Technical University of Berlin
Faculty of Electrical Engineering and Computer Science
Chair for Design and Testing of Communication Systems

Doctor of Engineering Dissertation

A Methodology For Pattern-Oriented Model-Driven Testing of Reactive Software Systems

by

Alain-Georges Vouffo Feudjio

Supervisors: Prof. Dr. Ing. Ina Schieferdecker (Technical University of Berlin),
Prof. Cesar Viho (IRISA/University of Rennes I, France)

Berlin, June 5, 2010

To Aimé-G. Vouffo-Mbotezo...

Contents

List of Symbols and Abbreviations

Abbreviation	Description	Definition
ABT	Action Based Testing	page 140
ATG	Automated Test Generation	page 8
ATL	Atlas Transformation Language	page 204
ATS	Abstract Test Suite	page 44
BVA	Boundary Value Analysis	page 277
cMOF	Complete MOF	page 38
CORBA	Common Object Request Broker Architecture	page 31
DS(M)L	Domain Specific (Modelling) Language	page 34
DVA	Default Value Analysis	page 277
EFSM	Extended Finite State Machine	page 22
eMOF	Essential MOF	page 38
EP	Equivalence Partitioning	page 277
ETSI	European Telecommunications Standards Organisation	page 47
FDT	Formal Description Techniques	page 20
FSM	Finite State Machine	page 22
GML	Generic Purpose Modelling Language	page 34
HIL	Hardware-In-the-Loop	page 40
ICS	Implementation Conformance Statement	page 266
IMS	IP Multimedia Subsystem	page 269
ISO	International Organization for Standardization	page 15
ISTQB	International Software Testing Qualification Board	page 16
LBS	Location Based Services	page 239
M2T	Model-to-Text (Transformation)	page 204
MBT	Model Based Testing	page 8
MDA	Model Driven Architecture	page 6
MDE	Model Driven Engineering	page 6
MDT	Model Driven Testing	page 8

Abbreviation	Description	Definition
MIL	Model-In-the-Loop	page 40
MLP	Mobile Location Protocol	page 239
MOF	Meta Object Facility	page 38
MPM	Machine Processable Model	page 27
OCL	Object Constraint Language	page 7
OMA	Open Mobile Alliance	page 239
OOA/D	Object Oriented Analysis and Design	page 22
OSI	Open System Interconnection	page 16
PIM	Platform Independent Model	page 23
PIT	Platform Independent Test model	page 23
PSM	Platform Specific Model	page 23
PST	Platform Specific Test model	page 23
QVT	Query/Views/Transformations	page 204
RLP	Roaming Location Protocol	page 239
RTE	Round Trip Engineering	page 7
RVA	Random Value Analysis	page 277
SIL	Software-In-the-Loop	page 40
SIP	Session Initiation Protocol	page 226
SUPL	Secure User Plane Location Protocol	page 239
SUT	System Under Test	page 1
SysML	System Modelling Language	page 37
TTCN-3	Testing and Test Control Notation Version 3	page 53
UML	Unified Modelling Language	page 8
UTML	Unified Test Modelling Language	page 1
UTP	UML Testing Profile	page 8
XSD	XML Schema Definition language	page 278

List of Figures

List of Tables

Abstract

The level of pervasiveness and complexity of software and computing systems has been growing continuously since their introduction. New technologies emerge at regular base, covering ever more aspects of our daily life and leading to shorter product delivery cycles. These ongoing trends are posing new challenges to traditional software testing approaches, because despite those constraints, software products are required to meet a certain level of quality prior to their deployment. Otherwise, confidence in new technologies and products could be harmed, potentially leading to their commercial failure. Therefore, tests have to be developed within harsher time constraints for increasingly complex systems.

Model driven engineering (MDE) has been very helpful in improving the development process of software products by raising the level of abstraction in software design, thus increasing the understandability and facilitating reuse of solutions. Furthermore, with the introduction of patterns, reuse could successfully be extended to concepts for producing new solutions through object-oriented (OO) software design patterns.

One of the benefits expected from MDE was that it would facilitate testing activities. Therefore, various approaches have been proposed for combining MDE and testing in the expectation that similar benefits could be achieved in the test development process, as previously obtained in model-driven software system development. However, most of those approaches have been focusing on automatically generating tests from models of the System Under Test (SUT), while neglecting manual and semi-automated test engineering [1]. This although, a large majority of test suites are not automatically generated, but rather developed with human intervention in what is a highly repetitive and error-prone development process. Furthermore, test engineering is an activity whereby expertise is essential. Therefore, it is assumed that capturing that expertise in the form of patterns, expressed with abstract models to enable its reuse for producing new test solutions will certainly lead to a gain of quality and productivity for the test engineering process.

This thesis describes an attempt to verify that intuitive assumption. Test engineering is addressed from a model-driven development perspective, taking into account test design patterns in the modelling process. For that purpose a methodology for model-driven testing is described that includes a set of recognised patterns in test engineering. The proposed methodology is built on a notation called Unified Test Modelling Language (UTML) that allows to design tests at a high, conceptual level of abstraction as abstract test models that can then be subsequently refined through a series of iterative transformation processes into executable test sequences or scripts. To illustrate and to evaluate the approach, an example test suite project and a case study from the communications application domains are discussed.

[1]The term *manual* test engineering is used here to denote a process, whereby human testing expertise is combined with test automation and model transformation techniques to produce software tests.

Abstract

Software Systeme werden heutzutage in immer mehr Bereichen eingesetzt und betreffen mittlerweile fast jeden Aspekt des täglichen Lebens in modernen Gesellschaften. Um den Anforderungen dieser verschiedenen Einsatzgebiete zu genügen, ist die Komplexität von solchen Systemen in den letzten Jahren rasant gestiegen. Ausserdem werden kontinuierlich neue Technologien entwickelt, um den Verbrauchern bessere Dienste zu günstigeren Preisen anbieten zu können. Dabei steht die Software-Industrie mächtig unter Druck, denn nicht nur sollen unter grossem Zeitdruck komplexere Produkte entwickelt werden, die auf diesen neuen Technologien basieren, sondern sollen diese Produkte ein Mindestmass an Qualität vorweisen, um die Akzeptanz der neuen Technologien nicht zu gefährden. Dies führt dazu, dass neue, bessere Software-Produkte unter erschwerten zeitlichen und finanziellen Bedingungenentwickelt werden müssen. Diese Trends stellen die Testentwicklung vor neuen Herausforderungen, die mit bisherigen Ansätzen nur schwer zu lösen sind.

Trotz der grossen Anstrengungen der letzten Jahre ist die Automatisierung im Testentwicklungsprozess noch nicht auf dem gleichen Niveau angekommen wie bei der generellen Softwareentwicklung, die erheblich durch den Einsatz der modellgetriebenen Softwareentwicklung (Model-Driven Engineering: MDE) und von Entwurfspattern beflügelt wurde.

Unter der Annahme, dass der MDE-Ansatz automatisch den Testentwicklungsprozess beschleunigen und verbessern würde, wurden viele Forschungsarbeiten mit dem Ziel geführt, MDE und Testentwicklung zusammenzuführen, um diese Vorteile praktisch auf die Testautomatisierung zu übertragen. Die Mehrheit dieser Arbeiten haben die automatische Ableitung von Testskripten oder Testsequenzen aus Systemmodellen als Schwerpunkt und berücksichtigen kaum die manuelle bzw. semi-automatische Testentwicklung [2]. Dies obwohl, ein wesentlich höherer Anteil an Testsuites nicht automatisch aus Systemmodellen generiert wird, sondern manuell oder semi-automatisch entwickelt werden. Dieser Prozess kennzeichnet sich jedoch durch ein hohes Maßan Wiederholungen der gleichen Vorgänge, das die Fehleranfälligkeit erhöht und zu höheren Kosten führt, bei nicht gewährleisteter Qualität. Es wird angenommen, dass der systematische Einsatz des MDE-Ansatz in diesem Bereich für den ganzen Softwareentwicklungsprozess vorteilhaft wäre und dazu beitragen könnte, diese Probleme zu lösen.

Diese Arbeit schlägt einen musterbasierten Ansatz zur modelgetriebenen Testentwicklung für reaktive Softwaresysteme vor, mit dem Ziel, die Testentwicklung für solche Systeme zu vereinfachen und zu beschleunigen ähnlich, wie das schon der Fall war mit Produktsoftware. Dabei beschreibt die Arbeit eine Methodologie für einen solchen Ansatz und untersucht anhand von konkreten Beispielen den erreichten Gewinn an Effizienz.

[2]Die Terminologie *manuelle* Testentwicklung bezeichnet einen Prozess, wobei menschliche Testexpertise mit Testautomatisierungsanssätzen kombiniert werden, um Softwaretests zu produzieren

Acknowledgements

This work would not have been possible without the continuous and decisive support of numerous people whom I would like to thank wholeheartedly[3].

- Prof. Dr. Schieferdecker for keeping the faith and pushing me with new ideas to complete this work.
- Prof. Viho for the support and inspiration.
- My parents, Papa Vouffo Prosper and Maman Colette for their patience, their love and care and for teaching me the most important values in life.
- My wife Sandy and my daughters for their patience.
- The whole Vouffo family for their continuously provided encouragement.
- My colleagues at Fraunhofer FOKUS and beyond for their helpful comments and challenging ideas.

[3]The list below is by no means exhaustive and I apologize for any inadvertent omission.

Chapter 1

Introduction

1.1 Introduction

The importance of testing as a mean for evaluating quality factors of software and to reveal errors in software products before they are deployed or commercialised has been growing continuously in recent years. It is currently estimated that 30 to 60 percents of the overall resource consumption in software development is done on testing [158]. This has underlined the need for approaches to keep test development costs under control by ensuring the efficiency of the efforts being invested. Those approaches aim at introducing a high level of automation and reuse in each phase of testing where applicable. Thus, the term *Test Automation* has been used to denote them.

Now, test automation can be understood in many different ways, depending on the intended goal. Automation of test execution has been the subject of a large amount of research in recent years, leading to the emergence of a plethora of notations, tools and frameworks to support automatic execution of test scripts, including features such as automatic scenario capturing and replay, automated evaluation of verdicts, tracing and reporting of test results, etc. Those might also include facilities for managing test suites, controlling distributed test infrastructures and beyond.

Another field of testing on which automation has been applied successfully, is that of test generation. Test generation is the process aiming at allowing tests to be automatically generated from system models or any other kind of formal representation of the System Under Test's behaviour or structure.

While those test automation approaches have significantly improved the testing process, a lot remains to be done to address the challenges of testing software systems that are becoming increasingly sophisticated and heterogeneous. Nowadays tests have to be developed within shorter time and using less resources for systems that present a much higher level of complexity. Testing has evolved into a full development discipline of its own, with a dedicated process running in parallel to software system development. As stated by Utting and Legeard in [158]:

> Writing tests is now a programming task, which requires different skills from test design or test execution.

Interestingly, the evolution of testing is quite similar to that of product software development in recent years. To address the challenges of ever shorter time to deliver software products of higher complexity mentioned above, the level of reuse and maintainability of test artifacts must be improved significantly. For example, Conrad [34] states that:

> The test notations which are often used when developing automotive control software, such as the direct description of the test scenarios in the form of time-dependent value courses or the use of test scripts, lead to a description of test scenarios on a very low level of abstraction, making maintainability and reuse difficult.

Although the above statement explicitly refers to the automotive application domain, it holds true for almost any domain in which software testing is performed.

Model-driven software engineering (MDE) approaches, e.g. the Model Driven Architecture (MDA)[1] proposed by the Object Management Group (OMG), were introduced to address exactly that kind of challenges for software system development. MDA is defined by the Object Management Group as:

> a way to organise and manage enterprise architectures supported by automated tools and services for both defining the models and facilitating transformations between different model types. [118]

The MDA approach of software system development which consists of transformations from a platform-indepent model (PIM) through platform-specific models (PSM)into lower-level source code.

Compared to "traditional" software development techniques, MDE has a lot of benefits including the following [96, 112]:

- Improved understandability, maintainability and reuse through higher abstraction and visualisation of concepts.

- High level of automation, leading to more consistent source code obtained through automated model transformations.

[1]MDA is a trademark of the Object Management Group (OMG)

- Round trip engineering (RTE): RTE is the ability to move from a system's highest level of abstraction into its lowest (i.e. implementing source code) and backward through model transformation. That means conceptual changes at the Platform Independent Model (PIM) level can automatically be propagated into the system's implementation, thus facilitating fixes on the product as well as development of new products or product lines.

- Early identification of design flaws through automated model validation e.g. based on constraints defined using formal notations e.g. the OMG's Object Constraint Language (OCL).

- Improved communication between stakeholders involved in the development process, leading to higher productivity for the whole business process.

Thanks to those benefits, the productivity gain resulting from the introduction of model-driven development is estimated somewhere between 25% [111], 35% [33], 69% [110] and even up to 500% [24].

Patterns are well-documented abstractions of solutions to recurrent problems that can be reused to resolve similar problems in any new context in which they might occur. Back in 1979, an approach for capturing patterns in a systematical manner was introduced by Alexander [3] to catalogue sound solutions in designing the architecture of buildings and cities. The adoption of that approach for Object-Oriented (OO) software development introduced by the Gang of Four [62] led to so-called software design patterns aimed at documenting proved solutions to recurrent problems in that field and to speed up the design and implementation of such solutions, through automated model or source code generation.

Patterns are a way of abstracting from the complexity of systems by focusing on the main aspects of the solution they provide. Because they address problems by defining concepts at a high level of abstraction (i.e. at a meta-level), integrating patterns in the MDE process has always appeared as a tempting idea, potentially improving the software development process both quantitatively and qualitatively. The aim is to allow new software engineering solutions to be designed, based on patterns and expressed in a formal modelling notation, so that they could be transformed automatically into complete source code or customisable stubs and skeletons. Examples of mechanisms for achieving that goal have been proposed in the existing literature by authors, such as Sunyé et al [153], Blazy et al [17], France et al [59], along with numerous others. Some of those approaches have even been successfully implemented in existing software design tools and frameworks available on the market and have contributed in maximising the benefits yielded with MDE.

With the hope of achieving similar kind of benefits for the test development process and following one of the trends predicted by Buschman et al [27], patterns in testing have been gaining more popularity as a research field. However, one of the difficulties faced with in that context is the fact that despite the large amount of works and approaches combining modelling and testing activities, few of those have managed to become popular among testers and developers alike. Those approaches can be classified under two main categories under terms such as model-based testing (MBT) and model-driven testing (MDT).

Model-based testing is defined in many different ways in the existing literature, but certainly, the most popular definition of MBT is that of an approach whereby test sequences are generated automatically from models of the system under test, using different kinds of computing algorithms to optimise that process. Therefore, model-based Automated Test Generation (ATG) is the key feature of MBT. In this thesis, whenever the term MBT will be used, that definition will apply. MBT is used in different flavors by several tools and projects. For example, the AGEDIS tool [77], the TOTEM method [21], the MODEST method [143] and numerous others [4, 15] use system models expressed in the Unified Modelling Language (UML) to automatically generate test sequences. A more detailed list of applications of that approach using various notations is presented by Utting et al in their *Taxonomy of Model-Based Testing* [157].

However, despite the huge progress in model-based automated test generation, a large amount of test cases are still developed manually or semi-automatically. That process is very repetitive, technically challenging and highly error-prone. Moreover, just as software systems have continuously been growing in complexity, so have the tests aiming at validating those systems with regard to their requirements also become highly complex pieces of software. This has underlined the need for approaches to optimise that process by integrating all phases of the test development process and by facilitating reuse of test artifacts. One such approach - labelled Model-Driven Testing (MDT) - consists in following the same model-driven engineering (MDE) approach that is already widely applied for generic software system development, in test development as well. Rather than attempting to generate tests automatically, the main feature of that approach is to address the growing complexity of test suites by raising the level of abstraction in the design phase and by supporting manual or semi-automatic test development with automatic model transformations. The UML Testing Profile (UTP) [70] is one such attempt to introduce MDE into the test development process. Figure 1.1 illustrates how the classical V-model of software development is transformed with the MDT process. As illustrated in that figure, MDT introduces a parallel thread dedicated to test activities into the classical MDA process, through which test development is performed as sequence of model transformations from a platform-independent test model (PIT) through platform-specific test models (PST) into executable test code.

The MDT is another illustration of the evolution of test automation into a full discipline of its own, confronted with the same type of issues already identified -

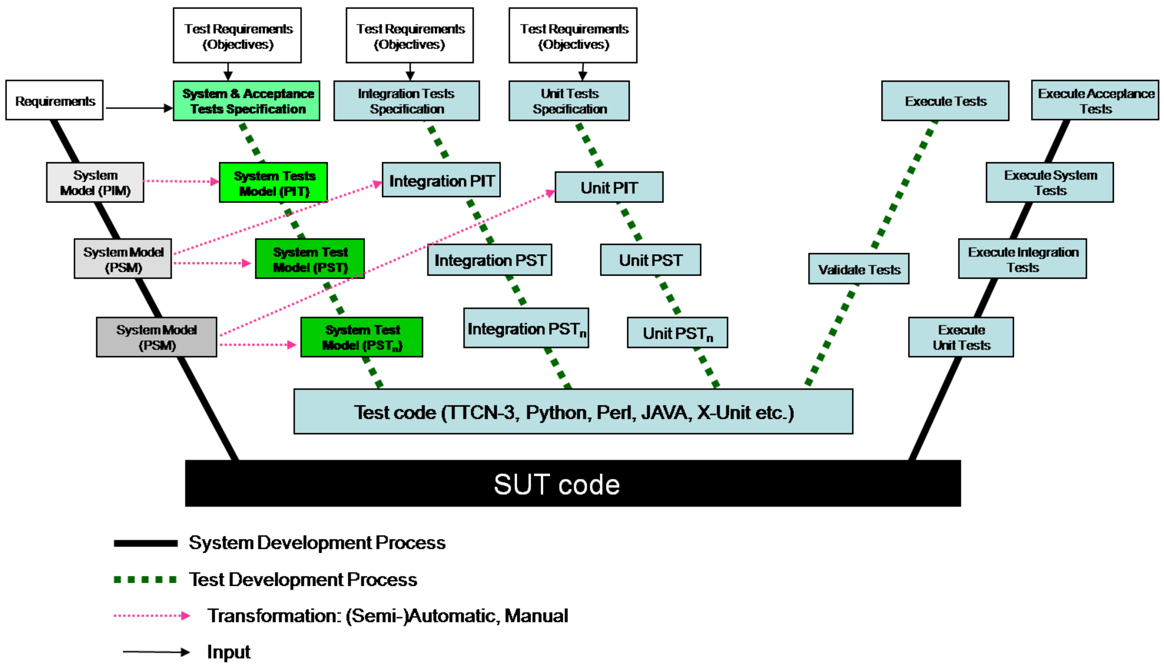


Figure 1.1: The model-driven test development process in the classical V-model

and possibly solved - for generic software development. In fact, as several authors pointed out [93, 7], test automation is indeed software development and requires the same level of discipline and methodology to be successful. Therefore, in the same manner as patterns in software engineering were catalogued and applied successfully to improve the development process, the concept of test design patterns has emerged and is gaining more popularity [159, 116, 109, 163]. Patterns in developing test automation aim at capturing knowledge gathered in those activities and at achieving more optimisation, to face the growing challenges of testing increasingly complex reactive software systems.

This thesis is based on the assumption that the identification and the exploitation of those patterns would be beneficial, not just for test development, but for the software development process as a whole. Given the fact that such an exploitation of test patterns would have a greater impact, if it tackles the issue from a high level of abstraction, a review of existing model-driven test development approaches was viewed as a necessary preliminary work to assess how that vision could be transformed into reality. Therefore, the thesis introduces concepts for a pattern-oriented model-driven testing approach, which enables test systems to be developed following an MDE process and along previously identified patterns in testing. Beyond the fact that it covers all phases of test development, the specificity of the approach lies in the fact that the abstract platform independent test models (PITs) are designed, taking into account a set of rules and templates based on identified test design patterns.

1.2 Scope and Purpose of this Thesis

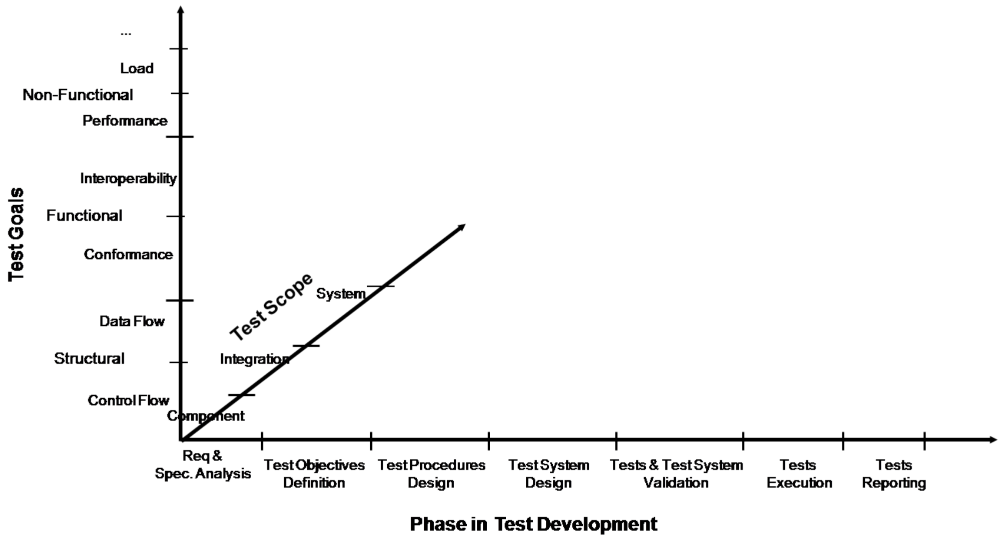


Figure 1.2: Classification of test approaches

Figure 1.2 depicts a usual method for classifying test approaches as a 3-dimensional plot, with each of the axis representing an aspect of testing used as classification criterion. As depicted in that figure, test approaches can be classified along the following criteria:

- Test goal: The test goal criterion distinguishes between possible intents of the testing activities. This leads to a division in three main categories:
  - Structural testing aims at verifying a SUT using knowledge of the internal structure of its source code. Because of that heavy reliance on the SUT's source code, structural testing is also referred to as white-box testing. Techniques for structural testing include control flow testing and data flow testing. In control flow testing, the tester attempts to exercise as many of the execution paths of the source code as possible and verifies that they produce the expected output. Given that the cost of testing rises with the number of executed paths, the approach for selecting a relevant subset of execution paths is critical for this category. Data-flow testing is a control-flow testing technique which besides examining the flow of control between the various components, also examines the lifecycle of data variables to select test cases [8, 76].

  - Functional testing aims at verifying that the SUT's behaviour meets its specified requirements. Functional test can be performed on a single entity to verify that its behaviour is compliant to a given standard or specification (conformance testing), or it can be performed by combining SUTs from different vendors to verify that they can work smoothly with each other, based on the same specification (interoperability testing).
  - Non-functional testing deals with quality aspects of SUTs that go beyond basic functionality, e.g. performance, stress-resistance, load-handling, robustness, etc.

- Test scope: The test scope denotes the SUT's level of granularity for which a test approach is applied. The finest level of granularity in object-oriented software systems is a class or its associated instantiating object. This finest level of granularity is also referred to as a "unit". Hence the term "unit testing" to denote that type of testing. Software modules (sub-systems) and whole software systems are other levels of granularity at which testing can also be applied. In which case, terms such as "integration testing" and "system testing" are used.

- Test phase: The test phase criterion refers to phases of the test development process in which a given approach is applicable. On this axis, the test development process is depicted as a process that starts with an analysis of the SUT's requirements with regard to testability, through various iterative steps to test reports describing the test results and thus reflecting the quality of the SUT.

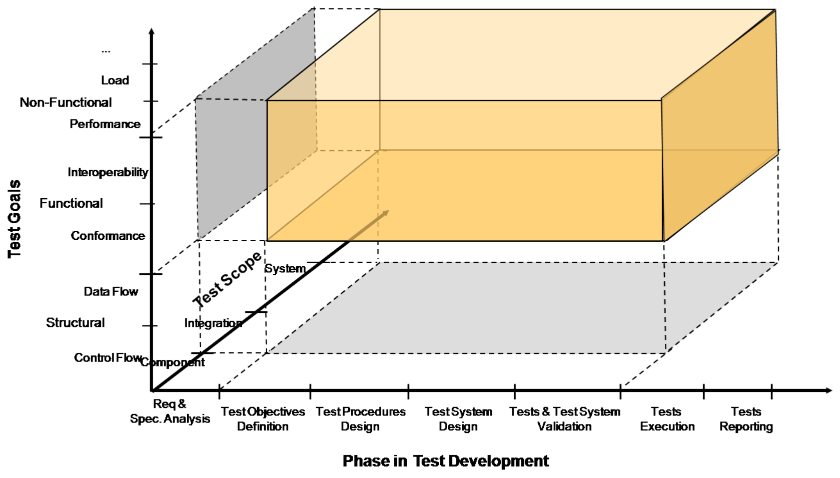


Figure 1.3: Scope of this work

Figure 1.3 illustrates the scope of this thesis, which can be described as follows:

- Test goal: This thesis discusses essentially conformance and interoperability (integration), i.e. functional testing of software systems. However, some of the findings might be applicable to non-functional testing (e.g. performance, load testing). Structural testing is out of scope, as it is best addressed with white box testing techniques (e.g. control flow or data flow analysis).

