Date: July 2005

UML Testing Profile

Version 1.0

formal/05-07-07

This is a testing profile for UML 2.0

OBJECT MANAGEMENT GROUP

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, 250 First Avenue, Needham, MA 02494, U.S.A.

TRADEMARKS

The OMG Object Management Group Logo®, CORBA®, CORBA Academy®, The Information Brokerage®, XMI® and IIOP® are registered trademarks of the Object Management Group. OMG™, Object Management Group™, CORBA logos™, OMG Interface Definition Language (IDL)™, The Architecture of Choice for a Changing World™, CORBAservices™, CORBAfacilities™, CORBAmed™, CORBAnet™, Integrate 2002™, Middleware That's Everywhere™, UML™, Unified Modeling Language™, The UML Cube logo™, MOF™, CWM™, The CWM Logo™, Model Driven Architecture™, Model Driven Architecture Logos™, MDA™, OMG Model Driven Architecture™, OMG MDA™ and the XMI Logo™ 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/technology/agreement.htm).

Table of Contents

List of Figures

1 Scope

The UML Testing Profile defines a language for designing, visualizing, specifying, analyzing, constructing, and documenting the artifacts of test systems. It is a test modeling language that can be used with all major object and component technologies and applied to testing systems in various application domains. The UML Testing Profile can be used stand alone for the handling of test artifacts or in an integrated manner with UML for a handling of system and test artifacts together.

The UML Testing Profile extends UML with test specific concepts like test components, verdicts, defaults, etc. These concepts are grouped into concepts for test architecture, test data, test behavior, and time. Being a profile, the UML testing profile seamlessly integrates into UML: it is based on the UML metamodel and reuses UML syntax.

The UML Testing Profile is based on the UML 2.0 specification. The UML Testing Profile is defined by using the metamodeling approach of UML. It has been architected with the following design principles in mind:

- UML integration: as a real UML profile, the UML Testing Profile is defined on the basis of the metamodel provided in the UML superstructure volume and follows the principles of UML profiles as defined in the UML infrastructure volume of UML 2.0.
- Reuse and minimality: wherever possible, the UML Testing Profile makes direct use of the UML concepts and extends them and adds new concepts only where needed. Only those concepts are extended/added to UML, which have been demonstrated in the software, hardware, and protocol testing area to be of central relevance to the definition of test artifacts and are not part of UML.

2 Conformance

2.1 Summary of optional versus mandatory features

Mandatory features for a UML Testing Profile implementation are the concepts for Test Architecture, Test Behavior, Test Data and Time Concepts.

2.2 Proposed compliance points

The compliance points are as follows:

1. UML Profile for Testing: A compliant implementation supports the UML profiling mechanism, the UML entities extended by the Testing Profile, and the stereotyped entities of the UML Testing Profile.
2. MOF-based Metamodel for Testing: The compliant implementation supports all of the entities in the MOF-based metamodel.
3. Notation: If graphical notation is used, the compliant implementation recognizably supports the notation defined by the Testing Profile specification.
4. XMI: An XMI compliant implementation of the Testing Profile and/or MOF metamodel provides the XMI exchange mechanism using the Testing Profile XMI schema definitions.
5. Static Requirements: The compliant implementation checks the specified constraints automatically.

Compliance requires meeting point 1 and/or 2. Points 3-5 are optional and can be claimed in any combination. A compliance statement should address every point.

3 Normative references

The following normative documents contain provisions which, through reference in this text, constitute provisions of this specification. For dated references subsequent amendments to, or revisions of, any of these publications do not apply.

- UML 2.0 Infrastructure Specification
- UML 2.0 Superstructure Specification
- UML 2.0 OCL Specification
- MOF 2.0 Specification

Note – You will find these documents in the OMG Specification Catalog at this URL: http://www.omg.org/technology/documents/modeling_spec_catalog.htm.

3.1 Informative References

- OMG ADTF: *RFP on a UML Testing Profile*, ad/01-07-08, 2001.
- OMG ADTF: *RFP on UML 2.0 Superstructure*, ad/00-09-02, 2000.
- OMG ADTF: *Final Adopted Specification on Unified Modeling Language: Superstructure, version 2.0*, ptc/03-08-02, 2003.
- OMG ADTF: *RFP on UML 2.0 Infrastructure*, ad/00-09-01, 2000.
- OMG ADTF: *Final Adopted Specification on: Unified Modeling Language: Infrastructure, version 2.0*, ptc/03-09-15, 2003.
- JUnit: http://www.junit.org.
- ETSI ES 201 873-1: *The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language*. V2.2.1 (2002-10), 2002; also an ITU-T standard Z.140.
- ETSI ES 201 873-3: *The Testing and Test Control Notation version 3; Part 3: TTCN-3 Graphical Presentation Format (GFT)*. V2.2.1 (2002-10), 2002; also an ITU-T standard Z.142.
- ITU-T Z.120: Message Sequence Charts (MSC), Nov. 1999.

4 Terms and definitions

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

4.1 Test Architecture

The set of concepts (in addition to the UML 2.0 structural concepts) to specify the structural aspects of a test context covering test components, the system under test, their configuration, etc.

Test Context — A collection of test cases together with a test configuration on the basis of which the test cases are executed.

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 Component — A test component is a class of a test system. Test component objects realize the behavior of a test case. A test component has a set of interfaces via which it may communicate via connections with other test components or with the SUT.

SUT — The system under test (SUT) is a part and is the system, subsystem, or component being tested. An SUT can consist of several objects. The SUT is exercised via its public interface operations and signals by the test components. No further information can be obtained from the SUT as it is a black-box.

Arbiter — A property of a test case or a test context 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*, *Inconc*, and *Error* as verdict, where these verdicts are ordered as *Pass* < *Inconc* < *Fail* < *Error*. The arbitration algorithm can be user-defined.

Scheduler — A property of a test context used to control the execution of the different test components. The scheduler will keep information about which test components exist at any point in time, and it will collaborate with the arbiter to inform it when it is time to issue the final verdict. It keeps control over the creation and destruction of test components and it knows which test components take part in each test case.

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.

4.2 Test Behavior

The set of concepts (in addition to the UML 2.0 behavioral concepts) to specify test behaviors, their objectives, and the evaluation of systems under test.

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 Case	A test case is a specification of one case to test the system including what to test with, which input, result, and under which conditions. It is a complete technical specification of how the SUT should be tested for a given test objective. A test case is defined in terms of sequences, alternatives, loops, and defaults of stimuli to and observations from the SUT. It implements a test objective. A test case may invoke other test cases. A test case uses an arbiter to evaluate the outcome of its test behavior. A test case is a property of a test context. It is an operation specifying how a set of cooperating test components interacting with a system under test realize a test objective. Both the system under test and the different test components are parts of the test context to which the test case belongs.
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 Objective	A test objective is a named element describing what should be tested. It is associated to a test case.
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.
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.
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.
Default	Default is a behavior triggered by a test observation that is not handled by the behavior of the test case per se. Defaults are executed by test components.
Verdict	Verdict is the assessment of the correctness of the SUT. Test cases yield verdicts. Verdicts can also be used to report failures in the test system. Predefined verdict values are *pass*, *fail*, *inconclusive*, 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. *Inconclusive* is used for cases where neither a Pass nor a Fail can be given. An *Error* verdict shall be used to indicate errors (exceptions) within the test system itself. Verdicts can be user-defined. The verdict of a test case is calculated by the arbiter.
Validation Action	An action to evaluate the status of the execution of a test case by assessing the SUT observations and/or additional characteristics/parameters of the SUT. A validation action is performed by a test component and sets the local verdict of that test component.

Log Action	An action to log information in the test log.
Test Log	A log is an interaction resulting from the execution of a test case. It represents the different messages exchanged between the test components and the SUT and/or the states of the involved test components. A log is associated with a verdict representing the adherence of the SUT to the test objective of the associated test case.

4.3 Test Data

The set of concepts (in addition to the UML 2.0 data concepts) to specify data used in stimuli to the SUT, observations from the SUT, and for coordination between test components.

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.
Data Pool	A data pool is a collection of data partitions or explicit values that are used by a test context, or test components, during the evaluation of test contexts and test cases. In doing so, a data pool provides a means for providing values or data partitions for repeated tests.
Data Partition	A logical value for a parameter used in a stimulus or in an observation. It typically defines an equivalence class for a set of values (e.g., valid user names, etc.).
Data Selector	An operation that defines how data values or equivalence classes are selected from a data pool or data partition.
Coding Rule	The interfaces of an SUT use certain encodings (e.g., CORBA GIOP/IIOP, IDL, ASN.1 PER or XML) that have to be respected by the test systems. Hence, coding rules are part of a test specification.

4.4 Time Concepts

The set of concepts (in addition to the UML 2.0 time concepts) to specify time constraints, time observations and/or timers within test behavior specifications in order to have a time quantified test execution and/or the observation of the timed execution of test cases.

Timer	Timers are mechanisms that may generate a timeout event when a specified time value occurs. This may be when a pre-specified time interval has expired relative to a given instant (usually the instant when the timer is started). Timers belong to test components. They are defined as properties of test components. A timer is started with an expiration time being the time when the timeout is to be issued. A timeout indicates the timer expiration. A timer can be stopped. The expiration time of a running timer and its current status (e.g., active/inactive) can be checked.
Timezone	Timezone is a grouping mechanism for test components. Each test component belongs to a certain timezone. Test components in the same timezone have the same time (i.e., test components of the same timezone are time synchronized).

5 Additional information

5.1 Acknowledgements

The following person served as chair person up to the submission adoption (June 2003):

- Ina Schieferdecker (schieferdecker@fokus.fraunhofer.de), Fraunhofer Fokus

During the FTF Dr. Ina Schieferdecker will also serve as co-chair together with:

- Øystein Haugen (oystein.haugen@ericsson.com), Ericsson

In addition the following persons have written parts of the specification and been central to the discussions:

- Paul Baker, Motorola
- Zhen Ru Dai, University of Lübeck
- Jens Grabowski, University of Lübeck
- Serge Lucio, IBM
- Eric Samuelson, Telelogic
- Clay Williams, IBM

Furthermore Softeam was also a submitter of the adopted document, and additional supporters were iLogix, IRISA, and Scapa Technologies.

5.2 Guide to material in the specification

So far, UML technology has focused primarily on the definition of system structure and behavior and provides limited means for describing test procedures only. However, with the approach towards system engineering according to model-driven architectures with automated code generation, the need for solid conformance testing, certification, and branding is increased.

The UML Testing Profile

- is based upon the UML 2.0 specification ,
- enables the specification of tests for structural (static) and behavioral (dynamic) aspects of computational UML models, and
- is capable of inter-operation with existing test technologies for black box testing.

The work is based on recent developments in black-box testing such as the results from TTCN-3 (Testing and Test Control Notation) and COTE (Component Testing using the Unified Modeling Language). The UML Testing Profile is defined such that mappings onto TTCN-3 and JUnit are possible in order to enable the reuse of existing test infrastructures.

Major generalizations are:

- The separation of test behavior and test evaluation by introducing a new test component: the arbiter. This enables the easy reuse of test behavior for other testing kinds.
- The integration of the concepts test control, test group, and test case into just one concept of a test case, which can be decomposed into several lower level test cases. This enables the easy reuse of test case definitions in various hierarchies. A test context is just a top-level test case.
- The integration of defaults for handling unexpected behaviors, verdicts, and wildcards.
- The support of data partitions not only for observations, but also for stimuli. This allows to describe test cases logically without having the need to define the stimulus data completely but as a set or range of values.

Furthermore, some additions widen the practical usability of the UML Testing Profile:

- A test configuration (as an internal structure of a test context) is used to enable the definition of test components realizing a test case, it describes the initial setup of the test components and the connection to the SUT and between each other.
- A deployment diagram is used to place requirements regarding test execution on certain nodes in a network.

The UML Testing Profile provides concepts for:

- test behavior addressing observations and activities during a test,
- test architecture (i.e., the elements and their relationships involved in a test),
- test data (i.e., the structures and meaning of values to be processed in a test), and
- time (i.e., the time constraints and time observation for test execution).

Chapter 6 contains the terminology, the metamodel, examples, and mappings for these concepts.

6 The UML Testing Profile

6.1 Overview

This section provides an overview and introduction to the UML Testing Profile.

The UML Testing Profile defines a language for designing, visualizing, specifying, analyzing, constructing, and documenting the artifacts of test systems. It is a test modeling language that can be used with all major object and component technologies and applied to testing systems in various application domains. The UML Testing Profile can be used stand alone for the handling of test artifacts or in an integrated manner with UML for a handling of system and test artifacts together.

The UML Testing Profile is defined by using the metamodeling approach of UML. It has been architected with the following design principles in mind:

- UML integration: as a real UML profile, the UML Testing Profile is defined on the basis of the metamodel provided in the UML superstructure volume and follows the principles of UML profiles as defined in the UML infrastructure volume of UML 2.0.
- Reuse and minimality: wherever possible, the UML Testing Profile makes direct use of the UML concepts and extends them and adds new concepts only where needed. Only those concepts are extended/added to UML, which have been demonstrated in the software, hardware, and protocol testing area to be of central relevance to the definition of test artifacts and are not part of UML.

Being a UML profile, the UML Testing Profile inherits the characteristics of UML:

- Layering: separate concerns across metalayers of abstraction (along the 4-layer metamodel architecture).
- Partitioning: use of packages for coarse- or fine-grained structures.
- Extensibility: profiling the test profile for an adaptation to particular platforms (e.g., J2EE/EJB, .NET/COM+) and domains (e.g., finance, telecommunications, aerospace).

6.2 Structure of the UML Testing Profile

The UML Testing Profile is organized in four logical groups of concepts:

- Test architecture defining concepts related to test structure and test configuration.
- Test data defining concepts for test data used in test procedures.
- Test behavior defining concepts related to the dynamic aspects of test procedures.
- Test time defining concepts for a time quantified definition of test procedures.

The UML Testing Profile is specified by:

- Giving the terminology for a basic understanding of the UML Testing Profile concepts.
- Defining the UML 2.0 based metamodel of UML Testing Profile.
- Defining a MOF model for the pure UML Testing Profile enabling the use of the Testing Profile independent of UML.

- Giving examples to demonstrate the use of the UML Testing Profile for component-level and system-level tests.
- Outlining mappings from the UML Testing Profile to test execution environments like JUnit and TTCN-3.

All sections form the definition of the UML Testing Profile. In case of ambiguities, the UML 2.0 based metamodel takes precedence.

6.3 The Profile

6.3.1 Test Architecture

This section contains the concepts needed to describe the elements that the test cases defined using the profile. These elements include the test context, which contains a collection of test cases. These cases are realized by a set of test components. The verdict of a test case is determined by an implementation of the arbiter interface.

Arbiter

Arbiter is a predefined interface provided with the 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 6.3.2, “Test Behavior,” on page 14 for more information on validation actions.)

Every test context must have an implementation of the arbiter interface, and the tool vendor constructing tools based on the Testing Profile will provide a default arbiter to be used if one is not explicitly defined in the test context.

Scheduler

Scheduler is a predefined interface provided with the Testing Profile. The purpose of a scheduler implementation is to control the execution of the different test components. The scheduler will keep information about which test components exist at any point in time, and it will collaborate with the arbiter to inform it when it is time to issue the final verdict. It keeps control over the creation and destruction of test components and it knows which test components take part in each test case.

Every test context must have an implementation of a scheduler. Any tool must provide such an implementation that will be used if there are no explicit realizations defined in the test context.

SUT

The SUT is the system under test. As the profile only addresses black-box conformance testing, the SUT provides only a set of operations via publicly available interface(s). No information on the internal structure of the SUT is available for use in the specification of test cases using the Testing Profile.

Test Elements

Two test elements are defined in the architecture section: test contexts and test components. A test context contains a collection of test cases, an instance of the arbiter interface, normally an instance of the SUT, and possibly high level behavior used to control the execution of the test cases. Test components are the various elements that interact with the SUT to realize the test cases defined in the test context.

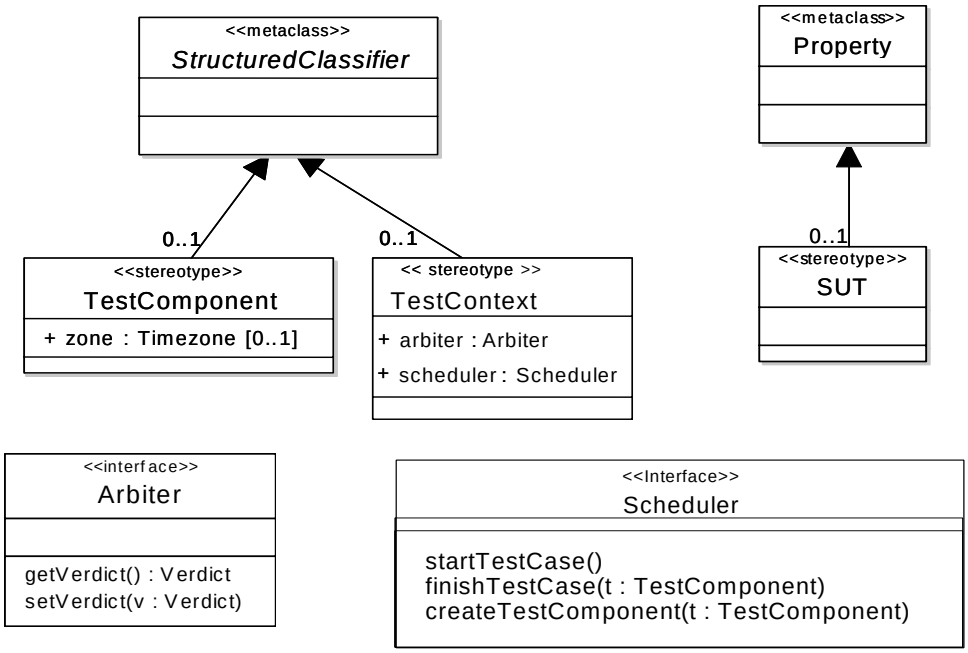


Figure 6.1 - Test Architecture

Arbiter (a predefined interface)

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 Section 6.3.2, “Test Behavior,” on page 14).

Operations

- getVerdict() : Verdict Returns the current verdict.
- setVerdict(v : Verdict) Sets a new verdict value.

Semantics

The verdict setting semantics is defined by the classifier realizing the arbiter interface. One example of how to implement the setVerdict operation is the following:

- If a verdict is pass, it can be changed to inconclusive, fail, or error only.
- If a verdict is inconclusive, it can be changed to fail or error only.
- If a verdict is fail, it can be changed to error only.
- If a verdict is error, it cannot be changed.

Examples

An example arbiter can be found in Figure 6.45 on page 68.

Scheduler (a predefined interface)

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.

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.

Examples

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

SUT

Description

Extends Property. The SUT stereotype 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.

Notation

The notation for SUT is a part with stereotype ≪SUT≫. It is to be used in the composite structure of a test context.

