature isPinCorrect
test behavior of HWEmulator	function invalidPIN_hwe
classifier default of HWEmulator	altstep HWEmulator_classifierdefault
composite structure of ATMSuite	function ATMSuite_Configuration
testCase invalidPIN	testcase invalidPIN_test

Annex D - Examples

(normative)

D.1 General

This annex contains an example of using the UML Testing Profile to specify test cases, from unit level, to integration and system level, to cross-enterprise system level. The example is motivated using an interbank exchange scenario in which a customer with a European Union bank account wishes to deposit money into that account from an Automated Teller Machine (ATM) in the United States.

The diagram in Figure D.1 provides an overview of the architecture of the system. The ATM used by this customer interconnects to the European Union Bank (EU Bank), through the SWIFT network, who plays the role of a gateway between the logical networks of the US Bank and the EU Bank.

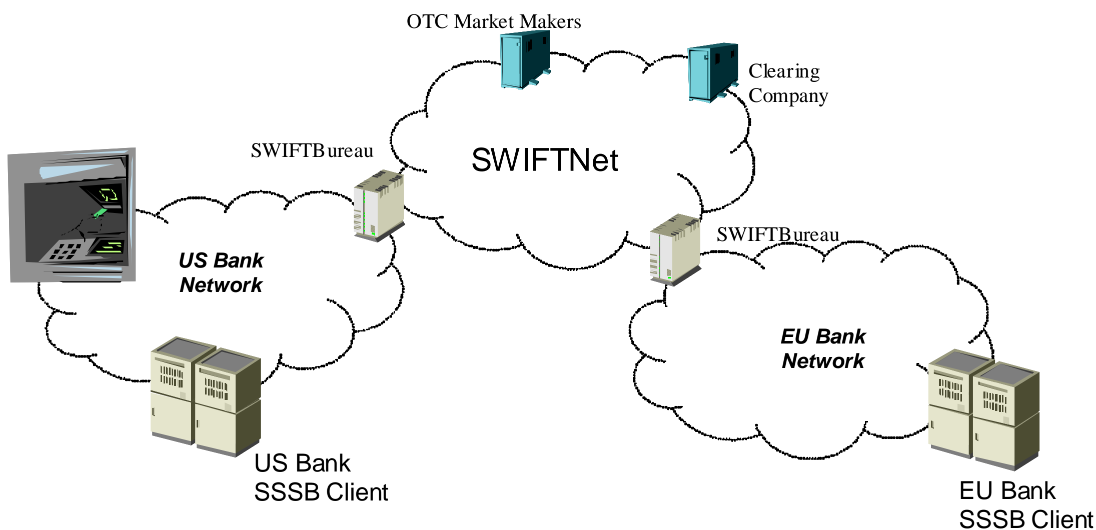


Figure D.1 - Overview on the InterBank Exchange Network

The packages for this example are shown in Figure D.2. They are used in the following sub clauses.

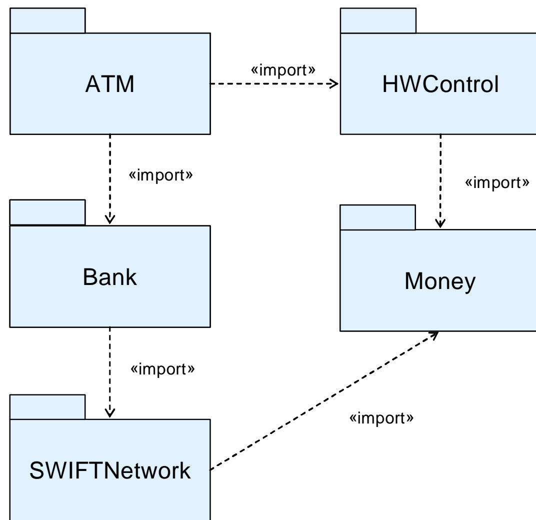


Figure D.2 - Packages for the InterBank Exchange Network

D.2 Money Example

This sub clause illustrates the use of the UML Testing Profile to define unit test level test cases. It reuses and extends the Money and MoneyBag classes provided as examples of the now famous JUnit test framework (see www.junit.org). In this scenario, these classes are used by the ATM to count the bills entered by a user when making a deposit in cash. As illustrated by the figure, these classes belong to the Money package.

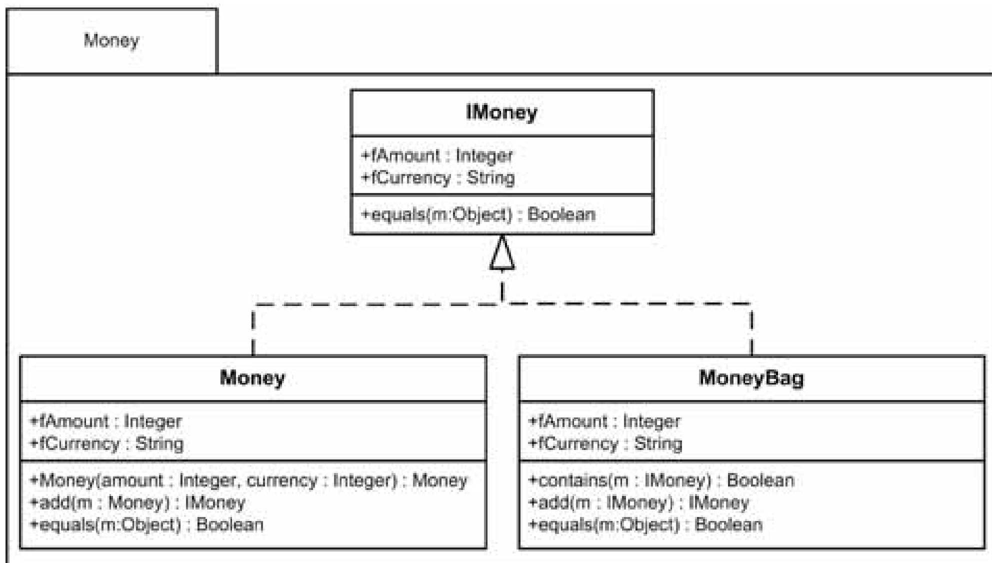


Figure D.3 - Money structure

The Test Objective is given as follows:

- Verify that the Money and MoneyBag classes are appropriately counting the bills added by the user:
    - When bills from the same currency are entered.
    - When bills from different currencies are entered.

The following diagram illustrates the test package. In this example, no test components are used, but rather the test behavior is implemented using the behavior of the Test Context classifier. The System Under Test is limited to the classes defined in the Bank package. As shown in the following diagram, the test package includes only the Test Context.

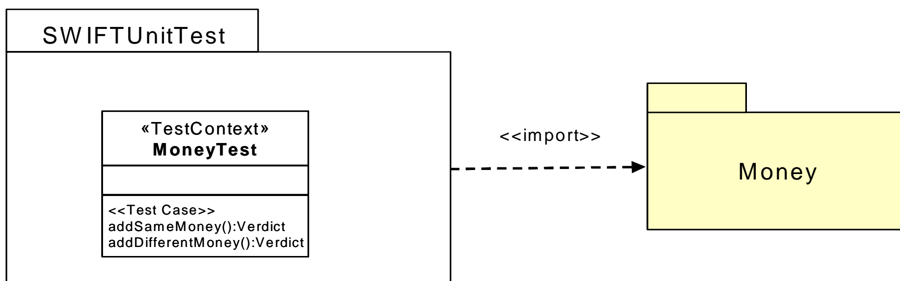


Figure D.4 - SWIFTUnitTest package

The first Test Objective primarily consists in exercising the Money interfaces and ensuring that the Money class returns an object of type Money with the correct amount and currency. The following diagram highlights the behavior of the addSameMoney behavioral feature

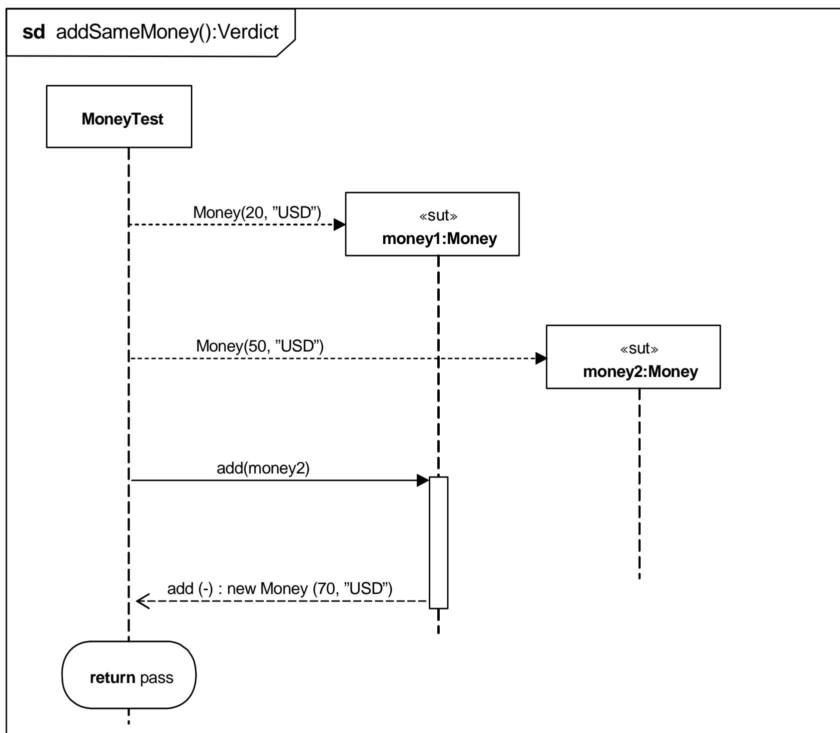


Figure D.5 - Unit test behavior for addSameMoney

The second Test Objective consists in exercising the Money interfaces and ensuring that the Money class returns an object of type MoneyBag with the currencies of the two Money objects that were added during the call to the add operation. The following diagram highlights the behavior of the addDifferentMoney behavioral feature.

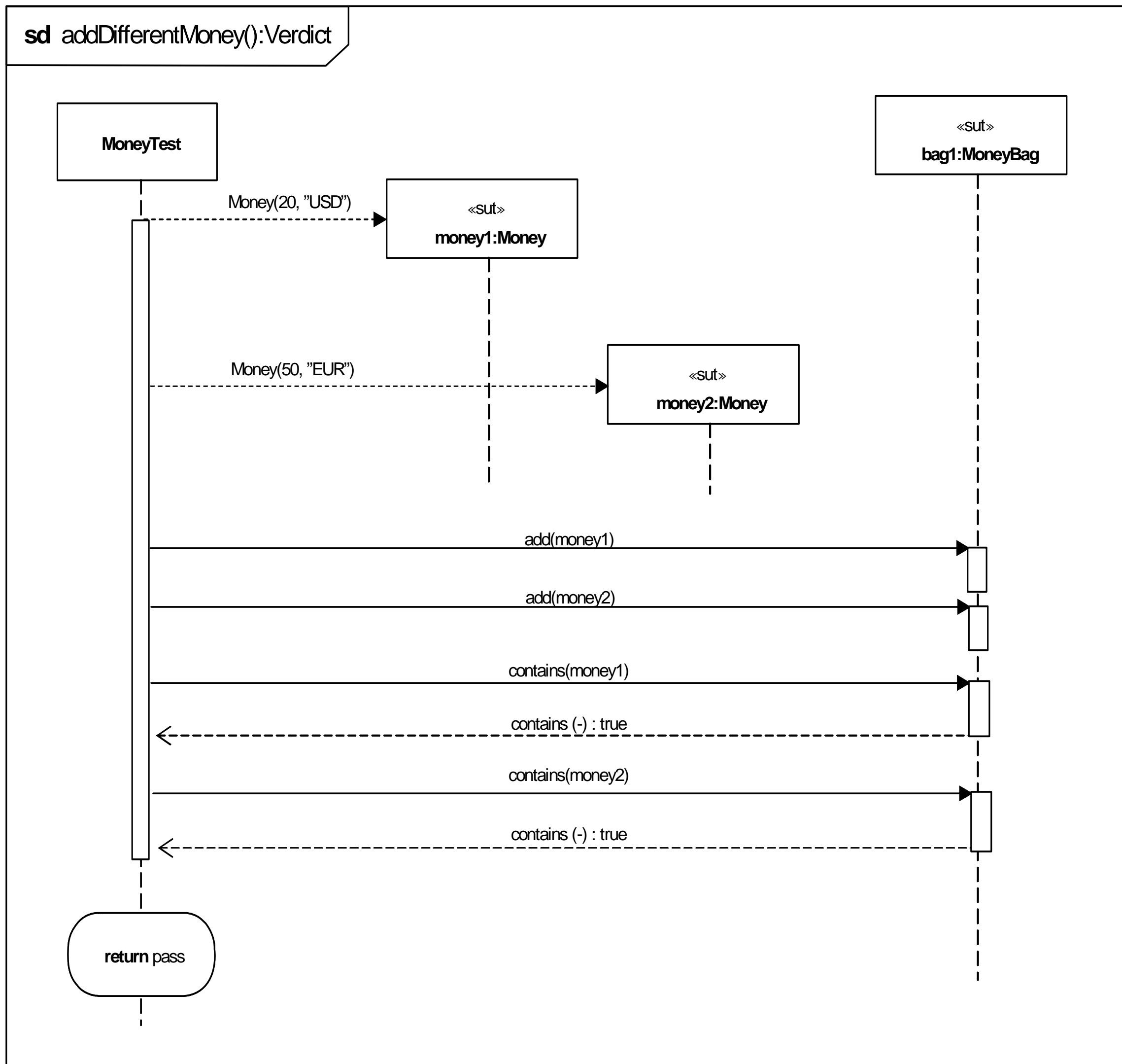


Figure D.6 - Unit test behavior for addDifferentMoney

D.3 Bank ATM Example

This part of the example illustrates how the UML Testing Profile can be used for specifying tests at integration and system levels. The purpose of these tests is to verify the logic of the ATM machine when a user initiates a transaction to deposit and wire money to an account in another part of the world. This includes authorizing a card and a pin-code and initiating communication with the bank network. The hardware, bank, and network connections are all emulated, since we are testing the logic of the ATM machine itself only.

The ATM logic is specified in the ATM package. The ATM package imports the HWControl package where the interfaces to the hardware is specified, and the Bank package, where the interface to the bank is specified. Figure D.7 shows the packages involved in this part of the example.

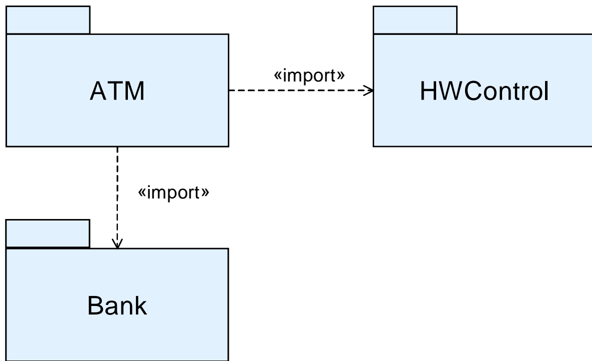


Figure D.7 - ATM and related packages

Figure D.8 shows the public parts of these packages. The BankATM class controls the ATM logic and is the focus of our tests. It implements the IATM interface for the control logic and relies on a number of interfaces to communicate with the hardware and the bank.

Since the hardware and the bank is emulated, only the interfaces of the HWControl and Bank packages used by the BankATM class are shown.