- Test scope: This thesis covers testing at the component (module, subsystem) and at the system level of granularity. An application to class-level unit-testing, though possible, appears to be less appropriate, because, existing testing approaches at the coding level are more effective for that purpose and the incentive for raising the abstraction level is not present.

- Test phase: This thesis covers the whole test engineering process, once the requirements on the SUT and the associated system specification have been analysed from the testing perspective. However, test execution and test reporting are covered to a lesser extend than the other phases of test engineering, as those areas have already been the object of numerous works to improve efficiency through automated test execution and result analysis. Therefore, the approach used in this thesis will consist in taking advantage of existing test execution and reporting platforms rather than proposing yet another new architecture for that purpose.

The purpose of this work is to propose a methodology for pattern-oriented model-driven testing, covering the whole test development process and to assess its potential impact on that process in particular and on the software engineering process as a whole.

1.3 Structure of this Thesis

The rest of this thesis is structured as follows:

- Chapter 2 provides the software testing background that serves as a basis for the remaining chapters, introducing the terminology used and describing how it is understood in the context of this work.

- Chapter 3 presents the current state of the art by describing existing work on model-driven testing, which is a pre-requisite for the approach proposed in this work.

- Chapter 4 provides an overview of the pattern-oriented model-driven testing approach and the principles it is based upon.

- Chapter 5 describes the UTML notation both in terms of syntax and semantics, through its meta-model which embodies the concepts of pattern-oriented test modelling.

- Chapter 6 describes the design and implementation of a prototype tool chain that will be used to evaluate the approach. That evaluation was achieved through an example usage of the approach to design a solution for test automation of a small web application, followed by a qualitative and quantitative evaluation through the application to a case study conducted with the prototype UTML tool chain.

- Chapter 7 summarises the main results of the thesis, then concludes the work and draws an outlook for further research in the field.

Chapter 2

Basics

2.1 Introduction

A reader trying to find out the difference between model-driven, model-based testing and any other combination containing the terms “model” and “testing” may get quite confused from the large amount of literature dealing with those two topics individually or in combination. A similar picture emerges if the term “patterns” is considered. In this section some of the key testing and modelling-related terms commonly used in this thesis will be introduced, including an explanation of how each of those terms is understood in this context. This chapter is organised as follows: The next section (2.2) introduces some basic principles of software testing, focusing on the terminology used in that context in general and in this thesis in particular. Then, section 2.3 discusses various approaches of combining testing and modelling, each time describing the potential benefits and pitfalls of the approach. Finally, section 2.4 introduces the background knowledge relative to patterns and their usage in software engineering as well as in testing, before section 2.5 summarises the chapter.

2.2 Principles of Software Testing

2.2.1 Terminology

To avoid misinterpretations and misunderstandings a clear and precise terminology is essential for any domain. This is particularly important for an activity like testing that plays a central role in the software development process. Therefore, testing terminology has been the object of many efforts from standardisation organisations and other groups. The International Organisation for Standardization’s (ISO) Conformance Testing Methodology Framework (CTMF) standard, published as ISO/IEC 9646, is just one example of such an effort. ISO's CTMF defines a framework for conformance testing of communication protocols based on the Open Systems Interconnection (OSI). Although the testing concepts defined in ISO/IEC 9646 originally had OSI communication protocols in mind, they have been adopted for conformance testing in other application domains. This is illustrated by the fact that the TTCN-3 [58] notation has adopted those concepts, although its scope now extends widely beyond testing of communication protocols.

Similar efforts from other institutions include the Institute of Electrical and Electronics Engineers (IEEE)'s IEEE-829 standard [83] and the International Software Testing Qualification Board's (ISTQB) glossary [86, 67], which both define a series of terms related to software testing. Many of the terms used in this thesis are understood consistently along the definitions provided in those standard documents. However, some of them needed to be redefined to align with the proposed approach and its underlying concepts.

The next sections enumerate those terms and describe how they are understood in this thesis.

Component

A component is an abstract entity that is part of the architectural context in which a test case can be executed. A component can be a representation of a part of the SUT - in which case it is called a *system component* - or a representation of an entity required to stimulate the SUT or to observe its behaviour to assess its correctness. In that case the component is called a *test component*. It is worth noting that a test component is understood all through this thesis as an abstract concept, which does not necessarily map to a piece of software running on a computing system. Rather, a test component can be mapped to any element of the testing environment which can cause an impulse on the SUT or through which the SUT's behaviour could be observed. For example, in the case of a coffee machine as SUT, a test component could be the representation of a person who interacts with the machine (SUT) through a set of buttons (input ports) and can observe the responses to her impulses.

(Abstract) Test Case

A test case is a complete and executable[1] specification of the set of actions required to achieve a specific test objective or a set thereof. A test case is considered to be abstract, if it cannot be executed automatically on a computing system, either because it has been specified using a language that does not allow such

[1]Please note that the term *executable* here does not necessarily mean automatically or entirely programmatically executable, because test execution may still include some steps to be performed manually by a physical person.

automated processing via a test execution platform (e.g. natural prose language) or because the notation used is an intermediary one that requires further transformations to obtain automatically executable test scripts.

(Abstract) Test Suite (ATS)

A complete set of (abstract) test cases, possibly combined into nested test groups that is needed to perform testing for one SUT or a family thereof implementing the same specification.

Conformance testing

Conformance testing aims at verifying the extent to which an SUT reflects its base specification. The base specification may be a document published by a standardisation body, a collection of requirements on the system, a prose description of the system or any document of that kind. *Requirements-based testing*, *acceptance testing*, *customer testing* [64] *specification-based testing* [134] are other terms used for conformance testing.

Executable Test case

A concrete realization of an abstract test case that can be executed to test an SUT. An executable test case is generally either a test script written in a notation that can be transformed directly into binary code for execution on a given computing platform or a series of clear and precise instructions to be followed by a person (test operator) to assess if the SUT's behaviour matches its specification. Another possibility consists of a combination of both manual and automated test execution into a form of semi-automated test execution.

Test Data

Test data is any form of data that can be used to stimulate a system under test or that can be observed as output from it. In this thesis the term *test data* defines an abstract concept, which can be mapped to anything that represents an input or an output on a given SUT. Examples of test data could be a communication protocol message, a method call on an Application Programming Interface (API), a physical control button on a machine that may be pushed to create a stimulus, a pop-up window on a graphical user interface, etc.

Test Action

A test action denotes any action that must be undertaken to execute a test case. An example of test action is the sending of test data to another component from a source test component, either to stimulate it (SUT component) or to achieve some other test-related purpose (e.g. reaching a certain pre- or post condition).

Test Event

An indivisible unit of test behaviour that is observable at the SUT's interfaces and can be evaluated to verify that the SUT's behaviour is correct, e.g. when it reacts to a given impulse.

Test Group

A named set of related test cases in a test suite. More generally, a group is merely a way of organising items in a test specification.

Test step

A named subdivision of a test procedure, constructed from test events and/or other test steps.

Test Architecture

A test architecture is a composition of test component(s) and SUT component(s) that are interconnected via ports through which they can exchange data (messages, signals, function calls, etc.) to execute a test case. A static test architecture is a predefined test architecture that can be reused for more than one test case. It is assumed that the interconnection of components within a static test architecture does not change during the whole test execution. In a way, a test architecture describes the topological context in which a test behaviour will occur. Dynamic test architectures are those that may be modified while the test case is still running. Example of such modifications include the instanciation new test components, the termination of existing one or the addition/suppression of connections between components. While such situations are rather scarce in conformance testing, they may be quite attractive for other kinds of testing, e.g. load and performance testing.

Test Architecture Model

A test architecture model is a model containing elements of architecture required for testing a particular SUT. Beside a collection of predefined static test architectures, the model includes type definitions required for designing static or dynamic test architectures, depending on the addressed test scenario.

Test System

A test system is the collection of test components within a test architecture ,i.e. excluding all SUT components.

Test Objective

A test objective is a prose description of a well defined goal of testing, focusing on a single requirement or a set of related requirements as specified in the associated SUT's specification (e.g. Verify that the SUT's operation *anOperation* supports a value of -*Xmax* for its parameter *p_IntParam*). It should be noted that the test objective merely specifies *what* needs to be tested, without any indication as to *how* that objective will be achieved.

Test Design Specification

A document specifying the test conditions (coverage items) for a test item, the detailed test approach and identifying the associated high level test cases [83, 86].

Test Case Specification

A document specifying a set of test cases (objective, inputs, test actions, expected results, and execution preconditions) for a test item [83, 86].

Test Specification

A test specification is defined as a document that consists of a test design specification (see 2.2.1), test case specification (see 2.2.1) and/or test procedure specification [86]. A test specification can be viewed as the equivalent to a software or system specification for generic software engineering.

Test Procedure

A test procedure - also labelled *test procedure specification* [86] - is defined as a prose description of a sequence of actions and events to follow for the execution of a test case. A test procedure describes how a test objective will be assessed. For example, the test procedure for the test objective mentioned above ("Verify that the SUT's operation *anOperation* supports a value of -*Xmax* for its parameter *p_IntParam*") might read as follows:

- Step 1: (Preamble) Initialise SUT
- Step 2: Instantiate a variable v for *p_IntParam* of the same type as *p_IntParam*
- Step 3: Assign -*Xmax* to variable v

- Step 4: Use variable v as a parameter to call the SUT's *anOperation* operation.
- Step 5: Check that the SUT returns normally to the call
- Step 6: (Post-amble) Cleanup test set (free memory, destroy objects etc.)

it should be noted that there is a 1:n relationship between a test procedure and the test objectives it addresses. I.e. a test procedure may cover 1 or many test objectives.

2.3 Testing and Models

The introduction of formal description techniques (FDTs) to specify software intensive systems created new perspectives for more efficient testing approaches of such systems. The hope was that the standardisation of FDTs (e.g. SDL, Estelle, LOTOS) and their usage for specifying software systems would provide a better basis for automated test derivation than with natural language specifications. With the emergence of semi-formal description techniques through model-driven engineering supported with notations such as UML, SysML etc. and their growing popularity, that hope has remained quite strong. This is illustrated by the large amount of research activities on automated test derivation based on such formal or semi-formal system specifications or models.

This section presents an overview of existing approaches in that area and describes how they relate to this thesis. Because of its popularity and the various different contexts in which it has been used, the term *model* might be one of the most difficult to define in computer science. This section will address the various aspects of the relationship between testing and modelling. It is organised as follows: The next section will review the terminology around the concept of models as it is used in this thesis. In particular, the question *what is a model?* will be the main point of interest for that section. Then, the next sections (2.3.2 and 2.3.3) discuss the most frequent associations of models and testing, namely model-based testing, model-driven testing and high level test design.

2.3.1 What is a model?

Because models have always been used in a wide variety of human activities, defining exactly what a model is, always appears like a sheer impossible task, inevitably leading to a controversial result. Therefore, instead of trying to provide a generic definition of that term, like a dictionary would do it, a more domain-specific definition appears to be a more realistic attempt. This thesis is concerned with models in software engineering. But even considering that area of computer science alone, the number of existing definitions of the term remains quite important. Nevertheless, a key characteristic of models (as they are understood in this thesis) is the fact that they are based on the idea of abstraction [104]. A model can be viewed as a description of the structural and behavioural design of a piece of software. It is similar to the set of plans used by engineers to build a house, with the difference that instead of a house, the result will be a piece of software. It is important to note the difference made here between a (miniature) representation of the object itself, for example to illustrate its usage and a representation of its design. In fact, the term *model* is also used in software engineering to denote simulations of various kinds of processes on infrastructures that would be otherwise too costly to build and to test in real-life (e.g. embedded software, telecommunication networks, large or high-value mechanical system etc.)

In the context of this thesis, a model is understood from a Model-Driven Engineering (MDE) perspective, i.e. as a partial and abstract, but yet exact representation of a system's design, out of which more concrete representations of that system can be derived (automatically/manually) following an iterative process. Such models are generally expressed graphically in the form of diagrams. A key characteristic of a model is that each of the associated diagrams allows the object to be analysed from a different view point, each time revealing (i.e. displaying and allowing access to) a particular aspect of the object or a combination of several aspects. The classification of possible views could be driven by the type of information (e.g. architectural view, data view and the behavioural view), the level of abstraction (e.g. logical view, technical view, physical view), or any other criteria of the data made accessible through those views.

However, in some parts of this thesis definition 4 above will be used, especially to clarify other concepts of model-related testing. But whenever that will be the case, it will be clearly indicated as such to avoid any possible confusion.

In the rest of this thesis a distinction will also be made between a *system model* and a *test model*. The term *system model* will be used to denote a model (according to the definition above) of the SUT. Whereas the term *test model* will be used for a model of the elements required for testing the SUT.

Other definitions of a model (e.g. as mathematical representations of physical processes) and the associated testing activities in those areas are considered out-of-scope for this thesis and will not be discussed further.

For a more detailed discussion on the definition of a model, please refer to Kühne [98], Utting et al [158] and Binder [16].

2.3.2 Model-Based Testing

With the growing popularity of models and MDE in software development, model-based testing, which was already successfully applied in hardware testing has become one of the main topics of research in software engineering in the last decades. One can identify two main flavours of model-based testing in the existing literature:

- Model-Based Testing as a generic term for any testing activity in which models are used in one way or another: Jorgensen [91], El-far et al [48] define MBT as

  > a general term that signifies an approach that bases common testing tasks such as test case generation and test result evaluation on a model of the application under test

- Model-Based Testing as Automated Test Generation (ATG) based on models of the SUT (also called *system models*): This flavour is the most common in the literature and in the tool landscape [157, 158], especially with the growing popularity of MDE and Object-Oriented Analysis and Design (OOA/D). It consists in using a model of the SUT - potentially enriched with some additional test-relevant information - to automatically generate tests. Those additional information are sometimes called *(test) requirements* or *constraints*, *annotations* etc. This form of Model-Based Testing relies heavily on algorithms aiming at achieving a possibly high level of coverage relative to the base model(e.g. coverage of all possible transitions, if the SUT's behaviour has been modelled as a labelled transition system (LTS) or as a Finite State Machine(FSM)). The generated tests range from structured plain-language descriptions of the test sequences (i.e. the set of actions to be performed to achieve the test goals), to directly executable test scripts, in the form of binary code or expressed in an intermediary notation, out of which automatically executable tests can be derived through compilation or interpretation.

- Model-Based Testing as the application of MDE to test development: This usage of the term MBT is rather seldom and appears mostly in works related to the UTP [122].

Models in MBT

Most of the models used in model-based automated test generation are representations of the SUT's usage from a black-box perspective and therefore correspond to definition 4 of the term *model* provided in section 2.3.1. Finite state machines (FSM) and their various extensions such as extended finite state machines (EFSM), state charts and markov chains are undoubtedly the most popular types of models used in MBT. The usage of FSMs in computer science can be dated back to the 1950s with the work of Mealy [108], Moore [113], Kleene [95] et al. FSMs are based on the principle that every software is always in a specific state and that the output it generates from any input will be determined by the state it currently finds itself in. Furthermore, FSMs are based on the assumption that the number of possible states for the SUT is finite. This is particularly the case for software running on computer hardware components, which additionally benefit from the fact that the number of states there is usually rather small.

While first experiences of using FSMs to test software date back to 1978 with the work of Chow [31], the large number of works on that field [61, 6, 32, 103][2] in the last decades clearly indicate the growing interest in industry and academia for it.

2.3.3 Model Driven Testing

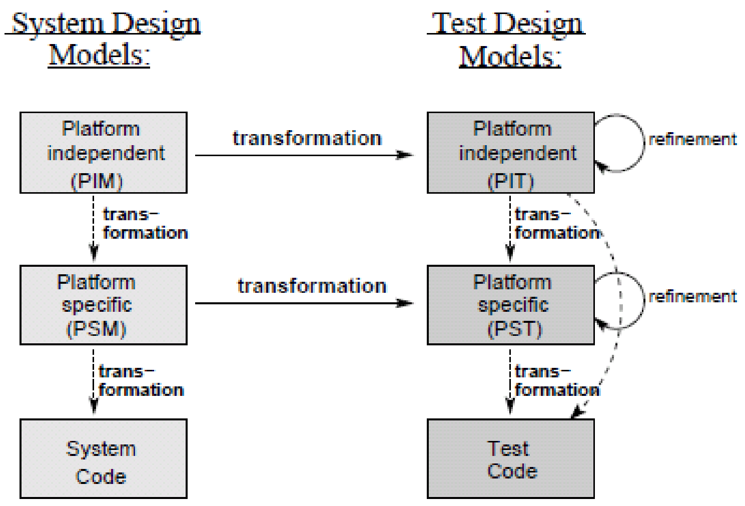


Figure 2.1: Model-Driven Testing Process

Model Driven Testing (MDT) is an approach of software testing whereby a model is used to model an abstract representation, not just of the SUT's structural design, but also of the SUT's testing context. In this thesis, MDT is defined as the application of MDE methodology, e.g. as proposed by the OMG's Model Driven Architecture (MDA), to the testing domain. As depicted in Figure 2.1 (from [162]), while the MDA features the (automated) transformation from platform independent system model (PIM) into source code via successive platform specific models (PSM), MDT introduces a parallel thread for the test development process. In that process, a platform independent test design model (PIT) is transformed automatically into platform specific test models (PST) and eventually into test script code that can be executed to assess the SUT. Optionally, an automated transformation of the system model into the test model could be achieved as well, both at platform independent level and at platform specific level.

[2]For a more exhaustive list, please refer to [48]

Rather than the automated generation of test cases, the main goal of MDT is to automate the manual test development process and facilitate its integration in the overall software development process to ensure requirements traceability and

Models in MDT

The models used and designed in MDT are mainly of two types:

- A system model, which is either a design model of the SUT (e.g. a UML model of the SUT) or a model describing the usage of the SUT from a black-box perspective (usage model).
- A test model, i.e. design of the testing context in which the SUT will be tested (e.g. a UTP model describing the testing context and potentially including the SUT's model as a whole or referring to it).

The testing context consists of three main elements, namely test architectures, test data and test behaviour. The technique used for modelling those elements will depend on the tooling environment and especially on the technique used for modelling the SUT. The tendency is to use the same technique for both elements to facilitate data exchange between them and a better understanding among stakeholders.

However, it is quite common in software development that no design of the SUT is available for reuse. In those cases, the test model directly incorporates the SUT's design elements required to design sensible tests, while ignoring aspects of the SUT that are irrelevant for the testing activities.

2.3.4 High Level Test Design

Test modelling is the activity of designing a test model and as such, it is performed in both MBT and MDT processes. However, there is a significant difference between the test design activities in those two processes. In MBT the focus will be laid on providing a compact model of the SUT's behaviour and in particular of how it should behave when it is being used, while at the same time neglecting the impact of the testing context. Whereas in MDT, the testing context plays a central role and the high level test design activity essentially consists of describing the SUT's expected behaviour within that testing context.

2.3.5 A few Words on Model-Based and Model-Driven Testing

Despite the undeniable benefits it brings into the test development process, MBT also raises a couple of challenges and open issues that need to be addressed for its successful application in a broader community. The next sections discuss each of those challenges.

Model granularity

Very often in integration testing, sub-system level testing or interoperability testing deriving meaningful tests from system models is a pointless task, because such testing implies taking into account functionalities that go beyond what could be represented as a single state machine, sequence or interactivity diagram. In fact, test scenarios in such situations involve several types of diagram at the same time, with the additional information on how to combine them being provided as natural language descriptions, which are inappropriate for automated processing.

Model testability

According to Binder [16], a testable model is one that contains sufficient information to allow automatic generation of test cases. Therefore, the model has to meet the following criteria:

- Completeness and accuracy: The model must represent all features that need to be tested. Also it should reflect the SUT's design as well as its specified behaviour.
- Balanced level of detail abstraction: the model should not contain too many explicit details of the SUT to keep design and maintenance cost at an acceptable level, but at the same time, it should preserve details that are essential for revealing faults and demonstrating conformance.
- Clearly defined concepts: The model must define the concepts on which it relies to describe structure and behaviour (e.g. state, events, actions) in a clear and precise manner so that they can be verified accordingly.

As one can easily imagine, providing models that effectively meet those criteria is a by no means trivial task that requires not just appropriate tools, but also a fairly good level of expertise. Moreover, as Briand et al [22, 20] pointed out, the usage of the UML notation - which is the de facto standard for modelling and is quite popular in MBT - does not help very much in that instance. They argue that

> ”since the use of the UML notation is not constrained by any particular, precise method, one can find a great variability in terms of the content and form of UML artifacts.”

Furthermore, it is worth noting that SUT models provide a user's or developer's perspective on the system. This might be explained by the fact that SUT models are provided by designers (and sometimes even developers) with the aim of implementing functionalities that fulfil the requirements on the software. Such requirements being generally user-driven, tend not to cover test-related aspects by focusing mainly on the expected behaviour in the system. Testing on the other hand has to go beyond the expected behaviour to uncover errors that might occur in case the SUT is confronted with unexpected inputs or unspecified user behaviour. Otherwise, instead of testing the system itself, the tests merely validate the SUT's model. Although it is commonly agreed that this type of model validation is an important step towards software quality, it is also widely acknowledged that it cannot be a substitute for “real” tests of the implemented system.

Model correctness and self-consistency

It should also be kept in mind that MBT cannot be viewed as the one solution to all testing problems, based on the mere fact that tests are automatically generated from models. Just as software will always contain failures, independently of their level of abstraction, models can also be faulty. Therefore, the quality of the formal model used as the base for MBT is fundamental for the whole process. As Heimdahl [79] put it,

> If the models are wrong, the testing effort will be inefficient or possibly outright deceiving (if we blindly accept the results of testing using poorly validated models).

However, according to Heimdahl, robust techniques for validating those models have been lacking ever since Dalal et al pointed out that challenge, back in 1999 [38]. Although domain expertise from project developers can be used to check the quality of MBT models as suggested by Dalal, it is obvious that those activities should be automated for more efficiency.

The human factor

Current MBT approaches and their associated commercially available tools expect the tester to be 1/3 developer, 1/3 system engineer, and 1/3 test engineer [38]. Such a combination of skills is hard to find in most test teams. Besides such skillful testers would hardly be affordable for many organisations. Therefore, the methods and tools for MBT should adjust to that fact.

Difficulty of creating and maintaining a SUT model

Creating and maintaining a model for a complex software system is less trivial, than it is sometimes suggested in many publications on MBT. In fact, the SUT model for test generation is generally more complex than the one used by developers, because it needs to combine several views on the system and correlate them as single model. That means difficult decisions have to be taken by the tester regarding for example which details to include in that model or to leave out, the algorithm to use for test generation, the appropriate level of abstraction [126] etc. The size and the manageability of the resulting model will ultimately depend on the complexity of the SUT's behaviour and the model coverage level targeted by the tester. These difficulties can partly be addressed by reusing elements of the SUT's design model already used by system developers in the MDE process. A way of achieving is by annotating the initial SUT model with additional testing-related information that will be exploited for automated test case generation.

Lower understandability of automatically generated test cases

The usage of MBT tends to make it difficult to identify reasons why a test case fails and to address the cause [38]. As Brinksma et al [23] rightfully stated:

> Not only detecting errors is important, but also locating and diagnosing errors.

This leads to the "loss of collateral validation and verification" [79], i.e. the ability to combine automatically generated tests with those developed in a manual process by experienced testers. Moreover, if it is not clear what functionalities of the SUT the tests actually assess, then confidence in the quality of the software will hardly be increased.

Model scalability

Another important challenge faced with MBT is that of scalability. Scalability refers here to the ability to handle SUT models of increasing size and complexity in a stable and satisfactory manner. MBT relies on algorithms originating from graph theory for traversing the various paths of a finite state model representing the SUT's behaviour [101]. Depending on the possible input/output combinations at each state, test cases can then be generated automatically to assess the SUT. While this works pretty well for simple SUTs consisting of only a few states, the number of generated test cases grows exponentially with increasing complexity of the SUT's state model, eventually leading to the well-known *state space explosion* problem [77, 77, 23, 48]. A consequence of this is the fact that the number of generated test cases may grow into unmanageable proportions, thus making the approach lose its intended efficiency.