Examples

Examples of SUTs can be found in Figure 6.26 on page 53 and Figure 6.40 on page 64.

TestComponent

Description

A test component is a structured classifier participating in test behaviors. A test component is commonly an active class with a set of ports and interfaces. Test components are used to specify test cases as interactions between a number of test components. The classifier behavior of a test component can be used to specify low level test behavior, such as test scripts, or it can be automatically generated by deriving the behavior from all test cases in which the component takes part.

Attributes

- zone : Timezone [0..1] Specifies the timezone to which a test component belongs (Timezones are discussed in Section 6.3.4, “Time Concepts,” on page 29).

Semantics

The zone attribute is available at run-time, and GetTimezoneAction and SetTimezoneAction are used to get and set the timezone in run-time. The run-time representation of the timezone is not specified. The initial timezone of the component is specified in the model as a tagged value on the zone attribute.

Notation

The notation for test component is a classifier with stereotype <<TestComponent>>.

Examples

Examples of test components can be found in Figure 6.25 on page 53 and Figure 6.38 on page 62.

TestContext

Description

A structured classifier acting 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 is used for test control.

Attributes

- arbiter : Arbiter [1] Realizes the arbiter interface.
- scheduler : Scheduler [1] Realizes the scheduler interface.

Constraints

[1] A test context must contain exactly one property realizing the Arbiter interface.

[2] A test context must contain exactly one property realizing the Scheduler interface.

Notation

The notation for test context is a classifier with stereotype <<TestContext>>.

Examples

Examples of test contexts can be found in Figure 6.20 on page 49, Figure 6.25 on page 53, and Figure 6.38 on page 62.

6.3.2 Test Behavior

The area of test behavior includes concepts to specify the behavior of tests in the context of a test context (see "TestContext" on page 13). The public test case operations of a test context are the test cases that represent the interface towards the testers. In addition there may be other private or protected test cases that are used as utilities within the concrete realization of the public test cases.

The implementation of test cases is specified by a test behavior. The test behavior may include a number of special constructs defined by the Testing Profile.

A test case will return a verdict.

Verdict

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. A **pass** verdict indicates that the test case is successful and that the SUT has behaved according to what should be expected. A **fail** verdict on the other hand shows that the SUT is not behaving according to the specification. An **inconclusive** verdict means that the test execution cannot determine whether the SUT performs well or not. An **error** verdict tells that the test system itself and not the SUT fails.

The final verdict of a test case is determined by an arbiter. Every test context has an arbiter and the tool vendor will provide a default arbiter if the test context does not explicitly specify one.

During the test execution of the test behavior, each test component will report verdicts to the arbiter, and the arbiter will produce the final verdict from these intermediate test component specific verdicts when all test components have finished executing this test case.

The predefined arbiter interface defines *setVerdict* and *getVerdict* that may be used directly if the arbiter is described explicitly. Even the default arbiter can appear explicitly in the specifications of the test behavior.

Furthermore the Testing Profile has defined a few actions that can be applied by test components to define and observe the verdict. The *ValidationAction* action is an implicit setVerdict message to the arbiter informing it about an intermediate (or final) local verdict from that test component.

Default

A UML specification 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 Testing Profile as a means to make partial definitions of test components complete in a compact, yet flexible way. The 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 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 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 FinishAction, 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.

Utilities

The Testing Profile has defined a few action utilities that may come in handy when defining test behavior.

FinishAction is an action that completes the test case for one test component. 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.

LogAction is an action that indicates that information about the test should be recorded for the test component performing the action.

determAlt is an interaction operator and is a shorthand for an Alternative where the operands are evaluated in sequence such that it is deterministic which operand is chosen given the value of the guards, regardless of the fact that the guard for more than one operand may be true.

TestLog

Both a test context and a test case may trace their executions. These traces are behaviors in general. They can be recorded as test logs and become part of the test specification. A test log has to be attached to a test context or a test case such that it is specified from which test context or test case that test log has been taken.

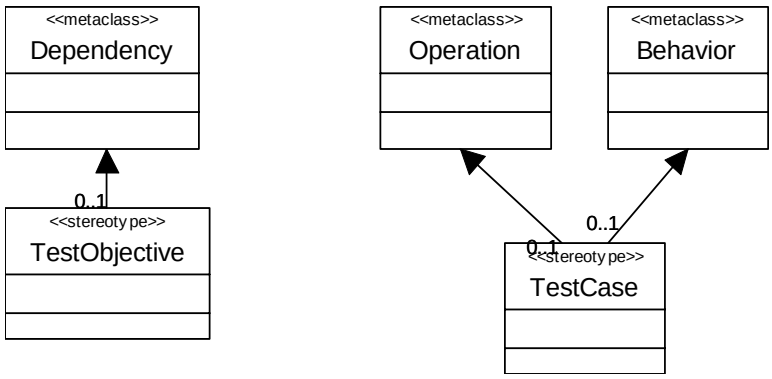


Figure 6.2 - Test case and test objective

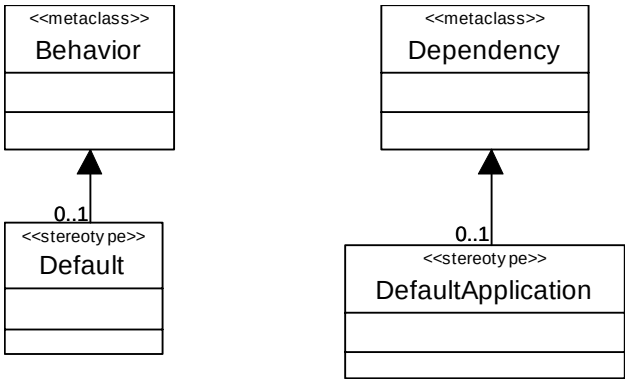


Figure 6.3 - Defaults

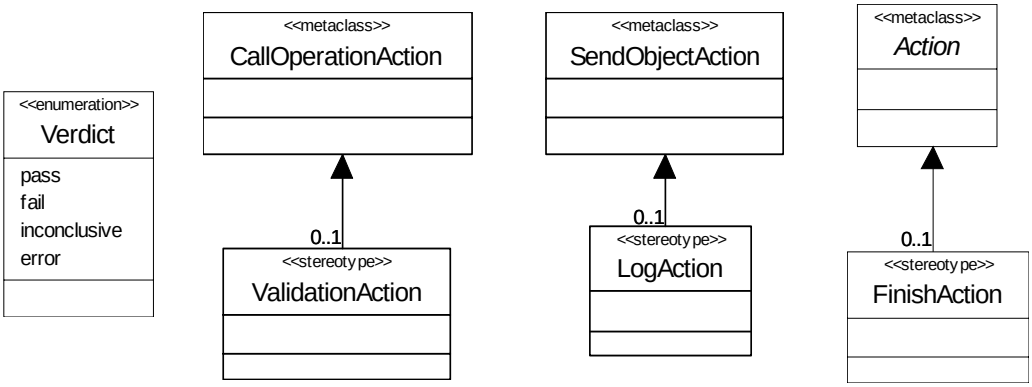


Figure 6.4 - Actions

Figure 6.5 - Deterministic Alt Operator (in Combined Fragments)

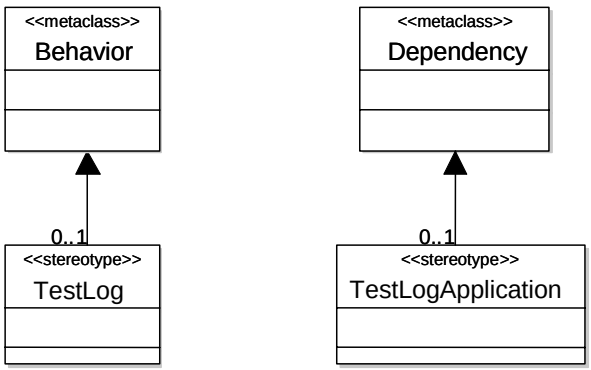


Figure 6.6 - Test Log

Default

Description

Extends Behavior. A default is a behavior used to handle unexpected or exceptional messages or events on one test component.

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 is defined as *continue* which is the situation if nothing is explicitly specified. When the default is defined as *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 *repeat*.

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).

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 “FinishAction” on page 20).
- The activity may conclude.

Notation

A Default is a Behavior and has no special notation. The Testing Profile gives examples of the usage of Interactions and State Machines as default notations in Figure 6.30 on page 57 and Figure 6.31 on page 57.

Examples

Examples of defaults can be found in Figure 6.30, Figure 6.31, and Figure 6.32.

DefaultApplication

Description

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

Constraints

[1] The default application relates a default to a unit of test behavior. That unit of test behavior must be one of the following: Package, Classifier, Behavior, InteractionFragment, State, Region, or Activity.

[2] The multiplicity of clientDependency is restricted to [0..1].

Semantics

Please refer to the semantics of Default.

Notation

The notation for a default 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 [**continue** | **repeat** [repetition-count]]

If nothing is given following the default identifier, *continue* is assumed. If no repetition-count is given, 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 6.29 on page 56 and Figure 6.31 on page 57.

determAlt (an interaction operator)

Description

The deterministic alternative is a shorthand for an Alternative where the operands are evaluated in sequence such that it is deterministic which operand is chosen given the value of the guards, regardless of the fact that the guard for more than one operand may be true.

Semantics

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))

Notation

The determAlt uses the notation for combined fragments in interactions: a determAlt in the small compartment in the upper left corner of the CombinedFragment frame.

Examples

An example determAlt operator can be found in Figure 6.30 on page 57.

FinishAction

Description

A FinishAction is an action that determines that the test component will finish its execution of the test case immediately. This does not mean that the test component is terminated.

Semantics

A FinishAction means that the test component will move 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 its FinishAction.

Notation

In Interaction 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 finish action can be found in Figure 6.30 on page 57.

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.

Semantics

The target of a log action refers to a logging mechanism in the run-time system. The request refers to the information that is logged. The representation of the logging mechanism is not specified. The LogAction records some snapshot of the test component.

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 to return a verdict value.

Test cases are owned by test contexts, and has therefore access to all features of the test context (e.g., the SUT and test components of the composite structure).

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).

Constraints

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

[2] A test case stereotype cannot be applied both to a behavior and its specification.

[3] If a test case stereotype is applied to an operation, the featuring classifier must have test context stereotype applied.

[4] If a test case stereotype is applied to a behavior, the context of the behavior must have the test context stereotype applied.

Semantics

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

Notation

The notation for Test Case is an Operation with stereotype <<TestCase>>.

Examples

Examples of test cases can be found in Figure 6.21, Figure 6.29, and Figure 6.41.

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.

Constraints

[1] The client of a test objective must be a named element with a test case or test context stereotype applied.

Notation

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

Examples

The test objective in Figure 6.7 specifies that the objective of the ValidWithdrawal test case is WithdrawMoney use case.

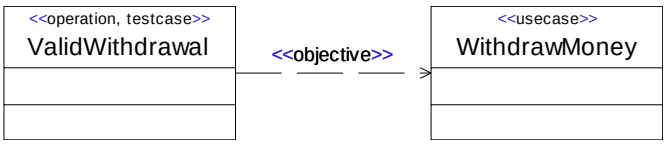


Figure 6.7 - Test objective

ValidationAction

Description

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

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 the test case stereotype is applied to the behavior or its specification).

Semantics

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

Notation

The notation for a validation action is an ordinary action with the validation action stereotype applied.

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.

Examples

In Figure 6.8 two validation actions 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.

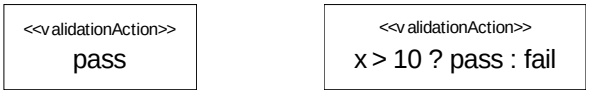


Figure 6.8 - Validation actions

Verdict (a predefined enumeration)

Description

A verdict is a predefined enumeration specifying the set of possible evaluations of a test. Four literals are defined: pass, fail, inconclusive, error.

- 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 has no semantics other than that of a plain enumeration. The order of the predefined literals are: error, fail, inconclusive, pass.

Notation

The predefined literals **pass**, **inconclusive**, **fail**, and **error** are shown as keywords (normally in bold face).

TestLog

Description

Extends behavior. 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.

Constraints

No constraints for TestLog defined.

Notation

No additional notation for TestLog is defined.

TestLogApplication

Description

A dependency to a test case or a test context.

Constraints

[1] The client of a test log application must be a named element with a test case or test context stereotype applied.

[2] The supplier of a test log must be a named element with a test log stereotype applied.

Notation

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

Examples

See the example in Figure 6.9. ATMSuite_log is a test log of the test context ATMSuite. invalidPIN_log is a test log of the test case invalidPIN.

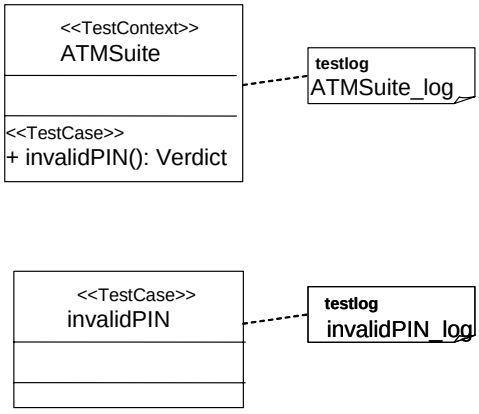


Figure 6.9 - Example on Test Log

6.3.3 Test Data

This section contains concepts additional to UML data concepts needed to describe test data. It covers wildcards for a flexible specification of test data, data pools, data partitions, data selection, and coding rules for the specification of test data transmission.

Wildcards

Wildcards are literals and denote an omitted value, any value, or any value or omit. These literals can be used wherever value specifications can be used. They are typically used for a loose specification of data to be expected from the SUT or provided to the SUT.

UML 2.0 provides LiteralNull, which is used by the Testing Profile for the representation of omitted values. Wildcards for any value (LiteralAny) and any value or omit (LiteralAnyOrNull) are extensions to UML 2.0.

Data Pool

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 explict values; and can only be associated with either a test context or test components.

Data Partition

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.

Data Selector

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.

Coding Rules

Coding rules are shown as strings referencing coding rules defined outside the Testing Profile such as by ASN.1, CORBA, or XML. Coding rules are basically applied to value specification to denote the concrete encoding and decoding for these values during test execution. They can also be applied to properties and namespaces in order to cover all involved values of the property and/or namespace at once.

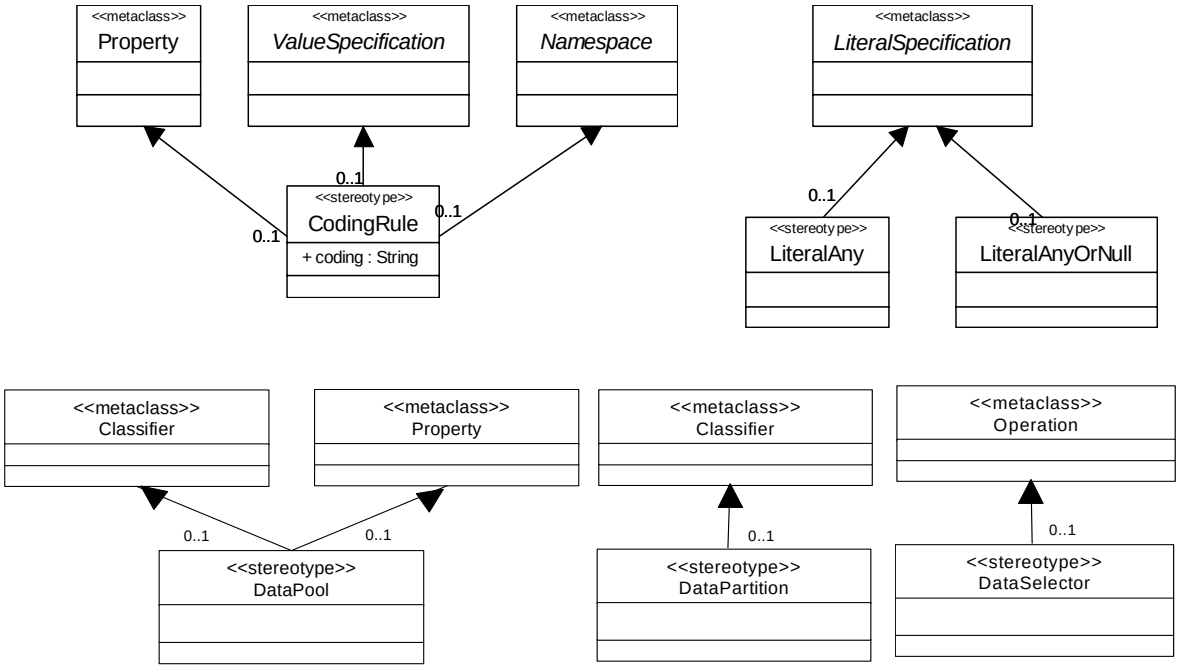


Figure 6.10 - Test data

Coding Rule

Description

A coding rule specifies how values are encoded and/or decoded. Coding rules are defined outside the Testing Profile and referred to within 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).

Attributes

- coding : String 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 coding rule 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 coding rule example is given in Figure 6.26 on page 53.

DataPool

Description

The data pool stereotype extends a classifier or property to specify 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.

Constraints

[1] A data pool stereotyped classifier can only be referenced by a test context or test component.

[2] A data pool stereotyped property can only be applied to a property associated connected with a test component within the context of a test context stereotyped classifier.

[3] A datapool stereotyped classifier cannot be associated with both a test context and a test component.

Semantics

The semantics of a data pool are given by the classifier, and contained data partitions, that realizes it.

Notation

The notation for data pool is a classifier with stereotype <<DataPool>>.

Examples

A data pool example is given in Figure 6.39 on page 63.

DataPartition

Description

The data partition stereotype extends a classifier to specify a container for a set of values. These data sets are used as abstract equivalence classes during test context and test case evaluation.

Constraints

[1] A data partition stereotyped classifier can only be associated with a data pool or another data partition.

Semantics

The semantics of a data partition are given by the classifier that realizes it.

Notation

The notation for data partition is a classifier with stereotype <<DataPartition>>.

Examples

A data partition example is given in Figure 6.39 on page 63.

DataSelector

Description

The data selector stereotype extends an operation to allow the implementation of different data selection strategies.

Constraints

[1] If a data selector stereotype is applied to an operation, the featuring classifier must have either a data pool or data partition stereotype applied.

Semantics

The semantics of a data selector are given by the behavior that realizes it.

Notation

The notation for data selector is an operation with stereotype <<DataSelector>>.

Examples

A data selector example is given in Figure 6.39 on page 63.

LiteralAny

Description

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

Semantics

If a LiteralAny 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 LiteralAny 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 LiteralAny is to be sent. The selection of this value can be done along different selection schemes such as default values or random values.

Notation

The notation for LiteralAny is the question mark character, '?'.

LiteralAnyOrNull

Description

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

Semantics

If a LiteralAnyOrNull 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 LiteralAnyOrNull 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 LiteralAnyOrNull 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, '*'.