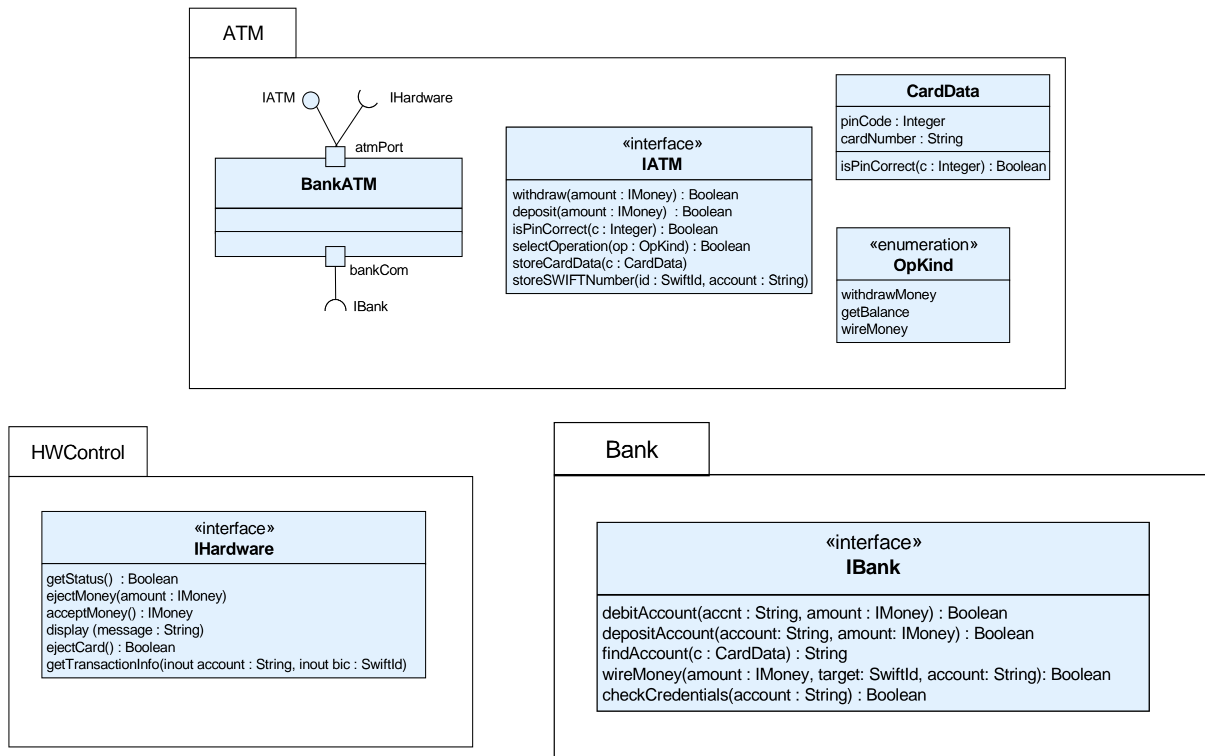


Figure D.8 - Elements of the system to be tested

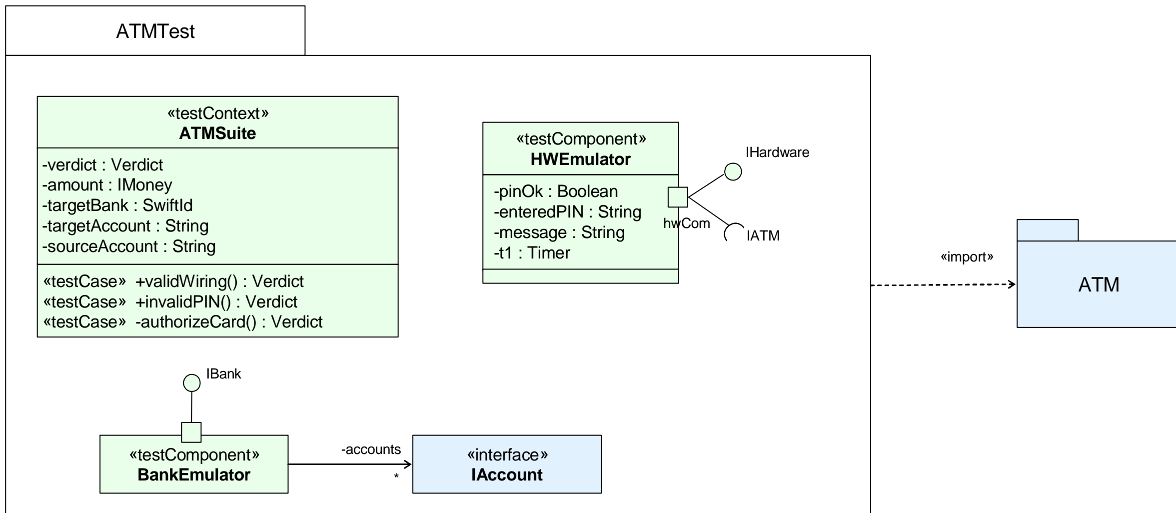


Figure D.9 - The ATMTest package

The ATMTest package shown in Figure D.9 contains all model elements necessary to fully specify our tests. ATMTest imports the ATM package to get access to the elements to be tested. ATMTest consists of one test context, ATMSuite, and two test components: BankEmulator and HWEmulator. ATMSuite has three testcases: validWiring(), invalidPIN(), and authorizeCard(). Two have public visibility and one private. The test components implement the interfaces of the HWEmulator and BankNetwork packages and will serve as emulators for these packages.

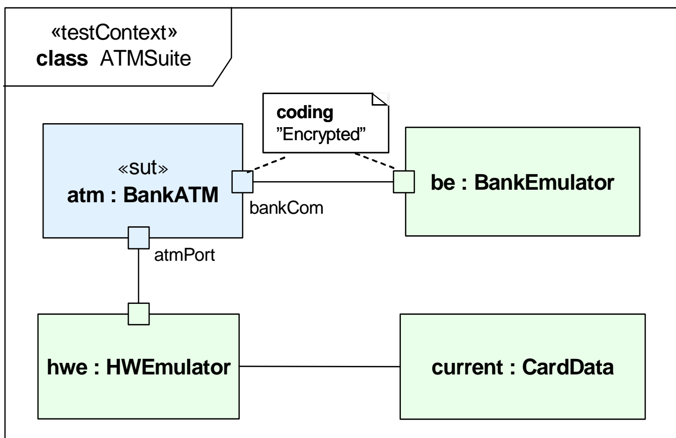


Figure D.10 - The composite structure of the ATMSuite test context

The test configuration (i.e., the composite structure of the test context) is shown in Figure D.10. The test configuration specifies how the SUT, a number of test components, and one utility part are used in a particular test context. Ports and connectors are used to specify possible ways of communication. Each test configuration must consist of at least one SUT.

The ATMSuite composite structure consists of one SUT, two test components, and one utility part. The SUT, atm, is typed by the BankATM class from the ATM package. The SUT is connected to two parts typed by test components, be and hwe. In addition, there's a utility part, current, used by hwe. A coding rule is applied to the ports of the "atm" and the "be" to show that the communication between these properties is encrypted.

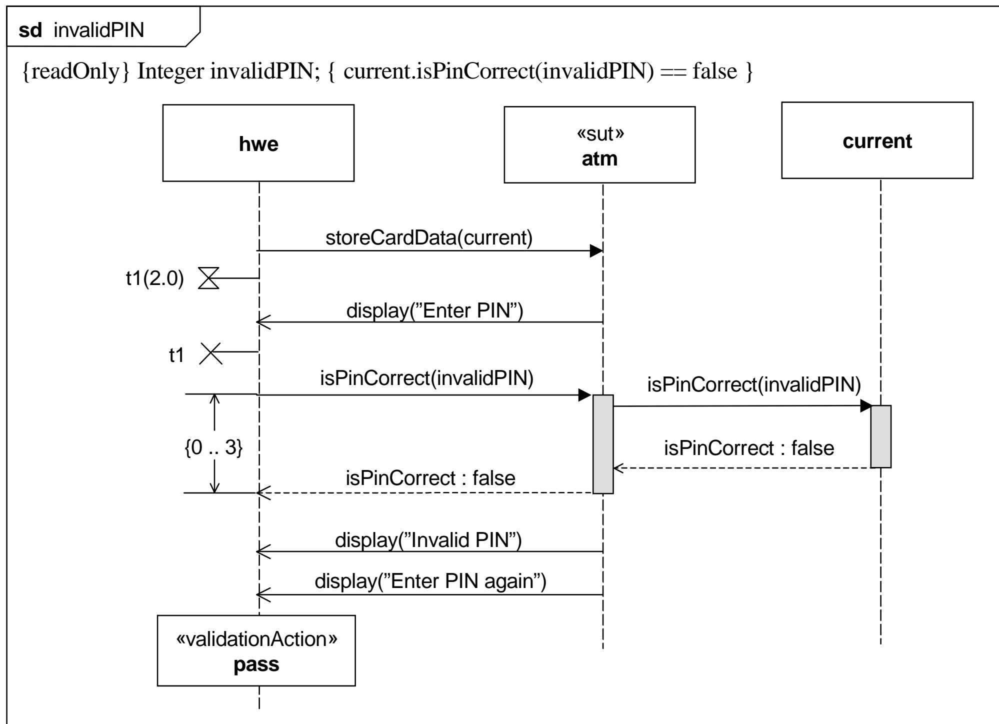


Figure D.11 - The behavior of the invalidPIN test case

The sequence diagram in Figure D.11 specifies the behavior for the invalidPIN() test case. The test objective of this test case 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 test contexts and test cases can be specified using any UML behavior, but in this case an interaction is used. When used as a test behavior, the interaction specifies the expected sequence of messages. During a test case, validation actions can be used to set the verdict. Validation actions use an arbiter to calculate and maintain a verdict for a test case. Test cases always return verdicts. This is normally done implicitly through the arbiter and doesn't have to be shown in the test case behavior. In the example above, an arbitrated verdict is returned implicitly.

The diagram above also illustrates the use of a timer and a duration constraint. The timer is used to specify how long the hardware emulator will wait for the display ("Enter PIN") message. Once the message has been received, the timer is stopped. If the message does not appear within the time limit, the timer times out and the specification is violated. In this example the timeout is handled in the default of this test component, see Figure D.15 for details. The timer t1 is an attribute of the HWEmulator test component, see Figure D.9.

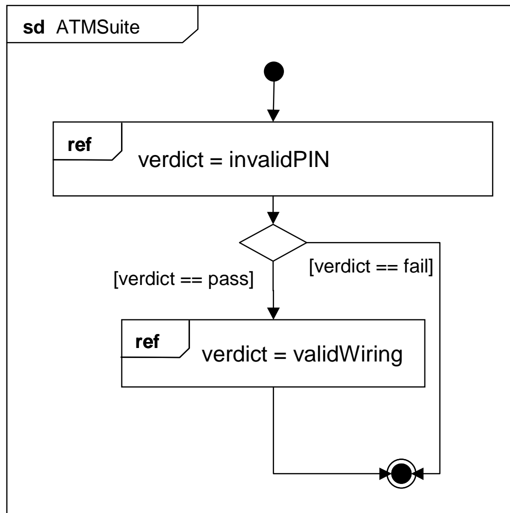


Figure D.12 - Classifier behavior for ATMSuite

Execution of the test cases of a test context, commonly referred to as test control, can be specified in the classifier behavior of the test context. Figure D.12 above shows an example of this using an interaction overview diagram. Another way to control the execution is to call the test cases just as ordinary operations from outside of the test context.

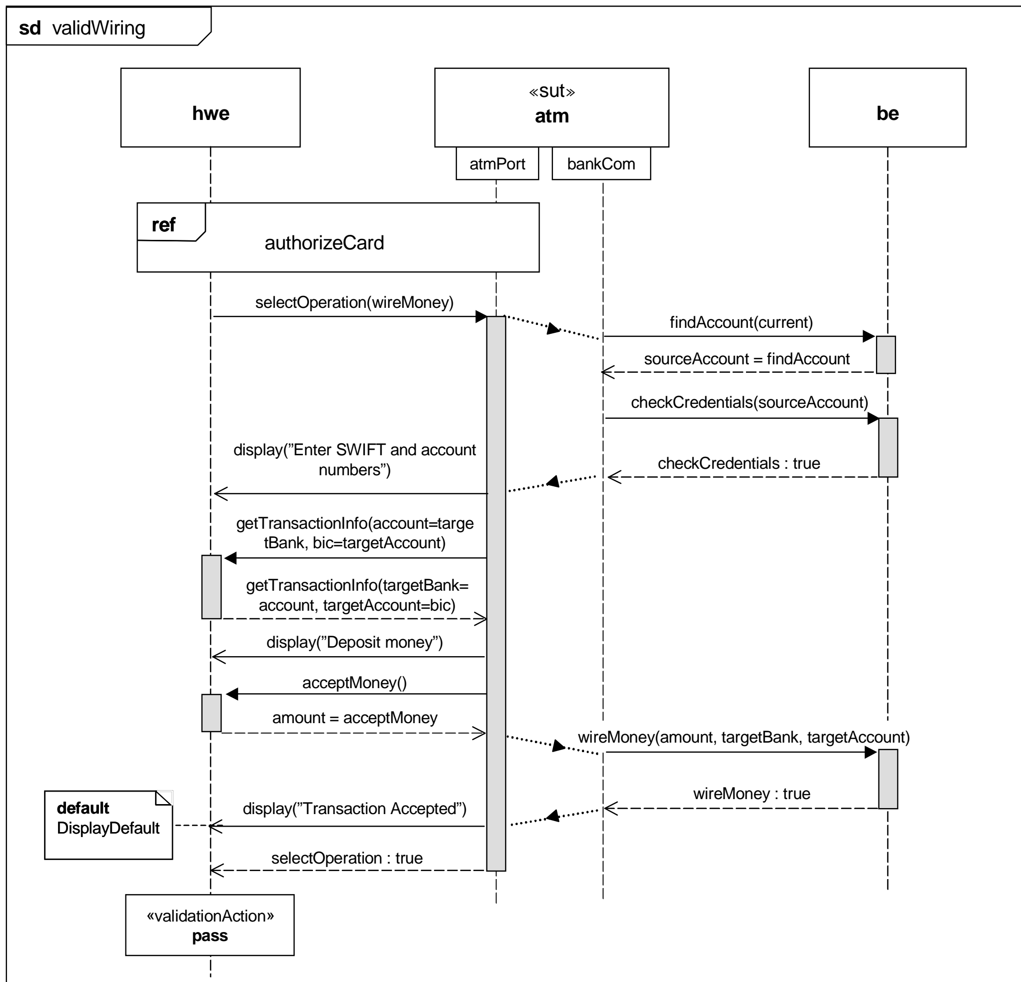


Figure D.13 - The validWiring test case

The validWiring test case is slightly more elaborated, with the following test objective:

Verify that when a user with the right credentials initiates a wiring of US dollars to his European account the transaction is correctly handled.

One of the other test cases in the context, authorizeCard() is referenced through an interaction occurrence. This illustrates how test case definitions can be reused within a test context. General ordering is used to illustrate how to sequence messages between two test components. Finally, the default concept is introduced.

A default specifies how to respond to messages or events not specified in the original test case behavior. Defaults are typically used for exception handling. To include all possible message sequences in the test case behavior would be cumbersome. Defaults can be applied on many different levels (e.g., test components and upon reception of individual messages).

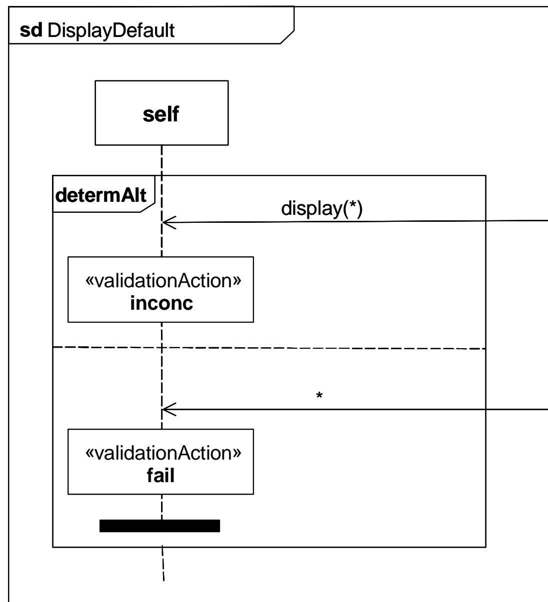


Figure D.14 - Default for individual message reception

Figure D.14 specifies the DisplayDefault, a default for the reception of the *display("Transaction accepted")* message in the *validWiring* test case. The DisplayDefault describes what happens when a different message is received. An inconc verdict is assigned if a display message is received with a parameter different to the expected one. Otherwise, fail is assigned and the test component finishes.