Dealing with the real (imperfect) world

MBT heavily relies on the availability of complete and correct requirements specification and SUT models. However, besides the fact that formal requirements are rare [135], a machine processable model (MPM) of the SUT is not always available. This situation might occur, either because it was simply not planned or because the cost for providing a system model solely for the purpose of test generation would outweigh the potential benefits of automated test generation. In many cases, only parts of the system model are expressed in a machine-processable notation, with the biggest part of the system specification being provided as natural language description documents. For example the type system and the application programming interfaces (APIs), i.e. the static aspects of the system, might be specified in the Unified Modelling Language (UML), the Interface Definition Language (IDL), the Web Service Definition Language (WSDL) or any other similar notation, while the dynamic parts would be provided as a combination of sequence charts, state diagrams and natural language descriptions. This emphasizes the need for MBT to integrate with the whole software engineering process as noted by numerous authors [23, 77, 41, 135].

Taking the testing context into account

For the sake of abstraction, MBT approaches tend not to take the testing context into account. The term *testing context* denotes the architectural environment in which the SUT needs to be put for test execution, the communication points it provides for input or output, and the constraints to be considered for ensuring a proper operation (e.g. required components and functionalities). As a conscequence of this, the generated test cases may remain at too high level of abstraction and thus unsuitable for test execution. Moreover, if the additional effort for transforming those abstract test cases into executable ones is too high, the benefits expected from applying MBT may be lost in the process.

2.3.6 Conclusions

As a technique that has already been advocated and put into practice in several works [162, 15, 18, 66, 65, 89], MBT appears to be a promising approach, potentially yielding the same type of benefits obtained with MDE for generic software products. However, its adoption by the testing community and the software industry as a whole has remained extremely low. As early as in 1995, Lai [102] raised that issue for testing of communication protocols. In later works [100, 101] he continued, stating that

> There is not muchprogress in the use of test sequence generation techniques for practical testing of communication networks. Test design is still largely performed by testers by interpreting the specifications written in a natural language.

The figures collected and analysed by Neto et Al in their recent survey of MBT approaches [41] and reports by other researchers [135] seem to indicate that this issue continues to affect testing, even beyond the communication domain. In an associated article [115], Neto et al go as far as questioning the existing amount of practical evidence brought up by researchers to prove their assumptions in that domain, despite many seemingly promising and conclusive case studies [38, 36, 128, 155, 30, 6, 127] describing its successful application in the research and the industrial context.

Lai explains this big gap between academic and industrial testing practices with the fact that academia has not been addressing the real-life testing issues and problems [100] One reason that is often mentioned to explain the low acceptance of MBT/MDT is the fact that the proposed methodologies are difficult to use and sometimes unnatural in their application.

While many of the challenges mentioned above have a lower impact for systems displaying a lower level of complexity in terms of their behaviour, approaches are yet to be developed to address them for more complex software systems.

For all those reasons mentioned above, “manual” test development is still a common practice and is even likely to remain so for a few more years to come, given the fact that it will undoubtedly require some more time until MBT reaches the level of maturity required to evolve from an academical discipline into a broadly established practice. Besides, independently of automation a certain level of manual intervention in test development will always be required for deriving meaningful tests from system specifications and requirements. MBT simply raises the level of abstraction at which that manual intervention occurs and facilitates automated test case generation based on those high-level models. Nevertheless, designing test cases requires a good knowledge of the SUT, as well as testing expertise, both of which are hard to find and costly assets. Amazingly, while a large amount of tests are developed following that manual process, the level of automation currently available to support those activities is still disappointingly low. This is where MDT comes into play, by providing a model-driven approach including automated validation and iterative transformations of test models towards executable test scripts. A detailed discussion on MDT is provided later on in the next chapter of this thesis.

2.4 (Design) Patterns

The concept of design patterns as it is currently known in the software development domain originates from the work of Alexander [3], an urban architect who had the basic idea of recording design wisdom in a canonical form. He defines a pattern as

> both a description of a thing which is alive, and a description of the process which will generate that thing.

As Buschmann et al [26] pointed out:

> A pattern is the result of abstracting from a given (set of) problem-solution pair(s) and distilling common factors, which can be reused to solve other problems.

This process of analysing existing solutions and extracting the essence of a set of problem-solution-benefit combinations is called *pattern-mining*, whereas the reverse process of producing a solution for a similar problem-benefit pair by applying the previously identified pattern is called *pattern instantiation*.

It soon became obvious that the concept of patterns introduced by Alexander for urban architecture could also apply to nearly any design and engineering field. In analogy to the design patterns for urban architecture, software designers acknowledged the existence of patterns in software design and the need for identifying and documenting them, in such a way that they would possibly be reused wherever the context might require it to generate new solutions. A key event in the history of software design patterns was the publication of the book "Design Patterns: Elements of Reusable Object-Oriented Software" by E. Gamma, R. Helm, R. Johnson and J. Vlissides, also called the Gang-Of-Four (GoF) [62].

Jacobson et al. [88] define a software architecture pattern as both a part of a software system and a description of how to build that part. The purpose of software architecture patterns is to identify and specify abstractions above level of single instances or components in a software system, as well as to document existing well proved design experiences, software architectures and guidelines. Also software patterns provide a common vocabulary and understanding for design principles and well-proven experiences.

There has been some amount of controversy around the concept of patterns in software engineering and how they relate to existing software methodologies. As described in [26], emphasis must be put on the fact that patterns can and should not be viewed as solution for all possible software engineering problems and one should not attempt to force patterns reuse in situations where they simply would not fit. Patterns should rather be viewed as a complementary approach to existing methodologies.

Also, patterns should harmonize with the fundamental principles of software construction commonly known as *enabling techniques* [26], which are independent of a specific software development method such as Abstraction, Encapsulation, Information Hiding, Modularization, Separation of Concerns, Coupling and Cohesion, Sufficiency, Completeness and Primitiveness, Separation of Policy and Implementation, Separation of Interface and Implementation, Single Point of Reference, and Divide-and-conquer [26]. While some patterns address some of those concepts explicitly, it is important to make sure that patterns do not affect those principles negatively.

This also holds true for the usual non-functional requirements on software systems, i.e. changeability, interoperability, efficiency, reliability, testability and re-usability [85]. It should be kept in mind that while some patterns will aim at enhancing some of those requirements and help in achieving them, it is also possible that a given pattern affects some of the non-functional requirements negatively. For example, the broker pattern, which is the base of many middleware architectures such as the Common Object Request Broker Architecture (CORBA), eases testing of individual client or server components in a distributed system. However it decreases the testability of client-server systems by introducing additional elements between the client and the server.

2.5 Summary

Testing has been the object of a significant amount of research activity in the last decades and has evolved from something performed “en passant” by software developers into a complete discipline of its own, following the same process as generic software development. Test development is just another way of approaching software development from the quality insurance perspective, but to be successful, testing also has to be done in a rigorous and systematic way. Otherwise that would ultimately lead to a lower quality for the resulting products or cause more costs. However, with testing evolving into a discipline of its own, methodologies are required to fasten test development to avoid an explosion of costs stemming from the growing complexity of both the systems to be tested and the test systems themselves.

With the growing popularity of models in software engineering, model-based testing has been the source of (too) high expectations as a means for addressing the challenges of ever complex software systems and shorter test delivery time spans. However, MBT has not yet reached a high level of popularity among testers and developers. Therefore, test engineering has remained a highly manual, repetitive and error-prone process in many domains. Automating those manual tasks and enhancing reuse both of artifacts and concepts through patterns is a promising approach for addressing those issues. Although this could also be achieved through patterns at the test scripting level (e.g. using some dedicated libraries or specific macro-like scripting language idioma), the introduction of a model-driven approach to test development will undoubtedly facilitate that task and increase both efficiency and usability, by allowing tests to be designed at the appropriate abstraction level for those activities. In the next section the current state of the art of such model-driven testing approaches is discussed.

Chapter 3

State of the Art in Model-Driven Test Automation

3.1 Introduction

Patterns are the result of an abstraction process, in which the common essence shared by a set of existing solutions to a recurrent problem is extracted, so that new solutions future occurrences of the same (or similar) problems would be instantiated more easily. Since they address issues at different levels of abstraction, patterns are described in many different ways, ranging from natural language through object model diagrams to source code snippets or templates. The choice of one method or the other is driven essentially by the targetted domain and the usability of the resulting solution.

As a mean for raising the abstraction level in software development, MDE can facilitate the exploitation of patterns in test automation. In Chapter 2, Model-Driven Testing was defined as an approach consisting in applying the MDE method to design solutions for test automation. Furthermore, the potential benefits of combining MDT and test design patterns were highlighted.

In this chapter, the current state of the art of existing MDT approaches is reviewed. As described in Chapter 2, the MDT process consists of successive transformations from a platform-independent test model (PIT) through platform-specific test models (PSTs) into executable test scripts. That process basically remains unchanged, independently of the technology used for achieving it. Therefore, the main differentiating factor between MDT approaches is the notation(s) used to express testing concepts as PIT at a high level of abstraction and the methodology they carry.

There is a wide variety of existing notations for test design available both in academia and in the industry [64]. Therefore, selecting one that is appropriate for one's needs is by no mean a trivial task. Basically, two main categories of approaches can be identified for model-driven testing, depending on whether a Generic Modelling Language (GML) or a (Test) Domain-Specific Modelling Language (DSML is used to model the tests.

Significant differences exist between the requirements for generic software design notations and those targeting test design. While the former tend to focus on expressive power with generic concepts that match the complexity of today's software and information systems, the latter rather aim at providing a simplified view on those systems to enhance their understandability and to help uncovering errors they may contain. Additionally, test development involves a process that includes its specificities, despite the many similarities it shares with software system development. Those specificities include [11]:

- The ability to model assertions and expectations.
- The ability to model test-related roles for entities.
- The ability to model means for verifying constraints: while constraints can be considered sufficient for modelling generic software systems, test models also require a description of how those constraints will be verified and which impact their violation would have on test results.
- The need to provide traceability to other aspects of the development process (e.g. requirements management, fault management).

Even though workarounds could be found for fulfilling those requirements on test development using GMLs, those are in many cases too inconvenient to be used efficiently in an MDT process.

Taking those specificities of test design into account while modelling tests would be highly beneficial for the process, because it would automate manual test development where it is needed most, while at the same time ensuring that the test models remain concise and precise.

3.2 Using GMLs for MDT

GMLs are notations primarily defined to model a wide range of software and computer system types. UML is probably the most popular GML in software development at the moment.

The UML is the industry standard notation for high level software design and has continuously been gaining popularity since its introduction. It provides a wide variety of diagramming possibilities to model software at any level of abstraction and benefits from solid tool support and a well-proven standardization process. Therefore, it appears to be a natural choice to consider UML for also designing tests in the same manner as other aspects of software engineering. Beyond the user-friendliness resulting from the familiarity of the UML and the usage of well-established CASE tools, it is also assumed that using the same language both for test and system design would make it easier to understand the links between both types of artifacts and facilitate automated transformations between them [121].

Proponents to the usage of UML as-is for test design argue that, it already provides all concepts required for that purpose and that using the same notation for modelling both the SUT and the test system will be beneficial for the whole development process. Baker et al [11] describe a case study conducted at Motorola, featuring a concrete example of such usage. Although no figures are put forward to support their statement, the authors claim that the approach has proven be very successful through automated test generation and reuse of test models in reducing the effort for developing tests, improving test coverage and increasing the failure detection capability of the test suites. However, as they also rightfully pointed, the usage of the same notation does not spare the test designer the challenges that are inherent to any application of MDE to testing. Namely [11]:

- insufficient tool support for
  - exchanging model artifacts between system architects, testers and developers,
  - migrating from legacy notations (e.g. SDL, MSCs),
  - integrating the various aspects of MDE as a comprehensive process,
  - handling large models (scalability),
- inadequacy of system models (Level of abstraction, incompleteness, platform specifics),
- lack of well-defined semantics originating from known *semantical variation points* of UML,
- difficulty of coupling data and behaviour in a reusable manner,
- team inexperience.

The authors describe how those various issues were addressed using some proprietary notations and tools besides standards such as TTCN-3, UML and UTP.

3.2.1 The UML Testing Profile (UTP)

The UML Testing Profile [70, 10, 162] (UTP) is an extension of UML standardised by the OMG for expressing test design concepts at higher level of abstraction. UTP adds test design concepts to the UML superstructure to define a language suitable to be used as stand-alone meta-model for test design or integrated with

UML for combined test and generic system design. UTP groups those concepts in four main categories: test architecture, test data, test behaviour and time. As a UML profile, UTP inherits the positive characteristics of UML, such as its extensibility (through the profiling mechanism), its included support of partitioning (through packages) and separation of concerns (layered metamodel architecture). While, this facilitates the dual usage of UTP, i.e. both for combining the design of test artifacts with that of generic systems, it also comes with some important drawbacks to be taken into account.

The UML Testing Profile defines a language for designing, visualising, specifying, analysing, constructing and documenting the artifacts of test systems. Apart from the numerous advantages of UML, this approach aims at facilitating the understandability of test models and their adoption by other stakeholders in the software development process. Furthermore, it is assumed that test design activities would benefit from the large variety of tooling facilities already available for UML, if that same language (or an extension thereof) were used.

However, the usage of UML and extensions thereof for test design requires that some of the issues regarding the semantics of that notation are addressed. Henderson-Seller [80] provides an overview of the pros and cons of UML, pointing at issues raised by experts regarding the impreciseness of that notation for modelling the behaviour and the structure of software systems. While SUT models may afford such impreciseness, test models must clearly specify what functionality of the SUT they assess and indicate precisely how that assessment will be performed. The impact of those issues on a potential usage of UML for high level test design has been analysed by authors such as Brinksma et al. [23] and Pickin et al. [121]. Moreover, the UML might be too complex, too difficult and too generic for the sole purpose of high level test design. In fact, many of the existing implementations of UTP are actually provided as plug-ins for UML modelling tools, which in most cases do not provide a specific process for test modelling, but rather for modelling generic object-oriented systems. This makes the process of modelling tests with those tools inefficient as they do not address the concerns of testing directly, but through some workarounds. Because of all these factors, the adoption of UML for test design appears to be coupled with high costs and risks with no guarantee of a beneficial impact at the end of the effort.

Also, it is very important that other non-functional aspects of the test development process (e.g. test planning, test analysis) are taken into account in the test modelling process, for it to be successful. Clearly, some of those aspects are considered only marginally, while others are not considered at all by GMLs. For example, while it extends the UML with concepts specific to testing such as test architecture, test behaviour and test data, the UTP covers many of those concepts by only defining their abstract syntax, with vague description of their semantics, if any. This is nevertheless not surprising and aligned to the UML, knowing that a standard is per se a compromise which rather than going too much in the details of things, aims at providing a common framework for a given methodology.

Furthermore, the GMLs neglect the visualisation of test artifacts, assuming that the means provided for generic software design would be sufficient. This is somewhat unrealistic, because if a diagram is used to represent test artifacts graphically, then the test-specific concepts contained in those artifacts should also be visible on the diagram. Otherwise the benefit of visualisation would be lost altogether.

3.3 Using DSMLs for MDT

DSMLs are notations especially tailored to allow the modelling of concepts specific to a particular domain. A domain in this context can be understood both as application domain (e.g. automotive, telecommunications, finance, etc.) or as problem domain (e.g. application deployment, testing, packaging, etc.). DSMLs are often extensions or restrictions of GMLs. For example, the SysML notation, which defines a DSML for systems engineering beyond the OO-paradigm, extends the UML notation.

Proponents to the usage of DSMLs for test design emphasise the need for notations that clearly address the purpose of modelling test concepts, without the burden of inheriting a large catalogue of concepts that are irrelevant in that specific context. A justified concern that arises in that scenario is that of the additional effort (and costs) in designing, maintaining and implementing yet another notation to the plethora of existing ones, both in terms of human resources and technical capabilities. However, the fact that the scope of DSMLs is narrower than that of UML may contribute in reducing the efforts required for learning and using them. Furthermore, with recent progress in MDE, a large set of tools and platforms exist that have made the effort of designing a new DSML and implementing an associated toolset a less daring adventure.

The UML standard [74] lists several potential motivations for designing a DSML by customising UML:

- To give a terminology that is adapted to a particular domain.
- To give a syntax for constructs that do not have a notation.
- To give a different notation for already existing symbols
- To add semantics that is left unspecified in the meta-model.
- To add semantics that do not exist in the meta-model.
- To add constraints that restrict the way you use the meta-model.

- To add information that can be used when transforming a model to another model or code.

Each of those motivations apply in the case of a DSML for test design.

3.3.1 Approaches for Designing a DSML

The choice of an appropriate approach for designing a DSML for MDT requires several factors to be taken in consideration. Firstly, to reduce the costs of designing and implementing a DSML, it would be more efficient to do so, based on a well-established existing GML, provided it offers the required customisation mechanisms.

Since UML provides such customisation mechanisms and has established itself as the standard notation for modelling in computer science,it was decided to take it a the base for any test design DSML.

One of the most interesting features of UML is the fact that it provides several possible customisation mechanisms. In their article on customisation approaches for UML [25], Bruck et Al. provide a detailed list of those possibilities that can be grouped in two main categories:

- The Meta-Object Facility (MOF) based approach: MOF is a standard [72] defined by the OMG as a 4-layers meta-modelling architecture to allow the definition of DSMLs in a way similar to Extended Backus-Naur Form (EBNF) for defining language grammars. Two variants of MOF are defined by the OMG, i.e. Essential MOF (eMOF) and Complete MOF (cMOF).

- UML "built-in" extension approaches (e.g UML profiling and Reuse through specialisation or copy/merge of UML meta-types): The usage of UML profiles is one of the most popular approaches in this category, thanks to the many advantages it provides [25]:

  – Easy to create such extensions
  – Well described with documentation in Superstructure Specification
  – Standard means to define icons
  – Well defined display options.
  – Application of profiles and how to use them is well defined.
  – Can add structure
  – Low development cost
  – Leverage existing UML editors
  – Ease of deployment.

Those advantages might have played an important role in motivating the definition of the UML Testing Profile, which is currently the best known notation for MDT. However, UML profiles also come with some disadvantages [25]:

- Inability to specify behaviour
- Impossibility to remove existing constraints.
- Clumsy programmatic usage
- Impossibility to modify existing structures

Alternatively to using a notation based on UML customisation, a test modelling language may follow a different approach, e.g. by using a generic format to express test specific concepts. The XML-based notation TML [56] (Test Modelling Language) is an example application of that approach, whereby instead of using a meta-model to define the abstract syntax of the notation, an XML schema descriptor is used. While this might appear as an attractive alternative, based on the assumption that the XML format is widely used, platform-independent and generic, it has some disadvantages that are likely to emerge in the long term:

- Editing interface: The process of editing XML files can be very tedious and error-prone, if an appropriate GUI is not provided for that purpose. The effort for designing, implementing and maintaining that infrastructure will have to be taken into account while considering going along that path.

- Customisation difficulties: Using an XML-based DSML makes it more difficult to use a whole set of facilities provided by MDA-infrastructures for semantically validating the test model (e.g. using OCL-constraints) and for model transformation into other notations. Although facilities provided by XML tools might be helpful in facing those issues, the effort required for integrating them into existing development infrastructures should not be underestimated.

3.4 Related Works

Most of the existing works on model-based and model-driven testing address automated test case generation from models of the SUT. Besides the work on the UTP, the following works were found, addressing the same issue as this thesis:

The approach proposed here is based on the same motivations as the work of Pickin et al. [120, 119, 121] towards a formal and yet user-friendly mean for describing tests at a higher level than notations such as TTCN-3. After assessing the suitability of UML 1.4/1.5 and UML 2 for that purpose, the authors come to the conclusion that there is a need for a new notation to address the shortcomings of existing languages. Their new notation - called *TeLa* - is based on UML sequence diagrams, but introduces specific semantics to address the issues identified with UML and MSCs. However the authors focus mainly on the behavioural aspects of high-level test design and although other aspects such as test data design and test architecture design are also briefly discussed, they are obviously not covered with the same level of detail. Furthermore, in this thesis, the focus was laid more on usability and and reuse based on patterns than on formalism.

Another work in the same area is that of Baerisch [9], who proposes an approach labelled Model-driven Test Case Construction (MTCC) which aims at decoupling SUT implementation details from system tests. The author argues that this would improve reuse of those tests, especially for product lines consisting of many variants of systems that share a certain amount of features.

Al Saad et al [141] present a visual model-driven testing framework for wireless sensor networks applications. Their approach consists in using a visual domain-specific language (DSL) to create a model of the test cases, that is then refined through a series of transformation steps into executable test case code (Java/C++) that can be run on an engine called *ScatterUnit*, developed specifically for that purpose. A limitation of their approach resides in the fact that it addresses a particular kind of applications exclusively, i.e. wireless sensor networks applications. Possibilities for applying the method beyond that domain, though not discussed explicitly in the paper, appear to be feasible.

Grossmann et al [69] propose to use a DSML called TestML to address the challenges faced with the testing of embedded software in the automotive industry. The authors describe TestML as an interchange format between the technologically heterogeneous test infrastructures present in that domain. One of the main characteristics of those infrastructures is the fact that they operate at different phases of the development process such as Model-in-the-Loop (MIL), Software-in-the-Loop (SIL) and Hardware-in-the-Loop (HIL). TestML is thus defined as an XML-based format for facilitating the reuse of test artifacts among those different phases and the heterogeneous tool landscape they imply. Although they do not mention an MDT process explicitly, a transformation of the abstract concepts expressed with TestML into more concrete representations for the respective test infrastructure appears to be a logical following step.

The potential benefits of cataloguing best practices and patterns in test design has been advocated by several authors before. Binder [16] discusses a test pattern template, based on a pattern language of object oriented testing (PLOOT) proposed by Firesmith [54] and introduces a collection of test patterns from the object-oriented software design domain. Meszaros [109] presents a collection of test patterns for unit testing. Howden [81] presents a collection of patterns in selecting tests for maximum error detection. It appears that existing work on test patterns tend to focus on interactions at the object level and are hardly applicable for higher level (i.e. integration, system, and acceptance-level) testing whereby the applied programming paradigm are less relevant. Delano et al [39] present a collection of patterns focussing more on the organisational aspects of test development as a process, rather than on test design itself. On the other hand, Dustin [47] covers all aspects of test development, with one chapter dedicated to test design and documentation. in 2005, the European Telecommunications Standards Institute (ETSI) started an initiative on patterns in test development (PTD) in which some of the patterns defined in this work were introduced and discussed. However, other attempts to formalise test design patterns, so that they could be instrumented to support the test development process in an automated manner, as proposed in this thesis, could not be found.

3.5 Conclusions and Summary

The MDT approach clearly appears to be more appropriate for allowing the formalisation and the exploitation of patterns in test engineering, because it provides the ability to work at the right level of abstraction, while at the same time keeping the amount of flexibility required to design a process that would support the usage of those patterns. Therefore, MDT plays a more important role in this thesis, while MBT is considered to a lesser extent.

However, it should also be kept in mind that MBT and MDT are not necessarily mutually exclusive alternatives. When transforming models into test sequences, most of the currently existing approaches tend to do so directly into a lower level test implementation language (e.g TTCN-3) or into source code for the target environment on which test will be executed (Java, C++, C etc.). As argued by [9], this coupling of the test cases with lower level implementation details make those difficult to maintain. Similar results were obtained in first experiences of applying MBT with test patterns to TTCN-3 test development [159]. MDT can be the mean for decoupling the test cases from lower-level implementation details. Instead of generating test cases directly into the target test environment's lower-level notation from a model of the SUT, the model-based automated test generation(ATG) tool would generate a platform independent test design model that can then be refined for each respective target environment using MDT techniques.

Furthermore, a DSML-based approach was chosen to express the test design concepts suitable for MDT, rather than an approach based on a GML. This decision was taken based on the discussion presented in Section 3.2 and Section 3.3.

The need for MDT and MBT has now widely been acknowledged in the testing community, but the methodologies proposed still hardly find their way into the test development process. Although it bears a high potential for enhancing reuse in test development and in optimising manual test development, MDT has not benefited from the same level of interest as model-based automated test case generation. This is partly explained by the fact that the existing proposed solutions have mostly tried to force the usage of generic modelling notations into something they were primarily not designed for. That has made those approaches inappropriate to take patterns in black-box test engineering into account and to reflect the specificities of the black-box test engineering process for reactive systems. As a result, existing solutions are often viewed as clumsy and inadequate by testers, which ultimately lead to their inefficiency. By combining MDT techniques with test design patterns using a dedicated high-level DSML for black-box test design, the pattern-oriented model-driven testing approach presented in the next chapter provides a solution to address those issues.

Chapter 4

Pattern Oriented Model Driven Testing

4.1 Introduction

Test patterns represent a form of reuse in test development, whereby the essences of solutions and experiences gathered in testing are extracted and documented to enable their application in similar contexts that might arise in the future. The idea is to capture test engineering knowledge from past projects in a canonical form, so that future projects would benefit from it.