6.3.4 Time Concepts

When specifying tests, time concepts are essential to provide complete and precise specifications. The simple time concepts of UML do not cover the full needs for test specification, and therefore the Testing Profile provides a small set of useful time concepts.

Timers are used as a means to manipulate and control test behavior, or to assure that a test case terminates properly. Timezones are used to group and synchronize test components within a distributed test system.

Timer and Timeout

Timer is a predefined interface. Timer propreties may only be owned by active objects. An active class may own multiple timers. Operations like start, stop and read are defined for the timer interface. 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.

When a timer expires after its predefined time, a special timeout message is generated automatically. It is sent immediately to the active class that owns the timer. A timeout is only allowed to be sent to the owning class of the timer.

To provide more flexibility, no restriction is made about the visibility of a timer. If a timer is private, the timer can only be accessed by its owning class. If the timer is public, any other active class of the test system is allowed to manipulate the timer.

The graphical syntax for starting, stopping a timer and receiving the timeout message within the owner class of the timer are adopted from the Message Sequence Charts (MSC) language. Herein, keywords like "**start**" or "**stop**" are not used. Only the name of the timer and its expiration time value need to be specified. When a timer is manipulated by other classes, common message with the keywords "**start**" and "**stop**" are used. No specific syntax is defined for reading a timer.

Timezone

Timezones serve as grouping mechanisms for test components within a test system. Each test component belongs at most to one timezone. Test components in the same timezone have the same perception of time (i.e., test components of the same timezone are considered to be time synchronized). The time zone of a test component can be accessed both in the model and in run-time.

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.

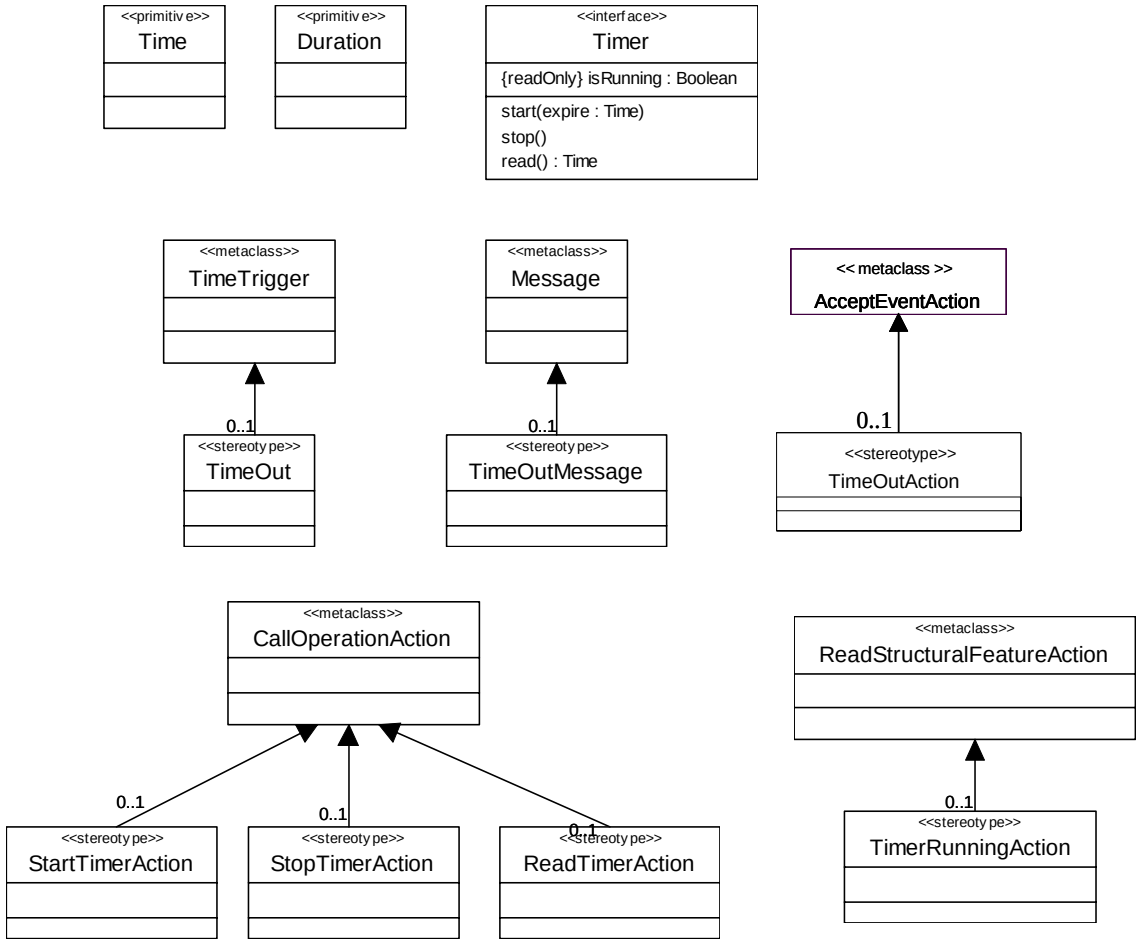


Figure 6.11 - Timer concepts

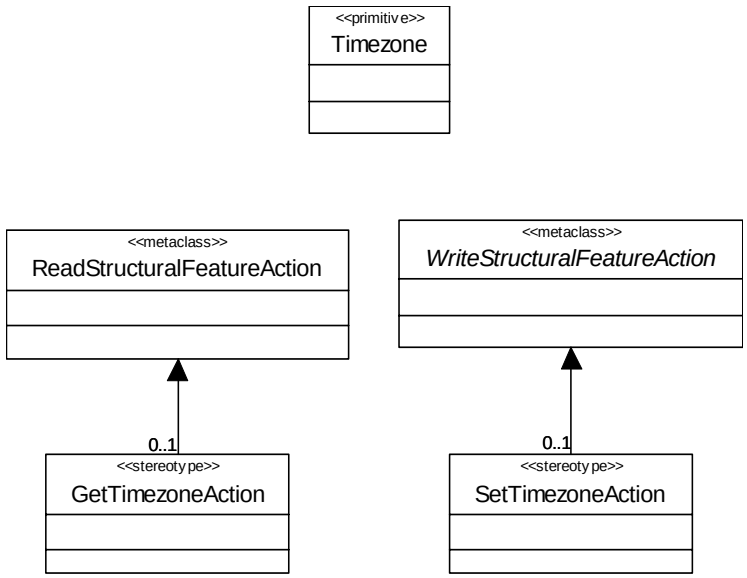


Figure 6.12 - Timezone concepts

GetTimezoneAction

Description

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

Constraints

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

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

Semantics

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

Notation

The notation for a get timezone action is an ordinary action with the get timezone action stereotype applied.

SetTimezoneAction

Description

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

Constraints

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

[2] The value must be a Timezone value.

Semantics

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

Notation

The notation for a set timezone action is an ordinary action with the set timezone action stereotype applied.

Duration (a predefined type)

Description

Duration is a predefined primitive type used to specify a duration. Durations are used in time constraints and together with time values. Adding or subtracting a duration to a time value results in a time value. Adding or subtracting two durations results in a duration.

Time (a predefined type)

Description

Time is a predefined primitive type used to specify concrete time values. Time values are used to express time constraints and to set timers. A predefined keyword ‘now’ may be used to represent the current time in a system. How this value is obtained is not specified. The difference between two Time values is a Duration.

TimeOut

Description

Extends TimeTrigger. A TimeOut is generated by a timer when it expires and may trigger an associated behavior (e.g., a transition in a statemachine).

Semantics

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

Notation

The notation for a TimeOut reuses the notation for TimeTrigger.

TimeOutMessage

Description

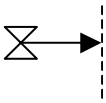
Extends Message. A TimeOutMessage is generated by a timer when it expires. The timeout message is sent to the active class that owns the timer.

Semantics

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

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 timeout message example can be found in Figure 6.27 on page 54.

TimeOutAction

Description

Extends AcceptEventAction. A timeout is enabled by a timer when it expires. An activity having the TimeOutAction as input condition occurs, when the timeout is enabled (and when all further input conditions are satisified).

Semantics

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

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.

Timer (a predefined interface)

Description

A predefined interface specifying features needed to specify a timer.

A timer is owned by an active class and is started with a predefined time of expiration. A timeout is generated automatically when a timer expires and is sent to the timer owner. A timer can either be private or public. If private, the timer can only be accessed by its owned active class. If public, anyone with sufficient visibility may access and manipulate the timer.

Attributes

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

Operations

- start(expires : Time)— Starts the timer and sets the time of expiration.
- stop() — Stops the timer.
- read(): Time— Reads the expiration time of the timer.

Constraints

[1] Only active classes may own properties realizing the timer interface.

[2] Timer may only be started with positive expiration time.

Semantics

Timers can be started, stopped, and checked by StartTimerAction, StopTimerAction, TimerRunningAction, and ReadTimerAction. A timeout is generated when the timer expires. A public timer is allowed to be started or stopped by any other active class. Start an active timer means a restart of the timer.

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 sections for StartTimerAction, StopTimerAction, and ReadTimerAction.

Examples

An example for a timer definition can be found in Figure 6.25 on page 53.

Timezone (a predefined type)

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.

Semantic Variation Point

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

StartTimerAction

Description

An action used to start a 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.

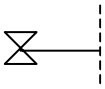
[3] The argument of the action must be a Time value.

Semantics

The StartTimerAction starts a timer. The StartTimerAction on a running timer restarts the timer.

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 start timer action example can be found in Figure 6.27 on page 54.

StopTimerAction

Description

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

Constraints

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

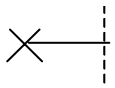
[2] The target of the action must refer to a classifier realizing the Timer interface.

Semantics

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

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 stop timer action example can be found in Figure 6.27 on page 54.

ReadTimerAction

Description

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

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.

Semantics

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

Notation

The notation for a read timer action is an ordinary action with the read timer stereotype applied.

TimerRunningAction

Description

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

Semantics

The TimerRunningAction checks the running status of a timer.

Notation

The notation for a timer running action is an ordinary action with the timer running action stereotype applied.

6.4 MOF-based Metamodel

This section 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 Testing Profile standard. The compliance provided by the MOF-based metamodel is limited to the architecture elements of the Testing Profile enabling traceability and management of test assets across tools. Specifically, the behavioral aspects of the testing profile are left out of the current metamodel as they would not provide any susbstantial 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 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 6.16 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.

6.4.1 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 6.13 presents the metamodel diagram for the Test Architecture and Test Behavior concepts.

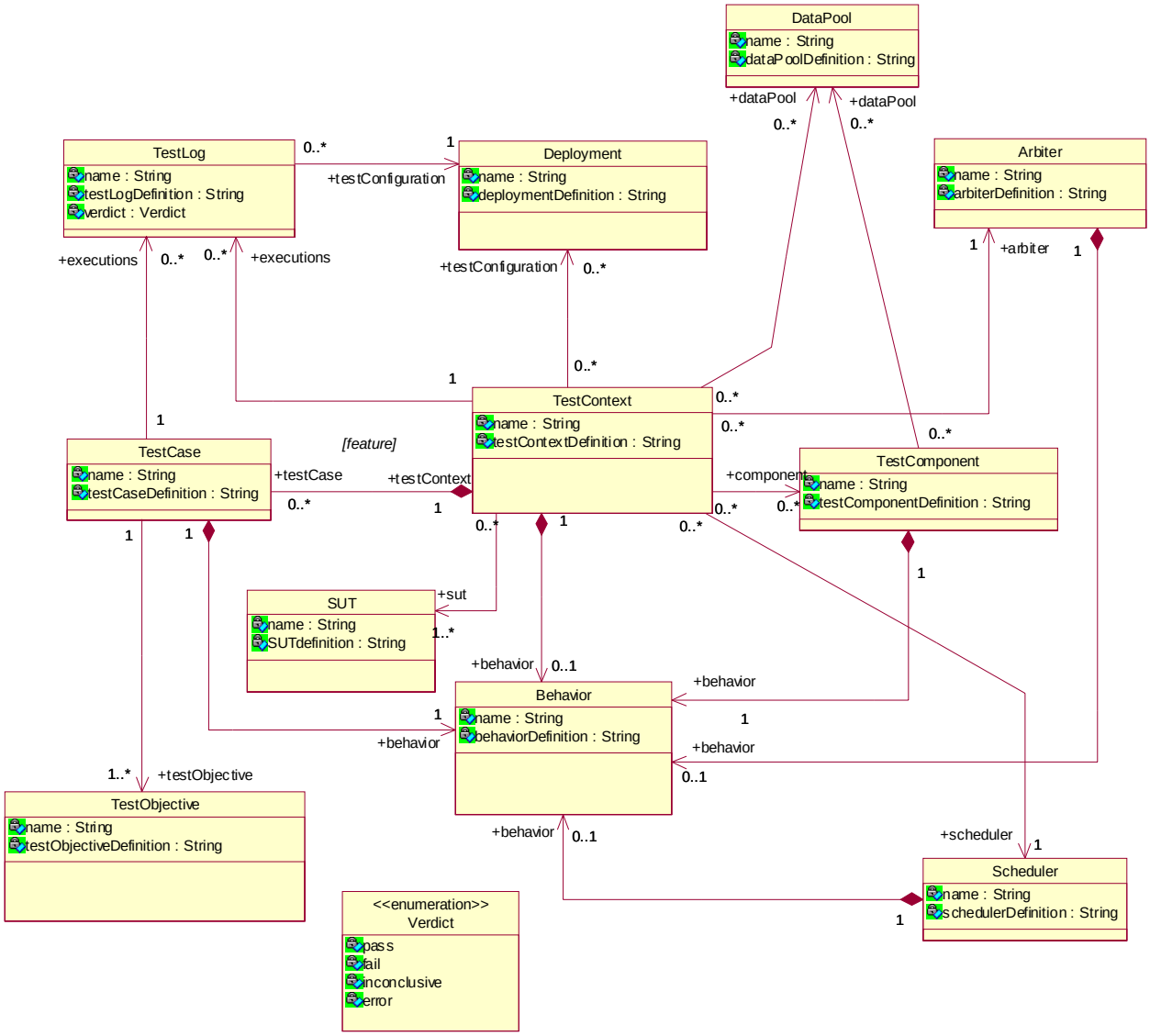


Figure 6.13 - Test architecture and test behavior portion of the MOF-based metamodel

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.

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 the 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).

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.

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).

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 6.16 on page 47.

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).

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).

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).

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.

Verdict

Verdict is defined exactly as in Section 6.3, “The Profile,” on page 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.

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.

6.5 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 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 6.14 is a diagram illustrating the data concepts.

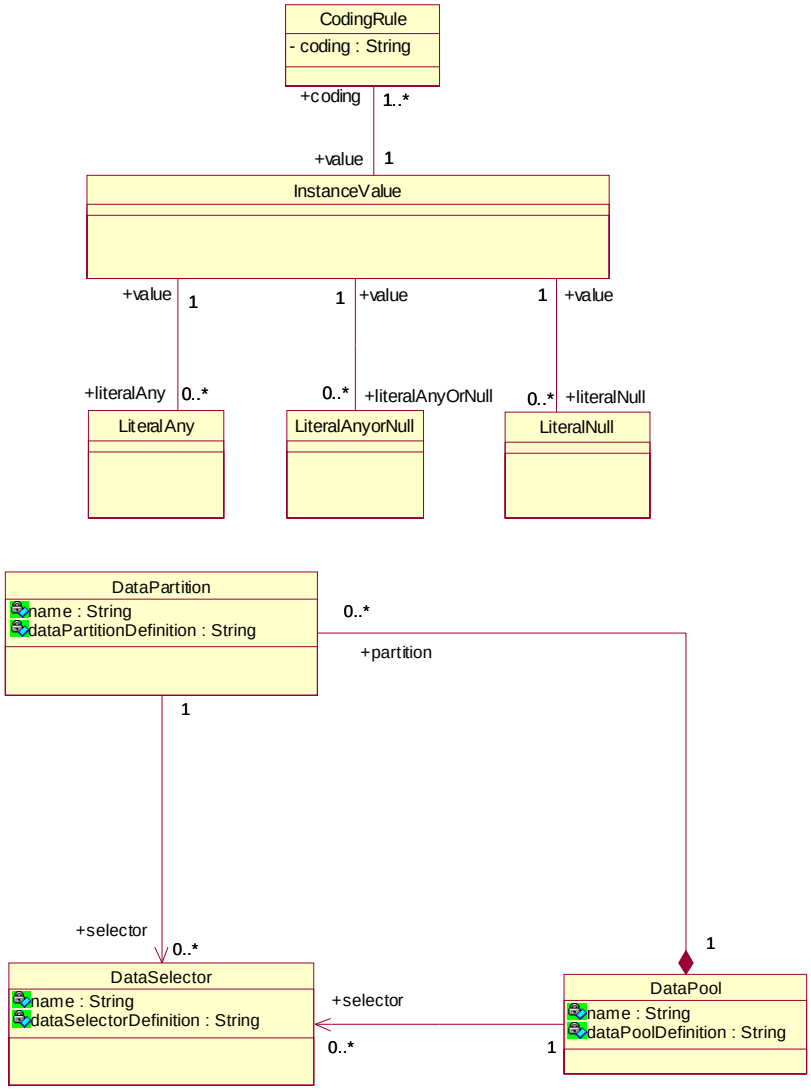


Figure 6.14 - Test data portion of the MOF-based metamodel

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.

CodingRule

Semantics

A rule that specifies how an instance value is to be encoded. As defined in Section 6.3, “The Profile,” on page 10, it contains a string attribute that specifies the coding scheme.

Associations

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

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.

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.

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.

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.

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.

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.

6.5.1 Time

As discussed in Section 6.3.4, “Time Concepts,” on page 29, 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 6.15 illustrates the time concepts for the metamodel.

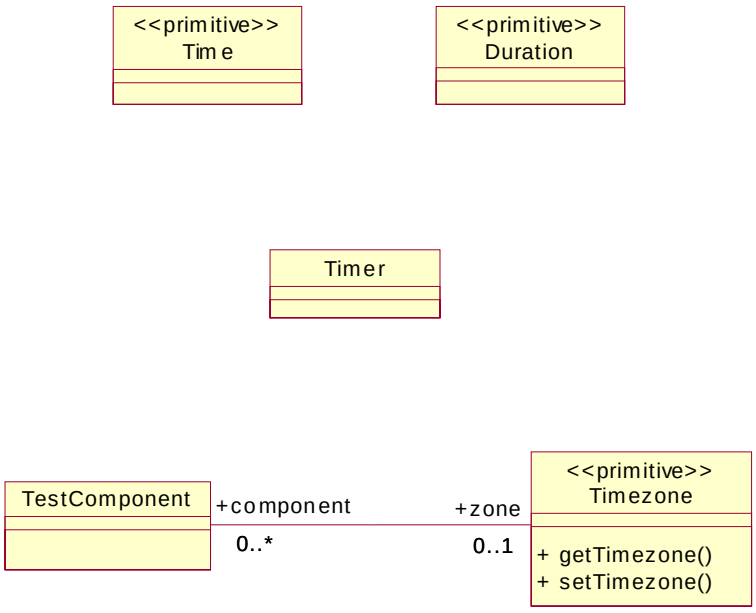


Figure 6.15 - Time portion of the MOF-based metamodel

Time

Time is defined exactly as in Section 6.3.4, “Time Concepts,” on page 29.