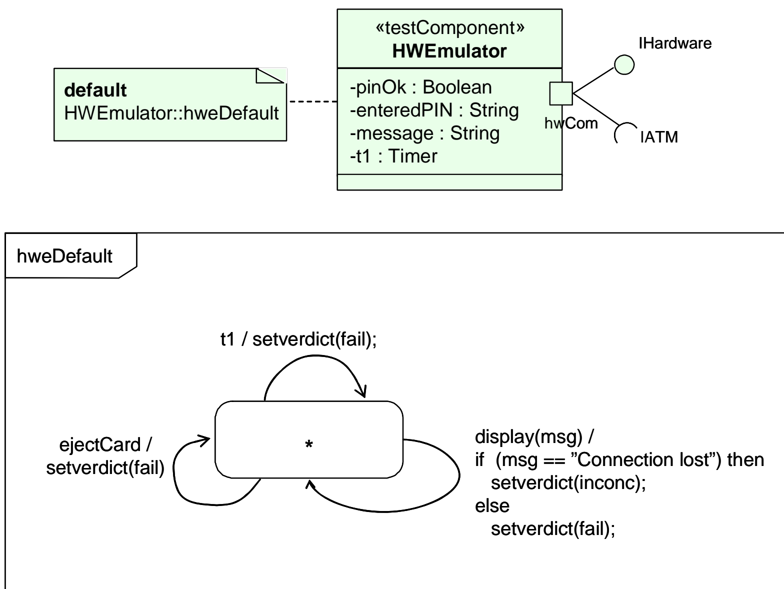


Figure D.15 - A statemachine default applied to a test component

Figure D.15 specifies and applies a default to a single test component, the HWEmulator test component. This default applies to all test behaviors owned by the test component.

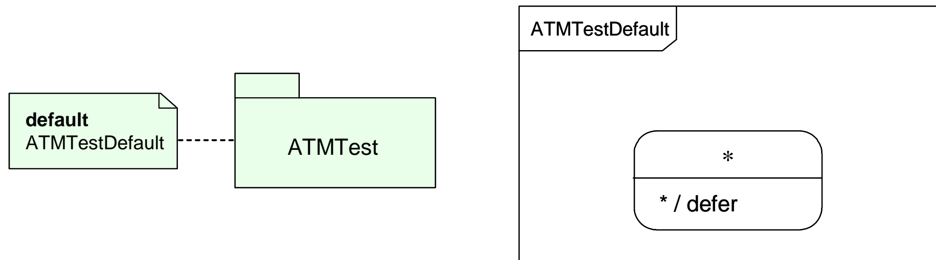


Figure D.16 - Package level default

In some cases it is necessary to apply a default to many or all elements in a test model. Assume, for example, that you need to adjust the UML semantics for events. Unless explicitly deferred, events will be discarded on reception. This might not be desirable from a testing perspective where all events are considered important. Figure D.16 shows how this can be accomplished by defining a default with the desired behavior and applying it to a package. Defaults applied to a package will apply to all test components in the package and therefore, they often need to be very general in their specification. The ATMTestDefault contains wildcards to be applicable to all test components in the package. The star '*' in the state symbol means 'any state,' and the star before the defer statement means any signal.

For any test behavior, several defaults may be applied at the same time. In our example when receiving the *display("Transaction Accepted")* message, three defaults are active: the *DisplayDefault*, the *hweDefault*, and the *ATMTestDefault*. If one default does not handle a message, the next default is checked, and so on. The evaluation order is given by the scope on which the defaults are applied. Evaluation starts at the innermost scope and traverses outwards.

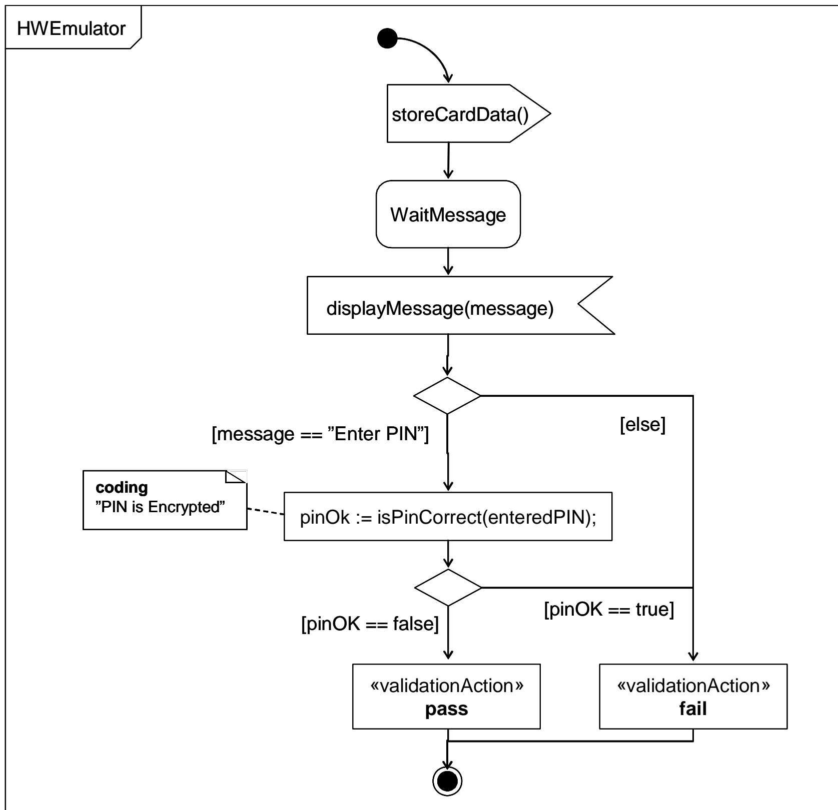


Figure D.17 - Test component behavior

In many cases, there's a need to specify the detailed behavior of individual test components (e.g., for test generation purposes). This is done by specifying the classifier behavior of a test component. Statemachine diagrams are suitable for this. Figure D.17 contains parts of the test behavior for the HWEmulator test component. (The part corresponding to the invalidPIN test case.)

D.4 Money Transfer Example

This sub clause contains an example of using the UML Testing Profile to specify cross-enterprise system level test cases. The example is motivated using an interbank exchange scenario in which a customer with an European Union bank account wishes to deposit money into that account from an Automated Teller Machine (ATM) in the United States. Likewise, we wish to emulate the same scenario being initiated from an ATM in Europe. This example builds on the unit level and sub-system level examples covered in the previous two sub clauses. This example illustrates several key concepts from the UML Testing Profile, including: test configuration (with multiple components), test control, arbiter, validation actions, data pools, as well as concepts from load/stress testing. The load/stress test objective states that a combination of European and US initiated transactions must behave correctly and 98% of them must be completed within 4 seconds.

.

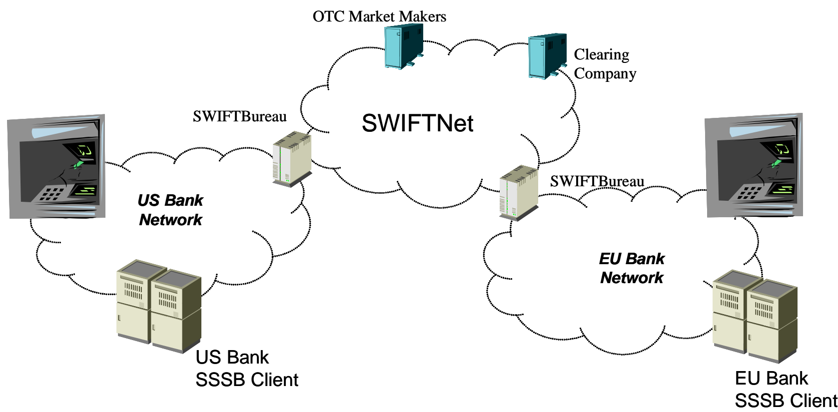


Figure D.18 - Interbank Exchange Network overview

Five packages are used to structure the classes and interfaces for the example. Three of these are known from the previous examples (ATM, Money, and HWControl). Two additional packages are introduced that are unique to this example (BankNetwork and SWIFTNetwork). Figure D.19 illustrates the package structure of the system under test.

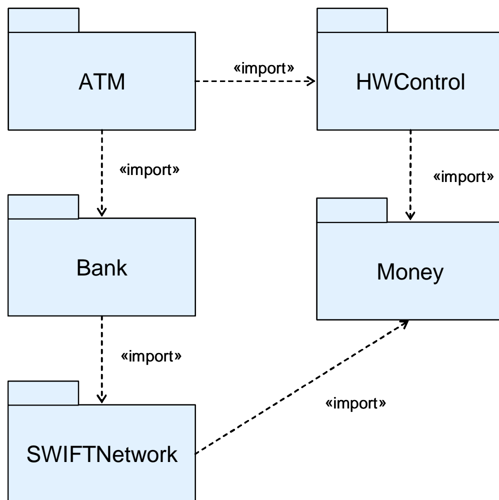


Figure D.19 - Package structure of the Interbank Exchange Network

Figure D.20 illustrates the contents of the Bank package, while Figure D.21 illustrates the contents of the SWIFT package. The IBank interface provides operations to find, credit, and debit accounts; to check credentials; and to wire money from one account to another. The IAccount interface provides operations to credit and debit accounts, and to check the balance of an account. The ISWIFT interface provides an operation to transfer a given amount from a source account to a target account. These interfaces are the focus of this sub clause, although the example also uses interfaces and classes from the previous two examples.

.

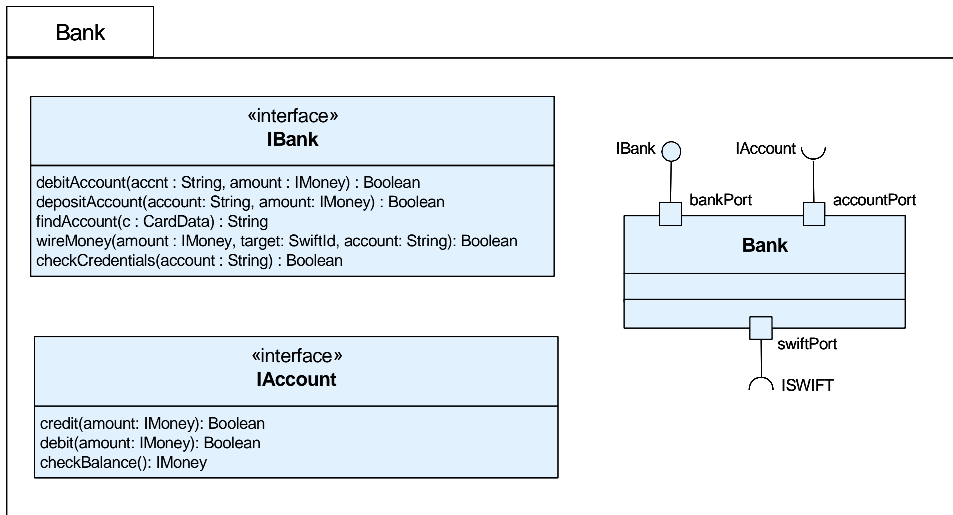


Figure D.20 - BankNetwork

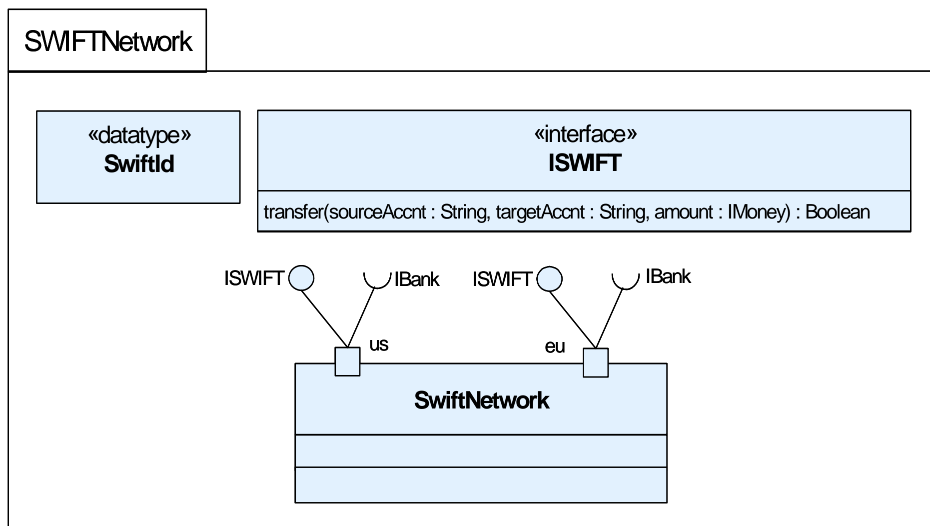


Figure D.21 - SWIFTNetwork

Figure D.22 is a package illustrating the test architecture for the example. The system under test is scoped to be the SWIFT network, the US and European Bank Networks, and ATM systems. Two test components provide the capability to execute and verify that the transfer occurred correctly: TransactionController and LoadManager. The TransactionController drives the ATMs and is used to represent the accounts for both the US and EU banks. These allow verification that transferred money is debited from the US account and deposited to the EU accounts and vice-versa. They also provide verification that an invalid transfer does not result in the same debit/credit cycle. The LoadManager controls

the workload of the test case. Additionally, a specialization of Arbiter is provided that supports the necessary capabilities to do load testing. Two additional types are also provided for users as utilities: DataPool provides data management and access services, and TrxnData contains the data provided by the DataPool for performing a transaction.

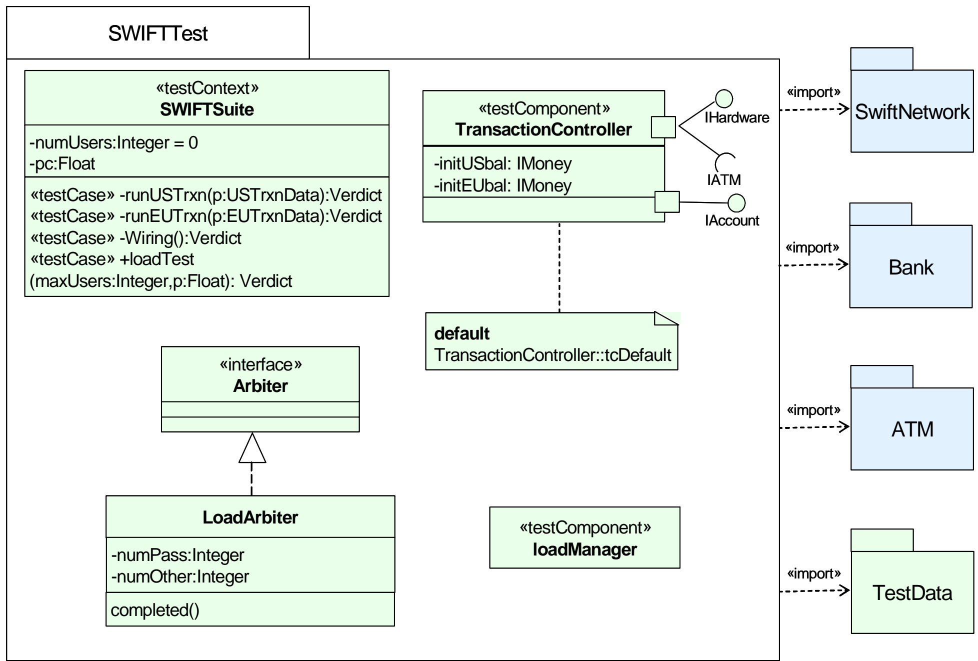


Figure D.22 - SWIFTTest package

Figure D.23 is a package illustrating the data pool, data partition and data selector concepts. The TestData package defines for TrxnData the data pool DataPool and the data partitions EUTrxnData and USTrxnData. The data partitions have two data samples defined each. Data selectors getEUTrxnData, getUSTrxnData, and getDistributionInterval are used for the access to the data pool and the data partitions.