Essentially, the following benefits can be expected from the exploitation of patterns in any software development process:

- Patterns facilitate and improve communication by providing a common vocabulary for computer scientists across domain barriers [62].
- Patterns help managing software complexity [26].
- Patterns support the construction of software with defined properties [26].
- Patterns provide a documentation and learning aid [62].
- Patterns facilitate refactoring of source code [62].
- Patterns capture (design) knowledge and experience [26, 1].

Although most of those benefits are hardly quantifiable, their potential quantitative and qualitative impact on the software development process is undeniable.

Test systems are a special type of software systems and with their growing complexity, the need for cataloguing good practices with regard to design, architecture, implementation and execution is becoming more and more urgent.

Just as for any other software product, well-proven experiences gathered while developing test systems need to be documented to ease their reuse.

A test pattern can be defined as a special kind of software pattern that applies specifically to the testing domain. Similarly to general software system engineering, the benefits expected from patterns in testing are both quantitative and qualitative.

The ISO/IEC 9126 standard [85] defines a model for internal and external quality of software, including quality characteristics and associated metrics. Applying that model to test development, Zeiss et al [165] identified the following characteristics of quality for test specifications[1]:

- *Test effectiveness*: Test effectiveness describes the capability of tests to fulfil their given test objective(s), including characteristics such as *coverage*, *correctness* and *fault-revealing capability*. However for the type of testing addressed in this thesis, emphasis is laid more on correctness than on other characteristics.

- *Reliability*: Reliability describes the capability of a system to maintain a specific level of performance under different conditions. When applied to tests, reliability includes *test repeatability* and *security* additionally to *maturity*, *fault-tolerance*, and *recoverability* mentioned in ISO/IEC 9126. Of all those sub-characteristics, *test repeatability* is the one that plays a more important role for black-box conformance testing as they are discussed in this thesis.

- *Usability*: In the context of testing, usability denotes the ease to actually manage, instantiate or execute a test suite. Although the management aspect is not explicitely mentioned by Zeiss et al, it is a key characteristic in this thesis, because focus is laid less on test execution and more on the process of achieving executable tests. Usability will therefore denote the ease of manageing test artifacts in such a way that the process would be facilitated.

- *Efficiency*:Efficiency is defined as the capability of tests to provide acceptable performance in terms of speed and resource usage, when executed [165]. Obviously, this characteristic is more applicable to executable test suites than for abstract test suites(ATS). In this thesis, efficiency will be understood in a similar manner to usability, i.e. more relatively to the test development process than to the executable test suite it produces as outcome.

- *Maintainability*:Maintainability denotes the capability of a test suite to be modified for error correction, improvement, or adaptation to changes in

[1]The term *test specification* is used here to denote an executable test suite

related artifacts (e.g. requirements, system specification). It includes characteristics such as *analyzability*, *changeability*, and *stability*. Obviously, maintainability will play an important role in assessing the quality of test suites in this thesis, because it is one of the characteristics claimed to benefit most from MDE in generic software system development. Therefore it will be interesting to assess whether similar benefits are possible with MDT.

- *Portability*: Portability include sub-characteristics such as *installability* (ease of installation in a specified environment), *co-existence* (with other test products in a common environment), *replaceability* (capability to be replaced by another item for the same purpose) and *adaptability* (capability to be adapted to different environments).

- *Reusability*: Although it is addressed separately by Zeiss et al as a characteristic of its own, reusability can actually be viewed as a sub-characteristic of *maintainability* discussed above. Therefore, a similar impact of MDT on this characteristic can be expected and will be measured in this thesis.

Although it can be assumed that test patterns will have a positive impact on each of the characteristics mentioned above, that impact is expected to be more important for *usability*, *efficiency* and *maintainability* (i.e. including *reusability*).

Additionally to those qualitative improvements, a quantitative improvement through a reduction of time-to-market and costs can also be counted among the benefits expected from pattern-oriented test engineering.[2]

In the next sections, different views on the concept of test patterns will be described and methodological aspects such as notation, test pattern mining and test pattern application will be discussed.

4.2 Classification of Test Design Patterns

4.2.1 Introduction

The issue of patterns in general and especially that of test patterns has very often been a source of some misunderstandings among experts. This stems from the generality of the concept, which leads to the fact that, depending on the abstraction level under consideration, different definitions and classifications will be obtained as a result. That abstraction level ranges from higher level generic discussions on the do's and dont's in organising and managing testing projects, to a more technical approach to the issue, aiming at optimising the engineering aspects of the testing process. For example, Delano and Rising [40] discuss the

[2]A measure of those improvements is provided in Chapter 6.

issue of patterns in test development at a high level of abstraction, which involves aspects such as the management of test projects and test organisations, the strategies for achieving higher efficiency in testing, etc.

On the other hand, whenever test patterns are addressed from an engineering perspective, their nature and results are also influenced by the three characteristics used to classify tests and illustrated in figure 1.2. Namely, test scope, test goal and testing phase. For example, the techniques for specifying, designing, implementing, executing and evaluating the tests will differ, depending on whether unit testing, integration testing or system testing is being performed. Examples of patterns for unit testing have been provided by Meszaros [109] whereas some for component testing of object-oriented software as described by McGregor et al [107] and Binder [16].

Test patterns are developed in this thesis according to the scope defined in Section 1.2 and illustrated on figure 1.3. As the engineering aspects in testing and test development are at the centre of this thesis, emphasis is laid on these aspects of the testing process rather than on the high-level test project management related ones.

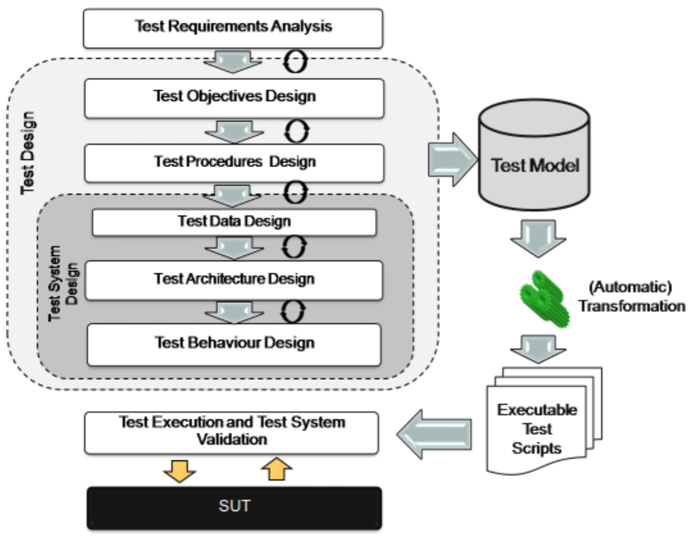


Figure 4.1: Overview of Model-Driven Test Engineering Process

Figure 4.1 illustrates the model-driven test engineering process and the various phases it comprises. As depicted in that figure, the process starts with an analysis of the test requirements for the system under test. Those requirements are then the base for designing a test model which can be transformed into executable test scripts to assess that the SUT's behaviour meets the specified requirements. The test modelling part of that process (i.e. the dashed grey-coloured box in the figure) is the main topic of this work. Therefore, the test patterns that are developed cover each of the phases of that part of the process. Accordingly, the approach for classifying test patterns is aligned to those different phases.

In the next sections, the different phases of that process are discussed, along with an analysis of which type of test patterns could be identified and possibly exploited, to facilitate the activities and to improve efficiency in that phase of the process.

4.2.2 Generic Test Design Patterns

Generic test design patterns are those that can be found and applied to all activities of test system design. They address concepts that spawn over the whole test modelling process and thus cannot be confined to a single activity.

4.2.3 Patterns in the Test Analysis and Planing Phase

As depicted in Figure 4.1, the test analysis and planing phase comprises two activities:

- An analysis of the SUT's requirements from a testing perspective to derive test objectives
- A design of test procedures to assess the defined test objectives on implementations.

Test Objectives Design Patterns

Test objectives definition is the first step in building a test system. It consists of extracting test objectives from the SUT's specification, depending on what the test goals are going to be. Test objectives can be viewed as the equivalent to system requirements in system development and are sometimes also referred to as test purposes or test directives in the literature. Test objectives design patterns are applicable when modelling which functionalities of the SUT the tests will have to assess.

When performed manually and without a clear systematic approach, the process of deriving test cases from test objectives can be quite costly and error-prone. Hence, the need for formalising how test objectives are described has arisen. In the Pattern for Test Development (PTD) group initiated by the European Telecommunications Standards Institute (ETSI) some patterns have been proposed for that purpose [51]. The initial intent of that work was to enable automatic derivation of test cases from such formalised and machine processable test objectives. However, one had to acknowledge that there was some level of contradiction that made that goal difficult to achieve: an automated transformation of test objectives (i.e. a description of WHAT needs to be tested) can by no means provide enough information for a test case, which is an implementation of HOW the test needs to be conducted. Therefore, a couple of additional steps are required to describe in the same systematic and potentially formalised manner, how each test objective will be checked.

Test Procedure Design Patterns

After the test objectives have been identified, comes the step of designing the test procedures for the SUT, i.e. to specify, how the test objectives identified in the previous step will be checked. Those descriptions of how test objectives will be checked are called test procedures[3].

Test procedures design patterns are those that are applicable when designing how each of the test objectives is going to be checked. While they need to follow a clear structure and provide as much information as possible, test procedures do not need to describe the technical means required for performing the tests. Therefore, as they are supposed to be understood by various stakeholders in the software system development process (e.g. system designers, test designers, test developers, quality management and product support personnel, etc.), they should (as much as possible) be expressed in natural language, however within clearly defined template structures.

4.2.4 Patterns in the Test System Design Phase

The lower part of figure 4.1 depicts the test system design phase in the test development process, whereby the composing elements the test system and their relations with the SUT are modelled. Those composing elements are used to provide the three main aspects of any test model, i.e. topology, data and behaviour.

In the process of designing each of those test system model elements, different types of test patterns can be identified and re-used. The next sections discuss those test patterns.

Test Architecture Design Patterns

The test architecture describes the topology of the test system, i.e. its composition as a set of (parallel) test components and the points of communication between those and elements belonging to the SUT. Depending on the goal of test(e.g. conformance, performance, functional, robustness, etc.), different test architectures might be more or less suitable. Test architecture design patterns define good practices and established recommendations in designing or selecting appropriate test architectures.

Architectural patterns address solutions as to how test architectures can be designed to solve or avoid specific recurring problems in producing high quality

[3]In previous works the term *test strategy* was used in this context, but it was rather changed to align to the IEEE-829 standard [83]

test solutions efficiently. This also includes patterns for the coordination and synchronisation of test components in a test system.

Test Data Design Patterns

Data patterns are test patterns describing reusable concepts and approaches for designing test data, i.e. data to be exchanged between entities in test architectures. Test data does not only mean concrete values or message objects, but also abstract values based on constraints defining properties, potentially used for evaluating data received from the SUT to assign a verdict to the test case.

Test Behaviour Design Patterns

Test Behaviour design patterns document approaches and principles for designing the behaviour of test systems, i.e. the interactions between entities within a test architecture.

Behaviour patterns might apply for a single entity of a test architecture (e.g. a test component) or for the interaction of test components with each other or with elements of a given SUT.

4.3 A Methodology of Pattern Oriented Model-Driven Test Engineering

A methodology for test patterns should not only address the various kinds of test patterns for the different approaches of test reuse, but also define how a test pattern is to be identified, specified, selected and applied. The following sections discuss those aspects of test pattern engineering.

4.3.1 Test Design Pattern Mining

Pattern mining is the process of abstracting from existing software design to identify patterns suitable for potential reuse in future contexts. Test design pattern mining is the application of pattern mining techniques to the testing domain. Several techniques are described in the existing literature for design pattern mining in generic software engineering [46, 45]. Those techniques aim at analyzing existing software artifacts (e.g. source code or high-level design models) automatically or semi-automatically (i.e. by involving human user interaction) to identify known design patterns or commonalities that may be elligible as candidates for new patterns. Those pattern mining techniques can be classified in three main categories, based on the aspects of software design they analyse to discover patterns. While some techniques analyse structural aspects of the artifacts, others analyse behavioural aspects and finally a last category of techniques cover both structural and behavioural aspects. A common point among all techniques is that they use an intermediary representation of the base artifacts to perform their pattern discovery algorithms, rather than the artifact in its original form itself.

The problem of test pattern mining does not differ much from that of pattern mining in generic software engineering, beyond the fact that here, the artifacts to be analysed are source code of abstract or executable test scripts , models of test systems or combinations thereof. Therefore, the pattern mining techniques for generic software engineering can also be applied for test pattern mining. However those techniques will have to be customized in such a way that instead of trying to identify generic design patterns, they would rather search for specific patterns that are relevant in their usage for testing. An example application of those techniques for test pattern mining in TTCN-3 test suites is reported by Neukirchen et al [117] who use an abstract syntax tree (AST) as intermediary representation of TTCN-3 source code to identify so-called *code smells*. Code smells are defined as patterns of inappropriate language usage that is error-prone or may lead to quality problems for the overall test suite. Besides facilitating reuse at the conceptual level and uncovering potential errors in existing artifacts, another motivation for pattern mining is to facilitate the understanding of existing artifacts so that they could be reegineered to address changed requirements. However, while the task of identifying the patterns and displaying their occurence in the artifacts can be done (semi-)automatically using the techniques described above, the task of making sense of the results generated by those techniques still needs to be performed by human beings based on their expertise. Visualisation can contribute in facilitating that human analysis by putting the collected pieces of information gathered through pattern mining in relationship to each other [45]. These findings can also be applied for test pattern mining. However, just as reverse engineering, design patterns are a rather recent topic in the testing domain. Obviously, as demonstrated in a case study in Chapter 6, the type of pattern-oriented test engineering described in this thesis can contribute in addressing those issues. Overall, although the process of going through existing test artifacts and trying to identify patterns for later reuse might appear costly and unrewarding at the first sight, in long term, it could effectively help in shortening the test development life cycle and hence reduce costs.

4.3.2 Test Design Pattern Template

Binder [16] defines a pattern template as a list of subjects (sections) that comprise a pattern. To unify the pattern definition process and to avoid misunderstandings between stakeholders involved in test development, such a template is required to serve as a guideline. Taking Binder's test pattern template as basis, a more refined test pattern template is proposed that is better adapted to the testing domain covered by this work. The content of the test pattern template depends on which of the benefits listed in section 4.1 are the main driving forces for pattern mining. While Binder's test pattern template is driven by the *test effectiveness* qualitative characteristic mentioned in Section 4.1, the pattern mining activities in this work are mainly motivated by other benefits such as *usability*, *(process) efficiency* and *maintainability*. For example, because it is more relevant for white-box testing and less for black-box testing, which is the main concern of this thesis, the *test effectiveness* criterium plays a less central role in this thesis than in Binder's work, although it is considered as well. Therefore the original template has been modified to reflect the fact that the focus is more on enhancing the test engineering process rather than on increasing the effectiveness of the tests. Another important difference between the test pattern template proposed in this thesis and the ones proposed in other publications is that, sections which play a less important role in the context of this thesis have been removed. For example, while the *subjects fault model*, *entry criteria* and *exit criteria* sections proposed by Binder [16] play a role for code-oriented, white-box testing, they are far less relevant for function-oriented, black-box testing, which are the main concern of this work. Therefore, the *applicable test scope* section was added instead, to capture the preconditions for applying test model patterns. This thesis' test modelling pattern template consists of the following subjects:

- *Pattern name*: A meaningful name for the test pattern.
- *Context*: To which specific context does it apply? This includes the kind of test pattern (organisational vs. design, generic, architectural, behavioural or test data etc.) as well as the test scope in which the pattern may be applied.
- *Problem*: What is the problem, this pattern addresses and which are the forces that come into play for that problem?
- *Solution*: A full description of the test pattern including examples of applications. Where applicable, UTML and TTCN-3 [58] will be used as notations for the examples.
- *Known uses*: Known applications of the test pattern in existing test solutions or existing concepts enabling the application of the test pattern in existing test specification or test modelling languages. Although this definition of the *known uses* section is slightly different from the one used in the patterns literature, it can still be considered a valid one, because test suites are not always publicly available to be referenced as known uses of a given pattern. However it can be assumed that the fact that a concept is provided in a test design notation indicates that there was a need for such a concept and subsequently that there are eventually existing usages of the concept, even if those have not been published.

- *Discussion*: A short discussion on the pitfalls of applying the pattern and the potential impact it has on test design in general and on other patterns applicable to that same context in particular.

- *Related patterns* (optional): Test design pattern related to this one or system design patterns in which faults addressed by this test pattern might occur. This section is optional and will be omitted, if no related pattern can be named.

- *References* (optional): Bibliographic references to the pattern. This section is also optional and will be omitted, if no reference can be provided.

4.3.3 Specification of Test Design Patterns

One of the key challenges to address with regard to pattern-oriented test engineering is that of selecting a suitable approach for specifying test patterns in such a way that their exploitation would be facilitated to automatically generate new test solutions. In the case of generic product software design patterns, J. Bosch [19] has identified the following three approaches for specifying design patterns:

- *Design environment support*: The design environment support approach consists in providing via the software design environment the capability to model new software designs along defined patterns and to annotate the corresponding source code accordingly.

- *Programming Language Extensions*: Design patterns come originally from Object-oriented (OO) software design. Therefore, the usage of OO programming languages or extensions thereof appears to be a natural choice for describing those patterns in a systematic manner. This approach is advocated by J. Bosch [19] and Hedin [78].

- *Generative approach*: The generative approach consists in using ontologies and meta-Modelling to describe design Patterns. For example, Fontoura et al [55] propose an extension to the UML notation to represent architecture design patterns. Kim et al [99] propose a UML-based meta-modelling language to specify design patterns. A similar approach is also advocated by Rauf et al [132], S.-K. Kim et al [94], Mak et al [105] and Le Guennec et al [75]

Although test design patterns have already been discussed in the literature in many instances, literature references on attempts to specify test design patterns are much harder to be found. Just as for software system development, one of the following three alternatives discussed above is applicable.

Specification of Test Design Patterns via Design Environment Support

One possibility for enabling the specification of test patterns consists in providing appropriate technical support through the test design environment. This can be achieved through so-called wizards, i.e. applications or applets that guide the test engineer stepwise through the process of specifying test patterns. While such wizards are already very common for generic software design and development tools, they are yet to gain the same level of popularity in test design environments.

An advantage of that approach lies in the fact that the test designer is not required to learn any new modelling notation, because he/she is only presented a template-like interface, through whichthe required information can be filled-out for specifying a new test pattern or instantiation thereof, while the test design environment takes care of verifying that those information are complete and translating them into any notation in the background, if required.

Specification of Test Patterns with Test Script Notations and Extensions

Test scripting notations are the equivalent to programming languages in test engineering. Therefore, in the same way that programming languages can be extended to support the specification of design patterns, it has been suggested that test scripting notations could also be extended to specify test design patterns. The TTCN-3 notation has been chosen as a candidate for that purpose, illustrated through idioms added to that language to specify test patterns.

TTCN-3 provides some concepts for test patterns such as the import mechanism, value parameterization and modifiable templates. Object-based concepts providing further means for the specification and application of test patterns do not exist, but are currently discussed with regard to their inclusion into TTCN-3. It is expected that with the new features of the language, support for test pattern specification should be improved. In the meantime specific annotations to TTCN-3 are used, in order differentiate the generic parts and specific parts of a test pattern. The generic parts are annotated with <> and are to be replaced when applying the test pattern. The specific parts are not annotated. They constitute the essence of the test pattern and should remain untouched when applying that pattern. These annotations are used to illustrate some of the test patterns in this work.

The specification of test patterns in TTCN-3 (and extensions thereof) provides several benefits. TTCN-3 test patterns

- are expressed formally.
- provide means for patterns in all phases of test system development and for the different approaches of test reuse.
- are defined already in the language of the target test suite.

However, using TTCN-3 or extensions thereof to specify test patterns also comes with some drawbacks that need to be seriously considered:

- Inability to express test patterns for certain aspects of test engineering (e.g. test objectives, test procedures)
- Difficulty to express concepts at higher level of abstraction.
- Tool support for new extensions. Validation of test patterns might be difficult to achieve, because patterns are generic, whereas syntax/semantics checkers require complete code.
- No visualisation, which leads to lower understandability of the specified test patterns.
- Difficulty to translate the test patterns into other test scripting notations and vice-versa.

Specification of Test Design Patterns with a Generative Approach by using Meta-Modelling

Alternatively to scripting extensions and design environment support, test patterns can also be specified using meta-modelling facilities, based on modelling languages such as UML and its extensions e.g. UTP, SysML, etc. The approach consists in defining an ontology of test patterns, i.e. a formal description of concepts embodying those patterns and of the relationships between them. Using a meta-model to specify that ontology defines a domain-specific modelling language (DSML) especially tailored for designing new test solutions as models instantiating the test patterns supported by the meta-model. This approach has several benefits:

- High level of abstraction: The test patterns concepts can be expressed at a high level of abstraction. This makes the approach independent of any lower-level test scripting notation. By that, its expressiveness and potential integration in existing test infrastructures remain unrestricted.
- Integration to existing MDE infrastructures: This enables access to elements of the SUT's model, if those have been specified in the same environment or using the same notation (e.g. UML). Additionally, the facilities provided by those environments can be used to generate tools for the DSML and enhance their functionalities towards improved usability. Such facilities include
  - better visualisation possibilities.
  - improved transformation capabilities.

  – built-in model checking and validation facilities.

However, despite the numerous benefits it bears, this approach also comes with some drawbacks that need to considered. Those include:

- Additional costs and risks of designing, implementing and disseminating a new notation: Although current existing MDE infrastructures (e.g. the Eclipse [148] IDE's Modelling Framework EMF [147]) have contributed to significantly reduce the risk and efforts associated with the introduction of a DSML, organisations should be well aware of those and analyse carefully, before opting for or against that approach. However it should be pointed out that the effort of introducing such a meta-model based DSML might be lower than that of introducing a new programming language, because the concepts defined by the meta-model play a more critical role than the syntax of the representation format used for the notation. This is a totally different situation from notations specified using (E)BNF, for which, additionally to the semantical concepts, the syntax of the language also needs to be learned.

- Difficulty in translating abstract concepts into concrete executable test scripts: patterns and high-level models are generic per se. Therefore solutions based on them require a certain amount of customisation to be completed, in such a way that automation towards generation of executable test scripts or valid code snippets thereof would be possible.

- Difficulty in ensuring bi-directional traceability between abstract test models and generated test scripting source resulting from model-transformation: If an appropriate solution for this issueis not found , then there is a risk of the code added manually to complete the generated source code, being overwritten whenever new source code is generated from the test model.