Duration

Duration is defined exactly as in Section 6.3.4, “Time Concepts,” on page 29.

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 6.16.

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.

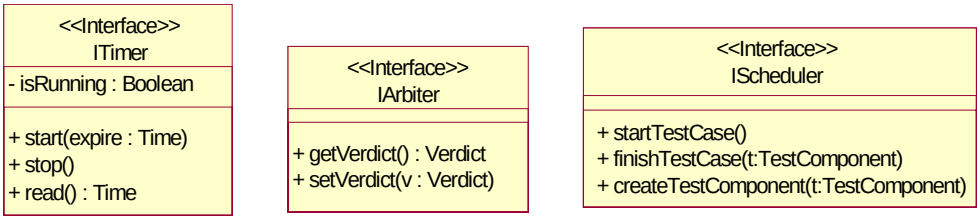


Figure 6.16 - Predefined interfaces for implementing timer, arbiter and scheduler

6.6 Examples

This section contains an example of using the 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 6.17 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.

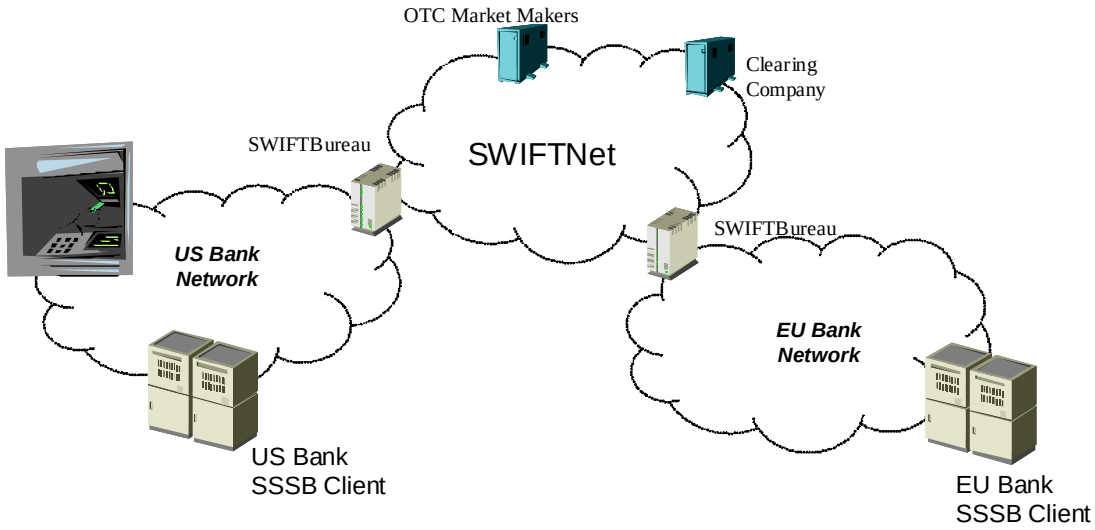


Figure 6.17 - Overview on the InterBank Exchange Network

The packages for this example are shown in Figure 6.18. They are used in the subsequent sections.

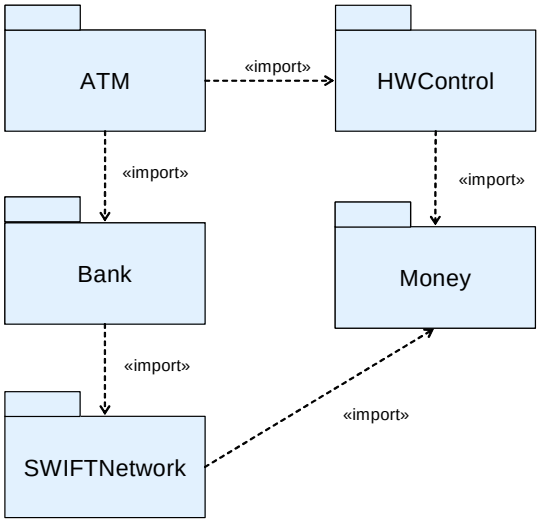


Figure 6.18 - Packages for the InterBank Exchange Network

6.6.1 Money Example

This subsection illustrates the use of the 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.

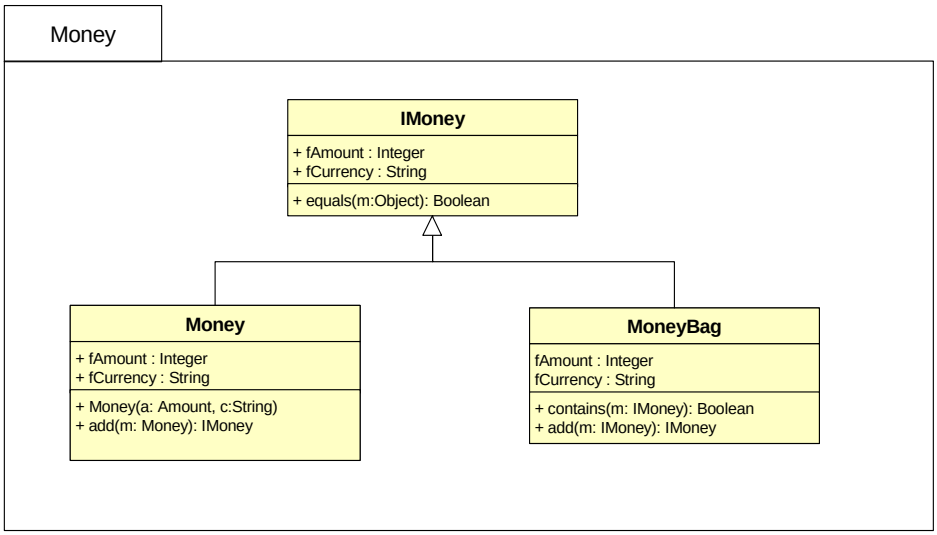


Figure 6.19 - Money structure

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.

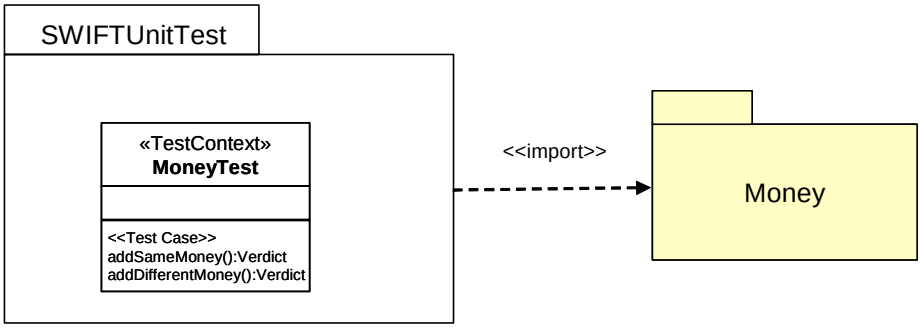


Figure 6.20 - SWIFTUnitTest package

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.

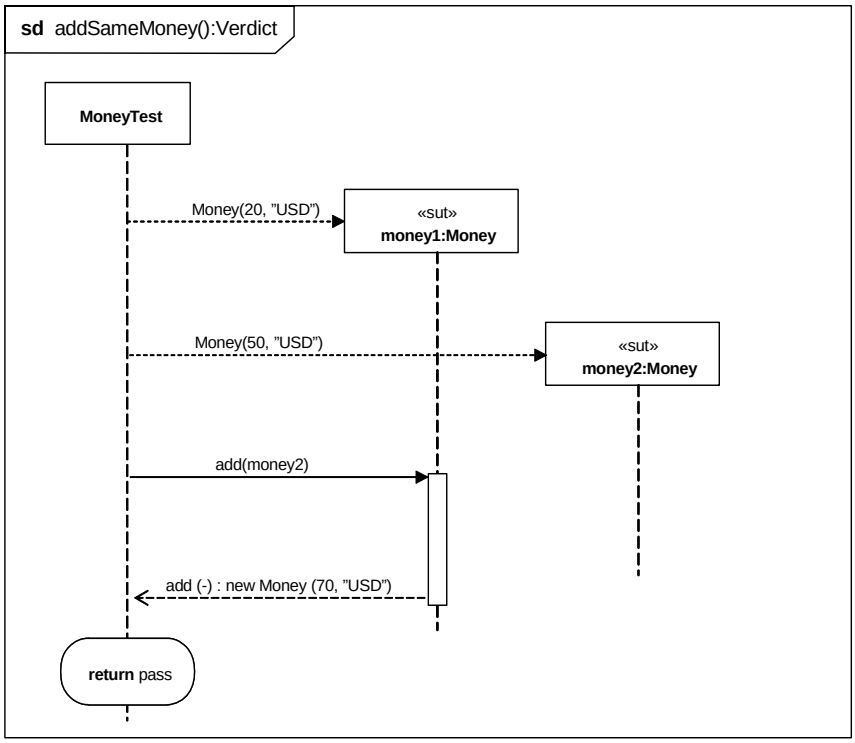


Figure 6.21 - Unit test behavior for addSameMoney

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.

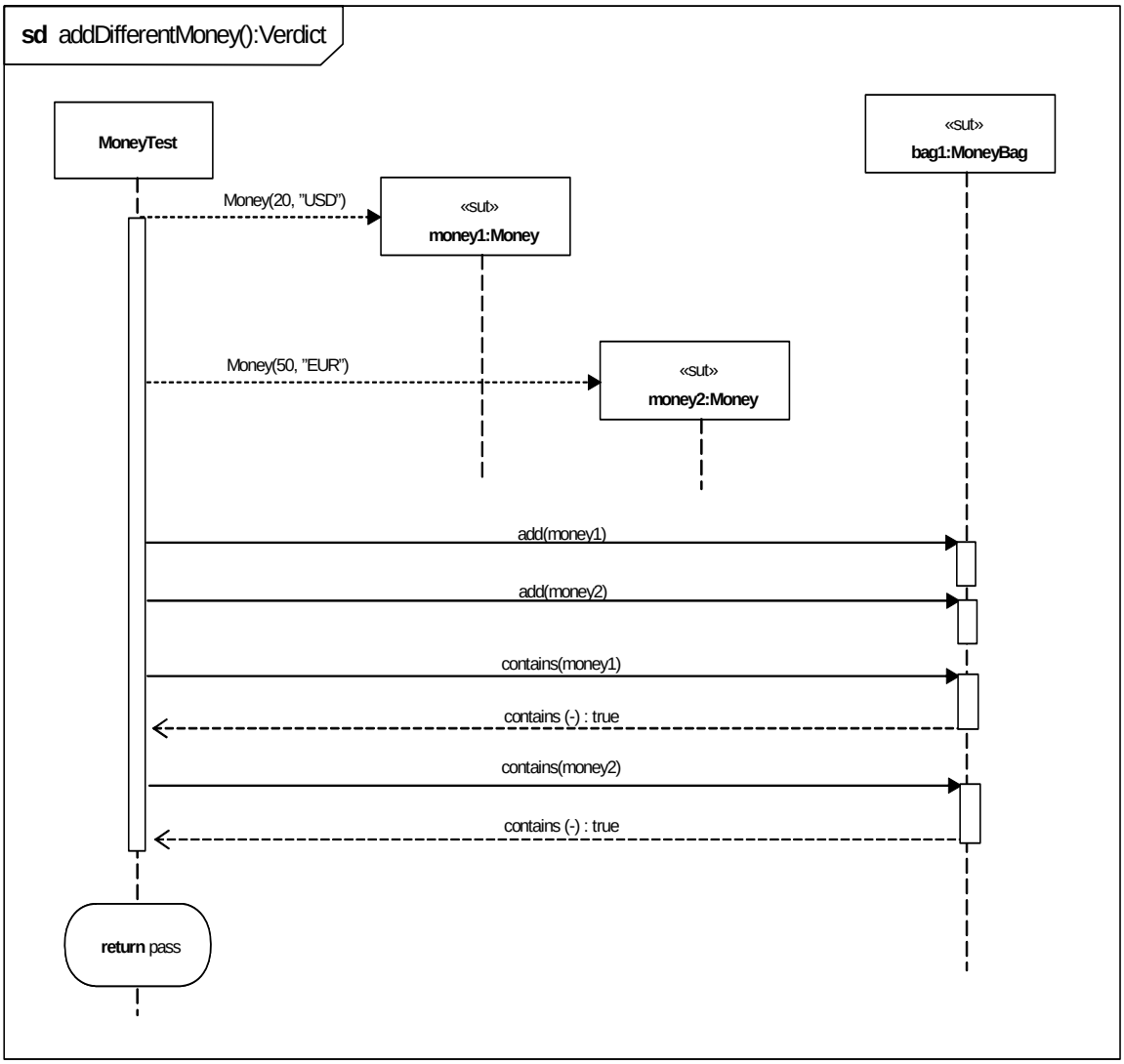


Figure 6.22 - Unit test behavior for addDifferentMoney

6.6.2 Bank ATM Example

This part of the example illustrates how the 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 6.23 shows the packages involved in this part of the example.

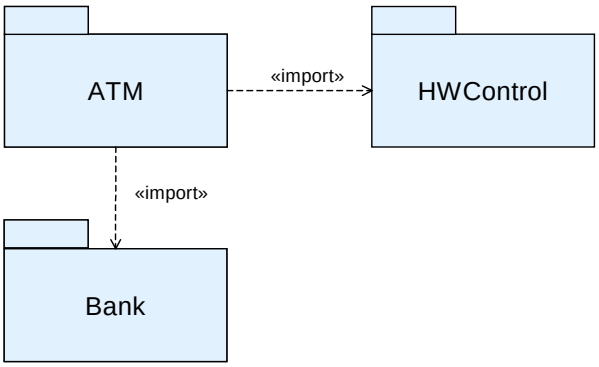


Figure 6.23 - ATM and related packages

Figure 6.24 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.

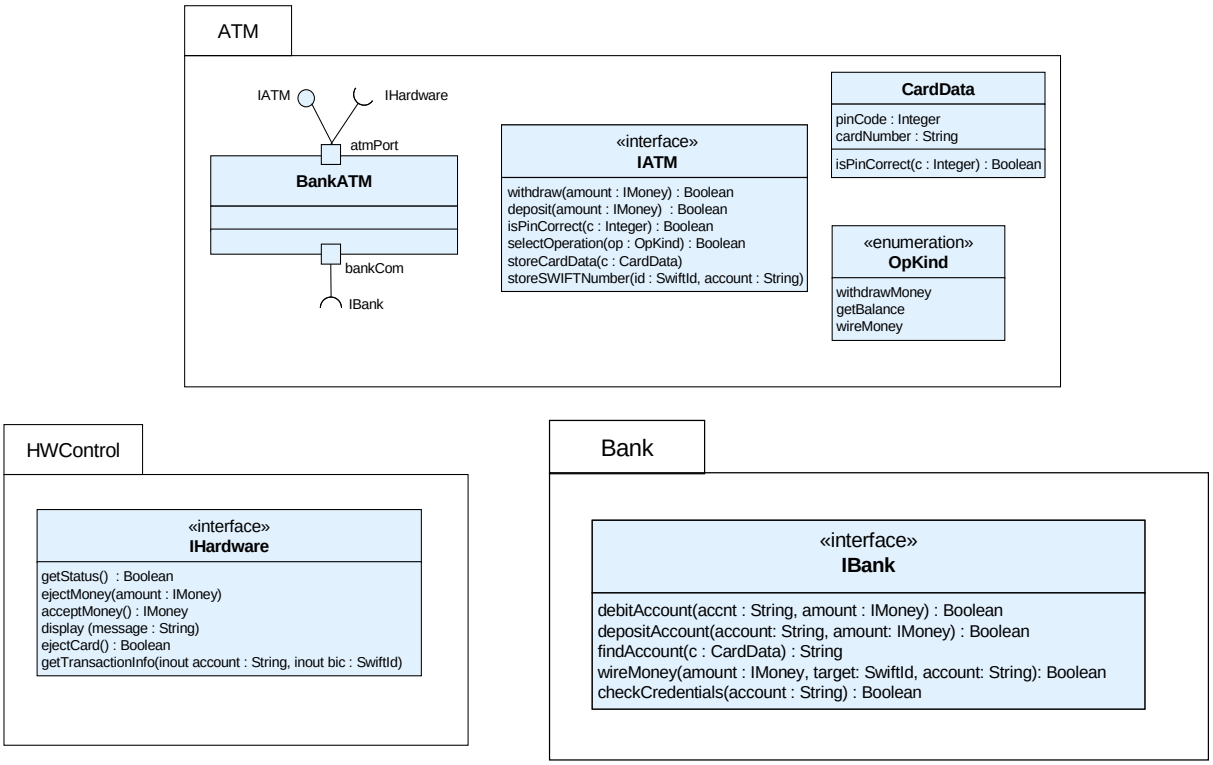


Figure 6.24 - Elements of the system to be tested

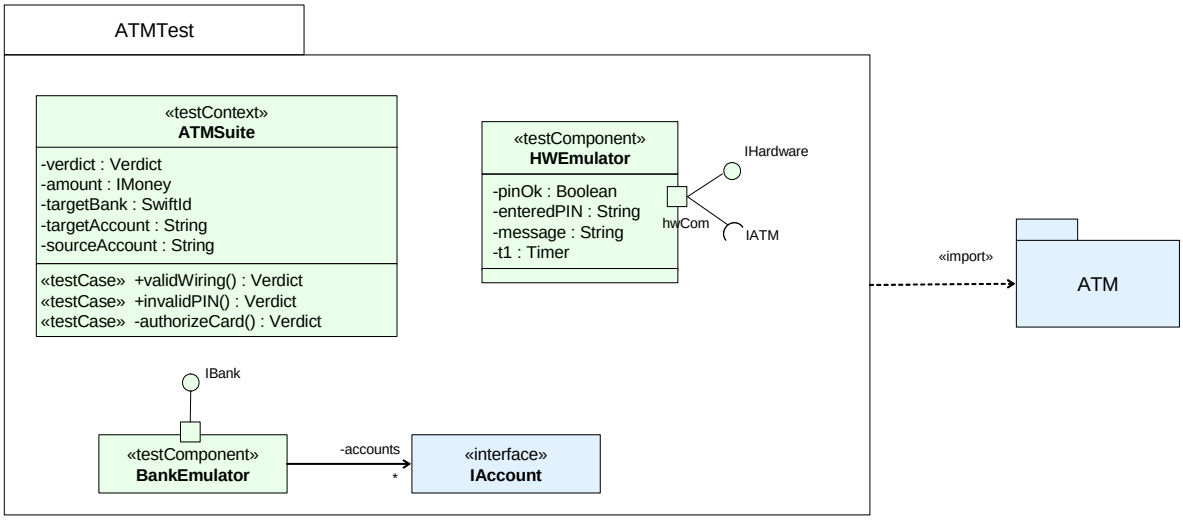


Figure 6.25 - The ATMTest package

The ATMTest package shown in Figure 6.25 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.