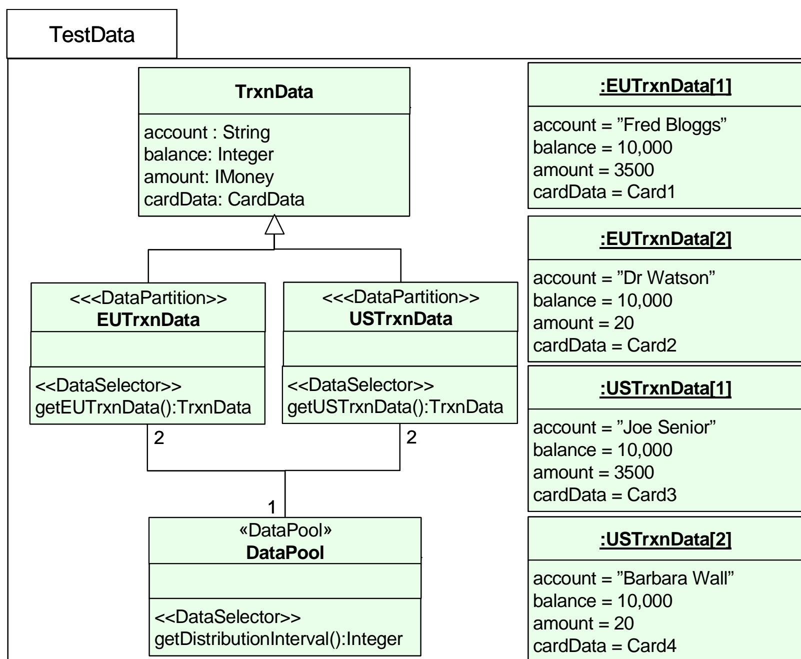


Figure D.23 - TestData package

Figure D.24 illustrates the internal structure of the TestContext. This shows how the SUT components and the test components are connected. ATMs are connected to the US & EU banks, both of which are connected via the SWIFT network. The TransactionController is connected to the ATM components and both the US and EU banks. The LoadManager is connected to the TransactionController and serves to ensure that the various properties of the load test are managed correctly. The LoadArbiter is connected to the LoadManager and TransactionController test components.

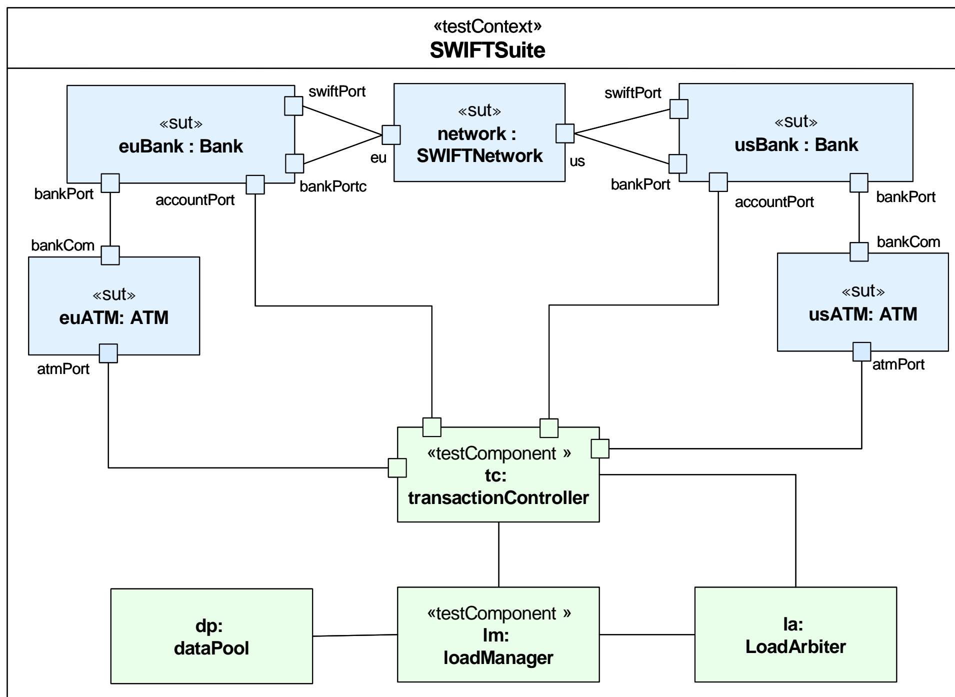


Figure D.24 - The composite structure of the SWIFTSuite test context

Figure D.25 and Figure D.26 illustrate the behavior of the loadTest method of the TestContext element. The test is invoked with two parameters: the maximum number of virtual users that are simulated at any one time and the data pool reference. Given these parameters, the system launches the appropriate number and types of test cases and monitors their outcomes. The load manager lm is used as a generation. At first, the total test duration is set by a timer. Then, the first Wiring is started. Following the start of the Wiring test scenario another timer is started, which is used to set the duration between generations when the second timer expires. This will loop as long as the test duration expires. Then, the loop is left. The load manager applies a special validation action value and the arbitration is entered to calculate the final test case verdict (see Figure D.27).

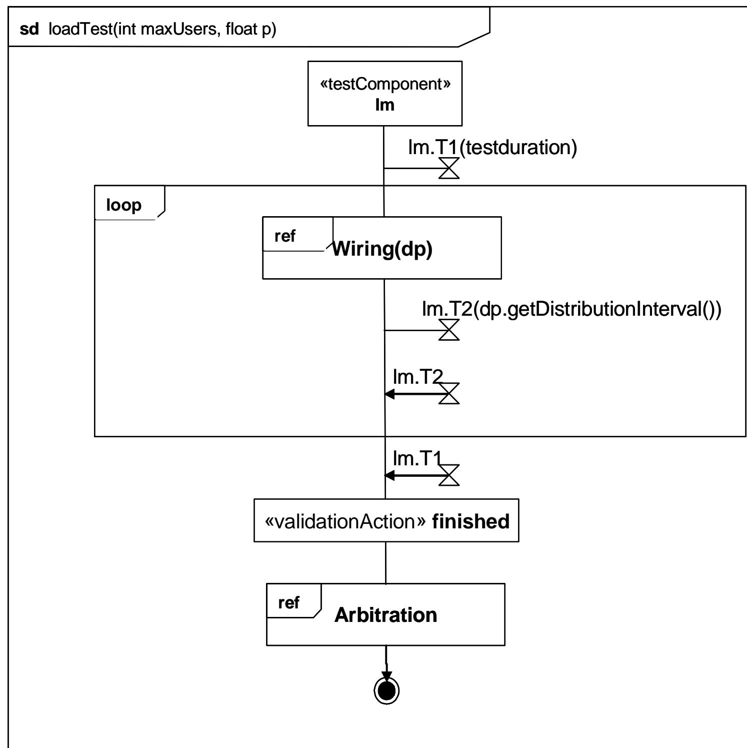


Figure D.25 - Main test behavior

Figure D.26 illustrates how an individual US wiring transaction is executed. The behavior is parameterized with the data pool dp. First, an instance of the transaction controller is created and started. Then the test is started and a transaction is started using either data taken from the getEUTrxnData data partition or from the getUSTrxnData partition. The timing constraint stipulates that the execution of the transaction should take less than 4 seconds, otherwise the default handler will inform the Load Arbiter that the transaction has failed.

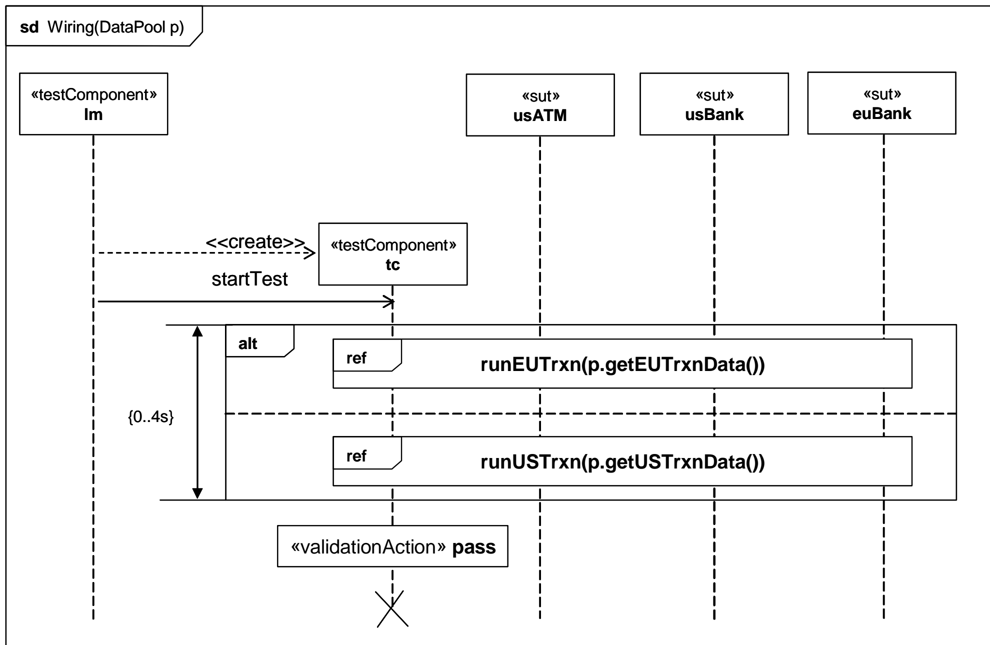


Figure D.26 - US initiated wiring transaction

The Arbitration behavior is given in Figure D.27. A pass verdict is assigned if the number of successful tests exceeds the given threshold, otherwise fail is assigned.

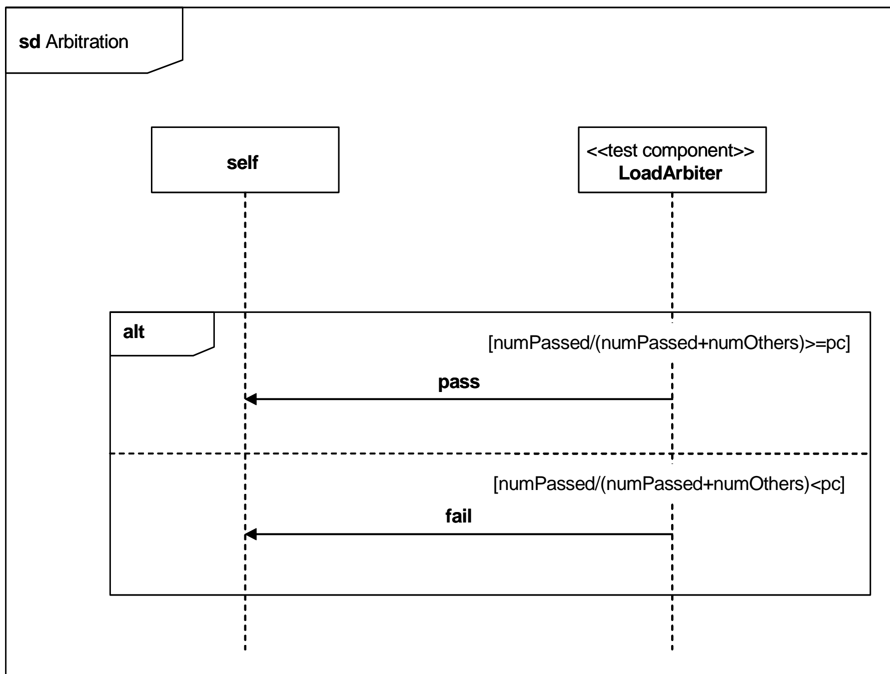


Figure D.27 - Arbitration behavior

Figure D.28 illustrates how the system obtains balance information from the banks prior to each transaction, as well as after each one is complete. These are used to validate that the transfer took place correctly. The information on the percentage of test cases that should be successful is used by the arbiter to determine whether or not the load test was successful.

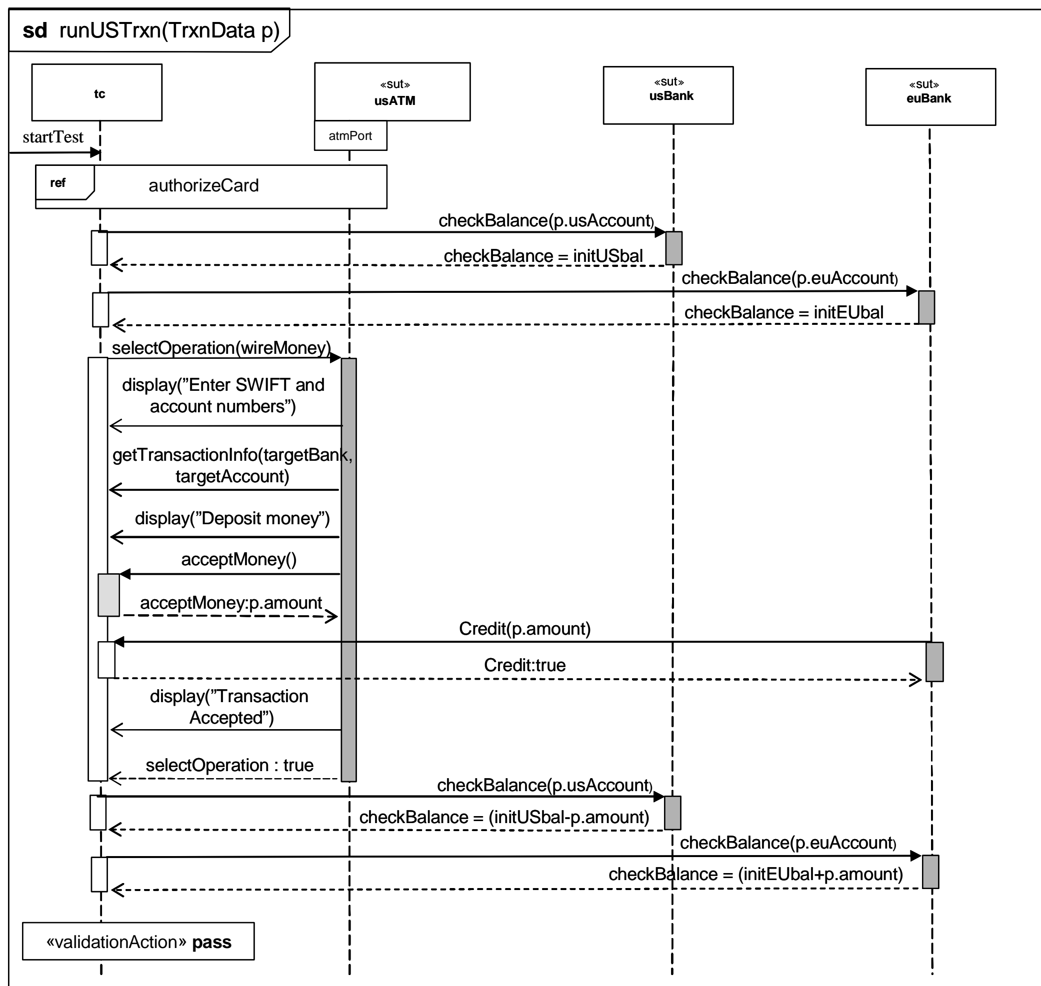


Figure D.28 - Transaction detail

Figure D.29 illustrates the behavior of the LoadArbiter component, which determines whether or not the load test was successful.

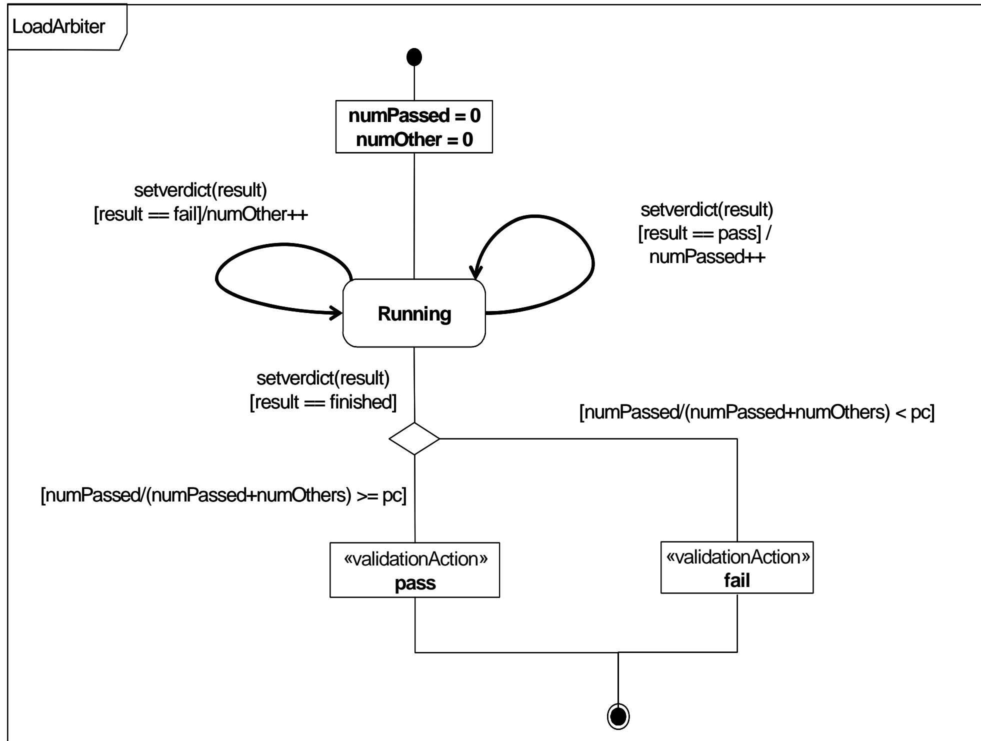


Figure D.29 - LoadArbiter behavior

D.5 Risk Integration Example

D.5.1 Introduction

Fundamental too many test organizations is the idea of risk based testing within test management. Typically management considers programmatic risk (e.g., cost or schedule), but this annex addresses product technical risk as related to modeling and prioritization of testing. This annex presents an example concept for modeling risks using existing UML, SysML, and UTP constructs. It is provided as information and reference to aid test modelers. Further examples of risk modeling are also defined in the [BMM].