4.4 The Pattern Oriented Model Driven Test Engineering Process

Pattern-oriented model driven test engineering is a process whereby test patterns are used to design a test model that is then transformed and refined into executable test cases, following a model-driven engineering approach. Figure 4.2 depicts a representation of that process in the form of a Business Process Modelling Notation (BPMN) diagram. The usage of test patterns may be explicit or implicit in pattern-oriented MDTE, depending on whether the person performing the test modelling activity is made aware of the patterns being applied (explicit) or not (implicit). Explicit usage of patterns is achieved through model design templates, based on which new model elements or skeletons thereof can be created automatically, before they are completed manually. For convenience,

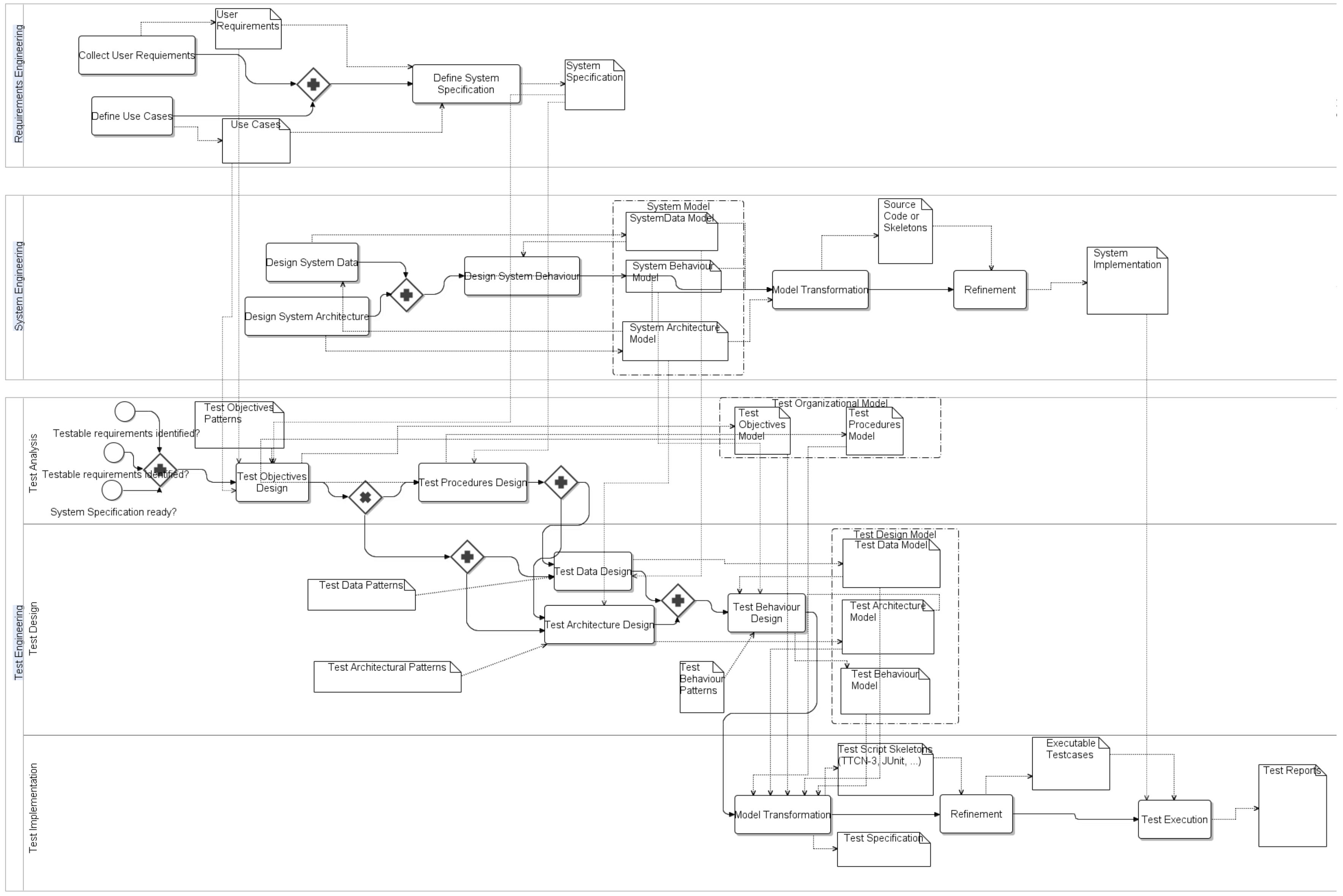


Figure 4.2: BPMN Diagram of the Pattern-Oriented MDTE Process

the process of providing the missing elements to fill out the design template may be accompanied with tool support in the form of so-called wizards. On the other hand, implicit usage of patterns is achieved through the enforcement of constraints and policies that guide the test modelling process. In which case, the test designer may not even be aware of the fact that he or she would be applying a given test pattern in the process. The top-most pools of the BPMN diagram in Figure 4.2 display the other processes that are related to the test engineering process, namely requirements engineering and system engineering, while the pool at the bottom displays the test engineering process. That process is subdivided in three lanes, each of which contains a sub-process dedicated to a phase of test engineering:

4.4.1 Test Analysis

The test analysis sub-process is triggered by requirements engineering or by high-level system design. It takes user requirements, use cases or a complete system specification as input and combines those with test objective patterns to produce a test objectives model. After the test objectives model is designed, the process continues either with the design of a test procedures model or moves to the next sub-process, i.e. test design.

4.4.2 Test Design

The test design sub-process starts with test data or test architecture design, both of which may run in parallel. Test data design takes the SUT's data model as input and combines it with test data patterns to produce a test data model. In a similar manner, the test architecture design process takes the SUT's architecture model and the test data model as inputs, combining them with test architectural patterns to produce the test architecture model. Then, the process continues with test behaviour design which, based on the others test models (objectives, procedures, data and architecture), uses test behaviour patterns to produce a test behaviour model.

4.4.3 Test Implementation

The test implementation sub-process takes the test design model resulting from the previous phases as input and transforms it into executable test scripts in a notation suitable for the target test environment. Depending on the level of details provided in the test design model, the generated test scripts will require more or less manual refinement to be complete. After those refinements, they can then be executed against the implemented SUT resulting from the system engineering process to produce the test reports.

4.5 A Collection of Test Design Patterns

A collection of test design patterns identified in various testing projects during this work is provided in Appendix A. The patterns are organised along the classification described in Section 4.2.

Table 4.1 presents an overview of those test design patterns and the page in which their description is located.

Category	Test Design Pattern	Page
Generic	Separation of Test Design Concerns	259
	Grouping of Test Design Concerns	261
Test Objectives	Prioritization of Test Objectives	262
	Traceability of Requirements to Test Artifacts	264
	Selection Criteria for Test Objectives	265
	Traceability of Test Objectives to Fault Management	266
Test Architecture	Extensibility/Restriction of Test Architecture	267
	One-on-One Test Architecture	264
	Flexibility of the Test Architecture Model	270
	Proxy Test Component	271
	Monitoring Test Component	273
	Central Test Coordinator	274
Test Data	Purpose-Driven Test Data Design	276
	Basic Static Test Data Pool	277
	Reusable Test Data Definitions	278
	Dynamic Test Data Pool	279
Test Behaviour	Assertion-Driven Test Behaviour Design	280
	Context-Aware Test Behaviour Design	281
	Test Component Factory	283
	Central Coordination of Test Components	284
	Distributed Coordination of Test Components	284
	Time Constraints on Test Behaviour	285

Table 4.1: Overview of Test Design Patterns Described in this Work

4.6 Summary

This chapter has described the concept of pattern oriented MDT, which combines a model-driven engineering approach with test design patterns for more efficiency in test automation. After defining the key concepts of the approach at the beginning of this chapter, a classification of test design patternshas been provided. Then, finally the methodology for applying the approach has been presented, together with the process it implies.

Obviously the pattern oriented MDT approach can help addressing some of the key issues currently faced with in test automation; Especially if the concepts contained in those patterns are expressed in such a way, that they can be processed automatically to optimize the test development process through automated code generation, model validation, etc. However, two additional elements are required for that to be possible: A suitable test design approach supporting the expression of test pattern instantiations and a mean for cataloguing identified test patterns.

In section 4.3.3, the existing possibilities for specifying test patterns have been described, including a discussion on the pros and cons of each approach. Eventually, a DSML based approach has been chosen to facilitate the usage of design patterns in test automation. In the next section, the concepts of that DSML - called UTML - are described, along with the relationships between them.

Chapter 5

UTML: A Notation for Pattern Oriented Model Driven Test Design

5.1 The Need to Formalise Test Patterns

One of the most important benefits expected when applying patterns in any domain is a facilitated instantiation of new solutions to recurrent problems through reuse of concepts. Such reuse at the conceptual level does not only have a positive impact on the key software quality factors of the resulting products, but also on the overall development process within the organisation. This leasds logically to higher productivity, shortened development cycles and last but not least, reduced costs.

As an activity in which expert knowledge plays an important role, test automation could benefit a lot from cataloguing that knowledge as patterns and providing tool support to facilitate reuse. This, however demands that the concepts described by the patterns are expressed in a formal and unambiguous manner to avoid confusion and facilitate automated processing. Furthermore, appropriate tool support is required to guide test designers in the process of applying patterns efficiently, because manual application of patterns is known to be tedious and error-prone [125].

Several efforts to formalise patterns have already been undertaken in the past, with the same motivations as mentioned above. Baroni et al [12] present an overview of approaches in formalising design patterns. A wide range of proposals are described, ranging from extensions to the classical object model [19] to new formalisms, extending existing object-oriented programming languages

(OOPLs) [78] or expressed as new text-based notations or as UML-based DSMLs [2] [106].

Test design aims at addressing the specific purpose of modelling test artifacts. Therefore it has a narrower scope than generic software design. Hence, the choice of an approach for formalising test patterns appears to be less difficult than in the case of generic software patterns. The main criteria in selecting an approach for describing test design patterns in this thesis were as follows:

1. Non-Dependency of any particular testing infrastructure.
2. Reuse or extension of existing notations and well-established concepts.
3. Support of graphical representation.
4. Integrability in the overall software model-driven engineering process.

Given the above criteria, a DSML-based approach was chosen, as it allows to define concepts at a level of abstraction that is high enough to be kept independent of any specific testing infrastructure, while at the same time providing all the mechanisms for defining the concepts precisely and unambiguously.

As discussed in Section 3.3.1, the approach for designing that DSML also needed to be selected. Considering criteria 2 from the list above, taking the UML notation, which is the lingua franca of software design as a basis appeared to be inevitable. Therefore it was essentially a matter of which of the standard-conformant extension mechanisms provided by that notation would be suitable. The choice was between a UML profile-based approach (light-weight UML extension) e.g. reusing the UTP, a (heavy-weight) extension to the UML Metamodel itself, and a new stand-alone metamodel with a specifically dedicated to the purpose of pattern-oriented test design.

Weisemöller et al. [161] provide a comparison of UML standard compliant ways of defining DSMLs and came up with the result displayed in Table 5.1 which clearly indicates that an approach based on a domain-specific metamodel for test design is an interesting option to consider. Despite the additional effort it implied with regard to tool support and the definition of the metamodel itself, the metamodel approach was chosen, based on the analysis made in this work, which confirmed the results displayed in Table 5.1. The metamodel was designed in such a way that it embodies concepts of test design patterns while reusing as much as possible the test design concepts introduced by the UTP. The defined DSML was called Unified Test Modelling Language (UTML).

5.2 Overview of UTML

As mentioned previously, the UTML notation reuses and extends concepts of the UTP into a stand-alone DSML specifically dedicated to model-driven test design. Table 5.2 presents an overview of UTP concepts and their equivalent in UTML

	UML profile	**UML MM-extension**	**New Meta-model**
Expressive power	-	+	+
Flexibility	-	o	+
Clarity of semantics	-	+	+
Simplicity of constraints	-	o	+
Model notation	-	-	+
Tool support	+	-	-

Table 5.1: Overview of approaches to specify DSMLs [161]

where applicable. Additionally, some comments on the motivations for adopting or leaving out the element are also provided.

Table 5.2: A Comparison of UTML and UTP

UTP Concept	**UTML Equivalent**	**Comments**
	Test Architecture Concepts	
Arbiter	-	Test behaviour in UTML is designed following the *Assertion-Driven Test Design* design pattern defined in Section A.5.1. Therefore, the test verdict is either implicitly or explicitly specified through the test behaviour and the *StopAction*(see Table 5.29) respectively. This makes the usage of an extra arbiter obsolete.

Scheduler	-	Test execution and the mechanisms for instantiating test components and controlling their lifecycle is out of scope for the UTML language, as those aspects can hardly be expressed at such a high level of abstraction. If a scenario for controlling the way test cases will be executed is required, this can be designed using UTML test activity diagrams.
SUT	*ComponentKind* property used for component instances	see Table 5.24
TestContext	*TestArchitecture*	see Table 5.31.
TestComponent	*Component-Instance* with *kind* property set to *TEST_-COMPONENT*.	see Table 5.29.
Test Behaviour Concepts		
Verdict	*Verdict*	see Table 5.68.
Default	*DefaultBeha-viourDef*	See Table 5.82.
FinishAction	*StopAction*	See Table 5.88.
TestLog	-	Logging is considered to be a functionality of the test execution environment that is inherently platform specific. Therefore, it is not viewed as essential for test design at this level of abstraction.
TestLogAppli-cation	-	See comments for *TestLog* element.
LogAction	-	See comments for *TestLog* element.
DefaultAppli-cation	*ActivateDefault-Action*	See Table 5.115

determAlt	*AltBehaviour-Action*	See Table 5.113.
TestCase	*Testcase*	See Table 5.74.
TestObjective	*TestObjective*	The UTML extends the syntax and the semantics of the TestObjective element, as defined by the UTP.
ValidationAction	*CheckAction* and extensions thereof (e.g. *ValueCheck-Action*, *External-CheckAction*).	See Table 5.106 and Table 5.105
Test Data Concepts		
Wildcards	Constraints on test data instances	See Table 5.54
Data partition	Abstract test data instances	See Table 5.54
Data pool	-	Rather than defining data pools, the UTML defines abstract data instances that may be mapped to data generators or data pools specific to a given test platform.
Data selector	-	The *data selector* concept is associated with the UTP's *data pool* concept. Given that the approach for designing test data with UTML follows a different strategy, this concept is also not supported.
Coding rules	-	In UTML coding rules are defined via the *coding_rules* property of a *TestDataType* element (see Section 5.7.13).
LiteralAny	-	See comments on *Wilcards* elements
LiteralAnyOrNull	-	See comments on *Wilcards* elements
Time Concepts		

TimeZone	-	Timezone are considered to be a platform-specific feature in UTML.
GetTimeZone-Action	-	Timezone and associated actions are considered to be a platform-specific feature in UTML.
SetTimeZone-Action	-	Timezone and associated actions are considered to be a platform-specific feature in UTML
Duration	-	UTML defines no specific concept for designing duration. Timing constraints can be defined on test actions and events to specify that those should be taken in account.
Time	-	In UTP *Time* is a predefined primitive type used to specify concrete time values. Although UTML does not define any such concept, an equivalent type can be created through the *BasicTestDataType* element as part of a library of primitive type definitions, if required.
TimeOut	*TimerExpiration-Event*	see Table 5.81
TimeOutMessage	-	As previously mentioned, in UTML, timers are simple declarative elements without any semantics. Therefore, they may not send messages to other entities as part of test behaviour.

TimeOutAction	-	In UTP the *TimeOutAction* element models an action to occur, after a given timer has expired. Given that, any test action is a potential *TimeOutAction*, this class cannot be instantiated meaningfully. Therefore, the UTML proposes the *WaitAction*(see Table 5.87) through which the timer's expiration could be awaited, before the following actions in the test sequence are executed.
Timer	*Timer*	Timers in UTML are purely declarative and do not bear any semantics in themselves, but only in combination with behaviour elements.
StartTimer-Action	*StartTimer-Action*(see Table 5.85)	Timers are started implicitly in UTML, depending on the context in which they are referenced. However in certain situations the *StartTimerAction* may be used to start a timer explicitly.
StopTimerAction	*StopTimer-Action*(see Table 5.86)	The semantics of the StopTimer-Action element is the same in UTML as defined in UTP.
ReadTimer-Action	-	The action of reading a timer is always implicitly associated to a specific test behaviour in UTML and is generally not required to be invoked explicitly. At least, at the time being, the need for explicitly reading a timer's value has not yet been identified.
TimerRunning-Action	-	The same comments made for the *ReadTimerAction* element also apply for the *Timer-RunningAction* element.

Beyond concepts adopted from the UTP, design of test automation in UTML is based on a series of principles of abstraction which guide the whole process to ensure that the resulting model remains concise while at the same time, being as complete as possible.

Prenninger et al. [126] identify four categories of such principles of abstraction, namely *functional*, *data*, *communication*, and *temporal*. The same categories have been used for UTML. However, the abstraction approach used in each of those categories shows some differences with those described in that work and will be described further in the next section.

Taking into account those principles of abstraction, the concepts of the UTML notation follows can be grouped in six main categories:

- Generic UTML concepts application field span over all phases of the test development process.
- Test planing modelling concepts specify the means for organising test plans in such a way that they can be integrated to the other phases of test development.
- Test procedures modelling concepts help in documenting test procedures and ensuring that those documentations follow specific patterns and guidelines.
- Test data design concepts define the means for specifying data used in test scenarios.
- Test architecture design concepts
- Test behaviour design concepts build on test data and test architecture for system, i.e. test data, test architecture and test behaviour.

Figure 5.1 depicts a UML class diagram displaying the hierarchy of UTML test models and illustrating the structure of the language. As depicted in that figure, the UTML metamodel defines five different views on the test model, each of them dealing with a specific aspect of test design and extending the abstract ***BasicTestModel*** element. Also depicted in that figure are the relationships between the categories of test models, which participate in defining a clear process for test design.

5.2.1 Visualisation

Test models are essential instruments of communication between all stakeholders involved in the software business process. Therefore, they need to be understood by technical (testers, designers, developers) and less technical staff (sales, support, managers) in their interactions before, during and after the development phase. At the same time, test models must meet certain requirements, so that they can

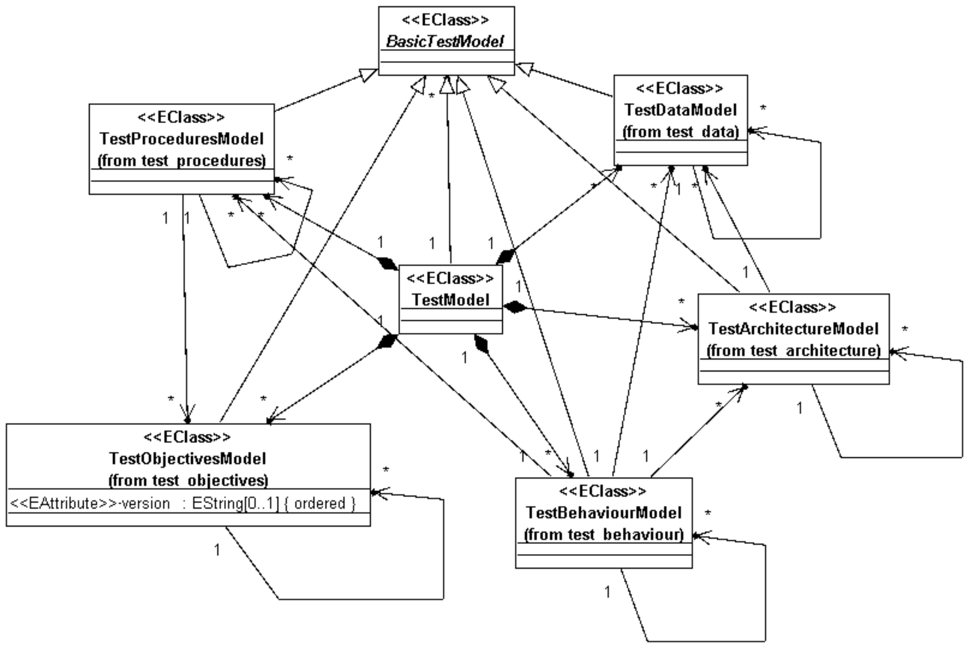


Figure 5.1: Overview of UTML Test Models

be exploited for automatic transformation in a model-driven testing approach to reduce the test development lifecycle.

It is well-known that graphics are the most appropriate way of sharing technical information. As Tufte states in his book *The Visual Display of Quantitative Information* [156]:

> At their best, graphics are instruments for reasoning about quantitative information. Often the most effective way to describe, explore, and summarise a set of numbers even a very large set is to look at pictures of those numbers. Furthermore, of all methods for analysing and communicating statistical information, well designed data graphics are usually the simplest and at the same time the most powerful.

Although Tufte is referring here to statistical information, this statement also holds true for nearly any type of information. This might explain why visualisation is such a key aspect of every modelling approach.

The need for visual notations has long been acknowledged in the testing community. This is illustrated by the important amount of research in that area, starting with works on the usage of Message Sequence Charts (MSCs) (including variants thereof) [66, 140, 65] and of the Specification and Description Language (SDL), via the graphical presentation format of the TTCN-3 notation [144](GFT), through to the UML and extensions thereof e.g. the UML Testing profile or notations based on the same concepts [120]. However, most of those approaches either try to use a generic modelling language (UML, SDL, MSCs) for test modelling or they provide a graphical representation of concepts originating from a textual test notation without raising the abstraction level(GFT). In the first case, the notation used was designed with product design as its main purpose and does not address specific testing concerns, while in the second case the one-to-one mapping of concepts from the textual to the graphical format makes the modelling process less efficient.

Besides being specifically design for modelling test solutions, the UTML notation defines testing concepts at a higher level of abstraction to facilitate their mapping to graphical elements. In fact the concepts of the UTML notation were tailored so as to facilitate the definition of visual elements to illustrate them and to allow a more natural process of test design.

Diagrams

UTML models are expressed in the form of diagrams in which elements of the test model can be added and modified graphically. For each of the seven types of UTML test models, a UTML diagram type is defined to visually represent the concepts supported by that model. Those seven diagram types are:

- Test model diagrams: Test model diagrams visualise instances of the UTML *TestDataModel* (Cf. Table 5.37). Their main purpose is to provide an overview of the structure of test model by displaying other models contained in the *TestDataModel* instance.

- Test objectives diagrams: Test objectives diagrams define visual elements for test objectives model instances, as defined by the *TestObjectivesModel* (Cf. Table 5.11) element of the UTML metamodel.

- Test procedures diagrams: Test procedures diagrams provide a graphical view on the content of a test procedures model, as defined by the *Test-ProceduresModel* (Cf. Table 5.17) element of the UTML metamodel.

- Test architecture types diagrams: Test architecture types diagrams visualise elements of a *TestArchitectureTypesModel*, i.e. type definitions to be used in a test architecture.

- Test architecture diagrams: Test architecture diagrams allow the graphical representation of test architecture models (Cf. Table 5.25). They provide a structural view on the topology of the test system, depicting groups, architectures, components, ports, etc.

- Test data diagrams: Test data diagrams can be used to visualise elements of a UTML test data model(Cf. Table 5.37). Therefore, they provide a view on the structure and content of the test data model.

- Test behaviour diagrams: The UTML notation defines two types of diagrams for designing test behaviour: test sequence diagrams and test activity diagrams. Test sequence diagrams are based on UML 1.4 sequence diagrams, which they modify with some specific semantics to allow the design of test scenarios. On the other hand, test activity diagrams are similar to UML activity diagrams and can be used to design composition of test behaviours involving several test scenarios modelled as test sequence diagrams. This approach was already chosen for the *TeLa* notation [120, 119, 121] which follows similar goals as UTML, to address the shortcomings of UML with regard to test design. However, the UTML approach is less formal and primarily based on well-established patterns and good practices in test automation design. Therefore the modifications proposed in this thesis are different from those proposed by the authors of TeLa. This was motivated by the fact the main concern here was to provide means for supporting model-driven test engineering by test designers and test developers, rather than to generate test sequences automatically from existing system design models.

Generic Visualisation Concepts

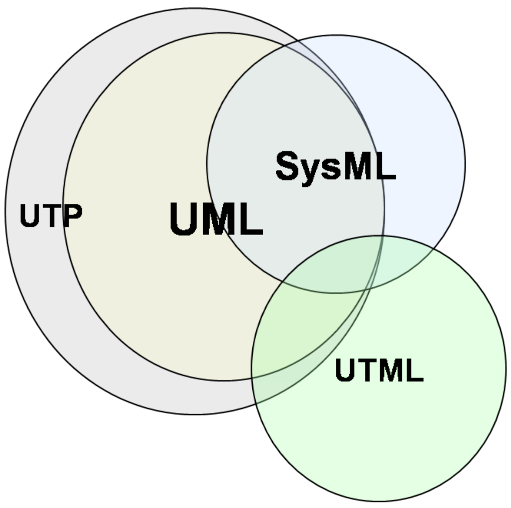


Figure 5.2: The UTML notation and its relation to UML and SysML

While designing the graphical elements for the UTML notation, special care was taken to reuse visual elements from existing well-known notations such as the UML and the SysML. Figure 5.2 illustrates the picture that emerges as a result of that effort. As depicted in that figure, the UTML notation uses as much as possible visual concepts introduced by SysML, while at the same time inheriting some of the concepts SysML adopted from the UML. However, as also visible from that figure, some additional visual elements had to be provided to express concepts specific to UTML for which satisfactory symbols were not available in the two other notations.

Some of the visualisation concepts defined for graphical test modelling apply to all types of UTML test models. The next sections list those concepts and the elements for which they have been used.

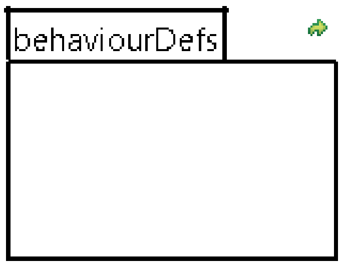


Figure 5.3: The *Package* Visual Element

The *Package* Visual Element The *Package* visual element, depicted on figure 5.3, is adopted from UML class diagrams and is used to display instances of the UTML metamodel that are containers for other elements. Examples of such containing UTML model elements include:

- Test Models (e.g. elements of meta-classes *TestArchitectureModel*, *TestDataModel*, *TestObjectivesModel*, etc.
- Group definitions (e.g. elements of meta-classes *TestArchitectureGroupDef*, *TestDataGroupDef*, *TestObjectivesGroupDef*, etc.)

The *Class* Visual Element The *Class* visual element, depicted on figure 5.4, is adopted from UML class diagrams and is used to display instances of the UTML metamodel that can be assimilated to classes or objects in the OO-programming sense (i.e. instances of leaves in the UTML metamodel). Examples of such UTML model elements include:

- Type definitions (e.g. elements of meta-classes *MessageTestDataType*), *OperationTestDataType*, *SignalTestDataType*, etc.
- Instances definitions (e.g. elements of meta-classes *TestDataInstance*, *TestObjective*, *TestProcedure*, etc.)