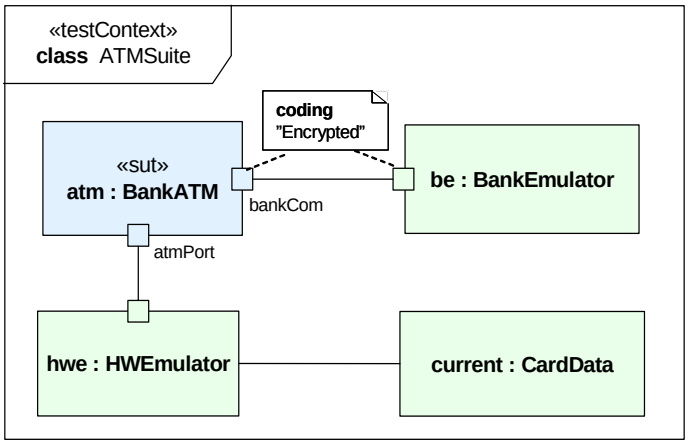


Figure 6.26 - The composite structure of the ATMSuite test context

The test configuration (i.e., the composite structure of the test context) is shown in Figure 6.26. 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.

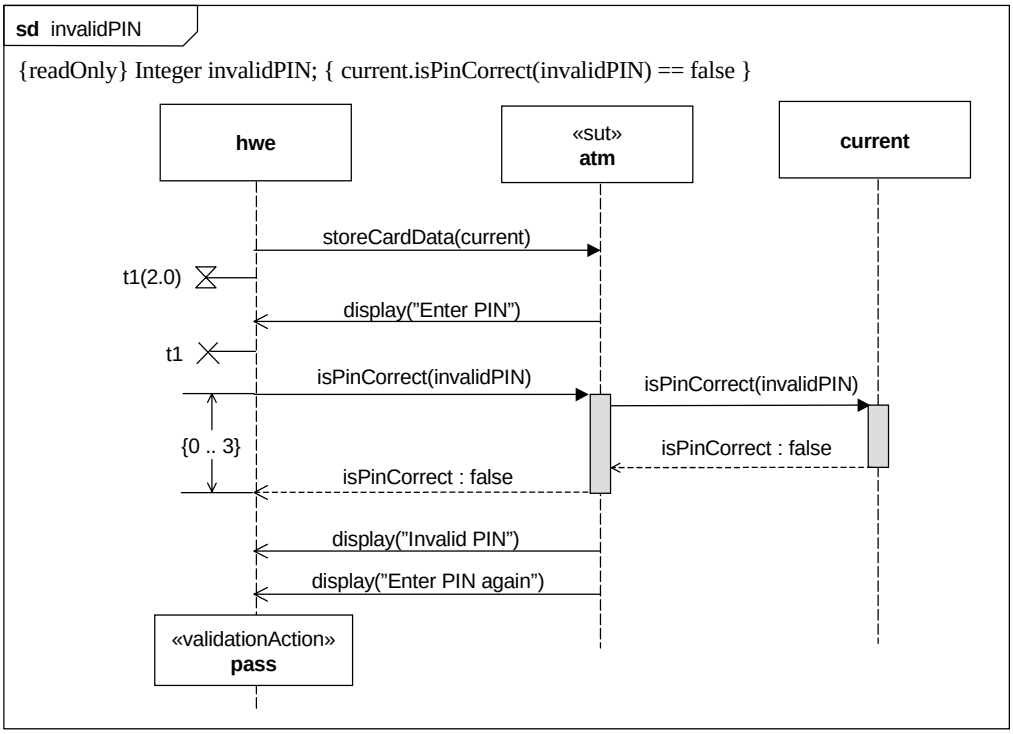


Figure 6.27 - The behavior of the invalidPIN test case

The sequence diagram in Figure 6.27 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 6.31 for details. The timer t1 is an attribute of the HWEmulator test component, see Figure 6.25.

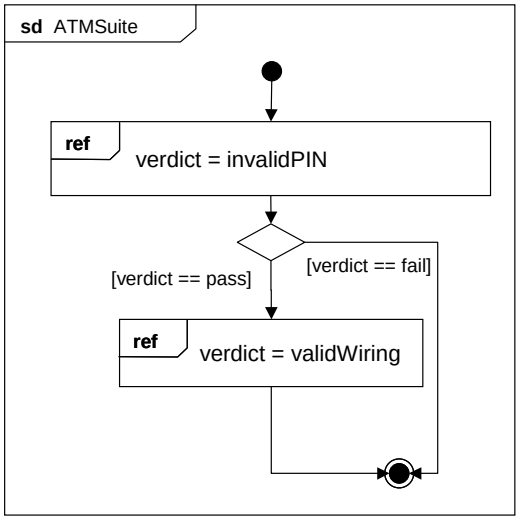


Figure 6.28 - Classifier behavior for ATMSuite

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 6.28 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.

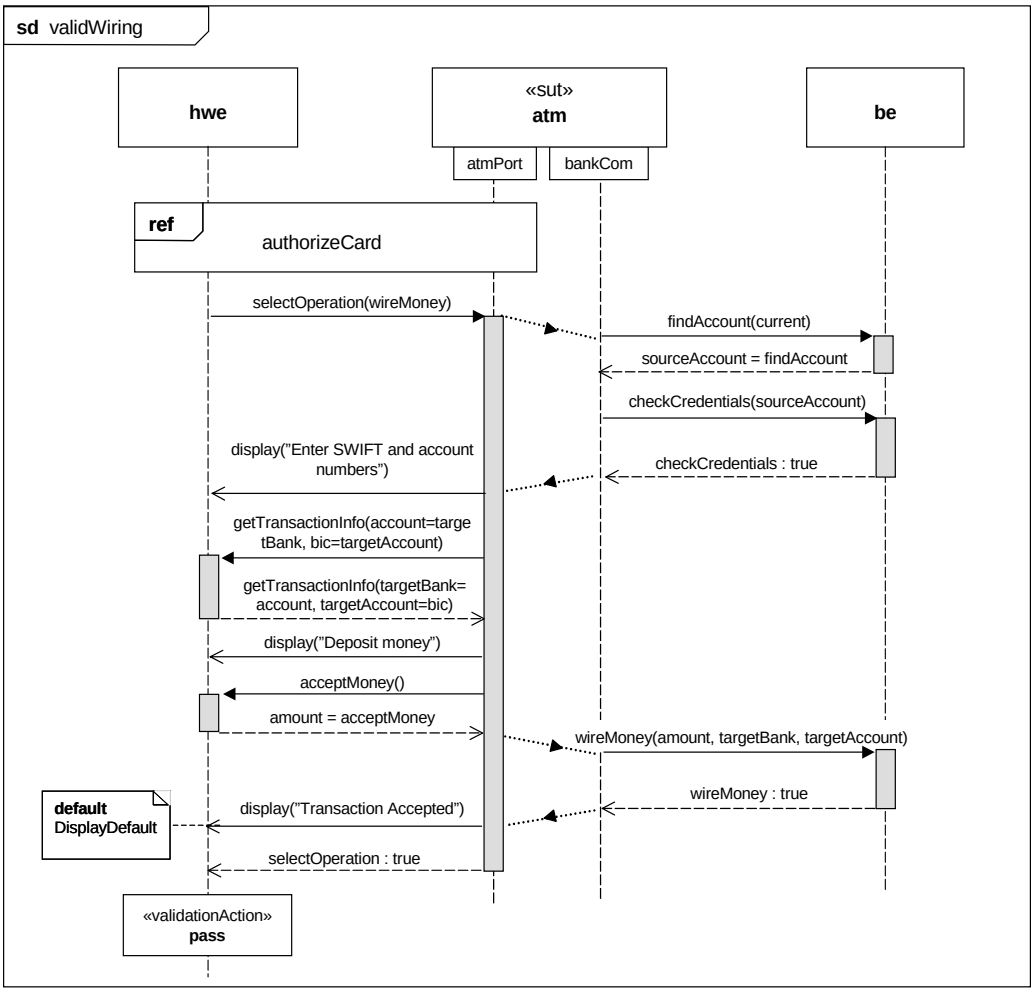


Figure 6.29 - The validWiring test case

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).

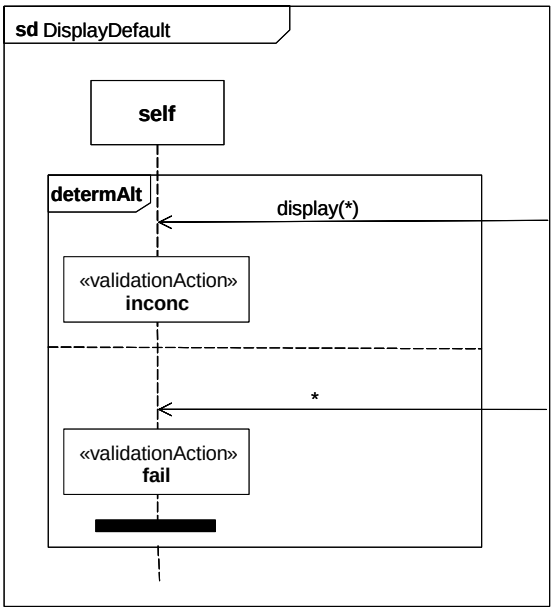


Figure 6.30 - Default for individual message reception

Figure 6.30 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.

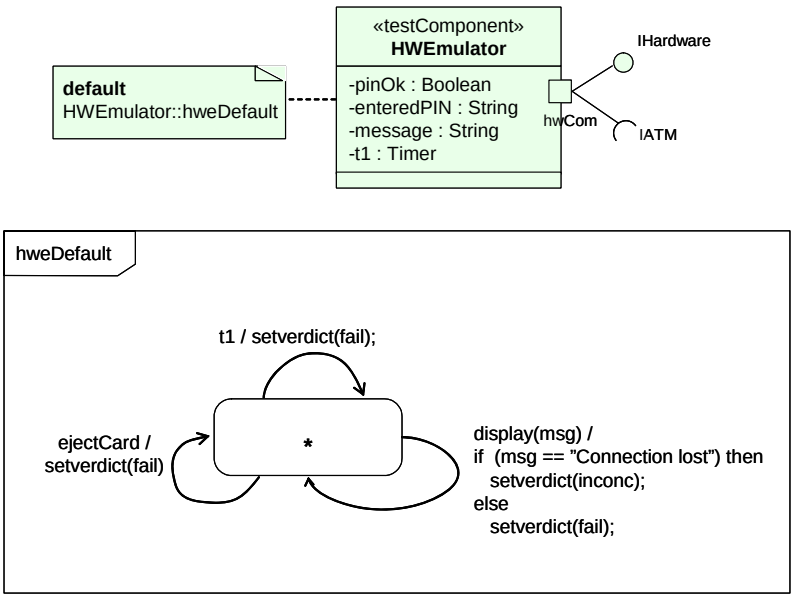


Figure 6.31 - A statemachine default applied to a test component

Figure 6.31 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.

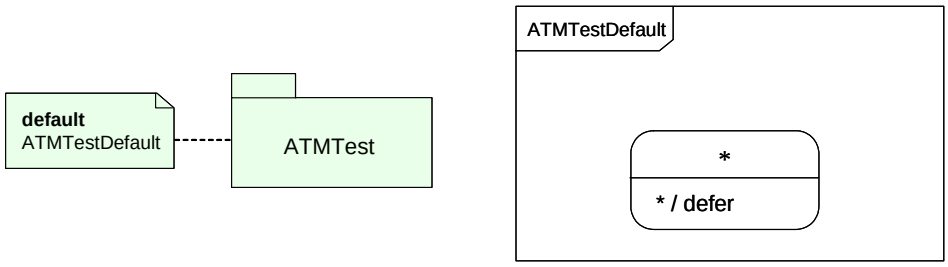


Figure 6.32 - Package level default

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 6.32 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.

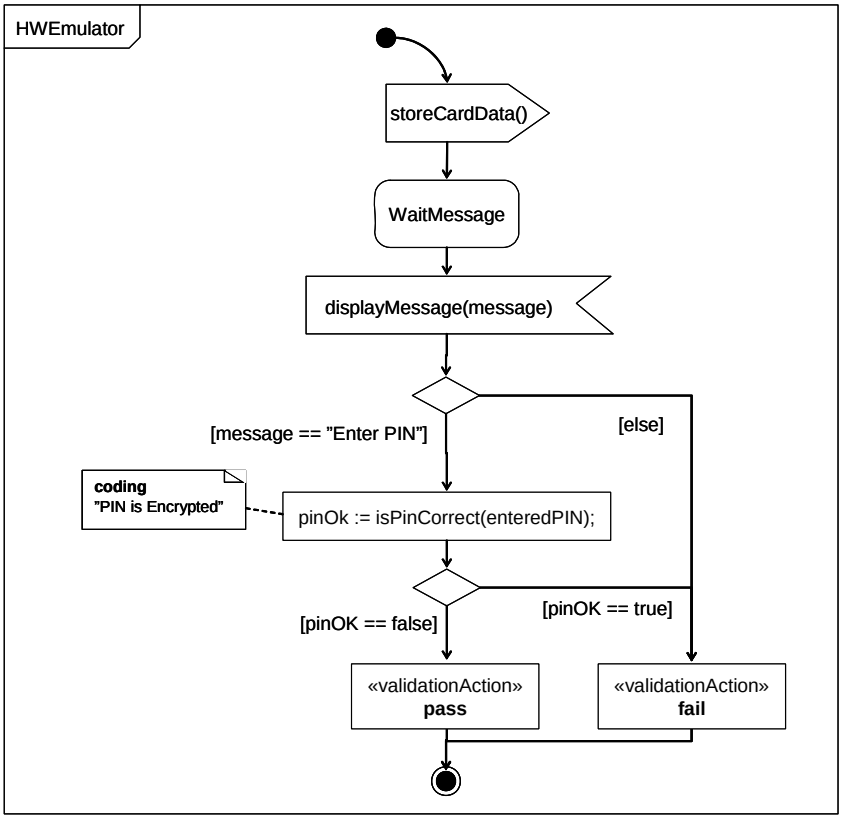


Figure 6.33 - Test component behavior

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 6.33 contains parts of the test behavior for the HWEmulator test component. (The part corresponding to the invalidPIN test case.)

6.6.3 Money Transfer Example

This subsection contains an example of using the 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 sections. This example illustrates several key concepts from the 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.

.

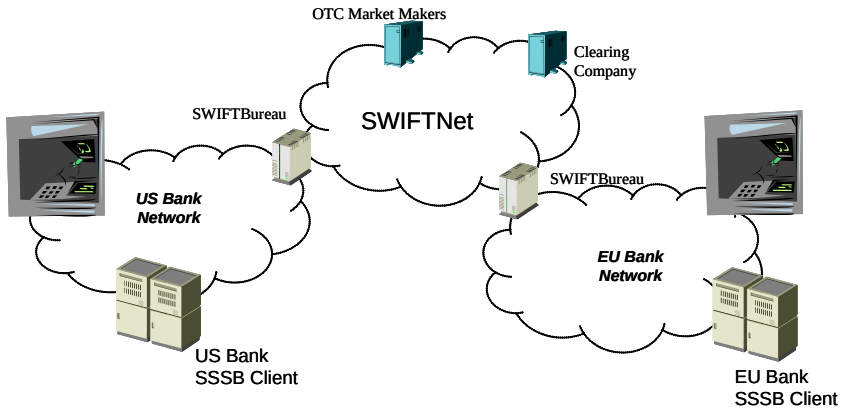


Figure 6.34 - Interbank Exchange Network overview

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 6.35 illustrates the package structure of the system under test.

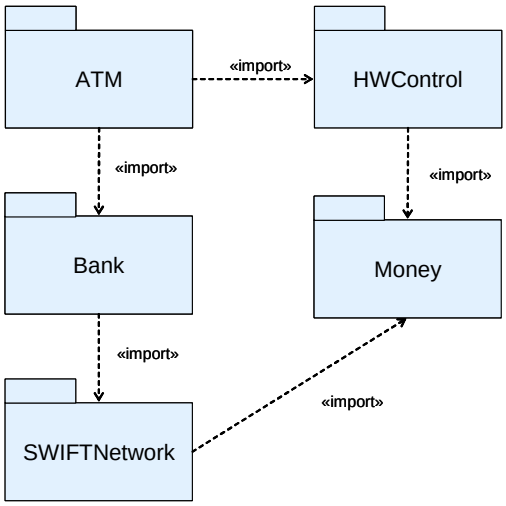


Figure 6.35 - Package structure of the Interbank Exchange Network

Figure 6.36 illustrates the contents of the Bank package, while Figure 6.37 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 subsection, although the example also uses interfaces and classes from the previous two examples.

.

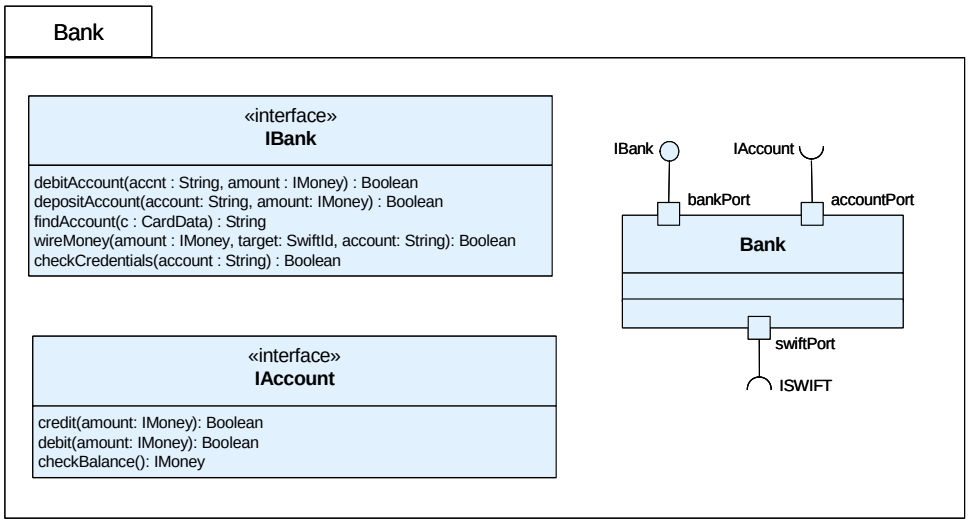


Figure 6.36 - BankNetwork

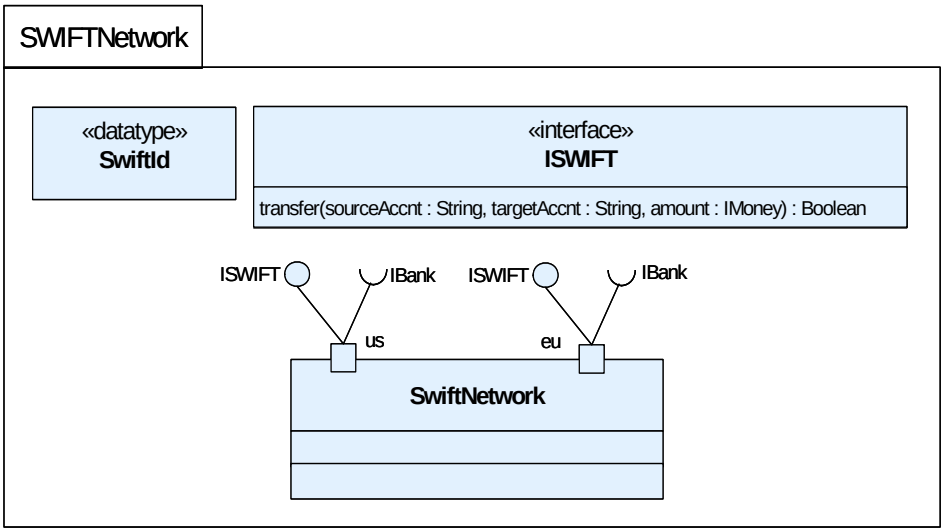


Figure 6.37 - SWIFTNetwork

Figure 6.38 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.