Modeling risks to support risk-based testing can aid in the following:

1. Identifying and selecting test cases,

2. Priority of test cases,

3. Overall test development,

4. Mitigation and opportunity planning,

5. Capturing rationales for review by stakeholders,

6. Assessment of system and software model elements linked to risk, and
7. Traceability to/from these items and requirements.

D.5.2 Basics

As part of test management, carrying out risk-based testing can be done so that the risks with the highest priority are paid the highest attention during testing. This can aid in test prioritization, scheduling, and other management activities. A risk is typically expressed with two aspects:

1. Impact, effect, or consequence of the risk – See Figure D.30 in example, Risk Tag “cost of occurrence.”
2. Probability and/or frequency of occurrence of the risk – See Figure D.30 in example, Risk Tag “probability of occurrence.”

Each of the aspects can be accounted for in different classification schemes, including:

- ordinal (e.g., 1 to 10),
- numeric (e.g., dollars or percentages as in example), or
- text ranking (e.g., low, medium, high).

With the use of comment tags with string values as shown in the example given below, different risk classifications can be accounted. A compilation of impact and probability can then be used to create a categorized risk profile, which is the identified set of risks and their classifications. The categorized risk profile is usually ordered, prioritized, or ranked from “high” to “low,” corresponding to where the risks may cause impact or not. These elements can be important to include in a test model.

The categorized risk profile can be used in test management on the project. For instance, the categorized risk profile can be used to determine:

- The rigor of testing (e.g., test phases, test techniques, test completion criteria, number of tests, etc.).
- The prioritization of risks may be used to determine the test schedule (e.g., testing associated with higher priority risks are typically addressed earlier).
- The type of risk can be used to decide the most appropriate types of testing to perform (e.g., safety, security, performance, etc).

When performing risk-based testing, risk profiling driven by models can be used to identify and score risks, so that the perceived risks in the development and delivered system can be scored, prioritized, and categorized to support testing. Finally, a risk based test model can share information with other tools, efforts, and assessments, e.g., risk management tools, test tools, management systems, etc.

D.5.3 Example

Figure D.30 presents an example risk model for reference. This is modeled with the “comment” construct.

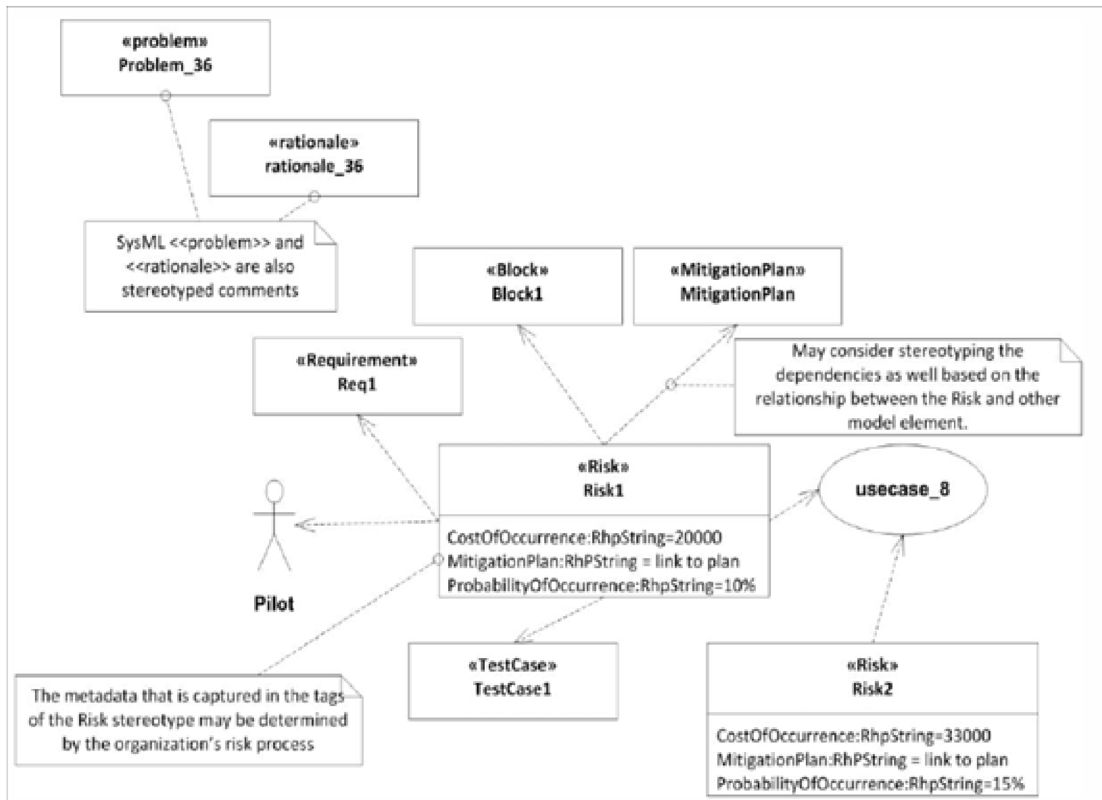


Figure D.30 - Example of modeling a risk with existing model construct

Product risks can relate to model constructs (be traced to or from as shown in Figure D.30) requirements, design information, plans, test cases, etc. Product risks jeopardize the customer satisfaction, business success, usability, safety, etc, and so in model based testing must be accounted for.

From a risk model, a risk profile can be generated (Figure D.31). Such profiles can be exported to other tools including risk management tools or test tools.

Name	CostOfOccurance	MitigationPlan	ProbabilityOfOccurance	Description
Risk1	20,000.00	link to plan...	10%	Description of Risk1 and link to external
Risk2	33,000.00	link to plan...	15%	Description of Risk2

Figure D.31 - Risk profile example

The nature and level of risk modeling is project unique.

Annex E - Stereotype and Type Overview

(normative)

This annex provides an overview of all types and stereotypes of the UML Testing Profile.

E.1 Graphical Summary

The following sub clause depicts a summary of the UML Testing Profile's types and stereotypes graphical syntax.

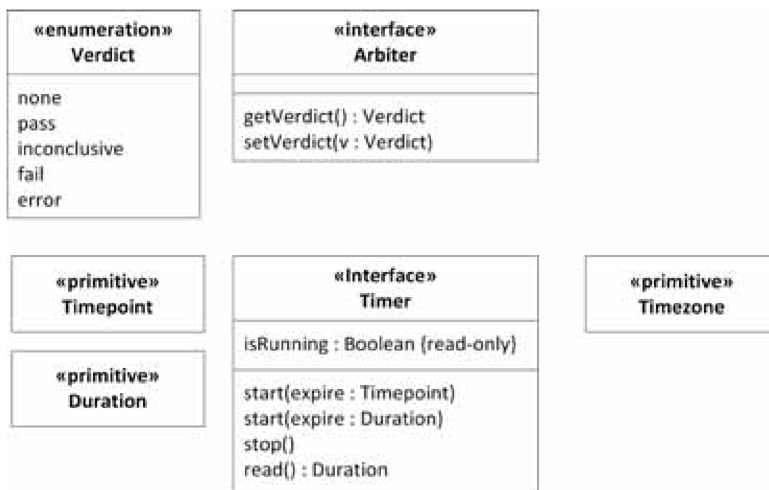


Figure E.1 - Abstract Syntax Predefined Type Library

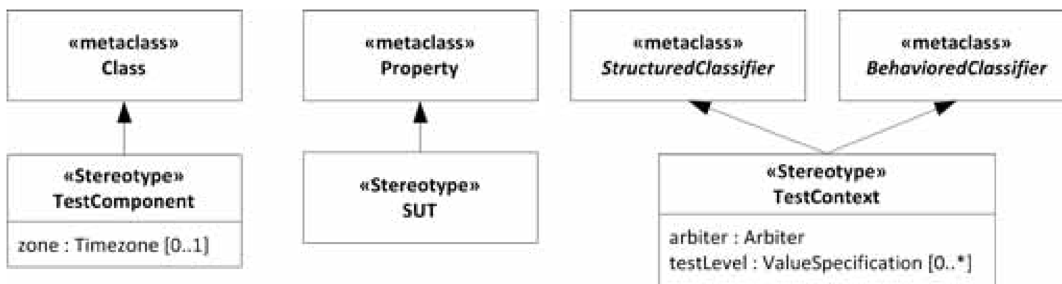


Figure E.2 - Abstract Syntax Test Architecture

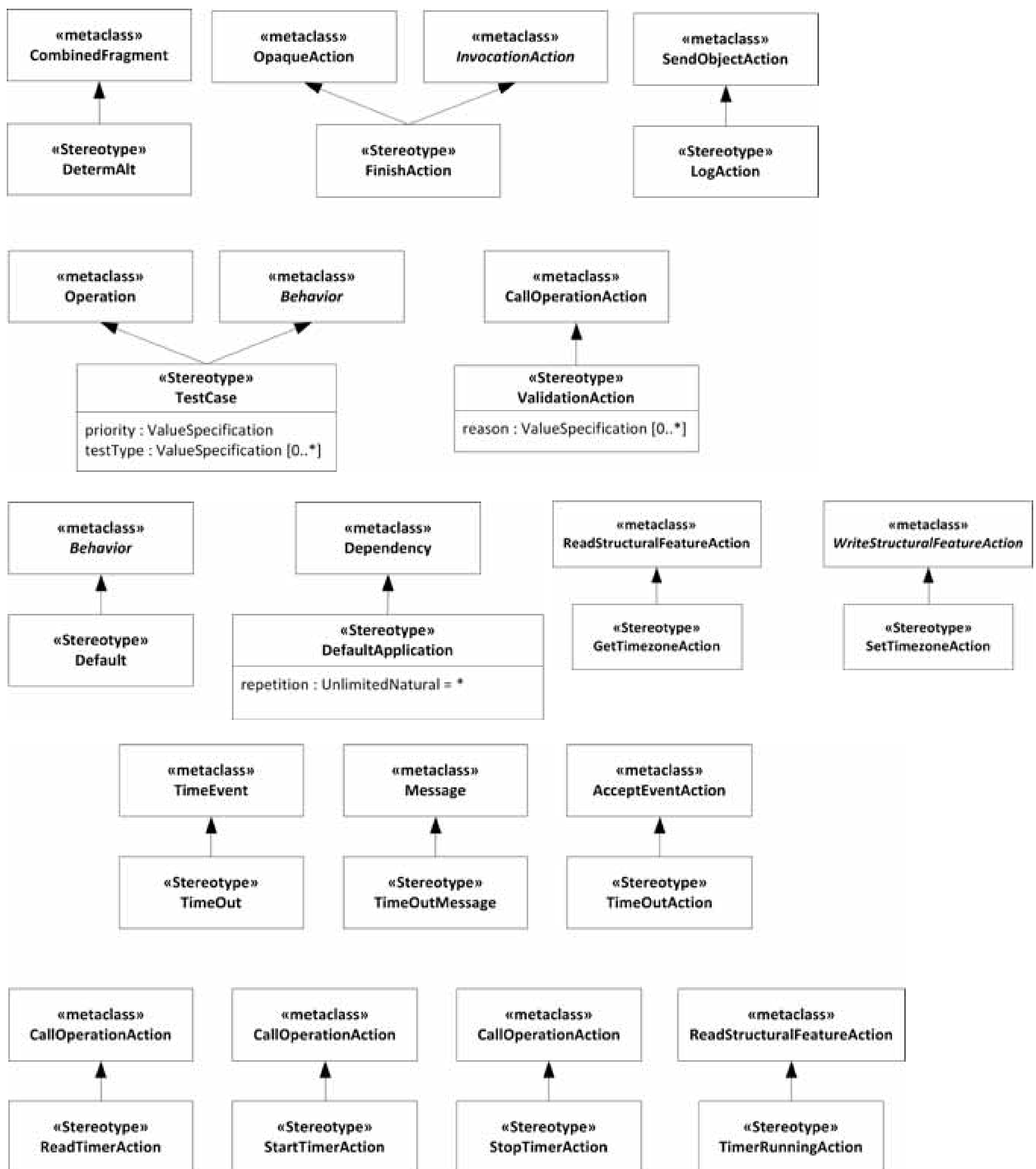


Figure E.3 - Abstract Syntax Test Behavior

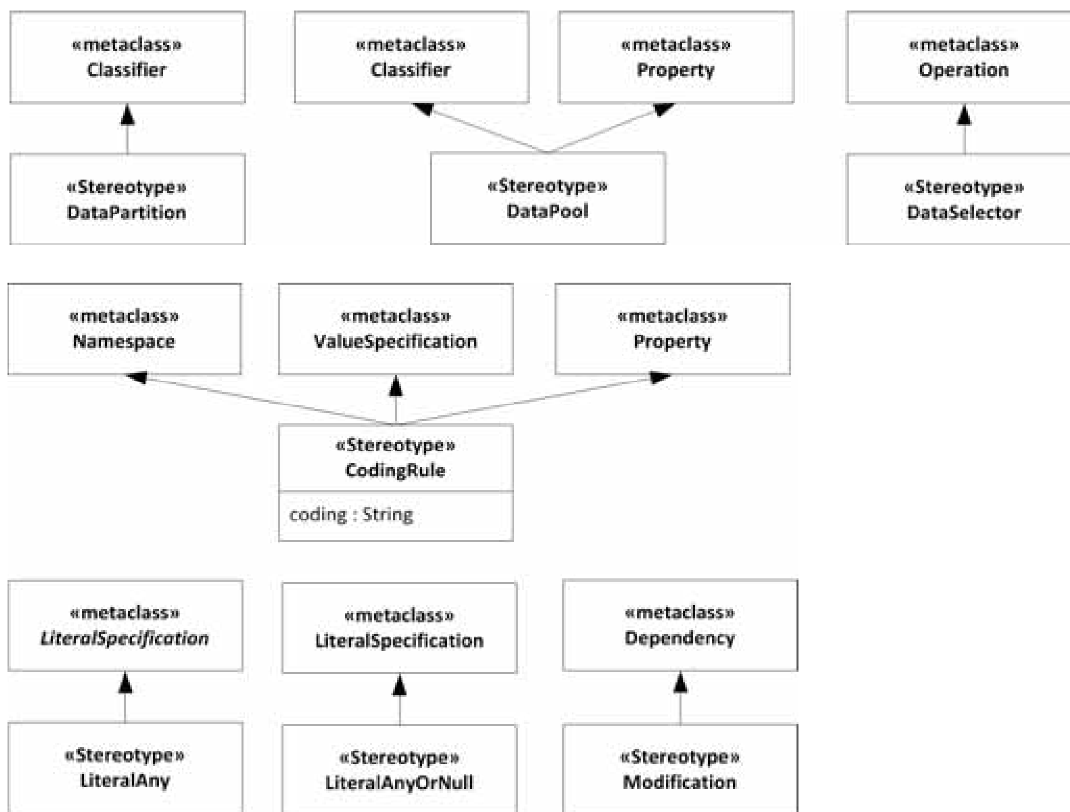


Figure E.4 - Abstract Syntax Test Data

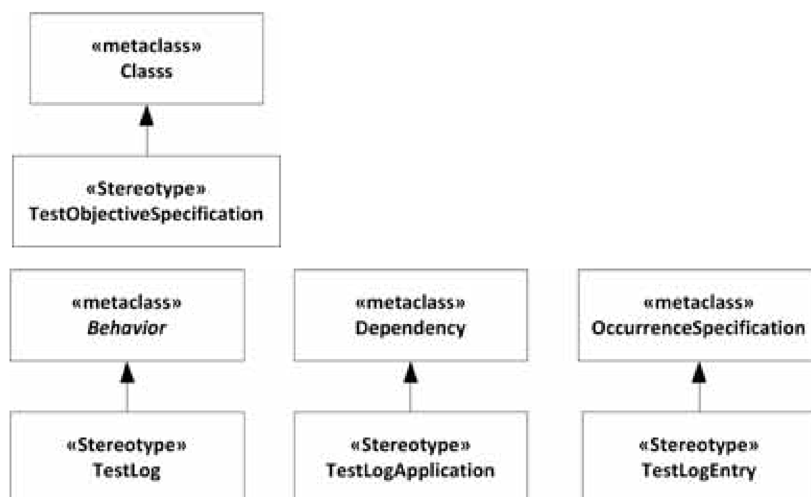


Figure E.5 - Abstract Syntax Test Management

E.2 Tabular Summary

The following sub clause summarizes the UML Testing Profile's types and stereotypes in a tabular manner.