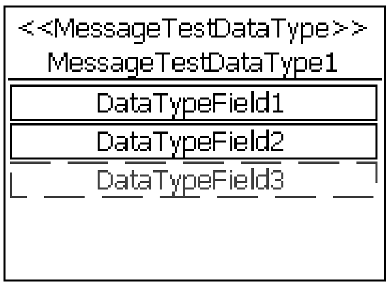


Figure 5.4: The *Class* Visual Element

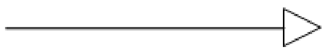


Figure 5.5: The *Generalisation* Visual Element

The *Generalisation* Visual Element The *Generalisation* visual element, depicted on figure 5.5, is adopted from UML class diagrams and is used to display generalisation relationships between model elements.

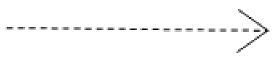


Figure 5.6: The *Dependency* Visual Element

The *Dependency* Visual Element The *Dependency* visual element, depicted on figure 5.6, is adopted from UML class diagrams and is used to display dependency links between model elements.

5.3 Generic UTML Metamodel concepts

Generic concepts of the UTML metamodel are those that provide a common base for other elements of the metamodel and therefore cannot be classified as belonging to any particular type of test model.

5.3.1 UtmlElement

Description

The ***UtmlElement*** element is an abstract classifier defined as the base classifier for all other classifiers in the UTML metamodel. It carries no particular semantic information.

5.3.2 BasicTestModel

Description

The ***BasicTestModel*** element is the abstract base classifier for all UTML test models.

Semantics

The ***BasicTestModel*** element is no specific semantics, besides being an abstract container for UTML model elements at the highest level.

Syntax

The ***BasicTestModel*** element extends the following elements of the metamodel:

- *UniqueNamedElement* (Cf. Table 5.7)
- *UtmlElement* (Cf. Section 5.3.1)
- *DescribedElement* (Cf. Table 5.4)

5.3.3 TestModel

Description

The ***TestModel*** UTML element defines a root test model. Root test models describe the static structure of a composite test model consisting of a number of test models of various kinds (e.g. test objective models, tests data models, test behaviour models, etc.). Figure 5.1 illustrates the relationship of the ***TestModel*** element with those other kinds of test models.

Semantics

Beyond their role as containers for the various other types of test models, ***Test-Model*** elements have no specific semantics.

Syntax

As described in Section 5.2.1, just like other UTML models, ***TestModel*** elements are represented graphically with the *Package* visual element. Each of the contained *Package* elements in a root test model diagram can be linked to another diagram visualising the content of the associated UTML test model.

Additionally, import relationships between UTML test models can be expressed using the *Dependency* visual element known from UML class diagrams.

The ***TestModel*** element extends the *BasicTestModel* element described in Table 5.3.2.

Table 5.3: Properties of the TestModel UTML element

Property	**Description**	**Type**	**Occurence**
testObjectivesModel	Test objectives models contained in this composite test model.	*TestObjectivesModel* (Cf. Table 5.11)	0..n
importedModel	References to other composite test models linked to this test model.	*TestModel* (Cf. Table 5.3)	0..n
testArchitectureModel	Test architecture models contained in this composite test model.	*TestArchitectureModel* (Cf. Table 5.25)	0..n
testBehaviourModel	Test behaviour models contained in this composite test model.	*TestBehaviourModel* (Cf. Table 5.66)	0..n
testDataModel	Test data models contained in this composite test model.	*TestDataModel* (Cf. Table 5.37)	0..n
testArchitectureTypesModel	Test architecture type models contained in this composite test model.	*testArchitectureTypesModel* (Cf. Table 5.20)	0..n

5.3.4 DescribedElement

Description

The UTML ***DescribedElement*** element is an abstract classifier used in all categories of test models to introduce textual documentation for the extending UTML meta-class.

Constraints

For certain UTML model elements, depending on the defined modelling policies, the description associated with the ***DescribedElement*** element may be made compulsory by activating the following OCL constraint:

```
self.description.oclIsTypeOf(OclVoid) = false
and self.description <> 'TODO: Add description'
```

Syntax

Table 5.4: Properties of the DescribedElement UTML element

Property	**Description**	**Type**	**Occu-rence**
description	A free textual description of the UTML element.	xsd:string	0..1

5.3.5 GroupItem

Description

The ***GroupItem*** UTML element is an abstract classifier used to design the grouping mechanism for elements of the test model. UTML elements extending ***GroupItem*** can be added as children to a group definition. The ***GroupItem*** UTML element has no fields and no attributes.

Semantics

A ***GroupItem*** element can be contained in a group definition.

5.3.6 GroupDef

Description

The ***GroupDef*** UTML element represents a group definition within a generic test model. Table 5.5 lists the properties of each ***GroupDef***.

Semantics

The ***GroupDef*** element defines a structural container for other model elements.

Syntax

The ***GroupDef*** element extends the following elements of the metamodel:

- *ElementWithUniqueID* (Cf. Table 5.9)
- *DescribedElement* (Cf. Table 5.4)

- *GroupItem* (Cf. Section 5.3.5)

Table 5.5: Properties of the GroupDef UTML element

Property	**Description**	**Type**	**Occurence**
groupItem	children elements contained in the group definition.	*GroupItem* (Cf. Section 5.3.5)	0..n

5.3.7 NamedElement

Description

The ***NamedElement*** element is an abstract classifier used as the base for named UTML elements.

Syntax

Table 5.6: Properties of the NamedElement UTML element

Property	**Description**	**Type**	**Occurence**
name	Name of the element.	xsd:string	1..1

5.3.8 UniqueNamedElement

Description

The ***UniqueNamedElement*** element is also an abstract classifier and provides the same functionality as the ***NamedElement*** element, with the difference that the name used in this case must be unique for the whole test model.

Syntax

Table 5.7: Fields and attributes of the UniqueNamedElement UTML element

Property	**Description**	**Type**	**Occurence**
name	Name of the element.	xsd:string	1..1

5.3.9 ElementWithID

Description

The ***ElementWithID*** element is an abstract classifier used as the base for UTML elements for which an identifier is required.

Syntax

Table 5.8: Properties of the ElementWithID UTML element

Property	**Description**	**Type**	**Occu-rence**
id	The identifier for the element.	xsd:string	1..1

5.3.10 ElementWithUniqueID

Description

The ***ElementWithUniqueID*** element is also an abstract classifier and provides the same functionality as the ***ElementWithID*** element, with the difference that the identifier used in this case must be unique for the whole test model.

Syntax

Table 5.9: Properties of the ElementWithUniqueID UTML element

Property	**Description**	**Type**	**Occu-rence**
id	The unique identifier for the element.	xsd:string	1..1

5.3.11 TestPatternKind

Description

The ***TestPatternKind*** UTML element is an enumeration defining a classifier for the various types of test modelling patterns.

Syntax

Table 5.10: The TestPatternKind UTML element

Literal	**Description**

ARCHITECTURE	Indicates a test architectural pattern.
BEHAVIOUR	Indicates a test behavioural pattern.
DATA	Indicates a test data pattern.

5.4 Test Objectives Design Concepts

The UTML metamodel's test objectives design concepts define the toolset required for modelling test plans in a rigourous and systematical manner. In this section the elements of the metamodel dealing with test planing are described, along with their relationships with other elements of the UTML metamodel.

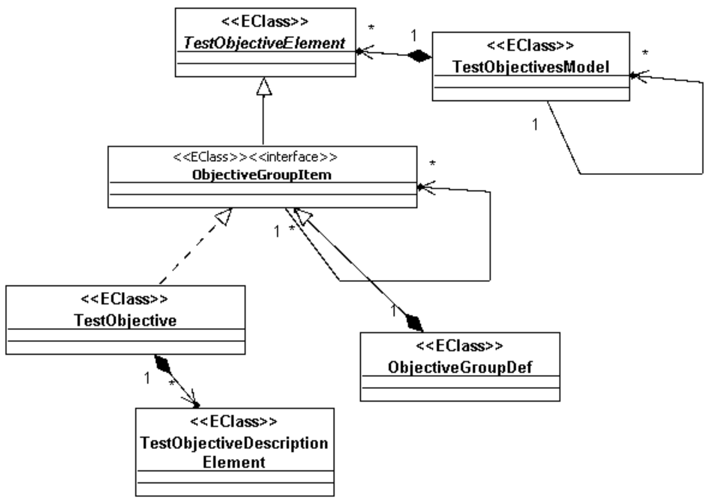


Figure 5.7: Class Diagram: UTML Metamodel for Test Objectives

Figure 5.7 displays a class diagram of the UTML metamodel for test objectives design.

5.4.1 TestObjectivesModel

Description

Syntax

Contained *TestObjectivesModel* elements and *TestObjectivesGroupDef* elements (defining groups of test objectives) are modelled using the aforementioned *Package* visual symbol, while leaves of the test objectives model (i.e. test objectives) are modelled using the *Class* visual element.

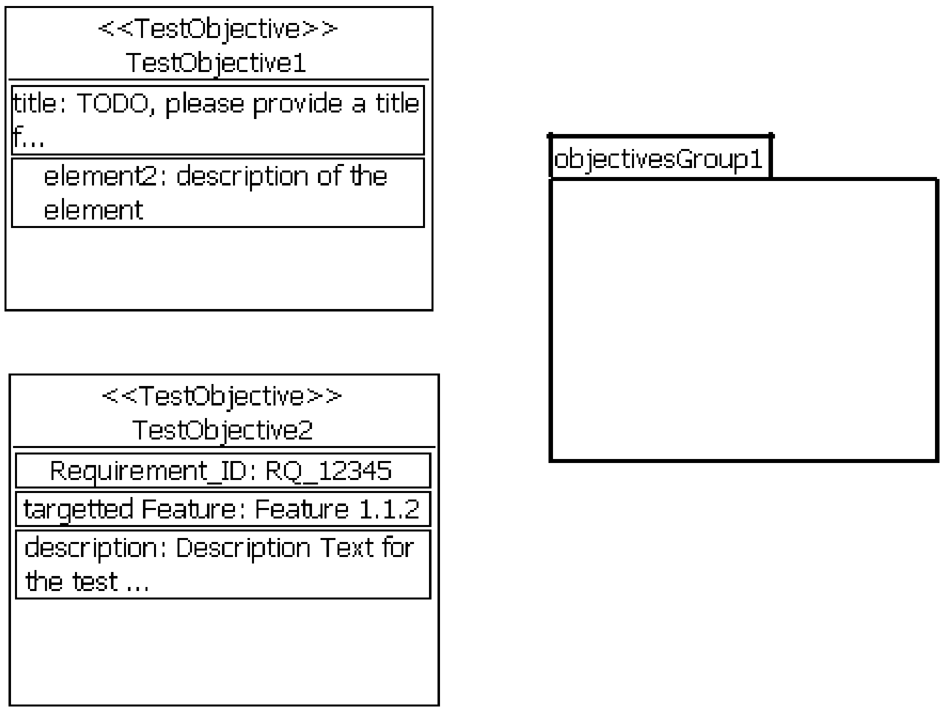

<<TestObjective>>
TestObjective1
title: TODO, please provide a title f...
element2: description of the element
objectivesGroup1
<<TestObjective>>
TestObjective2
Requirement_ID: RQ_12345
targetted Feature: Feature 1.1.2
description: Description Text for the test ...

Figure 5.8: Example UTML Test Objectives Diagram

Figure 5.8 displays an example UTML test objectives diagram containing one group of test objectives and two test objectives. The *TestObjectivesModel* element extends *BasicTestModel* (See Table 5.3.2)

Table 5.11: Properties of the TestObjectivesModel UTML element

Property	**Description**	**Type**	**Occurence**
test-Objective-Element	Test objective elements contained in the test specification.	*test-Objective-Element* (See Section 5.4.6)	0..n
test-Objectives-Model	References to other test objectives models linked to this test objective model.	*Test-Objectives-Model*	0..n

5.4.2 ObjectiveGroupDef

Description

The ***ObjectiveGroupDef*** element defines a group of objectives in a test specification model.

Syntax

The *ObjectiveGroupDef* element extends *DescribedElement* (See Table 5.4)

Table 5.12: Properties of the ObjectiveGroupDef UTML el-

imple-men-tation-Status	The current implementation status of the test objectives group.	*Imple-mentation-Status* (See Table 5.14)	1..1

5.4.3 ObjectiveGroupItem

Description

The ***ObjectiveGroupItem*** UTML element is an abstract class used to define a grouping mechanism for test objectives in a test specification model.

5.4.4 Priority

Description

The ***Priority*** element is an enumeration used for classifying the various priority levels for test objectives in a UTML test model. Based on those values, each test objective in the test plan is assigned a priority, which can guide decision taking in critical phases of the test project. The priority level of test objectives can be used as criterium to select which test objectives to implement first, if deadlines are approaching or resources scarce. Table 5.13 lists the pre-defined priority levels in the UTML metamodel.

Syntax

Table 5.13: The Priority UTML element

Priority Level
LOWEST
LOWER
LOW
NORMAL
HIGH
HIGHER
HIGHEST

5.4.5 ImplementationStatus

Description

The ***ImplementationStatus*** UTML element is an enumeration listing possible values for the implementation status of test objectives. Keeping track of the implementation status of test objectives can contribute significantly to improving productivity within a test project. In fact all other aspects of test modelling can benefit from those information, which can be used to filter test model elements for selection, edition or exporting (documentation, test script code, etc.). Table 5.14 lists the pre-defined implementation status values in the UTML metamodel, along with their meaning.

Semantics

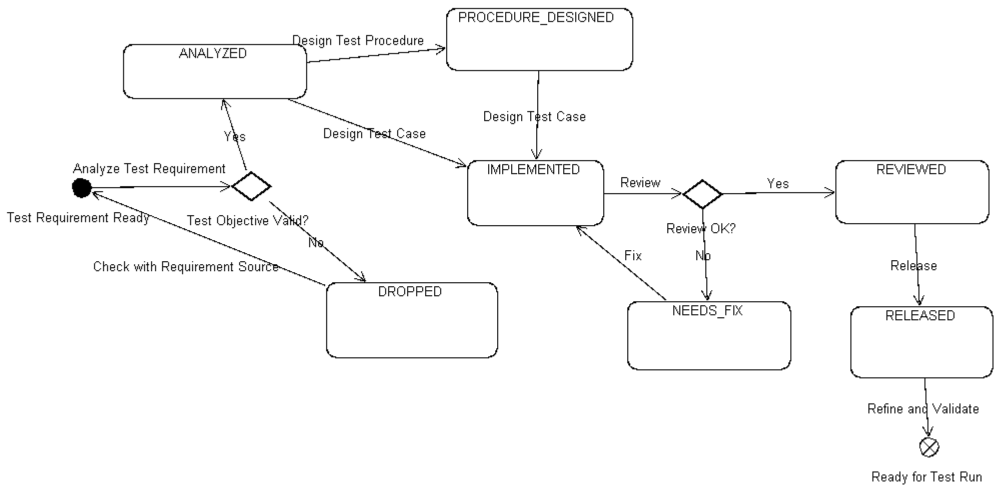


Figure 5.9: State Diagram: The Test Objective Lifecycle

Figure 5.9 displays a state diagram illustrating the usage of the *ImplementationStatus* to document the lifecycle of a test case from requirements analysis through to test execution. Each of the possible implementation status is represented as state in the diagram and the actions leading to state transition underline the semantics of this element.

Syntax

Table 5.14: The ImplementationStatus UTML element

Implementation Status	**Description**
ANALYZED	Indicates that the test objective has been analyzed and found valid based on the system specification (default).

PROCE-DURE_DE-SIGNED	Indicates that a test procedure has been designed for the test objective.
REVIEWED	A test case covering the test objective has been designed and reviewed.
IMPLEMENTED	Indicates that a test case has been designed covering the test objective.
REVIEWED	A test case covering the test objective has been designed and reviewed.
RELEASED	Indicates that a test case covering the test objective has been released.
NEEDS_FIX	Indicates that a bug has been discovered in the test case for this test objective. Therefore, the test case needs to be fixed.
DROPPED	Indicates the test objective has been dropped, e.g. because it has been found as not applicable or testable upon analysis.

5.4.6 TestObjectiveElement

Description

The ***TestObjectiveElement*** is an abstract class, modelling an element that can be added as child to a UTML test objectives model.

Syntax

The *TestObjectiveElement* element extends *UtmlElement* (See Section 5.3.1)

5.4.7 TestObjective

Description

TestObjective elements are the building entities of test objectives models (or test specifications). Their purpose is to document precisely and in a systematic manner what a test case will try to check on the SUT.

Semantics

The purpose of ***TestObjective*** elements is to enable a systematic approach to the test automation process by providing a common base for all parties, on which test cases will be designed and implemented. A ***TestObjective*** element represents a traceability link to phases in the software development process that preceded test design. A test objective element can be linked to one or several requirements.

Syntax

The *TestObjective* element extends the following elements of the metamodel:

- *ElementWithUniqueID* (See Table 5.9)
- *DescribedElement* (See Table 5.4)
- *TestObjectiveElement* (See Table 5.4.6)

Table 5.15: Fields and attributes of the TestObjective UTML element

Property	**Description**	**Type**	**Occurence**
objectiveDescElement	Description elements for the test objective.	*TestObjectiveDescriptionElement* (See Table 5.16)	1..n
testProcedure	A reference to test procedures that cover this test objective.	*TestProcedure* (See Table 5.19)	0..n
priority	The priority level assigned to the test objective.	*Priority* (See Table 5.13)	1..1
implementationStatus	The current implementation status of the test objective.	*ImplementationStatus* (See Table 5.14)	1..1
notes	Any additional notes to the test objective's description.	xsd:string	0..1

5.4.8 TestObjectiveDescriptionElement

Description

The ***TestObjectiveDescriptionElement*** element is an entity used to provide description of test objectives in a systematic manner. Each test objective description element is a name-value pair of free text. The *name* element provides the name of a description field for the test objective, while the value element provides the content of that description field.

Syntax

Table 5.16: Fields and attributes of the TestObjectiveDescriptionElement UTML element

Property	**Description**	**Type**	**Occurence**
name	Name of the description element.	xsd:string	1..1
value	Value of the description element.	xsd:string	1..1

5.5 Test Procedures Design Concepts

The UTML metamodel's test procedures concepts define the means for modelling test procedures for test objectives modelled in a test specification. A test procedure describes the sequence of steps that will have to be performed on the test system and the SUT to verify that a test objective is met satisfactorily. In this section the elements of the UTML metamodel's for test procedures are described, together with the relationships between them and other UTML elements.

Figure 5.10 provides an overview of the UTML metamodel for test procedures which illustrates the relationships between its components.

5.5.1 TestProceduresModel

Description

The ***TestProceduresModel*** UTML element is the root element for a test procedures model. Conceptually a test procedures model consists in a collection of test procedures, each of those covering at least one test objective.

Syntax

The *TestProceduresModel* element extends *BasicTestModel* (See Table 5.3.2)

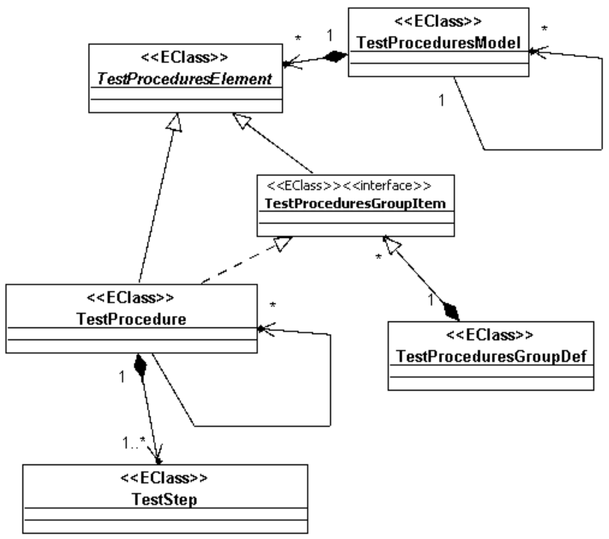


Figure 5.10: Class Diagram: UTML Metamodel for Test Procedures

Table 5.17: Properties of the TestProceduresModel UTML element

Property	Description	Type	Occurrence
testProcedures-Element	Test procedures or groups contained in the test procedures model.	*TestProcedures-Element* (See Section 5.5.3)	0..n
testObjectives-Model	References to test objectives models linked to this test procedures model.	*TestObjectivesModel* (See Table 5.11)	0..n
testProcedures-Model	References to test procedures models linked to this test procedures model.	*TestProcedures-Model* (See Table 5.17)	0..n

5.5.2 TestProceduresGroupItem

Description

The ***TestProceduresGroupItem*** UTML element is an abstract class used to provide a grouping mechanism for test procedures in a test procedures model.

Syntax

The *TestProceduresGroupItem* element extends *GroupItem* (See Table 5.3.5)

5.5.3 TestProceduresElement

Description

The ***TestProceduresElement*** is an abstract class that is the base for all other elements of the UTML test procedures model.

5.5.4 TestProceduresGroupDef

Description

The ***TestProceduresGroupDef*** UTML element defines a group within a test procedures model. A ***TestProceduresGroupDef*** can contain other ***TestProceduresGroupDef*** as subgroups or single test procedures.

Syntax

The *TestProceduresGroupDef* element extends the following elements of the metamodel:

- *DescribedElement* (See Table 5.4)
- *ElementWithUniqueID* (See Table 5.9)
- *TestProceduresGroupItem* (See Section 5.5.2)
- *TestProceduresElement* (See Section 5.5.3)

Table 5.18: Properties of the TestProceduresGroupDef UTML element

Property	**Description**	**Type**	**Occurence**
testProceduresGroupItem	children elements of the test procedures group.	*TestProceduresGroupItem* (See Section 5.5.2)	1..n

5.5.5 TestProcedure

Description

The ***TestProcedure*** element models a test procedure in the UTML metamodel.

Syntax

The *TestProcedure* element extends the following elements of the metamodel:

- *DescribedElement* (See Table 5.4)
- *UniqueNamedElement* (See Table 5.7)
- *TestProceduresGroupItem* (See Section 5.5.2)

- *TestProceduresElement* (See Section 5.5.3)

Table 5.19: Properties of the TestProcedure UTML element

Property	Description	Type	Occurence
testSteps	The test steps for this test procedure.	xsd:string	0..n
testcase	A reference to a testcase modelling the the test behaviour for this test procedure.	*Testcase* (See Table 5.74)	0..1
testObjective	References to the test objectives covered by this test procedure.	*TestObjective* (See Table 5.15)	0..n
subProcedure	References to other already defined test procedures which are parts of this test procedure.	*TestProcedure* (See Table 5.19)	0..n
remarks	Additional remarks concerning this test procedure.	xsd:string	0..1

5.6 Test Architecture Design Concepts

UTML test architecture concepts provide the means for designing test architectures following a pattern driven approach. Those concepts are based on the same principles of communication abstraction ([126]) used for the UML testing profile and the TTCN-3 notation.

UTML test architecture design concepts can be grouped in two main categories. The first group of concepts aims at defining type classifiers or descriptors for elements of test architectures, while the second group define concepts for designing instances based on the aforementioned descriptors.

Figure 5.11 displays the UML class diagram for the first group of UTML test architecture concepts.

Figure 5.12 depicts the UTML metamodel for the second group of test architecture concepts and illustrates the relationships between its composing elements.