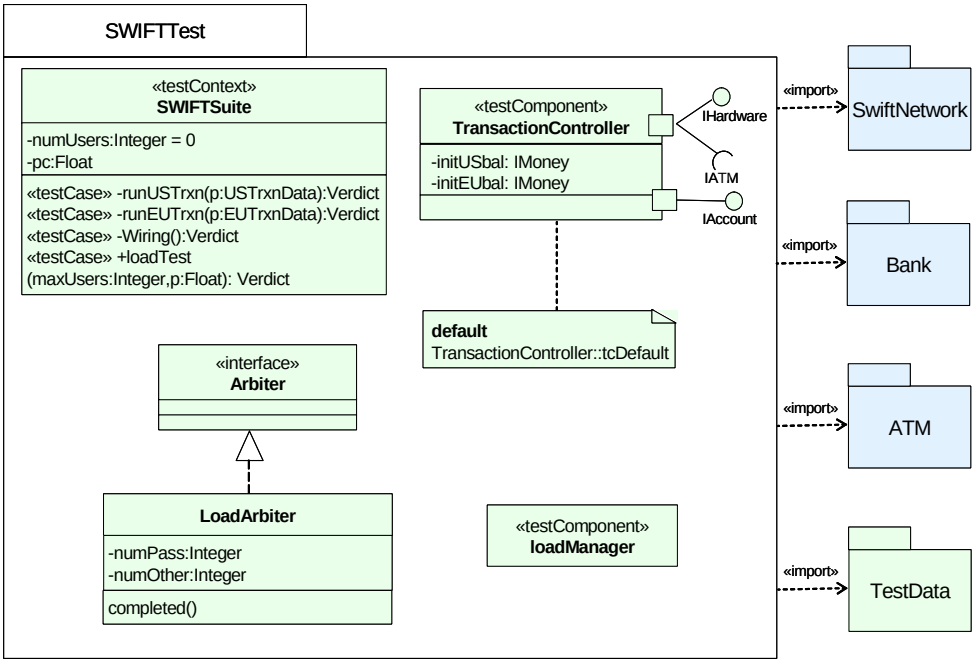


Figure 6.38 - SWIFTTest package

Figure 6.39 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.

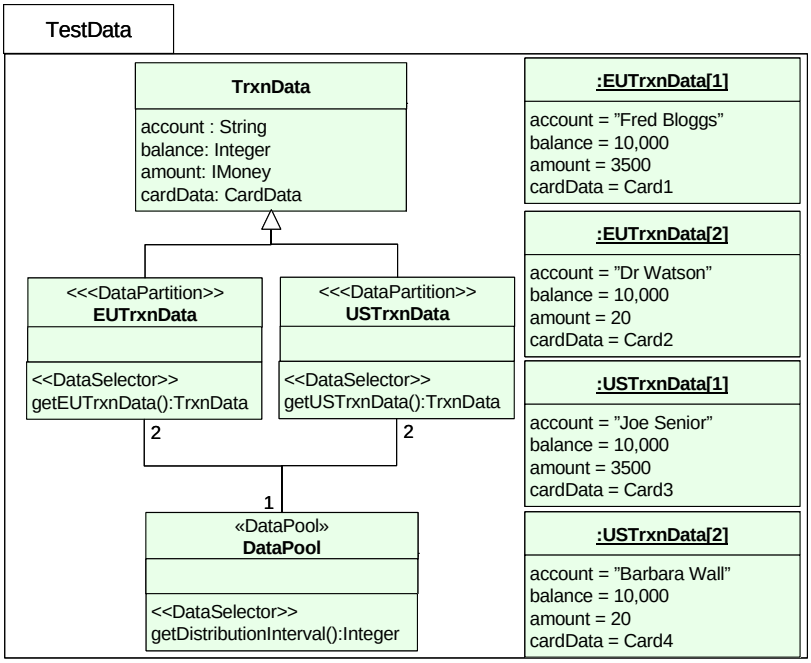


Figure 6.39 - TestData package

Figure 6.40 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.

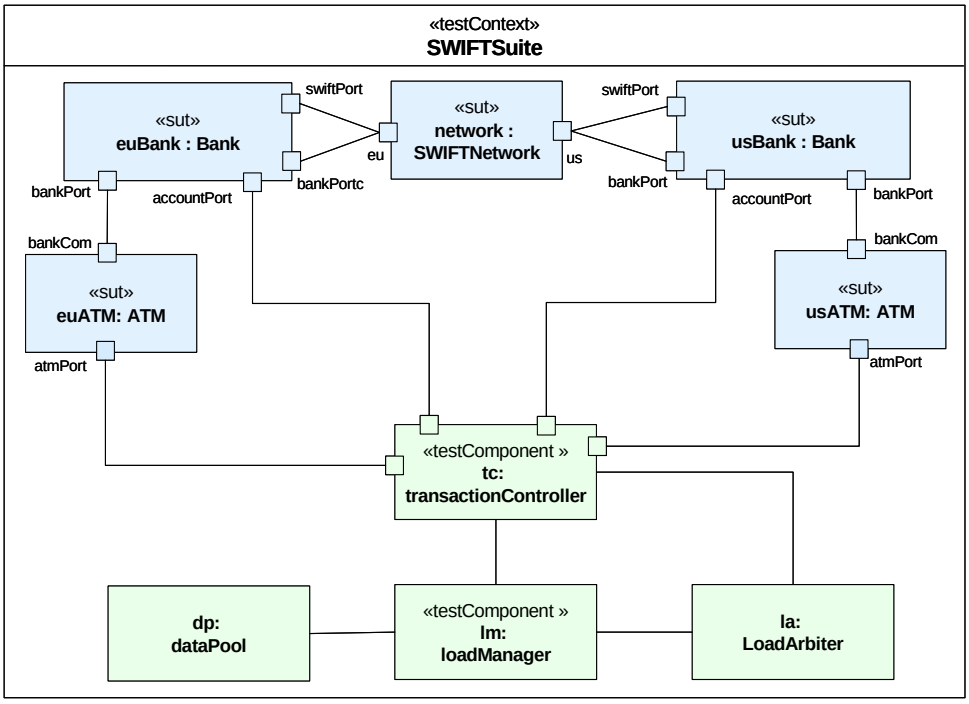


Figure 6.40 - The composite structure of the SWIFTSuite test context

Figure 6.41 and Figure 6.42 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 gneration. 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 6.43).

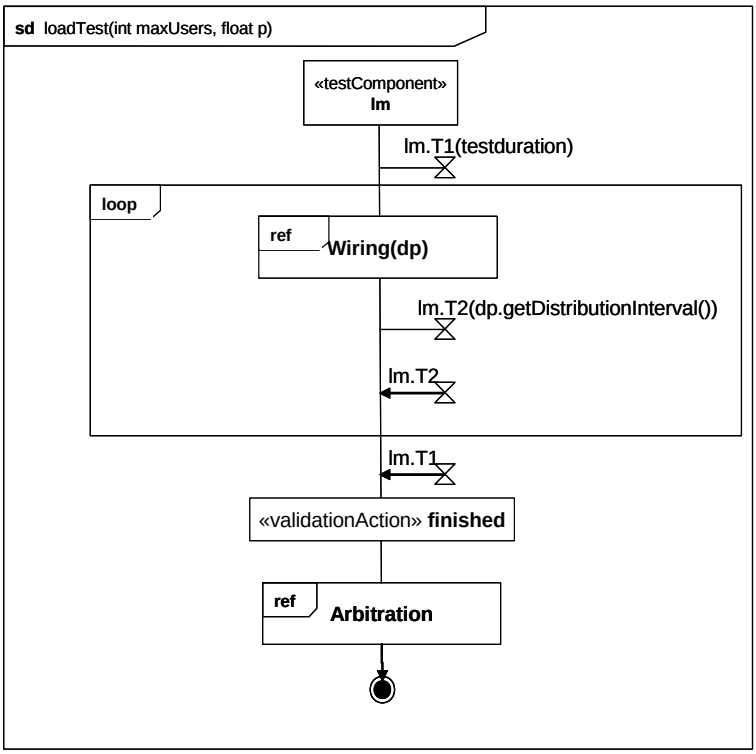


Figure 6.41 - Main test behavior

Figure 6.42 illustrates how an individual US wiring transaction is executed. The behavior is parameterized with the data pool dp. Firstly, an instance of the transaction controller is created and started. Then the test is started and a transation 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.

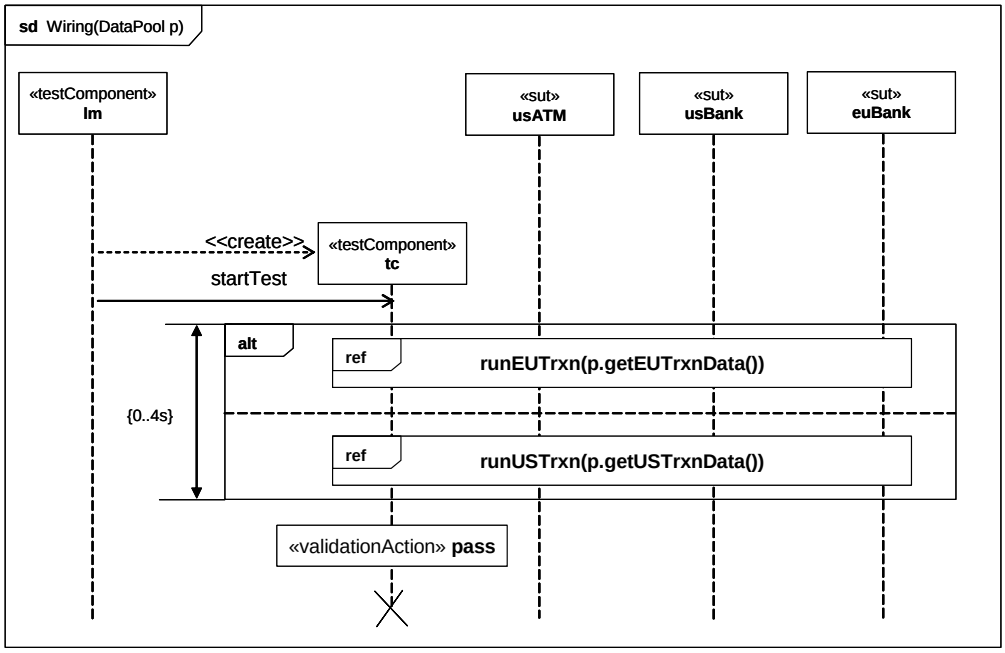


Figure 6.42 - US initiated wiring transaction

The Arbitration behavior is given in Figure 6.43. A pass verdict is assigned if the number of successful tests exceeds the given threshold, otherwise fail is assigned.

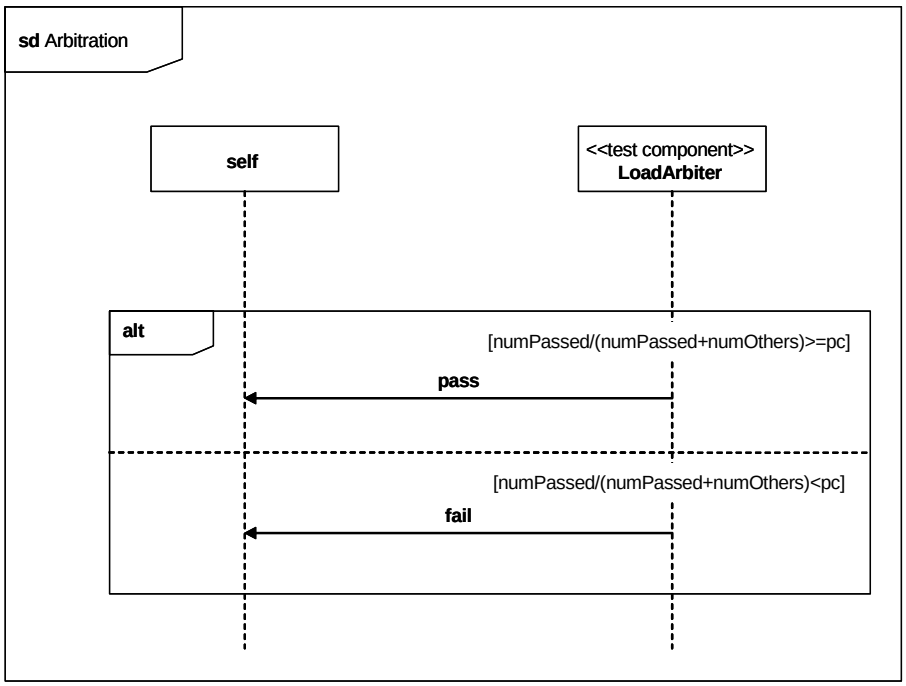


Figure 6.43 - Arbitration behavior

Figure 6.44 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.

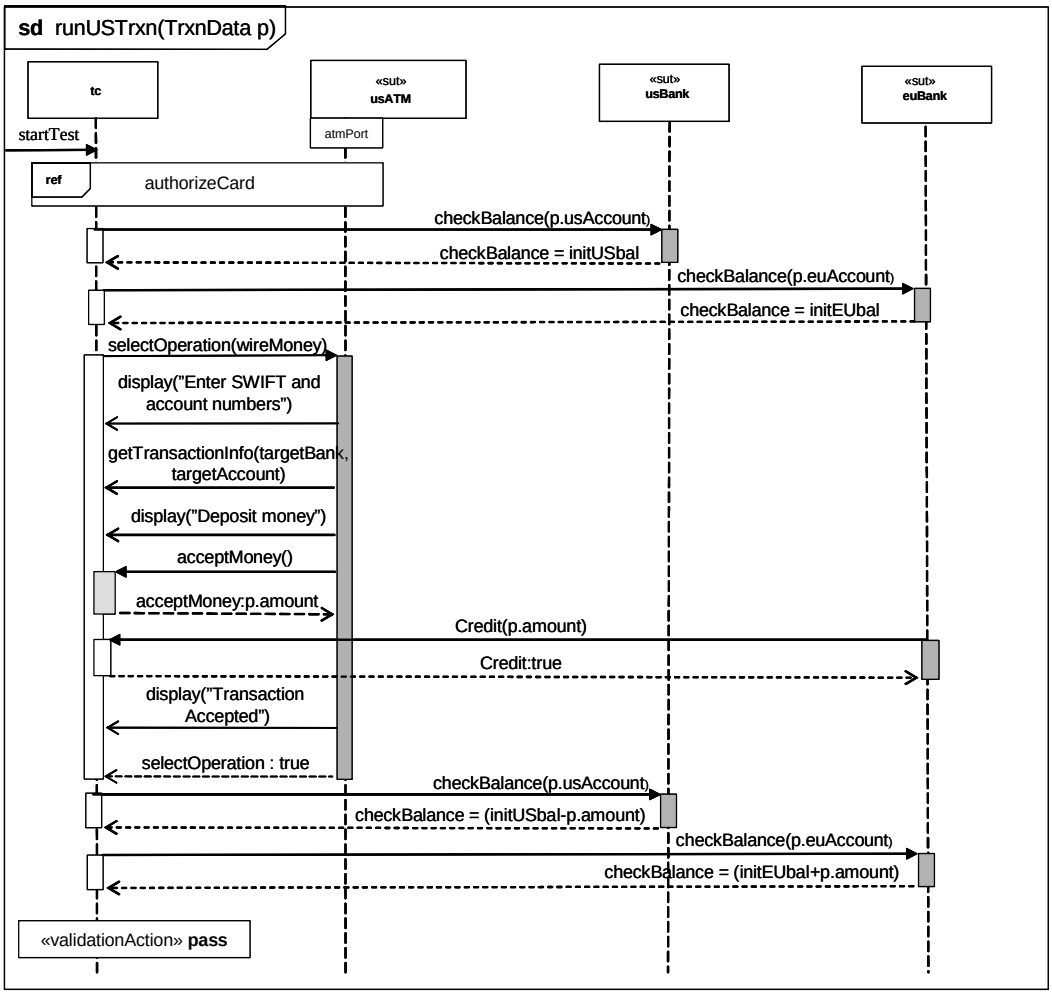


Figure 6.44 - Transaction detail

Figure 6.45 illustrates the behavior of the LoadArbiter component, which determines whether or not the load test was successful.