Table E.1 - Comparison between the UML Testing Profile and JUnit concepts

UML Testing Profile Concept	Metaclass/Type
Arbiter	Interface
CodingRule	*Namespace*, *ValueSpecification*, Property
DataPartition	*Classifier*
DataPool	*Classifier*, Property
DataSelector	Operation
Default	*Behavior*
DefaultApplication	Dependency
DetermAlt	CombinedFragment
Duration	PrimitiveType
FinishAction	*InvocationAction*, OpaqueAction
GetTimezoneAction	ReadStructuralFeatureAction
LogAction	SendObjectAction
Modification	Dependency
ReadTimerAction	ReadStructuralFeatureAction
SetTimezoneAction	*WriteStructuralFeatureAction*
StartTimerAction	CallOperationAction
StopTimerAction	CallOperationAction
SUT	Property
TestCase	Operation, *Behavior*
TestComponent	Class
TestContext	*BehavioredClassifier, StructuredClassifier*
TestLog	Behavior
TestLogApplication	Dependency
TestLogEntry	OccurrenceSpecification
TestObjectiveSpecification	Class
Timepoint	PrimitiveType
Timer	Interface
TimeOut	TimeEvent
TimeOutAction	AcceptEventAction

Table E.1 - Comparison between the UML Testing Profile and JUnit concepts

UML Testing Profile Concept	Metaclass/Type
TimeOutMessage	Message
TimerRunningAction	ReadStructuralFeatureAction
Timezone	PrimitiveType
ValidationAction	CallOperationAction
Verdict	Enumeration

Annex F - MOF-based Metamodel

(non-normative)

Note: The UTP MOF-based metamodel is obsolete since version 1.1 and will not be further maintained.

F.1 Metaclass Definitions

This sub clause provides a standalone metamodel for the UML Testing Profile. This metamodel is an instance of the MOF metamodel, providing the ability for MOF based tools to comply with the UML Testing Profile standard. The compliance provided by the MOF-based metamodel is limited to the architecture elements of the UML Testing Profile enabling traceability and management of test assets across tools. Specifically, the behavioral aspects of the UML Testing Profile are left out of the current metamodel as they would not provide any substantial improvement over this goal, while requiring a significant portion of the UML 2.0 metamodel to be included in the standalone metamodel.

We present the MOF metamodel diagrams and provide more detailed information as it pertains to the metamodel. A majority of the concepts from the UML Testing Profile are also present in the metamodel. Except for the information regarding profile alignment with the UML 2 Superstructure Adopted Specification, the information provided in the profile section (section 2.3) also applies to the MOF based metamodel unless we provide other information and state that it supersedes the information from the profile section.

Figure F.4 provides three interfaces that are not technically part of the MOF-based metamodel. These are used to describe the semantics that should be provided by instances of the Timer, Arbiter, and Scheduler metaclasses.

F.1.1 Test Architecture and Test Behavior

The test architecture concepts are related to the organization and realization of a set of related test cases. These include test contexts, which consist of one or more related test cases. The test cases are potentially realized by test components, and the verdict of a test case is assigned by an arbiter.

Similarly, test behavior concepts describe the behavior of the test cases that are defined within a test context. Associated with test cases are test objectives, which describe the capabilities the test case is supposed to validate. Test cases consist of behavior that includes validation actions, which update the verdict of a test case, and log actions that write information to a test log. Behavioral concepts also include the verdicts that are used to define the test case outcomes, and default behavior that is applicable when the something other than the specified behavior is observed.

Figure F.1 presents the metamodel diagram for the Test Architecture and Test Behavior concepts.

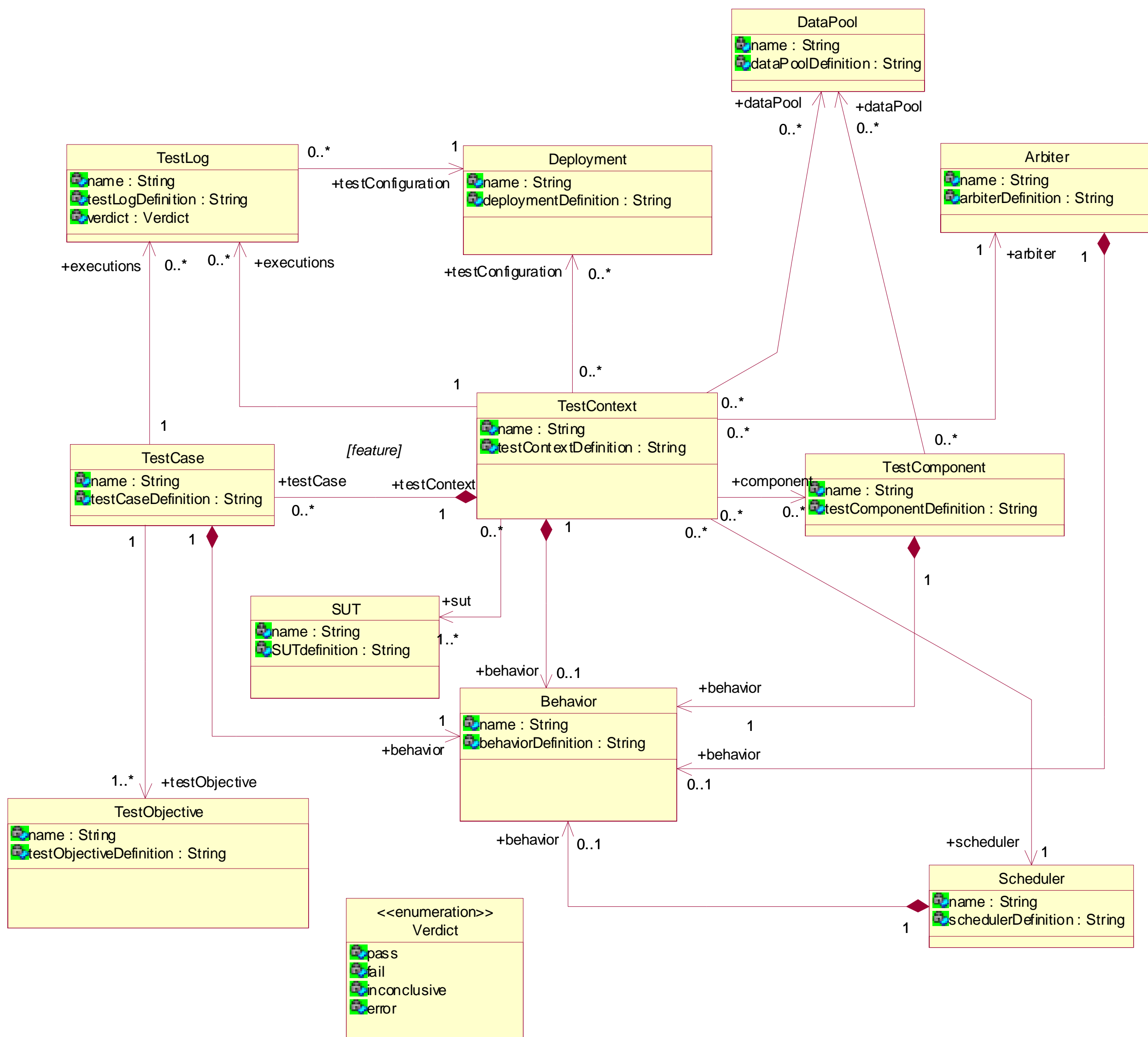


Figure F.1 - Test architecture and test behavior portion of the MOF-based metamodel

F.1.1.1 Behavior

Semantics

Behavior represents the dynamic behavior of a test context, test case, or test component in the testing system. In the MOF metamodel, it is a high level concept that allows the programmed behavior of the aforementioned elements to be explicitly referenced.

Attributes

- name: String [1]— The name of the behavior.
- behaviorDefinition: String [1]— The definition of the behavior.

F.1.1.2 TestContext

Semantics

A test context contains a set of zero or more test cases. The test cases are realized within the context by instances of test components that are deployed and act against the SUT. The test context has behavior, which is used to control the execution of the test cases that the context owns.

Associations

- testConfiguration:Deployement[0..*]
  A deployment describing the configuration of test components and SUT.
- testCase:TestCase[0..*]
  The collection of test cases owned by the test context.
- sut:SUT[1..*]
  The elements representing the system under test.
- behavior:Behavior[0..*]
  The behavior of the test context, which is used to control test case execution.
- component:TestComponent[0..*]
  The collection of test components that realize the test cases within the test context.
- arbiter:Arbiter[1]
  The implementation of the IArbiter interface used to determine the verdicts of the test cases within the test context.
- executions:TestLog[0..*]
  Traced elements representing the logged information for each execution of a test context.
- dataPool:DataPool[0..*]
  Data pools associated to the test context.

Attributes

- name: String [1]
  The name of the test context.
- testContextDefinition: String [1]
  The definition of the test context (in addition to the behavior definition of the test context).

F.1.1.3 SUT

Semantics

The SUT is a black-box from the point of view of the test specification. Thus, test components only have access to the public operations defined on the interface of the SUT.

Attributes

- name: String [1]
  The name of the SUT.
- SUTdefinition: String [1]
  The definition of the SUT.

F.1.1.4 TestComponent

Semantics

Zero or more test components realize the behavior associated with a test case. A test component executes sequences of stimuli and observations against the SUT. It can also take part in coordination with other components. Whenever a test component performs a validation action, the arbiter is notified of the outcome so that it may update the test case verdict if necessary. The timezone is an attribute that is available at runtime.

Associations

- behavior:Behavior[1]
  The behavior of the test component.
- zone:Timezone [0..1]
  Specifies the timezone to which a test component belongs.
- dataPool:DataPool[0..*]
  Data pools associated to the test component.

Attributes

- name: String [1]
  The name of the test component.
- testComponentDefinition: String [1]
  The definition of the test component (in addition to the behavior definition of the test component).

F.1.1.5 Arbiter

Semantics

An arbiter is an entity within a test context which is capable of determining the final verdict of a test case based on input from the test components realizing the test case. An instance of the Arbiter metaclass should provide the operations defined by the IArbiter interface shown in Figure F.4.

Associations

- behavior:Behavior[1]
  The behavior of the arbiter.

Attributes

- name: String [1]
  The name of the arbiter.
- arbiterDefinition: String [1]
  The definition of the arbiter (in addition to the behavior definition of the arbiter).

F.1.1.6 Scheduler

Semantics

A scheduler is an entity within a test context that controls the running of the test cases. It will keep track of the creation and destruction of test components and give instructions to the existing test components when to start executing a given test case. It will communicate with the arbiter when the time is right to produce the verdict for a test case.

Associations

- behavior:Behavior[1]
  The behavior of the scheduler.

Attributes

- name: String [1]
  The name of the scheduler.
- schedulerDefinition: String [1]
  The definition of the scheduler (in addition to the behavior definition of the scheduler).

F.1.1.7 TestCase

Semantics

A test case is a set of behavior performed against the SUT and owned by a test context. Test cases have access to all elements in a test context, including the SUT elements and test components. A test case produces a log containing all log actions and the verdict.

Associations

- behavior:Behavior[1]
  The dynamic behavior of the test case.
- executions:TestLog[0..*]
  Log elements representing the logged information for each execution of a test case.
- testObjective:TestObjective[1..*]
  A test objective is a description of the capability being validated by the test case.
- testContext:TestContext[1]
  The test context to which the test case belongs.

Attributes

- name: String [1]
  The name of the test case.
- testCaseDefinition: String [1]
  The definition of the test case (in addition to the behavior definition of the test case).

F.1.1.8 TestObjective

Semantics

A test objective is a dependency to another element that describes the purpose of a test case. Test objectives have no dynamic behavior, and only serve to describe the rationale for a test case.

Attributes

- name: String [1]
  The name of the test objective.
- arbiterDefinition: String [1]
  The definition of the test objective.

F.1.1.9 Verdict

Verdict is defined exactly as in “Verdict” on page 17.

F.1.1.10 Deployment

Semantics

The deployment represents a configuration of test components and the SUT. It specifies the execution architecture for a test context. In the profile, deployment concepts and capabilities come directly from the UML 2 Superstructure Adopted Specification.

F.1.1.11 TestLog

Semantics

The log represents a snapshot of the execution of a test case. It contains deployment information, the ordered set of log actions performed during the test case, and the final verdict.

Associations

- testConfiguration:Deployment[1]
  The deployment information for the test case producing this log.

Attributes

- name: String [1]
  The name of the test log.
- testLogDefinition: String [1]
  The definition of the test log (in addition to the behavior definition of the test log).
- verdict:Verdict [1]
  The final verdict achieved by the test case producing this log.

F.1.2 Test Data

The test data concepts provide the capabilities to specify data values, associate specific encodings with data values, and specify wildcards such as “?” (any value) and “*” (any value including none). These rudimentary features are all that are required in the UML Testing Profile, as higher level constructs such as data pools can be defined as test components within the standard or provided by specific tool implementations.

Figure F.2 is a diagram illustrating the data concepts.

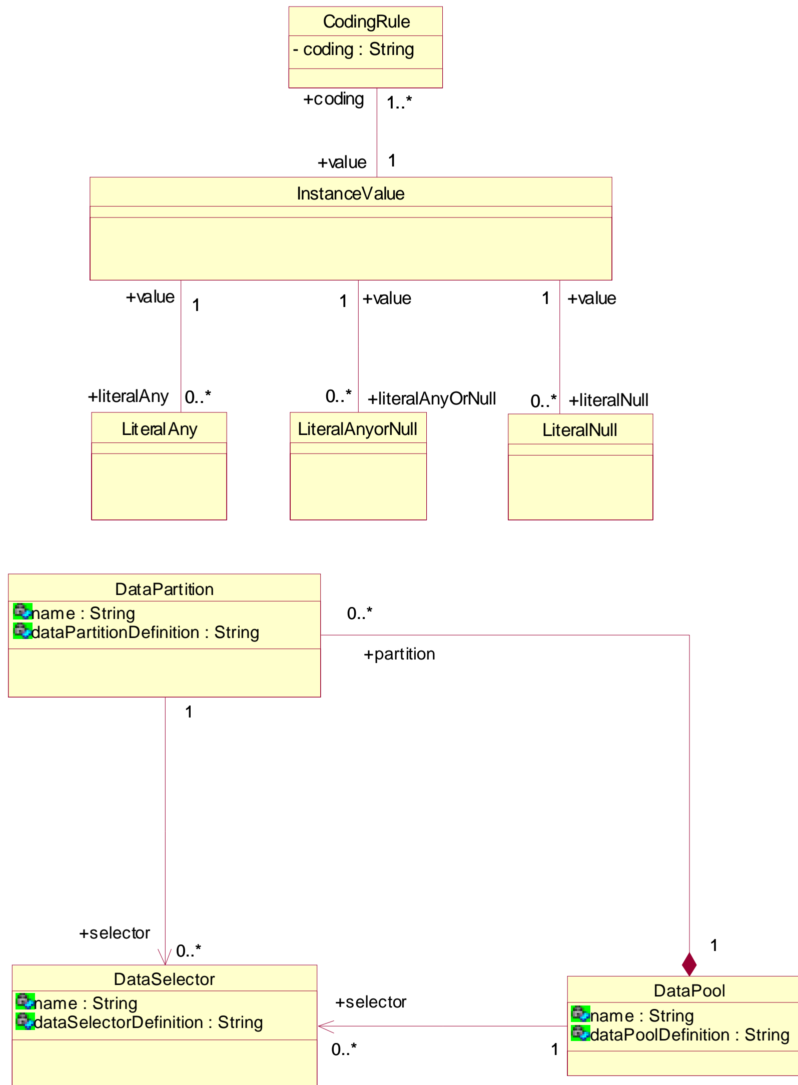


Figure F.2 - Test data portion of the MOF-based metamodel

F.1.2.1 InstanceValue

Semantics

An instance value is a specification of a data instance for use in the test context. It is based explicitly on the InstanceValue concept from the UML 2 Superstructure Adopted Specification.