5.6.1 TestArchitectureTypesModel

Description

A ***TestArchitectureTypesModel*** provides type definitions for instances in a test architecture model. Therefore, it is the basis for a test architecture model,

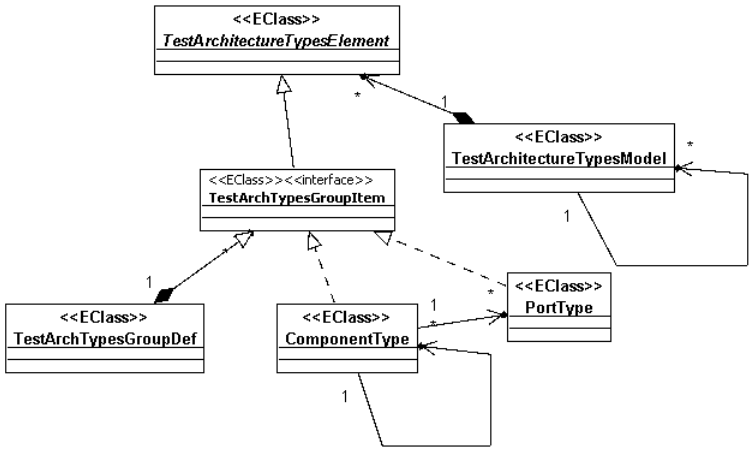


Figure 5.11: Class Diagram: UTML Metamodel for Type Definitions in Test Architectures

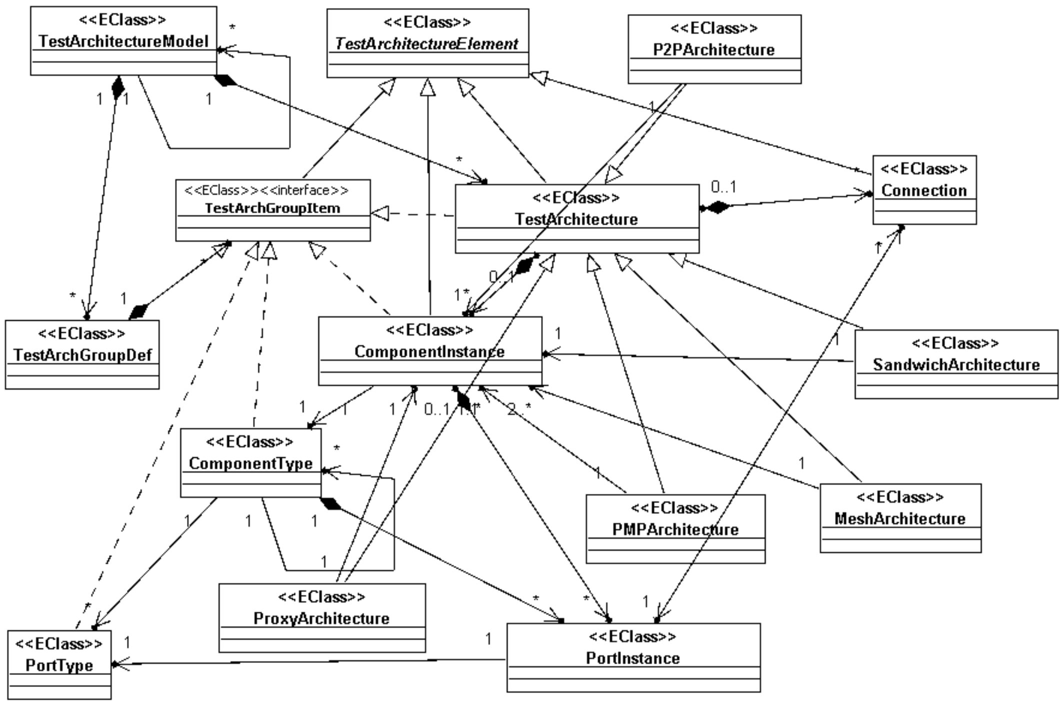


Figure 5.12: Class Diagram: UTML Metamodel for Test Architectures

which can be extended and reused in other test architecture models without affecting the existing test model artifacts.

The graphical modelling of test architecture type elements follows the same principles as those of other UTML models for which structure is the sole motivating factor. Accordingly, similar to those other diagram types, the *Package* and the *Class* graphical elements are used to model groups of test architecture type elements and instances of single elements (e.g. test component type definitions, port type definitions) respectively.

Syntax

The *TestArchitectureTypesModel* element extends *BasicTestModel* (See Table 5.3.2).

Table 5.20: Properties of the TestArchitectureTypesModel UTML element

Property	**Description**	**Type**	**Occurence**
testArchitectureTypesModel	References to other existing testArchitectureTypesModel elements.	*TestArchitectureTypesModel* (See Table 5.20)	0..n
testDataModel	References to data model documents from which test data definitions are used in this test architecture basic model.	*TestDataModel* (See Table 5.37)	
testArchitectureTypesElement	Test architecture elements contained in the model.	*TestArchitectureTypesElement* (See Section 5.6.2)	

5.6.2 TestArchitectureTypesElement

Description

The ***TestArchitectureTypesElement*** is an abstract class used as the base for UTML test architecture concepts that can be shared among several test architecture models.

Syntax

The *TestArchitectureTypesElement* element extends *UtmlElement* (See Section 5.3.1)

5.6.3 TestArchTypesGroupItem

Description

The ***TestArchTypesGroupItem*** element is an abstract class used to model the grouping mechanism in UTML test architectural basic models. Any object of a class extending ***TestArchTypesGroupItem*** can be added as a child to a group in a test architecture basic model.

Syntax

The *TestArchTypesGroupItem* element extends the following elements of the metamodel:

- *DescribedElement* (See Table 5.4)
- *TestArchitectureTypesElement* (See Section 5.6.2)

5.6.4 TestArchTypesGroupDef

Description

The ***TestArchTypesGroupDef*** element represents a group definition within a UTML model for test architecture type definitions.

Syntax

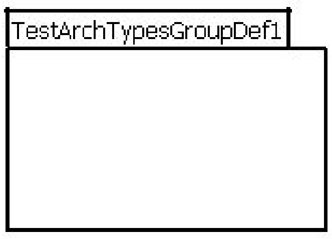


Figure 5.13: Example UTML Test Architecture Types Group

The *TestArchTypesGroupDef* element extends the following elements of the metamodel:

- *DescribedElement* (See Table 5.4)
- *ElementWithUniqueID* (See Table 5.9)

- *TestArchTypesGroupItem* (See Section 5.6.3)

Table 5.21: Properties of the TestArchTypesGroupDef UTML element

Property	**Description**	**Type**	**Occurence**
testArchTypesGroupItem	Basic test architecture model elements contained in the group.	*TestArchGroupItem* (See Section 5.6.9)	0..n

5.6.5 PortType

Description

The ***PortType*** provides a descriptor for a type of port.

Semantics

The main purpose of *PortType* is to define a classifier for modelling port instances. If a *PortInstance* element (see Section 5.6.12) is associated to a given *PortType* element through its ***type*** property, then the automatically inherits all the properties defined in the *PortType* element.

Syntax

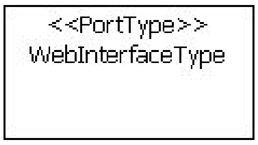


Figure 5.14: Example UTML Port Type

The *PortType* element extends the following elements of the metamodel:

- *DescribedElement* (See Table 5.4)
- *NamedElement* (See Table 5.6)
- *TestArchGroupItem* (See Section 5.6.9)
- *TestArchitectureTypesElement* (See Section 5.6.2)
- *TestArchitectureElement* (See Section 5.6.11)

Table 5.22: Properties of the PortType UTML element

Property	Description	Type	Occurrence
supported-Types	A list of test data types supported by the port type. If no type is indicated, it is assumed that port instances of this port type support all types of test data.	*TestData-Type* (See Table 5.46)	0..n

5.6.6 ComponentType

Description

The ***ComponentType*** element provides a descriptor for a type of test component to be used in test scenarios.

Semantics

The ***ComponentType*** element defines a mean for enabling the instantiation of components with a predefined set of properties. Component instances associated to a given *ComponentType* element through their ***type*** property automatically inherit all properties defined in that *ComponentType* element.

Syntax

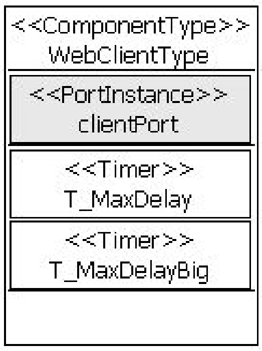


Figure 5.15: Example UTML Component Type

The *ComponentType* element extends the following elements of the meta-model:

- *DescribedElement* (See Table 5.4)
- *UniqueNamedElement* (See Table 5.7)

- *TestArchGroupItem* (See Section 5.6.9)
- *TestArchitectureElement* (See Section 5.6.11)

Table 5.23: Properties of the ComponentType UTML element

Property	**Description**	**Type**	**Occurence**
portType	A list of port types supported by components of this type.	*PortType* (See Table 5.22)	0..n
portInstance	Concrete port instances provided by the test component type. Every component instance having this component type as its ***type*** property implicitely inherits these port instances and may used them at any time for sending or receiving data in test actions.	*PortInstance* (See Table 5.28)	0..n
varDeclaration	Local variable declarations. These variables may be used by any test component instance having this ***ComponentType*** as *type* for storing and retrieving data while performing test behaviour.	*VariableDeclaration* (See Section 5.8.25)	0..n
timer	Local timers associated to the test component type. These timers must be contained in the component type element itself and not simply referenced.	*Timer* (See Table 5.83)	0..n
baseComponentType	Reference to component types that are extended by this component type. This field allows the specification of inheritance relationships between component types to facilitate reuse of existing elements in the test architecture model. If a component type B extends a component type A, then component type B inherits all ports, variables and timers declared in component type A	*ComponentType* (See Table 5.23)	0..n

5.6.7 ComponentKind

Description

The ***ComponentKind*** element is an enumeration listing the kinds of components possible in a UTML test architecture.

Syntax

Table 5.24: The ComponentKind UTML element

Component Kind	**Description**
TEST_COMPONENT	Used to indicate that the component is part of the test system
SUT	Used to indicate that the component is part of the System Under Test

5.6.8 TestArchitectureModel

Description

The ***TestArchitectureModel*** element is the root element for every UTML test architecture model document.

Syntax

The *TestArchitectureModel* element extends the BasicTestModel (See Table 5.3.2) element defined previously

Table 5.25: Properties of the TestArchitectureModel UTML element

Property	**Description**	**Type**	**Occurence**
testArchitectureModel	Links to other related test architecture model documents.	*TestArchitectureModel* (See Table 5.25)	0..n
testDataModel	Links to test data models.	*TestDataModel* (See Table 5.37)	0..n

testArch-GroupDef	Groups contained in this test architecture model.	*TestArch-GroupDef* (See Table 5.26)	0..n
testArchi-tecture	Contained test architectures.	*TestArchi-tecture* (See Table 5.31)	0..n

5.6.9 TestArchGroupItem

Description

The ***TestArchGroupItem*** element is an abstract class defining an element for a group in a test architecture model.

Syntax

The *TestArchGroupItem* element extends GroupItem (See Section 5.3.5)

5.6.10 TestArchGroupDef

Description

The ***TestArchGroupDef*** element represents a group definition within a UTML test architecture.

Syntax

The *TestArchGroupDef* element extends the following elements of the metamodel:

- *DescribedElement* (See Table 5.4)
- *ElementWithUniqueID* (See Table 5.9)
- *TestArchGroupItem* (See Section 5.6.9)

Table 5.26: Properties of the TestArchGroupDef UTML element

Property	**Description**	**Type**	**Occu-rence**
testArch-Group-Item	test architecture model elements contained in the group.	*TestArch-GroupItem* (See Section 5.6.9)	0..n

5.6.11 TestArchitectureElement

Description

The ***TestArchitectureElement*** element is an abstract class representing the base type for all other elements in the UT ML test architecture meta-model.

Syntax

The *TestArchitectureElement* element extends *UtmlElement* (See Section 5.3.1)

5.6.12 PortInstance

Description

A ***PortInstance*** element represents a communication point through which test components can exchange data with other test components or with SUT components.

Semantics

PortInstance elements are instantiations of *PortType* elements and can be used to model points of communication for components in a test architecture. A component instance may own one or more port instances. The type of data that can be exchanged via a given port instance is determined by the associated *PortType*'s ***supportedTypes*** property. This means, the communication paradigm(s) supported by the port instances depend(s) on the kinds of data types that can be exchanged through it. As described in Section 5.7.5, UTML defines three kinds of data types, namely ***Operation***, ***Message*** and ***Signal***.

The direction in which the port may be used to exchange data is defined by its ***direction*** property. It must also be ensured that the value of that property is also used to check if a connection can be created between two port instances or not. Table 5.27 provides a matrix for allowing/disallowing connections between port instances based on their directions.

	Target Port			
		INOUT	IN	OUT
Source Port	INOUT	Yes	Yes	Yes
	IN	Yes	No	Yes
	OUT	Yes	Yes	No

Table 5.27: Direction of Port Instances and Connection Support

Constraints

Constraint Port instances may only refer to model elements contained in the same test architecture.

```
(self.theConnection -> isEmpty() = false and self.theConnection ->
forAll(architecture.oclIsTypeOf(OclVoid)=false) and
self.theConnection ->
forAll(architecture.componentInstance -> notEmpty()))
implies
(self.theConnection.architecture.componentInstance -> exists(id =
self.theComponent.id))
```

Constraint A port instance's port type must be among the port types declared to be supported by the owning component's type definition element.

```
(self.theComponent.oclIsTypeOf(OclVoid) = false)
implies
(self.theComponent.type.portType -> exists(name = self.type.name))
```

Constraint Each port instances must have an associated port type.

```
 self.type.oclIsTypeOf(OclVoid) = false
```

Syntax

The *PortInstance* element extends the following elements of the metamodel:

- *DescribedElement* (See Table 5.4)
- *NamedElement* (See Table 5.6)

Table 5.28: Properties of the PortInstance UTML element

Property	**Description**	**Type**	**Occu-rence**
type	The port type for the port instance.	*PortType* (See Table 5.22)	1..1

the-Connection	Connections in which the port is involved, either as source or as target.	*Connection* (See Table 5.30)	0..n
the-Connection-Action	Connection actions in which the port instance is involved.	*ConnectionAction* (See Section 5.8.16)	0..n
direction	Direction in which communication through the port instance will occur.	*DataDirection* (See Table 5.39)	0..n
theComponent	The test component owning the port instance.	*ComponentInstance* (See Table 5.29)	1..1

5.6.13 ComponentInstance

Description

The ***ComponentInstance*** element represents an instance of a test component in a UTML test model. ***ComponentInstance*** elements are instantiations of *ComponentType* elements defined in Section 5.6.6.

Semantics

For more details on the semantics of component instances, see Section 5.8.1.

Constraints

Constraint Components belonging to the SUT must not be cloned.

```
(self.kind = utml::test_architecture::ComponentKind::SUT)
implies (self.clones = 0)
```

Constraint If a number of clones is provided for a component, then that number must be greater than or equal to zero. I.e. negative values are not allowed.

```
(self.kind = utml::test_architecture::ComponentKind::TEST_COMPONENT)
implies (self.clones >= 0)
```

Syntax

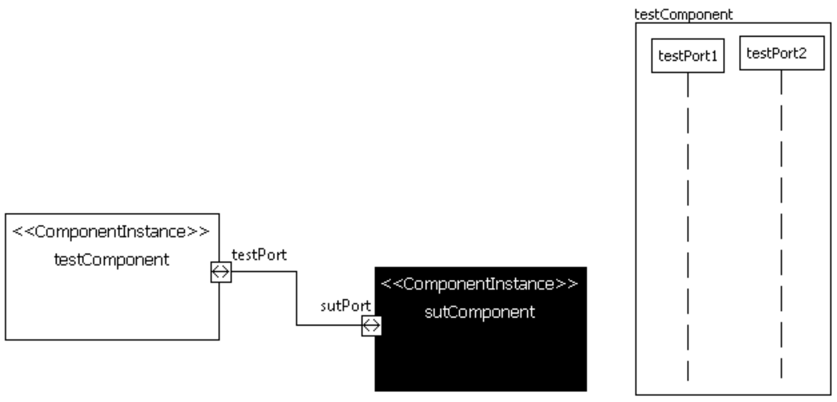


Figure 5.16: Component and Port Instances in UTML Diagrams

As depicted in Figure 5.6.13 components are visualised in UTML architecture diagrams through *Class* graphical elements similar to *block* elements in SysML. Each component carries the component's identifier as label. Components marked as being part of the SUT are colored in black color to underline the fact that they are considered black-boxes.

figure 5.6.13 also illustrates the visualisation of port instances belonging to components, as well as connections between those. For example, the solid line between *testPort* and *sutPort* in that figure indicates that those two ports are connected with each other in the containing architecture. The visualisation of ports owned by components is achieved using *flowport* symbols defined by the SysML notation. A flowport is represented graphically as a little box attached to the border of the owning component and containing an arrow that indicates in which direction the port may send or receive data. In the case of an *IN*-port or an *OUT*-port, the contained arrow will be directed inwards or outwards respectively, while for an *INOUT*-port a bidirectional arrow will be displayed.

Also, test behaviour naturally involves elements of the test architecture defined previously in the test architecture model. Therefore, the test behaviour sequence diagram defines graphical representation elements for those test architecture elements that might be used to model test behaviour. Those elements are component instances and port instances.

Figure 5.6.13 shows the graphical element for a component within a UTML test behaviour sequence diagram. As shown in that figure, test components are designed as a box potentially containing instance of port instances, represented by a *life line* graphical element.

The *ComponentInstance* element extends the following elements of the meta-model:

- *ElementWithID* (See Table 5.8)
- *TestArchitectureElement* (See Section 5.6.11)
- *TestArchGroupItem* (See Section 5.6.9)

Table 5.29: Properties of the ComponentInstance UTML element

Property	**Description**	**Type**	**Occurence**
type	A reference to the test component type which is instanciated by this component.	*ComponentType* (See Table 5.23)	1..1
architecture	The parent architecture to which this component instance belongs.	*TestArchitecture* (See Table 5.31)	1..1
port	Port instances owned by this component.	*PortInstance* (See Table 5.28)	0..n
kind	Indicate whether the component is marked as part of the SUT or as part of the test system (Default)	*ComponentKind* (See Table 5.24)	1..1
clones	Defines the number of clones to be created with this component. If the ***clones*** property is set to a value N_{clones}, then N_{clones} instances of ***type*** will be instantiated and started whenever this component instance element will be involved in test behaviour.	*Integer*	0..1

5.6.14 Connection

Description

The ***Connection*** element represents a connection, i.e. a data exchange channel between two ports in UTML.

Semantics

As described in Section 5.6.12, connections may only be created between ports if their direction allow it (See Table 5.27) and if they share supported data types as defined in the *supportedTypes* property or the port type referred to by their *type* property.

Constraints

Constraint Connections may only be created between port instances belonging to components within the same test architecture.

```
(self.sourcePort.oclIsTypeOf(OclVoid) = false and
self.destPort.oclIsTypeOf(OclVoid)) implies
(self.architecture.componentInstance -> exists(id =
self.sourcePort.theComponent.id) and
self.architecture.componentInstance -> exists(id =
self.destPort.theComponent.id))
```

Constraint A port instance must not be connected to itself.

```
self.sourcePort <> self.destPort
```

Constraint (Optional) A port instance should not be connected to another port instance belonging to the same component instance.

```
(self.sourcePort.oclIsTypeOf(OclVoid) = false
and self.destPort.oclIsTypeOf(OclVoid) = false)
implies
(self.sourcePort.theComponent <> self.destPort.theComponent)
```

Syntax

The *Connection* element extends the following elements of the metamodel:

- *ElementWithID* (See Table 5.8)
- *TestArchitectureElement* (See Section 5.6.11)

Table 5.30: Properties of the Connection UTML element

Property	**Description**	**Type**	**Occu-rence**

sourcePort	The source port for the connection.	*PortInstance* (See Table 5.28)	1..1
destPort	The destination port for the connection.	*PortInstance* (See Table 5.28)	1..1
architecture	The test architecture in which the connection has been created.	*TestArchitecture* (See Table 5.31)	1..1

5.6.15 TestArchitecture

Description

The ***TestArchitecture*** element represents a test architecture in UTML. I.e. a collection of component instances that are connected via ports to build a setup on which test behaviour will be executed to assess the system under test.

Semantics

TestArchitecture elements define the architectural context in which test behaviour will take place. Therefore, test architectures will be associated to UTML behaviour elements so that test behaviour design will take into account the constraints defined by the test architecture.

Constraints

Constraint (Optional) Every test architecture should contain at least one component belonging to the SUT.

```
self.componentInstance
-> exists(kind = utml::test_architecture::ComponentKind::SUT)
```

Syntax

Figure 5.17 displays an example test architecture diagram for a test model containing a group of test architecture elements and two test architectures. As depicted in that figure, each test architecture is visualised as a *Class*-type graphical element, with the test architecture's identifier as label. The usage of *Package* graphical element to visualise groups in test architecture diagrams is also illustrated on that picture.

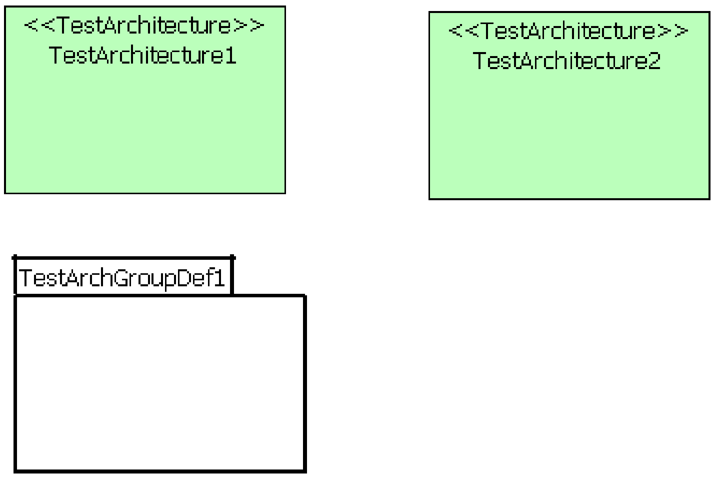


Figure 5.17: Example UTML Test Architecture Diagram with contained Architectures and Group Definitions

The *TestArchitecture* element extends the following elements of the meta-model:

- *DescribedElement* (See Table 5.4)
- *ElementWithUniqueID* (See Table 5.9)
- *TestArchGroupItem* (See Section 5.6.9)
- *TestArchitectureElement* (See Section 5.6.11)

Table 5.31: Properties of the TestArchitecture UTML element

Property	**Description**	**Type**	**Occurence**
componentInstance	Component instances contained in the test architecture.	*ComponentInstance* (See Table 5.29)	0..n
connections	Connections between the component instances through ports.	*Connection* (See Table 5.30)	0..n

setup-Function	A list of references to behaviour function definitions that initialize the test architecture. The behaviour functions listed are called sequentially to initialize the test architecture, in the same order in which they have been added.	*TestBehaviour-ActionDef* (See Table 5.72)	0..n
teardown-Function	A list of reference to behaviour function definitions that terminates the test architecture (e.g. performing some cleanup operations after test execution). The behaviour functions listed are called sequentially to teardown the test architecture, in the same order in which they have been added.	*TestBehaviour-ActionDef* (See Table 5.72)	0..n
associated-Default	References to default behaviour definitions to which the test architecture is associated. A default behaviour being associated to a test architecture means that, when the architecture is setup the default is activated and when the architecture is teared down (e.g. at the end of a testcase), the default is deactivated.	*Default-Behaviour-Def* (See Table 5.82)	0..n
execution-Mode	The execution mode defines how the component instances defined in the test architecture will be started when test behaviour designed to be run on that architecture is executed. Possible values are *SEQUENTIAL*(Default), indicating that the components will be instanciated and started sequentially in the same order of their creation in the UTML model or *PARALLEL*, indicating that all will be started in parallel according to an appropriate synchronization scheme.	*Execution-Mode* (See Table 5.32)	1..1

5.6.16 ExecutionMode

Description

The ***ExecutionMode*** UTML element is a classifier defining possible modes of execution for elements of a test architecture, when test behaviour designed on that architecture is executed.

Semantics

The only usage of the ***ExecutionMode*** element is as an attribute of a ***TestArchitecture*** element. Therefore, the ***ExecutionMode*** element has no semantics as such, but its value has an impact on the syntax of the test architecture to which it is associated.

Syntax

Table 5.32: The ExecutionMode UTML element

Literal	**Description**
SEQUENTIAL	Indicates that test component instances in the test architecture will be started sequentially, according to their order of creation in the UTML model.
PARALLEL	Indicates that test component instances in the test architecture will be started in a synchronised manner, according to a given synchronisation scheme. Currently, the definition of the synchronisation scheme is out of UTML's scope, but could be added at a later stage, if required as an extension to the metamodel.

5.6.17 TestArchPatternKind

Description

The ***TestArchPatternKind*** UTML element is an enumeration defining a classifier for the various types of architectural patterns identified so far.

Syntax

Table 5.33: Fields and attributes of the TestArchPatternKind UTML element

Pattern Kind	**Description**
P2P	Indicates a Point-to-Point kind of architecture, involving two component instances as endpoints with each of their ports connected at most once to the other one's. This corresponds to the *One-on-One Test Architecture* design pattern described in Section A.3.2.
PROXY	Indicates a proxy kind of architecture involving a test component placed as a proxy to intercept and monitor the data exchange between two components belonging to the SUT. This corresponds to the *Proxy Test Component* test architecture design pattern described in Section A.3.5.
SANDWICH	Indicates a sandwich kind of test architecture involving two test components, each of which is placed exchange data with an SUT component that is virtually placed in the middle between those two test components. This corresponds to the *Sandwich Test Architecture* pattern described in Section A.3.6.
PMP	Indicates a Point-to-Multi-Point(), i.e. one whereby the connections between the ports involved provide multiple paths from a single location to multiple locations [35] kind of architecture. This corresponds to the *Point-to-Multi Point Test Architecture* design pattern described in Section A.3.3.

MESH	Indicates a mesh kind of architecture. I.e. one whereby, each of the components involved is connected to a port of the other component instances.
CUSTOM	Reserved for user customisation. This literal indicates a user-defined architecture scheme.

5.6.18 P2PArchitecture

Description

The ***P2PArchitecture*** element is a realization of the *One-on-One* (or *Peer-To-Peer*) test architecture pattern, which consists of a test system composed of a single component, that is connected to the SUT, also represented as a single component. The *One-on-One* test architecture is described with further details in section A.3.2.