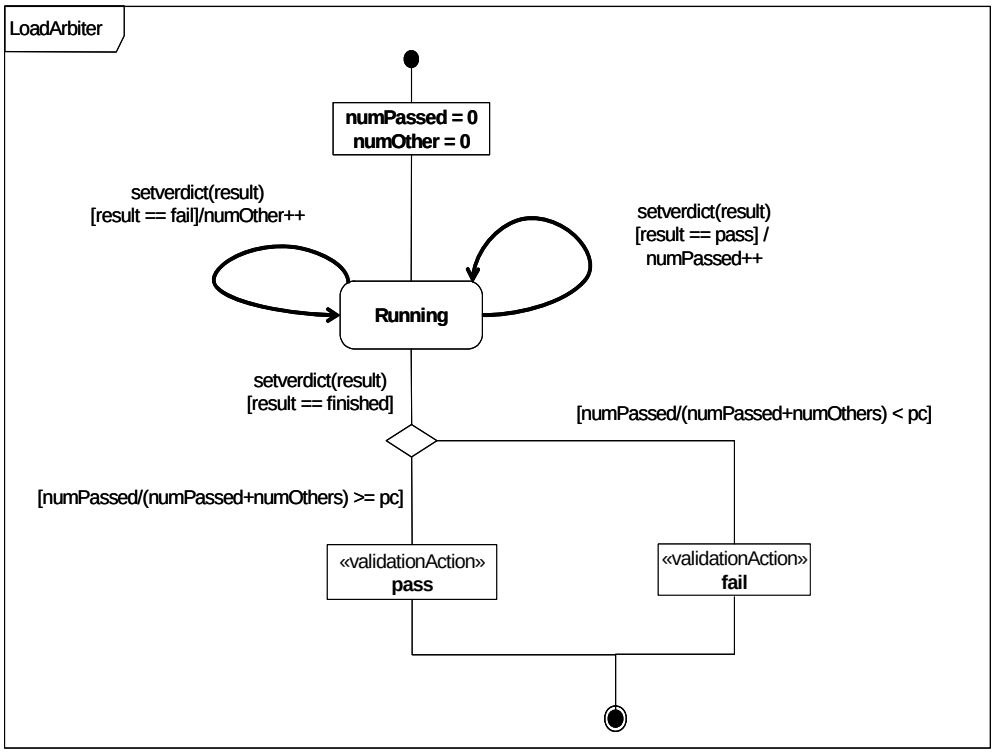


Figure 6.45 - LoadArbiter behavior

6.7 Mappings

6.7.1 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 section 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:

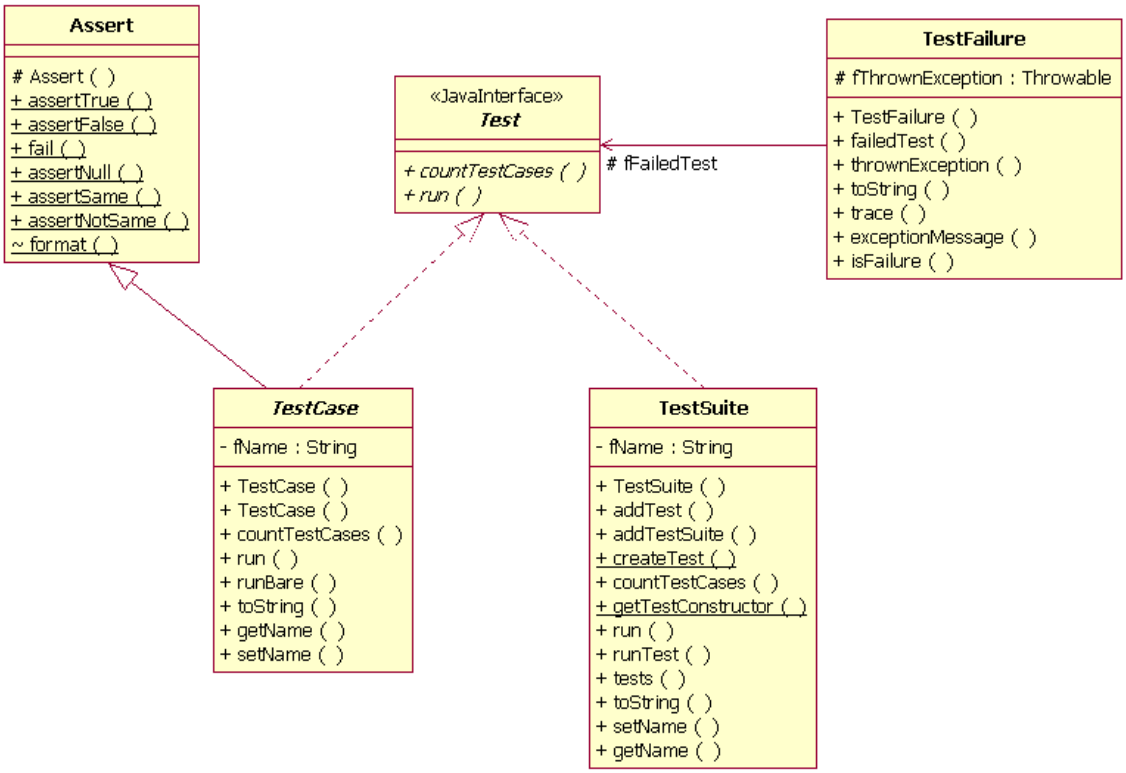


Figure 6.46 - JUnit framework overview

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

Table 6.1 - Comparison between 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.

Table 6.1 - Comparison between Testing Profile and JUnit concepts

UML Testing Profile	JUnit
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 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:	
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.
Scheduler	The scheduler can be realized as a property of Test Context. There should be a default scheduler along the protocol defined in Annex C.
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.

Table 6.1 - Comparison between Testing Profile and JUnit concepts

UML Testing Profile	JUnit
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 section - for the Test Cases given in Figure 6.21 and Figure 6.22.

```
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();
    }
}
```

6.7.2 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:

- type definitions for test data structures
- template definitions for concrete test data
- function and test case definitions for test behavior
- control definitions for the execution of test cases

TTCN-3 was one basis for the development of the Testing Profile. Still, they differ in several respects. The 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 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 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 6.2 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 6.2 - Comparison between 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.

Table 6.2 - Comparison between Testing Profile and TTCN-3 concepts

UML Testing Profile	TTCN-3
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.
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 Testing Profile default arbiter is a TTCN-3 built-in. User-defined arbiters are realized by the MTC.
Scheduler	There is a default scheduler built in TTCN-3. User defined schedulers can be 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.

Table 6.2 - Comparison between Testing Profile and TTCN-3 concepts

UML Testing Profile	TTCN-3
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 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 section - for the test case **invalidPIN** given in Figure 6.27.

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 6.3 - 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 A
(normative)

Profile Summary

A.1 The Profile

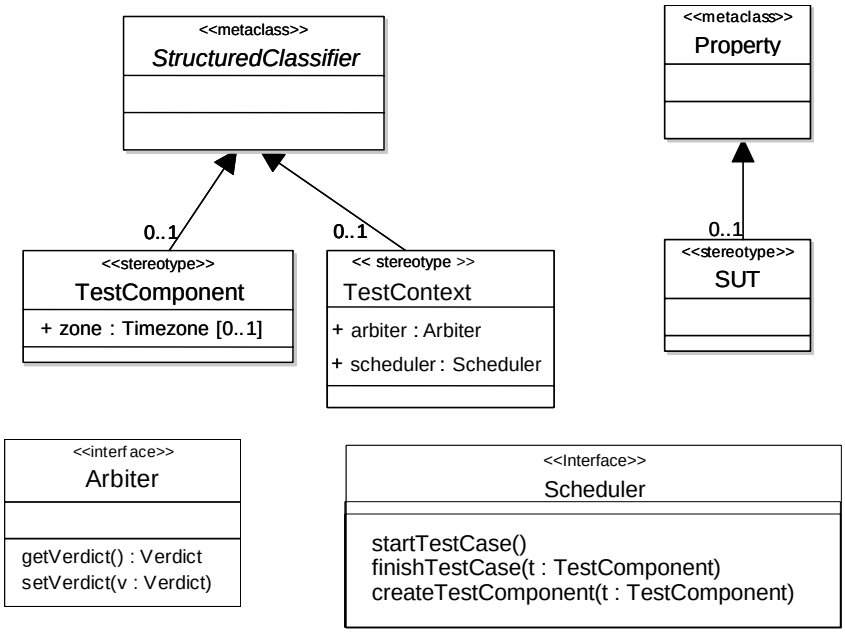


Figure A.1 -Test Architecture

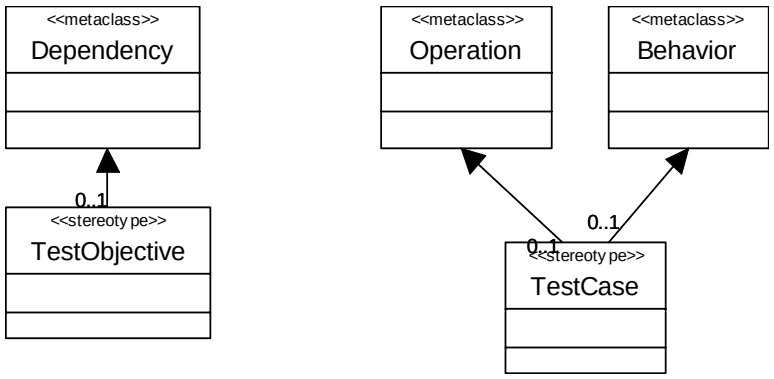


Figure A.2 - Test case and test objective

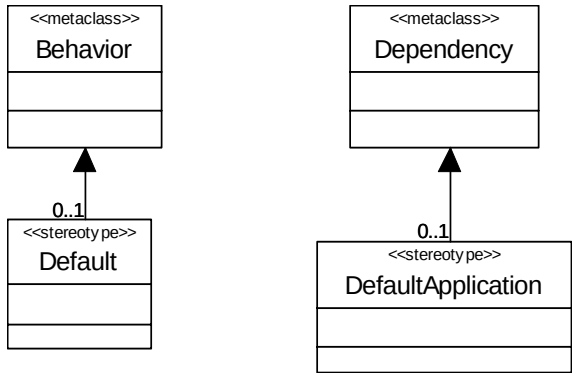


Figure A.3 - Defaults

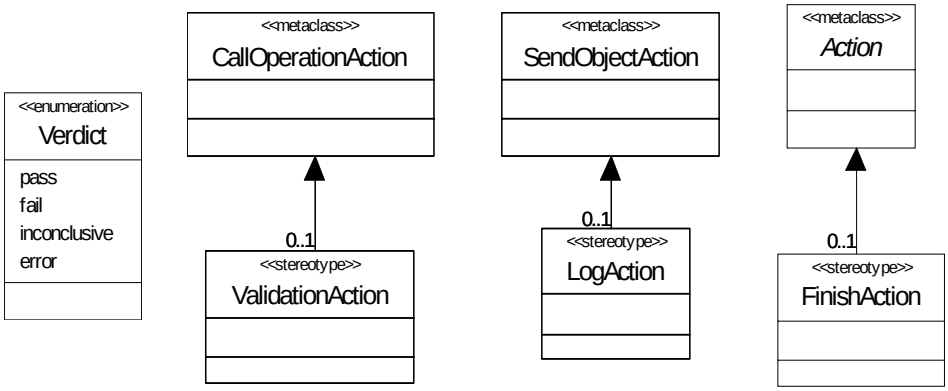


Figure A.4 - Actions

Figure A.5 - Deterministic Alt Operator (in Combined Fragements)

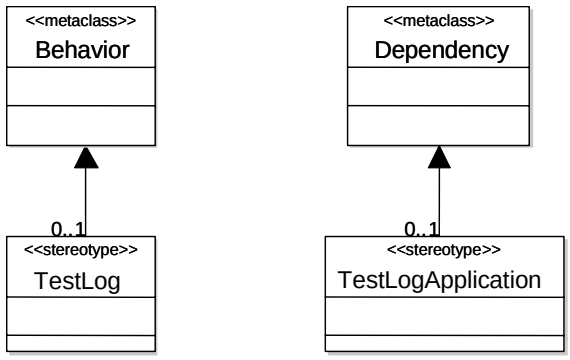


Figure A.6 - Test Log

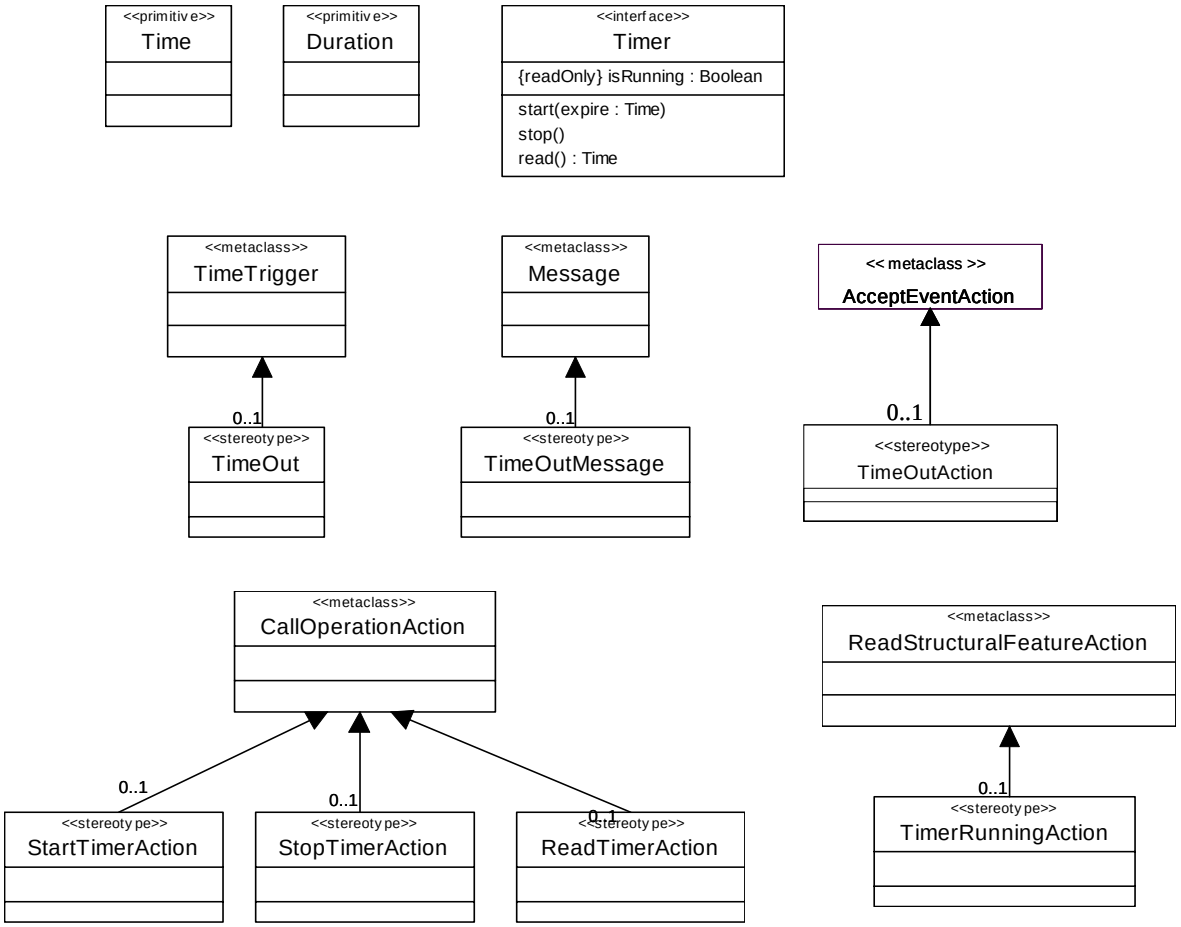


Figure A.7 - Timer concepts

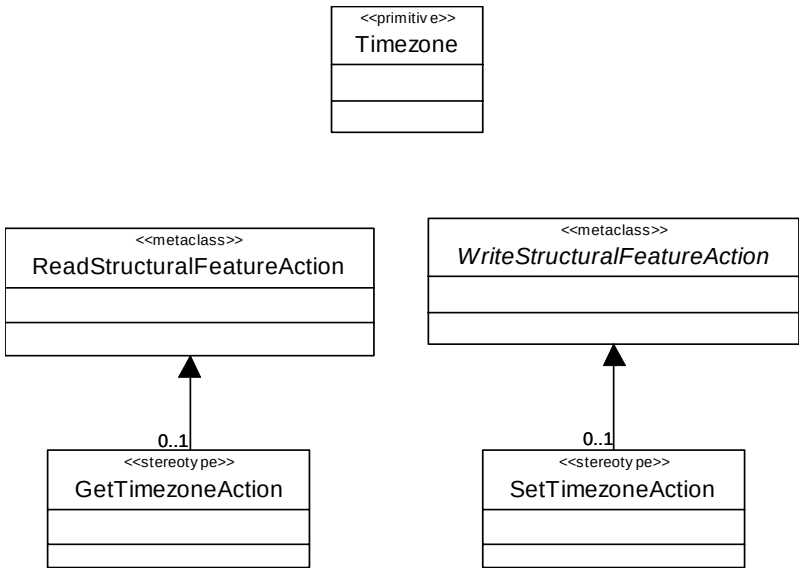


Figure A.8 - Timezone concepts

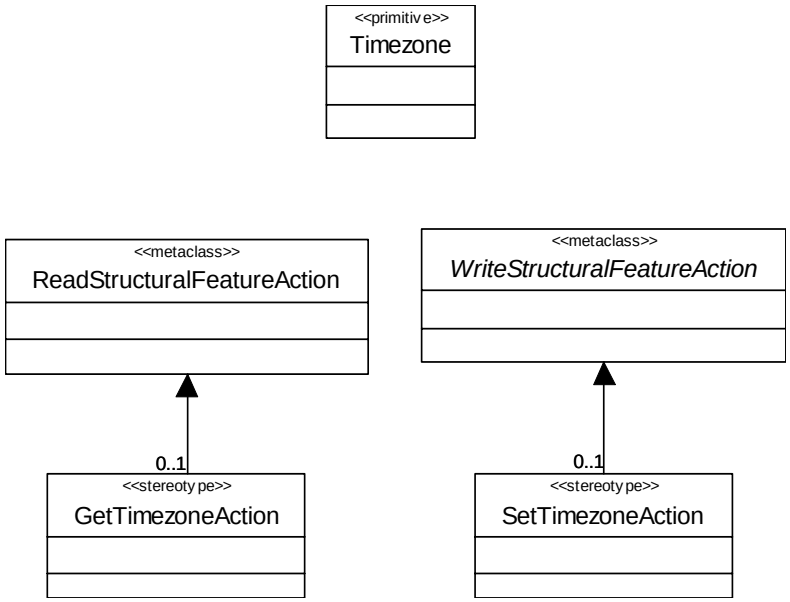


Figure A.9 - Timezone concepts

A.2 The MOF-based Metamodel

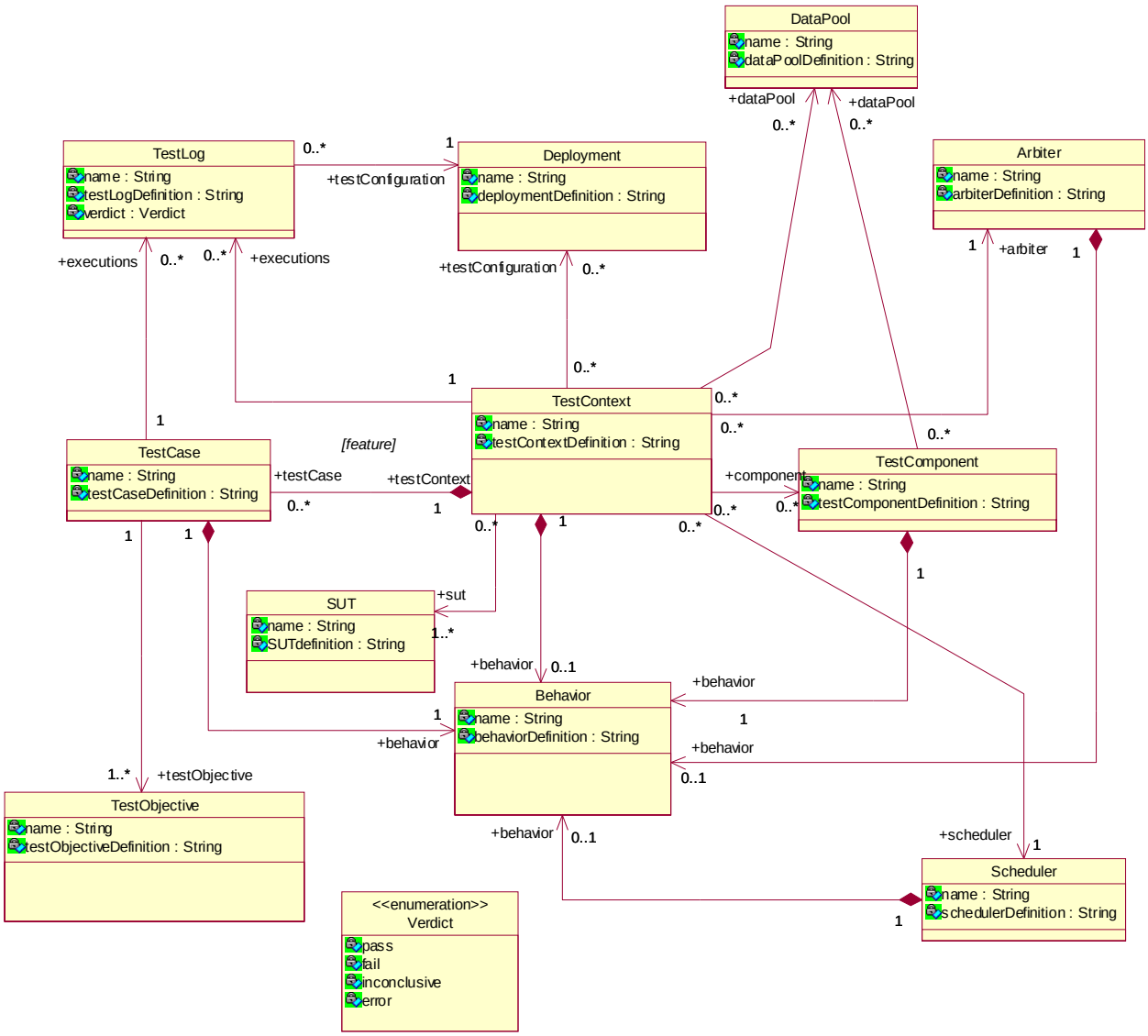


Figure A.10 - Test architecture and test behavior portion of the MOF-based metamodel