Associations

- coding:CodingRule [1..*]
  The set of coding rules that are applied to the instance value.

- literalAny:LiteralAny [0..*]
  A "?" character indicating that the instance value can be any non-null value.
- literalAnyOrNull:LiteralAnyOrNull [0..*]
  A "*" character indicating that the instance value can be any value (including null.)
- literalNull:LiteralNull[0..*]
  A "-" character indicating that the instance value can be null.

F.1.2.2 CodingRule

Semantics

A rule that specifies how an instance value is to be encoded. As defined in "CodingRule" on page 51, it contains a string attribute that specifies the coding scheme.

Associations

- value: InstanceValue [1]
  The instance value to which the coding rule applies.

F.1.2.3 LiteralAny

Semantics

A "?" symbol specifying that any value except null may be associated with a particular instance value.

Associations

- value: InstanceValue [1]
  The instance value to which the specification applies.

F.1.2.4 LiteralAnyOrNull

Semantics

A "*" symbol specifying that any value (including null) can be associated with a particular instance value.

Associations

- value: InstanceValue [1]
  The instance value to which the specification applies.

F.1.2.5 LiteralNull

Semantics

A "-" symbol specifying the absence of a value for a particular instance value.

Associations

- value: InstanceValue [1]
  The instance value to which the specification applies.

F.1.2.6 DataPool

Semantics

Zero or more data pools can be associated to test contexts or test components. Data pools specify a container for explicit values or data partitions. They provide an explicit means for associating data values for repeated tests (e.g., values from a database, etc.), and equivalence classes that can be used to define abstract data sets for test evaluation.

Associations

- partition:DataPartition[0..*]
  A set of data partitions defined for this data pool.
- selector:DataSelector[0..*]
  A set of data selectors defined for this data pool.

Attributes

- name: String [1]
  The name of the data pool.
- dataPoolDefinition: String [1]
  The definition of the data pool.

F.1.2.7 DataPartition

Semantics

Zero or more data partitions can be defined for a data pool. A data partition is a container for a set of values. These data sets are used as abstract equivalence classes within test context and test case behaviors.

Associations

- selector:DataSelector[0..*]
  A set of data selectors defined for this data partition.

Attributes

- name: String [1]
  The name of the data partition.
- dataPartitionDefinition: String [1]
  The definition of the data partition.

F.1.2.8 DataSelector

Semantics

Zero or more data selectors can be defined for data pools or data partitions. Data selectors allow the definition of different data selection strategies.

Attributes

- name: String [1]
  The name of the data selector.
- dataSelectorDefinition: String [1]
  The definition of the data selector.

F.1.3 Time

As discussed in “Timepoint” on page 15,” key time concepts are required to allow the specification of complete and precise test contexts and components. Within the MOF-aligned metamodel, we provide a subset of the timing concepts defined in the profile. This is because many of the actions defined in the profile are not required in MOF-aligned model as they can be implemented directly in tools realizing the metamodel. The concepts included in the model include two primitive types (time and duration), as well as timers and timezones.

Figure F.3 illustrates the time concepts for the metamodel.

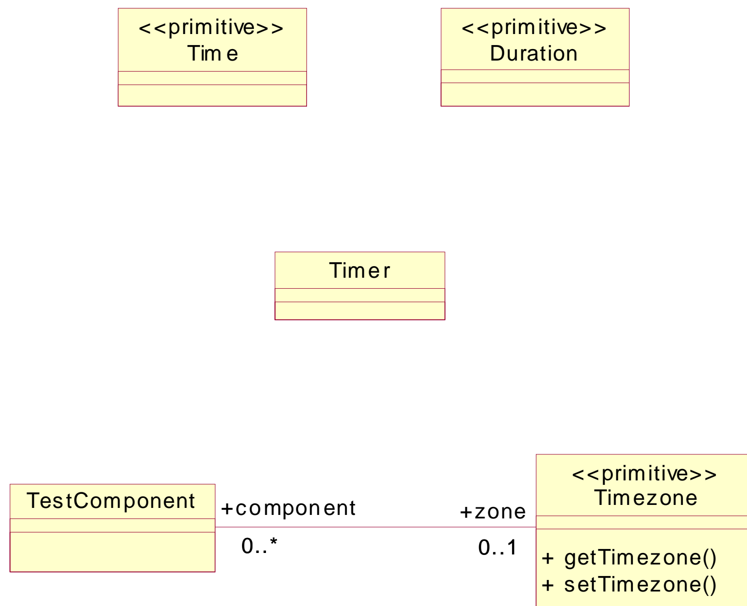


Figure F.3 - Time portion of the MOF-based metamodel

F.1.3.1 Time

Time is defined exactly as in “Timepoint” on page 15.”

F.1.3.2 Duration

Duration is defined exactly as in “Duration” on page 14.”

F.1.3.3 Timer

Semantics

A timer is an element that can provide information regarding time to a test component. It can be started, stopped, queried for the current time, and queried to determine whether or not it is running. Instances of the Timer metaclass should provide these capabilities by implementing the ITimer interface defined in Figure F.4.

F.1.3.4 Timezone

Semantics

Timezone is a primitive type that serves as a grouping concept for test components. Test components that are associated with the same timezone are synchronized and can share time values.

Associations

- component:TestComponent [0..*]
  The set of test components belonging to the timezone.

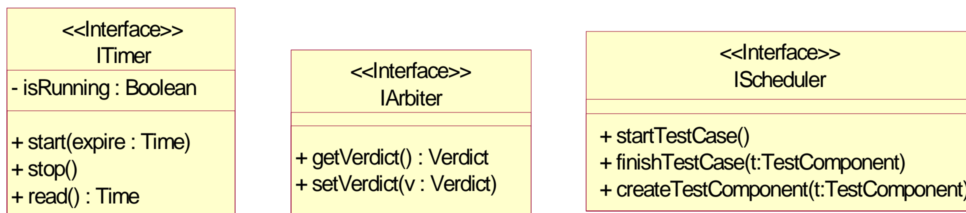


Figure F.4 - Predefined interfaces for implementing timer, arbiter and scheduler

Annex G - Scheduling

(informative)

G.1 Scheduling

G.1.1 Interface Definition

G.1.1.1 Scheduler

Description

Scheduler is a predefined interface defining operations used for controlling the tests and the test components. None of the operations of the Scheduler are available to the UML specifier.

Generalizations

None.

Attributes

None.

Operations

- Scheduler()
  The constructor of Scheduler. It will start the SUT and the Arbiter.
- startTestCase()
  The scheduler will start the test case by notifying all involved test components.
- finishTestCase(t:TestComponent)
  Records that the test component t has finished its execution of this test case.
  When all test components involved in the current test case have finished, the arbiter will be notified.
- createTestComponent(t:TestComponent)
  Records that the test component t has been created by some other test component.

Semantics

The implementation of the predefined interface will determine the detailed semantics. The implementation must make sure that the scheduler has enough information to keep track of the existence and participation of the test components in every test case. The test context itself will ensure the creation of a scheduler.

Constraints

None.

Notation

No additional notation for Scheduler is defined.

Examples

Example protocols for how the scheduler could work are found in Annex C.

G.1.2 Scheduler and Arbiter Protocol

This sub clause shows how the Scheduler and the Arbiter should work together with the test components and the SUT such that the execution of the test cases are performed properly and the verdict delivered at the right time. It is not necessary that a valid implementation conforms in detail to these sequence diagrams, but the net effect should be the same.

In the diagrams we have, in addition to standard notation, also applied a special kind of continuations that we have called synchronous continuations denoted by the prefixing “synch” keyword. This is a shorthand for introducing a number of general ordering relations such that all events above the synchronous continuation precedes all events below the synchronous continuation. This holds for all events on those lifelines that are covered by the continuation.

The protocol scenario in Figure G.1 shows the normal situation when the test components are static. Notice that the Scheduler starts the SUT and the Arbiter explicitly before any test case can be started. When a test case is initiated the Scheduler will give notice to the test components that are involved in it. They will in turn notify the Scheduler when they are at the end of executing that test case. None of this is seen explicitly in the user specifications. Finally when the Scheduler has recorded that no test components have pending results, the Arbiter will be asked to produce the final verdict for the test case.

The scenario in Figure G.2 shows what should happen when test components are created dynamically within the execution of the test case. The test component that creates another test component will notify the Scheduler about the creation. Notice that the later notification from that test component that it is finished with the test case must go along the same communication channel as the creation notification. This is to avoid possible race conditions between the creation notification and the finishing notifications. Such race conditions would have made it theoretically possible to create a situation where the Scheduler knows about no pending test components, while the newly created test component is still running. The Arbiter could therefore have been instructed to give final verdict before it should.

In case of test component termination (destruction) this must also be notified to the Scheduler by the test component. It is assumed that the test component being destroyed is able to transmit its last verdict to the Arbiter before it is deleted.

In some test cases not all existing test components will take part. It is assumed that the Scheduler has proper information about this from its description of the test case such that it will not initiate more test components than necessary for a particular test case.

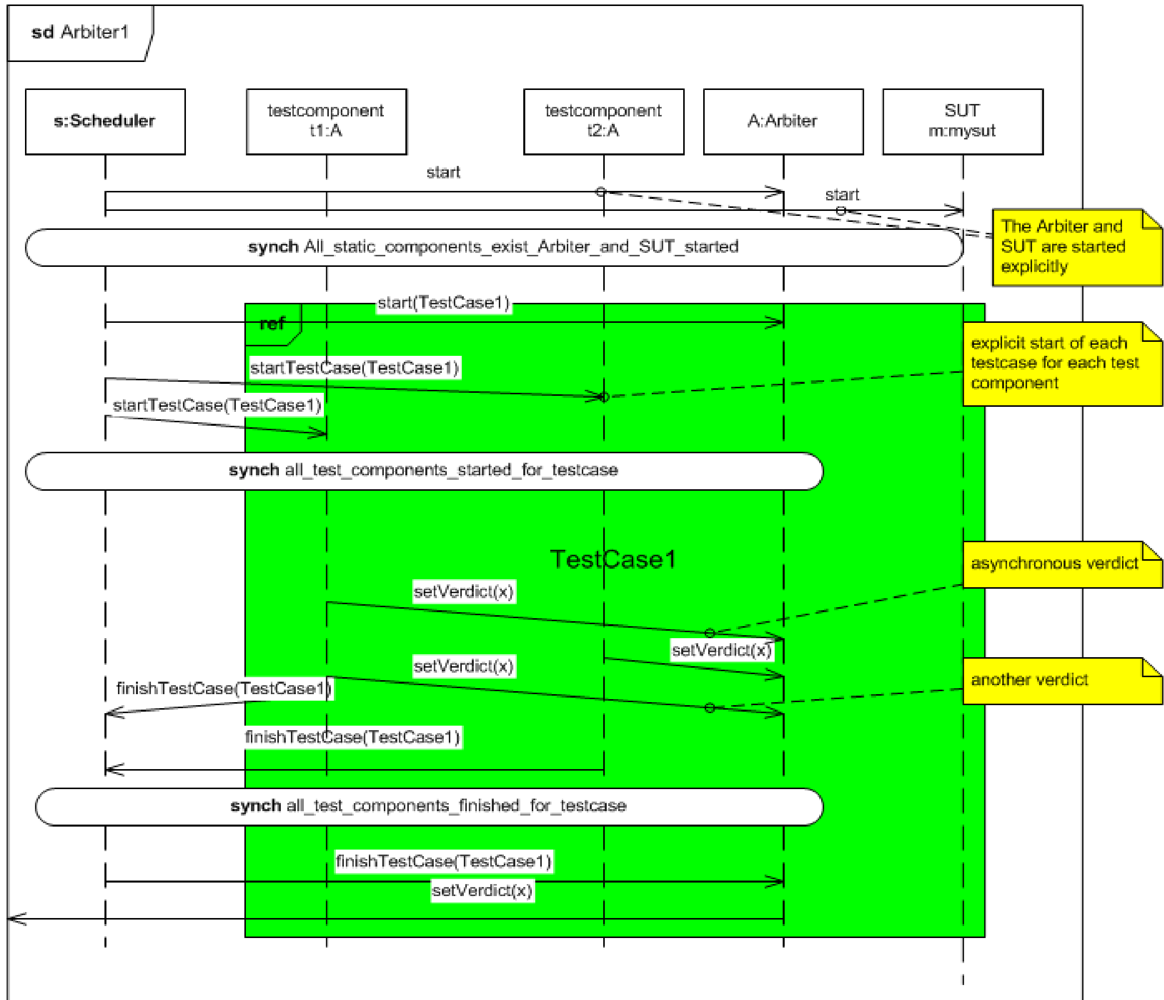


Figure G.1 - Scheduler/Arbiter protocol #1

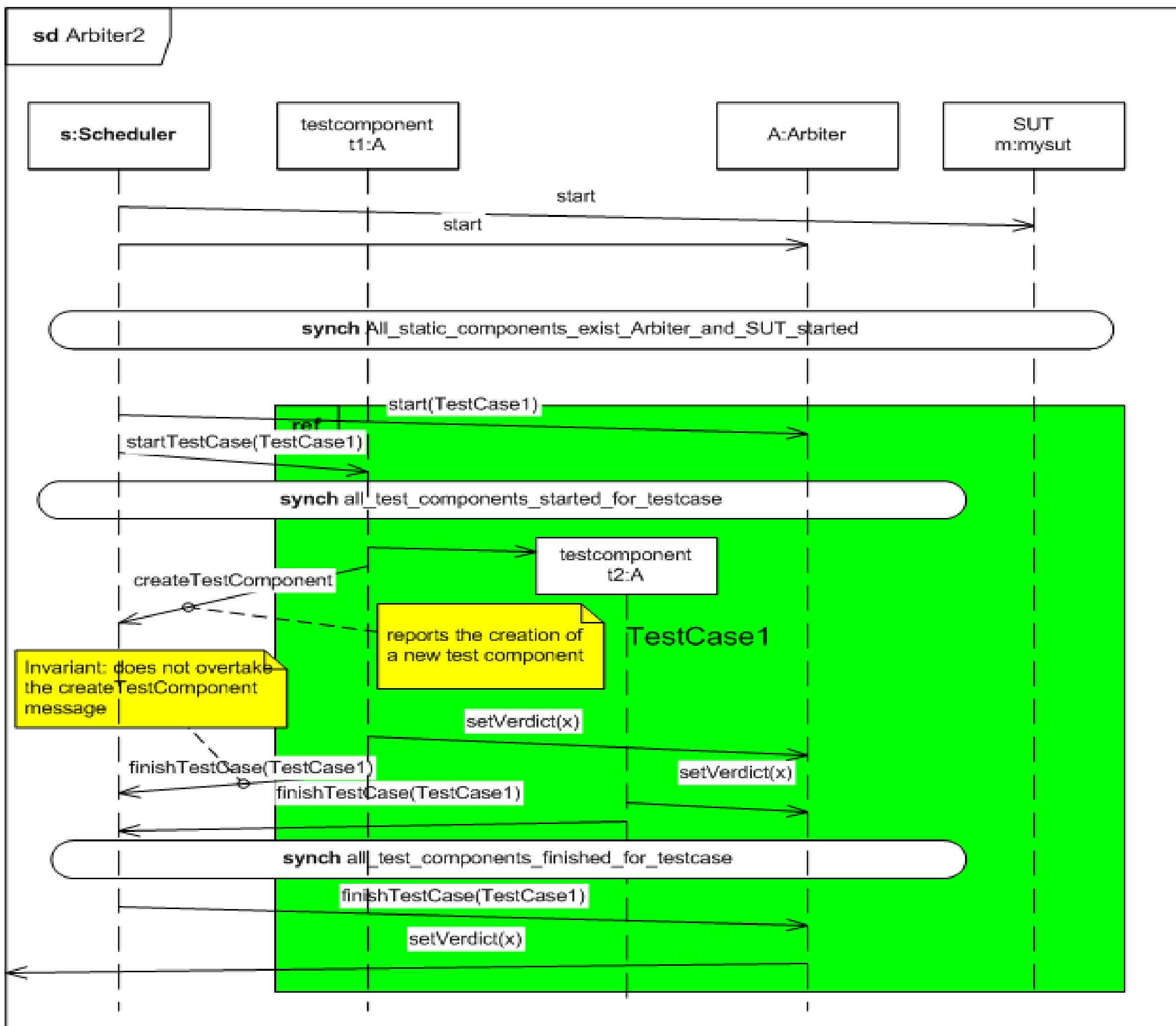


Figure G.2 - Scheduler/Arbiter protocol #2

Annex H - XMI Schema

(normative)

H.1 The Profile

The XMI schema definition for the exchange of UTP profile specifications follows the XMI schema definition of UML 2.4.1 for UML 2.4.1 profiles.