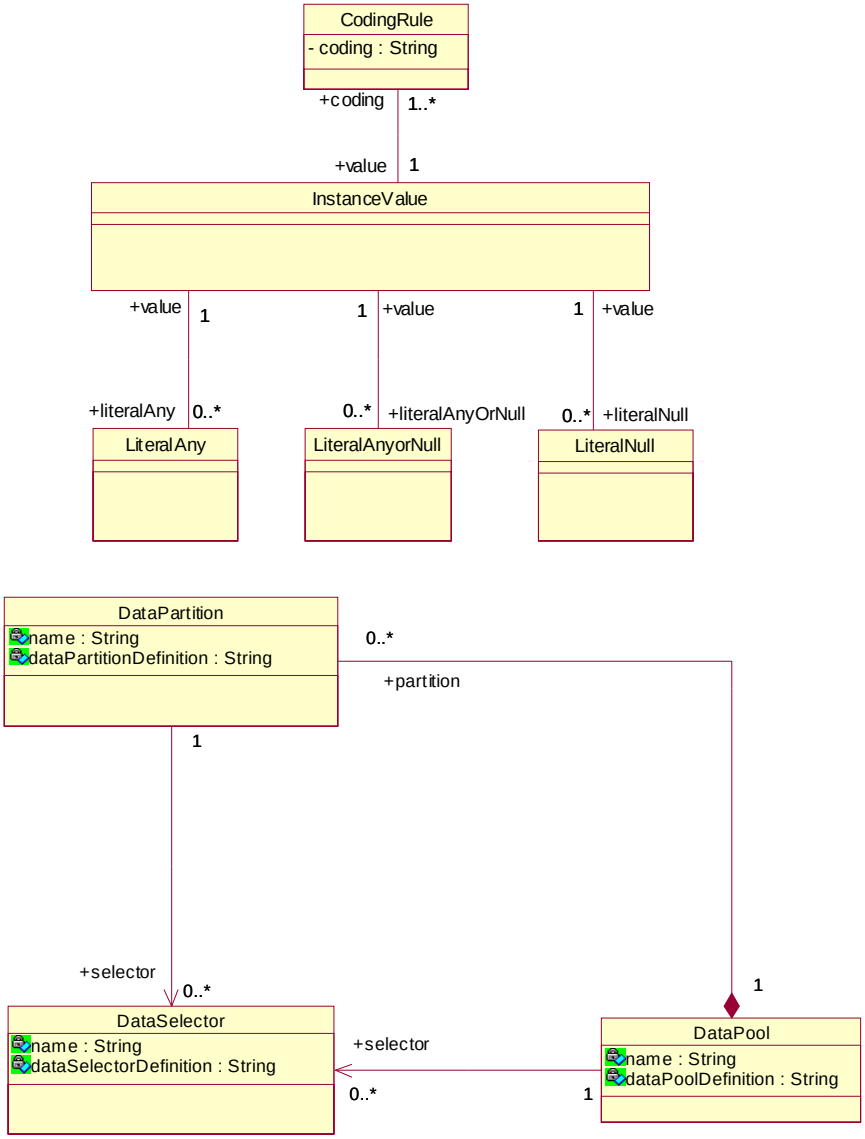


Figure A.11 - Test data portion of the MOF-based metamodel

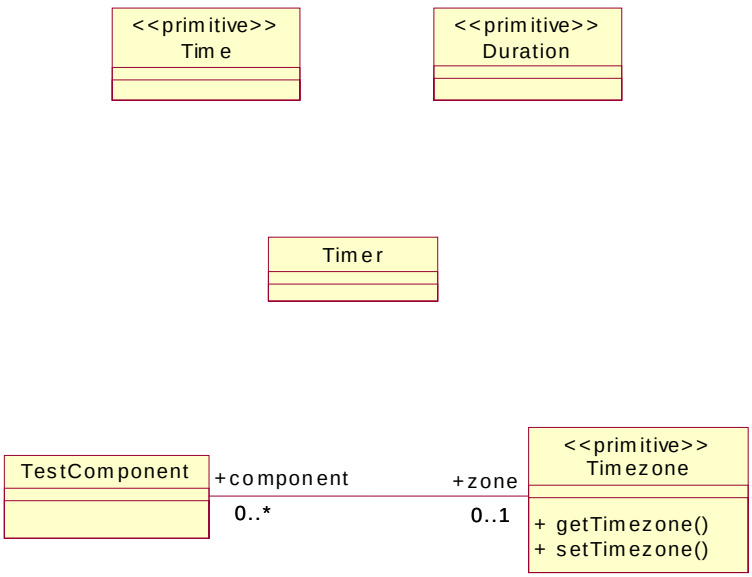


Figure A.12 - Time portion of the MOF-based metamodel

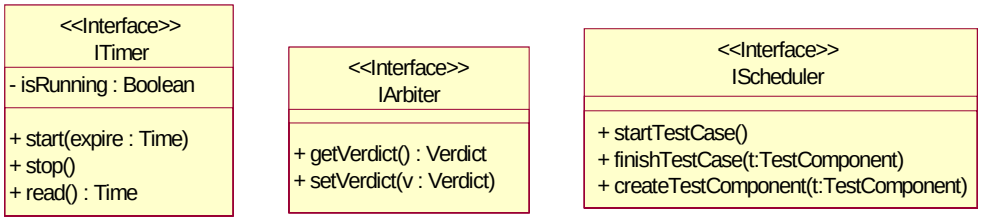


Figure A.13 - Predefined interfaces for implementing timer, arbiter and scheduler

Annex B
(normative)

XMI Schema

B.1 The Profile

The XMI schema definition for the exchange of U2TP profile specifications follows the XMI schema definition of UML 2.0 for UML 2.0 profiles. Please refer to the schema definition of UML 2.0.

B.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/U2TPSA"
    xmlns:u2tp="http://www.omg.org/U2TPSA"
    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="u2tp: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="u2tp: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="u2tp: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="u2tp: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="u2tp: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="u2tp: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="u2tp:Behavior"/>
        <xsd:element name="dataPool" type="u2tp: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="u2tp: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="u2tp:SUT"/>
            <xsd:element name="component" type="u2tp:TestComponent"/>
            <xsd:element name="arbiter" type="u2tp:Arbiter"/>
            <xsd:element name="scheduler" type="u2tp:Scheduler"/>
            <xsd:element name="behavior" type="u2tp:Behavior"/>
            <xsd:element name="testConfiguration" type="u2tp:Deployment"/>
            <xsd:element name="testCase" type="u2tp:TestCase"/>
            <xsd:element name="executions" type="u2tp:TestLog"/>
            <xsd:element name="dataPool" type="u2tp: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="u2tp: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="u2tp: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="u2tp:Verdict"/>
            <xsd:element name="testConfiguration" type="u2tp: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="u2tp:Verdict"/>
        <xsd:attribute name="testConfiguration" type="xsd:string"/>
    </xsd:complexType>
    <xsd:element name="TestLog" type="u2tp: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="u2tp: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="u2tp: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="u2tp:Behavior"/>
        <xsd:element name="executions" type="u2tp:TestLog"/>
        <xsd:element name="testObjective" type="u2tp: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="u2tp: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="u2tp:TestObjective"/>
<xsd:complexType name="CodingRule">
    <xsd:choice maxOccurs="unbounded" minOccurs="0">
        <xsd:element name="coding" type="xsd:string"/>
        <xsd:element name="value" type="u2tp: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="u2tp:CodingRule"/>
<xsd:complexType name="InstanceValue">
```

```
    <xsd:choice maxOccurs="unbounded" minOccurs="0">
        <xsd:element name="literalAny" type="u2tp:LiteralAny"/>
        <xsd:element name="literalAnyOrNull" type="u2tp:LiteralAnyorNull"/>
        <xsd:element name="literalNull" type="u2tp:LiteralNull"/>
        <xsd:element name="coding" type="u2tp: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="u2tp:InstanceValue"/>
<xsd:complexType name="LiteralAny">
    <xsd:choice maxOccurs="unbounded" minOccurs="0">
        <xsd:element name="value" type="u2tp: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="u2tp:LiteralAny"/>
<xsd:complexType name="LiteralAnyorNull">
    <xsd:choice maxOccurs="unbounded" minOccurs="0">
        <xsd:element name="value" type="u2tp: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="u2tp:LiteralAnyorNull"/>
<xsd:complexType name="LiteralNull">
    <xsd:choice maxOccurs="unbounded" minOccurs="0">
        <xsd:element name="value" type="u2tp: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="u2tp: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="u2tp: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="u2tp: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="u2tp: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="u2tp: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="u2tp: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="u2tp: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="u2tp: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="u2tp: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="u2tp:IScheduler"/>
</xsd:schema>
```

Annex C
(normative)

Arbiter and Scheduler Protocols

This annex 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.

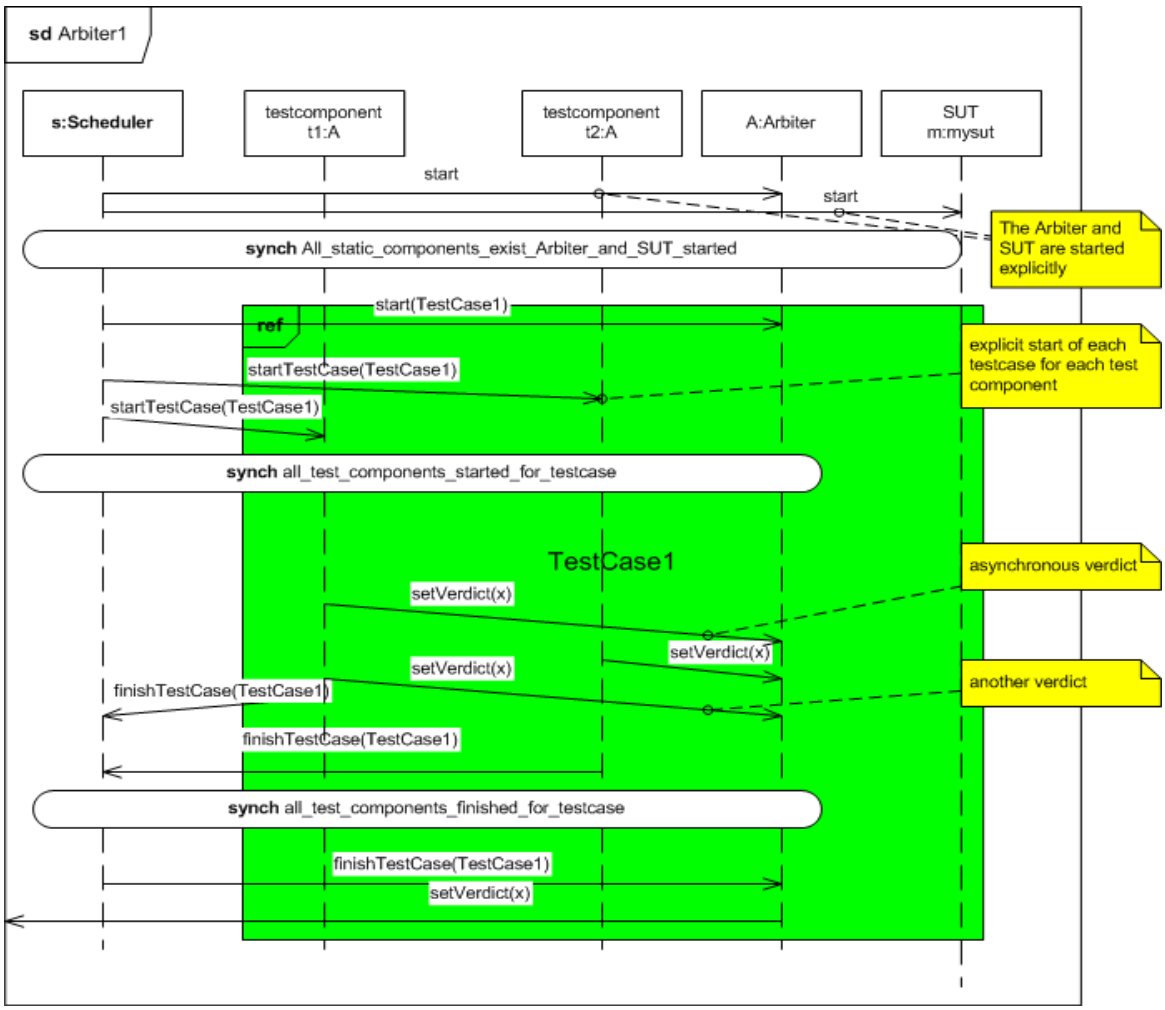


Figure C.1 - Scheduler/Arbiter protocol #1

The protocol scenario in Figure C.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.

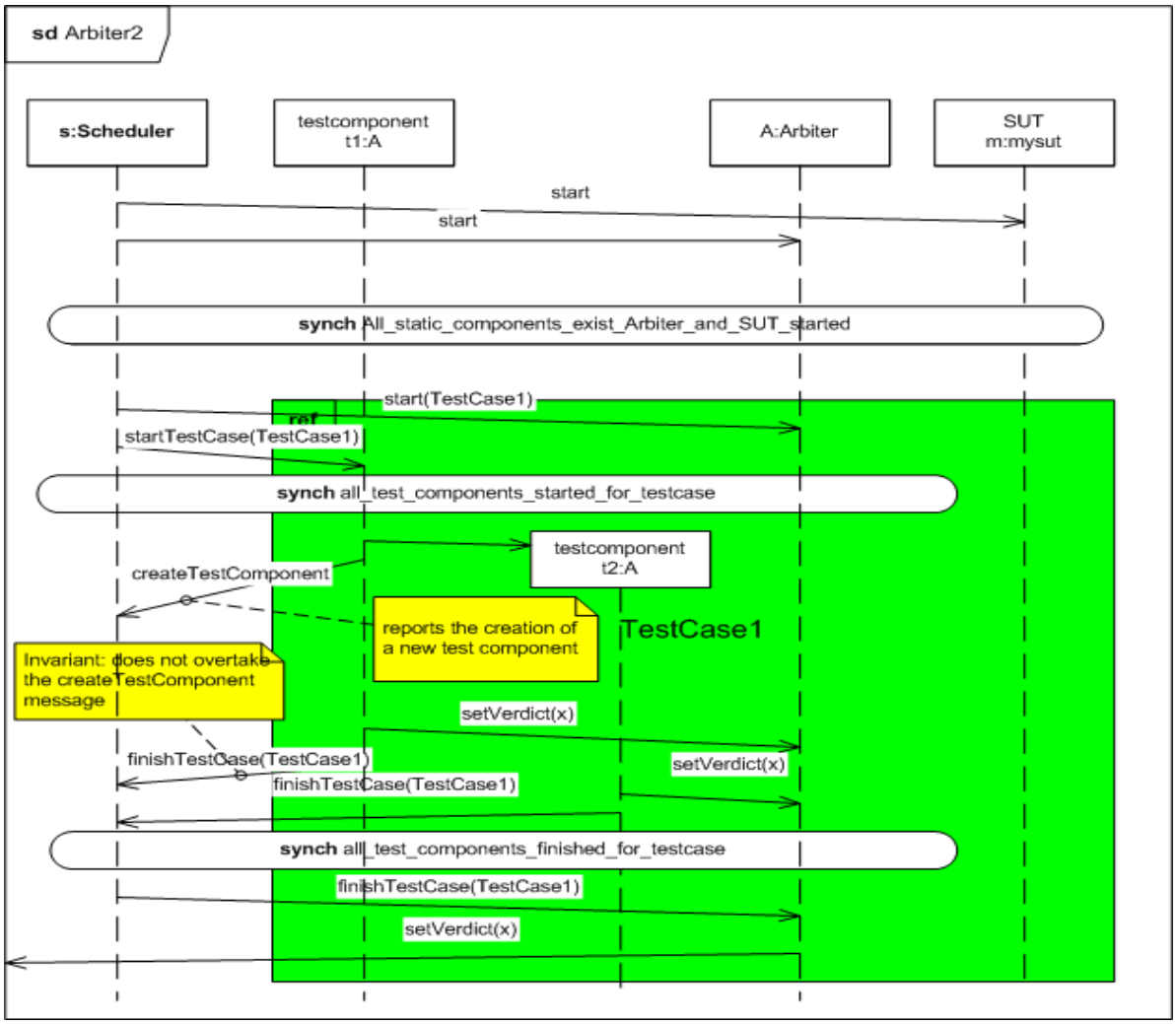


Figure C.2 - Scheduler/Arbiter protocol #2

The scenario in Figure C.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 ondition 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 eisting 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.

General Index

Class Index of the Profile

Class Index of the MOF-based Metamodel