H.2 The MOF-based Metamodel

The XMI schema definition for the MOF-based metamodel is given below.

```
<?xml version="1.0"?>
<xsd:schema targetNamespace="http://www.omg.org/UTPSA"
    xmlns:utp="http://www.omg.org/UTPSA"
    xmlns:xmi="http://www.omg.org/XMI" xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:import namespace="http://www.omg.org/XMI" schemaLocation="XMI.xsd"/>
    <xsd:simpleType name="Verdict">
        <xsd:restriction base="xsd:NCName">
            <xsd:enumeration value="pass"/>
            <xsd:enumeration value="fail"/>
            <xsd:enumeration value="inconclusive"/>
            <xsd:enumeration value="error"/>
        </xsd:restriction>
    </xsd:simpleType>
    <xsd:complexType name="Arbiter">
        <xsd:choice maxOccurs="unbounded" minOccurs="0">
            <xsd:element name="name" type="xsd:string"/>
            <xsd:element name="arbiterDefinition" type="xsd:string"/>
            <xsd:element name="behavior" type="utp:Behavior"/>
            <xsd:element ref="xmi:Extension"/>
        </xsd:choice>
        <xsd:attribute ref="xmi:id"/>
        <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
        <xsd:attribute name="name" type="xsd:string"/>
        <xsd:attribute name="arbiterDefinition" type="xsd:string"/>
    </xsd:complexType>
    <xsd:element name="Arbiter" type="utp:Arbiter"/>
    <xsd:complexType name="Scheduler">
        <xsd:choice maxOccurs="unbounded" minOccurs="0">
            <xsd:element name="name" type="xsd:string"/>
            <xsd:element name="schedulerDefinition" type="xsd:string"/>
            <xsd:element name="behavior" type="utp:Behavior"/>
            <xsd:element ref="xmi:Extension"/>
        </xsd:choice>
        <xsd:attribute ref="xmi:id"/>
```

```
        <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
        <xsd:attribute name="name" type="xsd:string"/>
        <xsd:attribute name="schedulerDefinition" type="xsd:string"/>
    </xsd:complexType>
    <xsd:element name="Scheduler" type="utp:Scheduler"/>
    <xsd:complexType name="Deployment">
        <xsd:choice maxOccurs="unbounded" minOccurs="0">
            <xsd:element name="name" type="xsd:string"/>
            <xsd:element name="deploymentDefinition" type="xsd:string"/>
            <xsd:element ref="xmi:Extension"/>
        </xsd:choice>
        <xsd:attribute ref="xmi:id"/>
        <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
        <xsd:attribute name="name" type="xsd:string"/>
        <xsd:attribute name="deploymentDefinition" type="xsd:string"/>
    </xsd:complexType>
    <xsd:element name="Deployment" type="utp:Deployment"/>
    <xsd:complexType name="SUT">
        <xsd:choice maxOccurs="unbounded" minOccurs="0">
            <xsd:element name="name" type="xsd:string"/>
            <xsd:element name="SUTdefinition" type="xsd:string"/>
            <xsd:element ref="xmi:Extension"/>
        </xsd:choice>
        <xsd:attribute ref="xmi:id"/>
        <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
        <xsd:attribute name="name" type="xsd:string"/>
        <xsd:attribute name="SUTdefinition" type="xsd:string"/>
    </xsd:complexType>
    <xsd:element name="SUT" type="utp:SUT"/>
    <xsd:complexType name="TestComponent">
        <xsd:choice maxOccurs="unbounded" minOccurs="0">
            <xsd:element name="name" type="xsd:string"/>
            <xsd:element name="testComponentDefinition" type="xsd:string"/>
            <xsd:element name="zone" type="xsd:string"/>
            <xsd:element name="behavior" type="utp:Behavior"/>
            <xsd:element name="dataPool" type="utp:DataPool"/>
            <xsd:element ref="xmi:Extension"/>
        </xsd:choice>
        <xsd:attribute ref="xmi:id"/>
        <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
        <xsd:attribute name="name" type="xsd:string"/>
        <xsd:attribute name="testComponentDefinition" type="xsd:string"/>
        <xsd:attribute name="zone" type="xsd:string"/>
        <xsd:attribute name="dataPool" type="xsd:string"/>
    </xsd:complexType>
    <xsd:element name="TestComponent" type="utp:TestComponent"/>
    <xsd:complexType name="TestContext">
        <xsd:choice maxOccurs="unbounded" minOccurs="0">
            <xsd:element name="name" type="xsd:string"/>
            <xsd:element name="testContextDefinition" type="xsd:string"/>
            <xsd:element name="sut" type="utp:SUT"/>
```

```
        <xsd:element name="component" type="utp:TestComponent"/>
        <xsd:element name="arbiter" type="utp:Arbiter"/>
        <xsd:element name="scheduler" type="utp:Scheduler"/>
        <xsd:element name="behavior" type="utp:Behavior"/>
        <xsd:element name="testConfiguration" type="utp:Deployment"/>
        <xsd:element name="testCase" type="utp:TestCase"/>
        <xsd:element name="executions" type="utp:TestLog"/>
        <xsd:element name="dataPool" type="utp:DataPool"/>
        <xsd:element ref="xmi:Extension"/>
    </xsd:choice>
    <xsd:attribute ref="xmi:id"/>
    <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
    <xsd:attribute name="name" type="xsd:string"/>
    <xsd:attribute name="testContextDefinition" type="xsd:string"/>
    <xsd:attribute name="sut" type="xsd:string"/>
    <xsd:attribute name="component" type="xsd:string"/>
    <xsd:attribute name="arbiter" type="xsd:string"/>
    <xsd:element name="scheduler" type="utp:Scheduler"/>
    <xsd:attribute name="testConfiguration" type="xsd:string"/>
    <xsd:attribute name="executions" type="xsd:string"/>
    <xsd:attribute name="dataPool" type="xsd:string"/>
</xsd:complexType>
<xsd:element name="TestContext" type="utp:TestContext"/>
<xsd:complexType name="TestLog">
    <xsd:choice maxOccurs="unbounded" minOccurs="0">
        <xsd:element name="name" type="xsd:string"/>
        <xsd:element name="testLogDefinition" type="xsd:string"/>
        <xsd:element name="verdict" type="utp:Verdict"/>
        <xsd:element name="testConfiguration" type="utp:Deployment"/>
        <xsd:element ref="xmi:Extension"/>
    </xsd:choice>
    <xsd:attribute ref="xmi:id"/>
    <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
    <xsd:attribute name="name" type="xsd:string"/>
    <xsd:attribute name="testLogDefinition" type="xsd:string"/>
    <xsd:attribute name="verdict" type="utp:Verdict"/>
    <xsd:attribute name="testConfiguration" type="xsd:string"/>
</xsd:complexType>
<xsd:element name="TestLog" type="utp:TestLog"/>
<xsd:complexType name="BaseDefault">
    <xsd:choice maxOccurs="unbounded" minOccurs="0">
        <xsd:element ref="xmi:Extension"/>
    </xsd:choice>
    <xsd:attribute ref="xmi:id"/>
    <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
</xsd:complexType>
<xsd:element name="BaseDefault" type="utp:BaseDefault"/>
<xsd:complexType name="Behavior">
    <xsd:choice maxOccurs="unbounded" minOccurs="0">
        <xsd:element name="name" type="xsd:string"/>
        <xsd:element name="behaviorDefinition" type="xsd:string"/>
```

```
        <xsd:element ref="xmi:Extension"/>
    </xsd:choice>
    <xsd:attribute ref="xmi:id"/>
    <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
    <xsd:attribute name="name" type="xsd:string"/>
    <xsd:attribute name="behaviorDefinition" type="xsd:string"/>
</xsd:complexType>
<xsd:element name="Behavior" type="utp:Behavior"/>
<xsd:complexType name="TestCase">
    <xsd:choice maxOccurs="unbounded" minOccurs="0">
        <xsd:element name="name" type="xsd:string"/>
        <xsd:element name="testCaseDefinition" type="xsd:string"/>
        <xsd:element name="behavior" type="utp:Behavior"/>
        <xsd:element name="executions" type="utp:TestLog"/>
        <xsd:element name="testObjective" type="utp:TestObjective"/>
        <xsd:element ref="xmi:Extension"/>
    </xsd:choice>
    <xsd:attribute ref="xmi:id"/>
    <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
    <xsd:attribute name="name" type="xsd:string"/>
    <xsd:attribute name="testCaseDefinition" type="xsd:string"/>
    <xsd:attribute name="executions" type="xsd:string"/>
    <xsd:attribute name="testObjective" type="xsd:string"/>
</xsd:complexType>
<xsd:element name="TestCase" type="utp:TestCase"/>
<xsd:complexType name="TestObjective">
    <xsd:choice maxOccurs="unbounded" minOccurs="0">
        <xsd:element name="name" type="xsd:string"/>
        <xsd:element name="testObjectiveDefinition" type="xsd:string"/>
        <xsd:element ref="xmi:Extension"/>
    </xsd:choice>
    <xsd:attribute ref="xmi:id"/>
    <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
    <xsd:attribute name="name" type="xsd:string"/>
    <xsd:attribute name="testObjectiveDefinition" type="xsd:string"/>
</xsd:complexType>
<xsd:element name="TestObjective" type="utp:TestObjective"/>
<xsd:complexType name="CodingRule">
    <xsd:choice maxOccurs="unbounded" minOccurs="0">
        <xsd:element name="coding" type="xsd:string"/>
        <xsd:element name="value" type="utp:InstanceValue"/>
        <xsd:element ref="xmi:Extension"/>
    </xsd:choice>
    <xsd:attribute ref="xmi:id"/>
    <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
    <xsd:attribute name="coding" type="xsd:string"/>
    <xsd:attribute name="value" type="xsd:string"/>
</xsd:complexType>
<xsd:element name="CodingRule" type="utp:CodingRule"/>
<xsd:complexType name="InstanceValue">
    <xsd:choice maxOccurs="unbounded" minOccurs="0">
```

```
        <xsd:element name="literalAny" type="utp:LiteralAny"/>
        <xsd:element name="literalAnyOrNull" type="utp:LiteralAnyorNull"/>
        <xsd:element name="literalNull" type="utp:LiteralNull"/>
        <xsd:element name="coding" type="utp:CodingRule"/>
        <xsd:element ref="xmi:Extension"/>
    </xsd:choice>
    <xsd:attribute ref="xmi:id"/>
    <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
    <xsd:attribute name="literalAny" type="xsd:string"/>
    <xsd:attribute name="literalAnyOrNull" type="xsd:string"/>
    <xsd:attribute name="literalNull" type="xsd:string"/>
    <xsd:attribute name="coding" type="xsd:string"/>
</xsd:complexType>
<xsd:element name="InstanceValue" type="utp:InstanceValue"/>
<xsd:complexType name="LiteralAny">
    <xsd:choice maxOccurs="unbounded" minOccurs="0">
        <xsd:element name="value" type="utp:InstanceValue"/>
        <xsd:element ref="xmi:Extension"/>
    </xsd:choice>
    <xsd:attribute ref="xmi:id"/>
    <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
    <xsd:attribute name="value" type="xsd:string"/>
</xsd:complexType>
<xsd:element name="LiteralAny" type="utp:LiteralAny"/>
<xsd:complexType name="LiteralAnyorNull">
    <xsd:choice maxOccurs="unbounded" minOccurs="0">
        <xsd:element name="value" type="utp:InstanceValue"/>
        <xsd:element ref="xmi:Extension"/>
    </xsd:choice>
    <xsd:attribute ref="xmi:id"/>
    <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
    <xsd:attribute name="value" type="xsd:string"/>
</xsd:complexType>
<xsd:element name="LiteralAnyorNull" type="utp:LiteralAnyorNull"/>
<xsd:complexType name="LiteralNull">
    <xsd:choice maxOccurs="unbounded" minOccurs="0">
        <xsd:element name="value" type="utp:InstanceValue"/>
        <xsd:element ref="xmi:Extension"/>
    </xsd:choice>
    <xsd:attribute ref="xmi:id"/>
    <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
    <xsd:attribute name="value" type="xsd:string"/>
</xsd:complexType>
<xsd:element name="LiteralNull" type="utp:LiteralNull"/>
<xsd:complexType name="ITimer">
    <xsd:choice maxOccurs="unbounded" minOccurs="0">
        <xsd:element name="isRunning" type="xsd:boolean"/>
        <xsd:element ref="xmi:Extension"/>
    </xsd:choice>
    <xsd:attribute ref="xmi:id"/>
    <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
```

```
    <xsd:attribute name="isRunning" type="xsd:boolean"/>
</xsd:complexType>
<xsd:element name="ITimer" type="utp:ITimer"/>
<xsd:complexType name="Timer">
    <xsd:choice maxOccurs="unbounded" minOccurs="0">
        <xsd:element ref="xmi:Extension"/>
    </xsd:choice>
    <xsd:attribute ref="xmi:id"/>
    <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
</xsd:complexType>
<xsd:element name="Timer" type="utp:Timer"/>
<xsd:complexType name="IArbiter">
    <xsd:choice maxOccurs="unbounded" minOccurs="0">
        <xsd:element ref="xmi:Extension"/>
    </xsd:choice>
    <xsd:attribute ref="xmi:id"/>
    <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
</xsd:complexType>
<xsd:element name="IArbiter" type="utp:IArbiter"/>
<xsd:complexType name="DataPool">
    <xsd:choice maxOccurs="unbounded" minOccurs="0">
        <xsd:element name="name" type="xsd:string"/>
        <xsd:element name="dataPoolDefinition" type="xsd:string"/>
        <xsd:element name="selector" type="utp:DataSelector"/>
        <xsd:element ref="xmi:Extension"/>
    </xsd:choice>
    <xsd:attribute ref="xmi:id"/>
    <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
    <xsd:attribute name="name" type="xsd:string"/>
    <xsd:attribute name="dataPoolDefinition" type="xsd:string"/>
    <xsd:attribute name="selector" type="xsd:string"/>
</xsd:complexType>
<xsd:element name="DataPool" type="utp:DataPool"/>
<xsd:complexType name="DataSelector">
    <xsd:choice maxOccurs="unbounded" minOccurs="0">
        <xsd:element name="name" type="xsd:string"/>
        <xsd:element name="dataSelectorDefinition" type="xsd:string"/>
        <xsd:element ref="xmi:Extension"/>
    </xsd:choice>
    <xsd:attribute ref="xmi:id"/>
    <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
    <xsd:attribute name="name" type="xsd:string"/>
    <xsd:attribute name="dataSelectorDefinition" type="xsd:string"/>
</xsd:complexType>
<xsd:element name="DataSelector" type="utp:DataSelector"/>
<xsd:complexType name="DataPartition">
    <xsd:choice maxOccurs="unbounded" minOccurs="0">
        <xsd:element name="name" type="xsd:string"/>
        <xsd:element name="dataPartitionDefinition" type="xsd:string"/>
        <xsd:element name="selector" type="utp:DataSelector"/>
        <xsd:element ref="xmi:Extension"/>
```

```
        </xsd:choice>
        <xsd:attribute ref="xmi:id"/>
        <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
        <xsd:attribute name="name" type="xsd:string"/>
        <xsd:attribute name="dataPartitionDefinition" type="xsd:string"/>
        <xsd:attribute name="selector" type="xsd:string"/>
    </xsd:complexType>
    <xsd:element name="DataPartition" type="utp:DataPartition"/>
    <xsd:complexType name="IScheduler">
        <xsd:choice maxOccurs="unbounded" minOccurs="0">
            <xsd:element ref="xmi:Extension"/>
        </xsd:choice>
        <xsd:attribute ref="xmi:id"/>
        <xsd:attributeGroup ref="xmi:ObjectAttribs"/>
    </xsd:complexType>
    <xsd:element name="IScheduler" type="utp:IScheduler"/>
</xsd:schema>
```