Semantics

The *P2PArchitecture* element extends the *TestArchitecture* element (See Table 5.31). Therefore, it inherits the basic semantics defined for that element.

Constraints

Constraints The ***P2PArchitecture*** element inherits all the constraints defined for the *TestArchitecture* element as defined in Section 5.6.15

Syntax

Table 5.34: Properties of the P2PArchitecture UTML element

Property	**Description**	**Type**	**Occurence**
firstComponent	First component instance involved in the P2P test architecture.	*ComponentInstance* (See Table 5.29)	1..1

system-Component	Other component instance involved in the P2P test architecture.	*ComponentInstance* (See Table 5.29)	1..1
pattern-Kind	Test architecture pattern kind: *P2P*(See 5.33)	*TestArch-Pattern-Kind* (See Table 5.33)	1..1

5.6.19 PMPArchitecture

Description

The ***PMPArchitecture*** element is a descriptor for a Point-to-Multi Point(PMP) test architecture, which consists of a component connected via a single port to a set of other components. This architecture is an instantiation of the PMP architecture pattern, which is described in section A.3.3.

Semantics

The *PMPArchitecture* element extends the *TestArchitecture* element (See Table 5.31). Therefore, it inherits the basic semantics defined for that element.

Constraints

The ***PMPArchitecture*** element inherits all the constraints defined for the *TestArchitecture* element as defined in Section 5.6.15

Syntax

Table 5.35: Properties of the PMPArchitecture UTML element

Property	**Description**	**Type**	**Occurence**
firstComponent	First component instance involved in the PMP test architecture.	*ComponentInstance* (See Table 5.29)	1..1
other-Components	Other component instances involved in the PMP test architecture.	*ComponentInstance* (See Table 5.29)	1..n

pattern-Kind	Test architecture pattern kind: *PMP* (See 5.33)	*TestArch-Pattern-Kind* (See Table 5.33)	1..1

5.6.20 MeshArchitecture

Description

A fully connected mesh network is a network in which all nodes are interconnected. Such networks are used in wireless networks and other application fields for which self-healing capacity is required. Self-healing is the ability for nodes in the network to reconnect themselves to overcome the failure of a part of the network.

The UTML ***MeshArchitecture*** element models a test architecture following the mesh networking pattern. However, only the topological aspects of mesh networks are expressed with a ***MeshArchitecture*** element, while the behavioural aspects (e.g. the self-healing) are not implied. Therefore, a test architecture created with the mesh architecture architectural pattern has all its associated component instances connected to each other wherever their owned ports allow it.

Semantics

The *MeshArchitecture* element extends the *TestArchitecture* element (See Table 5.31). Therefore, it inherits the basic semantics defined for that element.

Constraints

The ***MeshArchitecture*** element inherits all the constraints defined for the *TestArchitecture* element as defined in Section 5.6.15

Syntax

The *MeshArchitecture* element extends *TestArchitecture* (See Table 5.31)

Table 5.36: Properties of the MeshArchitecture UTML element

Property	**Description**	**Type**	**Occurence**

compo-nents	Component instances building the mesh test architecture.	*ComponentInstance* (See Table 5.29)	0..n
pattern-Kind	Test architecture pattern kind: *MESH* cf. Table 5.33	*TestArch-PatternKind* (See Table 5.33)	1..1

5.7 Test Data Design Concepts

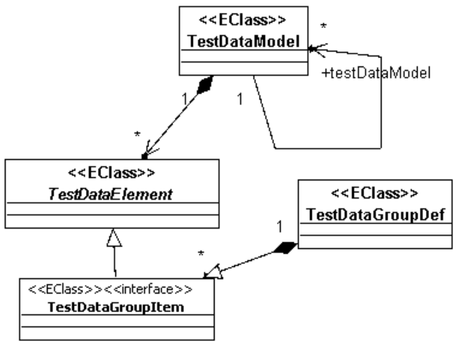


Figure 5.18: Class Diagram: Hierarchy of UTML Metamodel for Test Data Modelling

The UTML metamodel's test data concepts provide the means for modelling data types and data instances to be used for testing. Data abstraction [126] is an essential aspect of UTML test modelling. The idea is to find the right balance between the need for abstraction and the necessity of providing enough information to allow the design of sensible test cases. Therefore, constraints play a central role in the specification of test data in UTML, as they enable to define the requirements that need to be met by data to be suitable as input (i.e. as stimuli) or as output (response) in test sequences. Based on those requirements, concrete instances of data may be generated dynamically for testing purpose or validated to check if they meet the requirements.

To facilitate exchange with other notations, UTML's test data concepts reuse a lot of the concepts of the XML Schema Definition (XSD) language, adding elements specific to the testing domain. In this section, the elements of the UTML test data metamodel are described, along with the relationships between themselves and other aspects of the UTML metamodel.

5.7.1 TestDataModel

Description

The ***TestDataModel*** element is the root container for UTML test data design. It contains model elements for designing the structure of data to be exchanged in testing and the mechanisms for generating concrete test data.

Constraints

Constraint (Optional) The same field definition should not be duplicated in more than one message type definition. If so, then the usage of inheritance should be considered.

```
utml::test_data::MessageTestDataType.allInstances() ->
isUnique(dataTypeField)
```

Syntax

The ***TestDataModel*** element extends the *BasicTestModel* (See Table 5.3.2) element defined previously

Table 5.37: Properties of the TestDataModel UTML element

Property	**Description**	**Type**	**Occurence**
testDataModel	References to other related test data models.	*TestDataModel* (See Table 5.37)	0..n
testDataElement	Contained test data model elements.	*TestDataElement* (See Section 5.7.2)	0..n

5.7.2 TestDataElement

Description

The ***TestDataElement*** is an abstract classifier that is the base for all elements in the UTML test data metamodel

Syntax

The ***TestDataElement*** element extends *UtmlElement* (See Section 5.3.1)

5.7.3 DataTypeIndicator

Description

The ***DataTypeIndicator*** element is an enumeration listing the different categories of type indicators in the UTML test data metamodel.

Semantics

The ***DataTypeIndicator*** element's purpose is to be used as a property for data type model elements to define their structure in a more precise manner.

Syntax

Table 5.38: The DataTypeIndicator UTML element

Literal	**Description**
SEQUENCE	Used for a data structure containing ordered fields.
CHOICE	Used for a data structure in which only one of the listed fields may be present.
ALL	Used for a data structure containing unordered fields.
ENUMERATION	Used for an enumeration kind of data structure.

5.7.4 DataDirection

Description

The ***DataDirection*** element is an enumeration for indicating the direction in which a test data instance may be used in test scenarios. It should be kept in mind that in UTML test modelling, according to the black-box paradigm, all definitions are expressed from the test system's perspective.The details provided for ***DataDirection*** follow that same rule. For example, whenever direction *IN* is mentioned, it refers to data moving into the test component.

Semantics

The ***DataDirection*** element has no specific semantics, beyond being used as a property for data instances (see Section 5.7.23) and port instances (see Section 5.6.12). It should be noted that the direction of data and port instances is always relative to the component hosting the port instance or using the data instance for test behaviour.

Syntax

Table 5.39: The DataDirection UTML element

Literal	**Description**
INOUT	Used for a data instance that may be used both for sending data with other components (Test-, SUT-) or for verifying data received from it. When used in association with port instances, this indicates that the port instance may be used both for receiving data *IN*to the owning test component or for sending data *OUT* to other entities.
IN	Used for test data modelled for verifying data received from the SUT or from other test components. When, used in association with port instances, it indicates that data can be received via the referencing port instance into the owner test component.
OUT	Used for test data modelled for sending data out to other components in a test architecture. When used in association with port instances, this indicates that the port can be used to send data out of the owning test component to other entities.

5.7.5 DataKind

Description

The ***DataKind*** element is an enumeration listing the different kinds of data currently supported by the UTML metamodel for designing test data.

Semantics

The ***DataKind*** element has no specific semantics, beyond being used as a property for data types.

Syntax

Table 5.40: The DataKind UTML element

Literal	**Description**
MESSAGE	Indicates a test data kind used for asynchronous communication.
OPERATION	Indicates a test data kind used for modelling synchronous communication schemes, i.e. those whereby entities invoke operations or methods provided by others to exchange data.
EXCEPTION	Test data of this kind can be used to refine *OPERATION* kind data types by specifying potentially exceptional behaviour that could occur when the operation is called.
CONTINUOUS	Indicate a test data kind exchanged through continuous signals.

5.7.6 DataPatternKind

Description

The ***DataPatternKind*** UTML element is an enumeration defining a classifier for types of test data patterns.

Semantics

The main purpose of the ***DataPatternKind*** element is to serve as a property of test data instance to model data that will be generated during test execution based on the selected test data pattern. The implementation of the mechanism for generating concrete data instances based on the UTML test data model element is left to the implementing body and is therefore out-of-scope for this thesis.

Syntax

Table 5.41: The DataPatternKind UTML element

Pattern Kind	**Description**
DOMAIN_PARTITION	Indicates data obtained through domain partition. [133]
DEFAULT_VALUE	Default value used as test data.
BOUNDARY_VALUE	Indicates a scheme whereby boundary values are used as test data.

RANDOM_VA-LUE	For values obtained by selecting randomly within a range.
CUSTOM	This literal is for user-defined data pattern.

5.7.7 ConstraintKind

Description

The ***ConstraintKind*** element is an enumeration listing mechanisms for defining constraints for test data in UTML. Most UTML constraint kinds are inherited from the XSD language and follow the same semantics defined in the XSD specification [63] and described in table 5.42.

Semantics

The main purpose of the ***ConstraintKind*** element is to be used as a property for constraints, e.g generic constraints (see Section 5.7.27) or field constraints (see Section 5.7.27).

Syntax

Table 5.42: The ConstraintKind UTML element

Constraint kind	**Description**
ENUMERATION	Indicates a constraint reducing acceptable choices to those present in an enumeration of values.
FRACTION_-DIGITS	Defines a constraint indicating that a test data instances total number of decimal places should be equal to a given value. Obviously constraints of this kind are only applicable to types of test data where they make sense, namely those representing rational values (e.g. float).
LENGTH	Defines a constraint on the length of a test data value or that a contained field.
MAX_EXCLU-SIVE	Defines a constraint indicating that a test data instance's value should be lower than a given value.

MAX_INCLUSIVE	Defines a constraint indicating that a test data instance's value should be lower than or equal to a given value.
MAX_LENGTH	Defines a constraint indicating that a test data instance's length should be not exceed a given value.
MIN_EXCLU-SIVE	Defines a constraint indicating that a test data instance's value should be greater than a given value.
MIN_INCLU-SIVE	Defines a constraint indicating that a test data instance's value should be lower than or equal to a given value.
MIN_LENGTH	Defines a constraint indicating that a test data instance's length should be lower than a given value.
PATTERN	Defines a constraint indicating that a test data instance's value should match a given pattern. The pattern matching mechanism used is out of UTML's scope and is to be handled by the lower level test infrastructures. Obviously, (Perl/-Posix)regular expressions are good candidates for that purpose, but other similar approaches could be used instead.
TOTAL_DIGITS	Defines a constraint indicating that a test data instance's total number of digits should be equal to a given value. Obviously constraints of this kind are only applicable to types of test data where they make sense (e.g. integer values).
HAS_ELEMENT	Defines a constraint indicating that a test data instance is a list containing a given value as element.
IS_PRESENT	Defines a constraint indicating that a given field is present in a test data instance of a complex data type.
EQUALS	Defines a constraint indicating that a given test data instance equals a given value.

IS_NOT_PRE-SENT	Defines a constraint indicating that a given field is not present in a test data instance of a complex data type.

5.7.8 TestDataGroupItem

Description

The ***TestDataGroupItem*** is an abstract classifier providing the base for the grouping mechanism in UTML test data models. Each classifier extending ***Test-DataGroupItem*** can be added as a child of a group in a test data model.

Syntax

The ***TestDataGroupItem*** element extends *GroupItem* (See Section 5.3.5)

5.7.9 TestDataGroupDef

Description

The ***TestDataGroupDef*** element models a group in a test data model.

Syntax

The ***TestDataGroupDef*** element extends the following elements of the meta-model:

- *DescribedElement* (See Table 5.4)
- *ElementWithUniqueID* (See Table 5.9)
- *TestDataGroupItem* (See Section 5.7.8)

Table 5.43: Properties of the TestDataGroupDef UTML element

Property	**Description**	**Type**	**Occu-rence**
testData-Group-Item	test data elements contained in the group.	*TestData-GroupItem* (See Section 5.7.8)	0..n

5.7.10 RelationKind

Description

The ***RelationKind*** element is an enumeration of the kinds of relationship between test data types.

Semantics

A test data type may extend or restrict another existing test data type element to provide a mechanism similar to object inheritance for data types. The details of that mechanism are described in Section 5.7.11.

Syntax

Table 5.44: The RelationKind UTML element

Literal	**Description**
EXTENSION	Indicates that a test data type extends another existing one.
RESTRICTION	Indicates that a test data type restricts an existing test data type.

5.7.11 DataTypeRelationship

Description

The ***DataTypeRelationship*** element is used to model a relationship between a new test data type and an already existing one in UTML.

Semantics

The UTML notation defines two kinds of relationship between data types: extension and restriction. If test data type B *extends* test data type A(referred to by its *baseDataType* property), then B may modify fields defined in A (e.g. through redefinition, or by adding new constraints) and at the same time add new fields to those defined in A. On the other hand, if test data type B *restricts* test data type A, then B may only add new constraints or redefine fields defined in B without being able to add any new field.

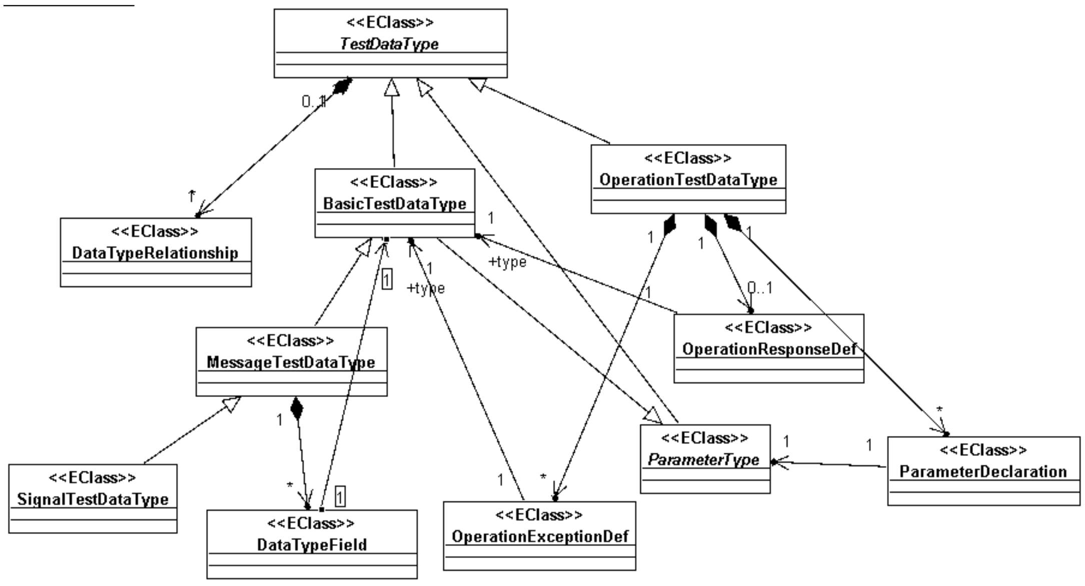
Syntax

Table 5.45: Properties of the DataTypeRelationship UTML element

Property	**Description**	**Type**	**Occu-rence**
baseData-Type	The base data type this relationship refers to.	*TestData-Type* (See Table 5.46)	1..1
dataCons-traint	Data constraints being added to the base data type.	*DataCons-traint* (See Table 5.57)	0..n
relation-Kind	Kind of relationship. One of the following val ues: • EXTENSION (Default) • RESTRICTION	*Relation-Kind* (See Table 5.44)	1..1

5.7.12 BasicTestDataType

Description



<<EClass>> TestDataType
<<EClass>> DataTypeRelationship
<<EClass>> BasicTestDataType
<<EClass>> OperationTestDataType
<<EClass>> MessageTestDataType
<<EClass>> OperationResponseDef
<<EClass>> SignalTestDataType
<<EClass>> DataTypeField
<<EClass>> OperationExceptionDef
<<EClass>> ParameterType
<<EClass>> ParameterDeclaration
0..1
*
1
+type
0..1

Figure 5.19: Class Diagram: UTML Metamodel for Modelling Test Data Types

The ***BasicTestDataType*** element models a basic simple test data type in UTML. A basic test data type has no fields.

Syntax

The ***BasicTestDataType*** element extends *test_data:TestDataType* (See Table 5.46)

5.7.13 TestDataType

Description

The ***TestDataType*** is an abstract classifier that is the base for test data type definitions in UTML.

Syntax

The ***TestDataType*** element extends *UniqueNamedElement* (See Table 5.7)

Table 5.46: Properties of the TestDataType UTML element

Property	**Description**	**Type**	**Occurence**
dataType-Relation-ship	Defines a relationship with another test data type.	*DataType-Relation-ship* (See Table 5.45)	0..1
kind	Kind of test data.	*DataKind* (See Table 5.40)	1..1
name	Unique identifier of the test data type.	xsd:string	1..1
coding_rule	Encoding rule of the test data type. The mechanism for evaluating the character string used for expressing encoding rules is out of UTML's scope.	xsd:string	0..1

5.7.14 MessageTestDataType

Description

The ***MessageTestDataType*** element models a structured *MESSAGE* kind of test data type, suitable for useage in asynchronous exchange of data in a UTML test model.

Semantics

The MessageTestDataType is a descriptor which can be used at a later stage to design test data instances for systems supporting asynchronous communication.

Syntax

A *Class* visual element is used to represent test data types.

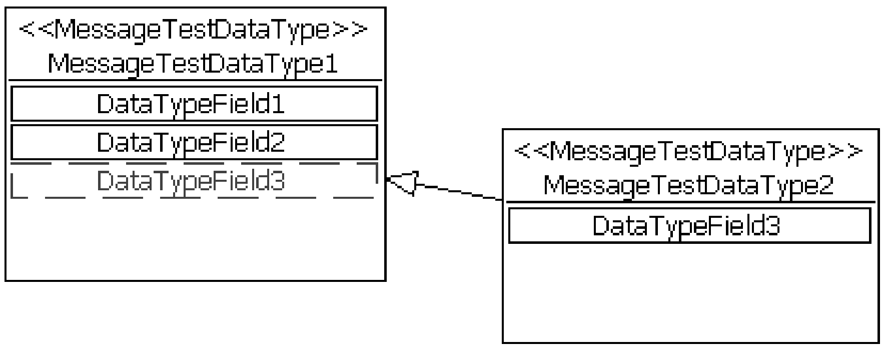


Figure 5.20: Example UTML Test Data Diagram

Figure 5.20 shows an example UTML test data diagram, featuring a *Message-TestDataType* being extended by another one. As depicted in that figure, fields contained in *MessageTestDataType* elements are represented graphically in a manner similar to the representation of attributes in UML class diagrams. The same rule applies to the representation of parameter declarations in *Operation-TestDataType*. Also visible in that figure is the graphical representation of optional fields (e.g. *DataTypeField2*, *DataTypeField3* and *DataTypeField4* in *MessageTestDataType2*) caracterized by a surrounding dash-styled border line and the associated dark gray color.

Additionally, the *Dependency* graphical element is used to represent import-type relationships between elements contained in test data diagrams, while the *Generalisation* graphical element is used to represent inheritance-type relationships (extension/restriction) between test data type definitions and test data instances.

The ***MessageTestDataType*** element extends *BasicTestDataType* (See Section 5.7.12)

Table 5.47: Properties of the MessageTestDataType UTML element

Property	**Description**	**Type**	**Occurrence**

dataType-Field	Fields of the message data type.	*DataType-Field* (See Table 5.48)	0..n
dataType-Indicator	Selector for indicating the kind of structured data type.	*DataType-Indicator* (See Table 5.38)	1..1

5.7.15 DataTypeField

Description

The ***DataTypeField*** element models a field in a structured test data type.

Syntax

The ***DataTypeField*** element extends the following elements of the metamodel:

- *NamedElement* (See Table 5.6)
- *DescribedElement* (See Table 5.4)

Table 5.48: Properties of the DataTypeField UTML element

Property	**Description**	**Type**	**Occu-rence**
fieldData-Type	Reference to type definition for the field.	*BasicTest-DataType* (See Section 5.7.12)	1..1
optional	Indicates whether the field is optional (Default is ***false***)	xsd:boolean	1..1
minOccurs	If message kind is a list, then this indicates, the minimal number of occurences of elements of the list. The provided string is one that can be evaluated to an integer value in the lower level test infrastructure.	xsd:string	0..1

maxOccurs	If message kind is a list, then this indicates, the maximal number of occurences of elements of the list. The provided string is one that can be evaluated to an integer value in the lower level test infrastructure.	xsd:string	0..1
dataType-Indicator	A selector for indicating the type of structured data type.	*DataType-Indicator* (See Table 5.38)	1..1
default-Value	A reference to a previously defined ***ValueInstance*** element, serving as default value for insta nces of this field.	*Value-Instance* (See Table 5.53)	0..1
default-Value-Literal	A string literal representing the de fault value for the field. This is an alternative to providing a reference for the default value via the ***defaultValue*** child element mentioned above.	xsd:string	0..1

5.7.16 ParameterDeclaration

Description

The ***ParameterDeclaration*** element models a parameter declaration for a test data type, a test data instance or a behaviour definition (i.e. a function) in the UTML metamodel.

Semantics

The ***ParameterDeclaration*** element provides the mean for modelling the declaration of a parameter for any other UTML elements for which parameterization is to be supported. Therefore, this element can only be used in combination with those other UTML elements.

Syntax

The ***ParameterDeclaration*** element extends the following elements of the metamodel:

- *NamedElement* (See Table 5.6)
- *AbstractDataInstance* (See Section 5.7.21)

Table 5.49: Properties of the ParameterDeclaration UTML element

Property	**Description**	**Type**	**Occurence**
type	Type of the parameter.	*BasicTestDataType* (See Section 5.7.12)	1..1
direction	Direction of the parameter: one of *IN*, *INOUT* or *OUT*. The direction of parameter declarations follows the same semantic as defined in the Interface Definition Language (IDL) or in the TTCN-3language.	*DataDirection* (See Table 5.39)	1..1

5.7.17 OperationTestDataType

Description

The ***OperationTestDataType*** element models an *OPERATION* kind of test data type in the UTML metamodel.

Semantics

The ***OperationTestDataType*** element is a descriptor which can be used to design test data instances for systems supporting synchronous communication using a semantics of operation calls that block until the called party returns a result.

Syntax

The ***OperationTestDataType*** element extends *TestDataType* (See Table 5.46)

Table 5.50: Properties of the OperationTestDataType UTML element

Property	**Description**	**Type**	**Occurence**
operationResponseDef	Response declaration for the operation test data type.	*OperationResponseDef* (See Table 5.51)	1..1

parameter-Declaration	Parameters declarations for the operation data type.	*Parameter-Declaration* (See Table 5.49)	0..n
operation-Exception-Def	Exceptions declarations for the operation data type.	*Operation-Exception-Def* (See Table 5.52)	0..n

5.7.18 OperationResponseDef

Description

The ***OperationResponseDef*** element models a response definition in a UTML test operation specification.

Semantics

The ***OperationResponseDef*** element is only used in combination with the *OperationTestDataType* element. It represents the modelling of the response part of an operation.

Syntax

The ***OperationResponseDef*** element extends the *DescribedElement* element of the metamodel(See Table 5.4).

Table 5.51: Properties of the OperationResponseDef UTML element

Property	**Description**	**Type**	**Occurrence**
type	Data type of the operation response.	*TestData-Type* (See Table 5.46)	1..1

5.7.19 OperationExceptionDef

Description

The ***OperationExceptionDef*** element models an exception definition in a UTML test operation specification.

Semantics

In a similar manner as the ***OperationResponseDef*** element, the ***Operation-ExceptionDef*** is also only used in combination with the *OperationTestDataType* element. It can be used to model exceptional responses potentially returned by an operation.

Syntax

The ***OperationExceptionDef*** element extends the *DescribedElement* element of the metamodel(See Table 5.4).

Table 5.52: Properties of the OperationExceptionDef UTML element

Property	Description	Type	Occurence
type	Data type of the operation exception.	*TestDataType* (See Table 5.46)	1..1

5.7.20 SignalTestDataType

Description

The ***SignalTestDataType*** element models a test data type descriptor for continuous signals in UTML.

Syntax

The ***SignalTestDataType*** element extends *MessageTestDataType* (See Table 5.47)

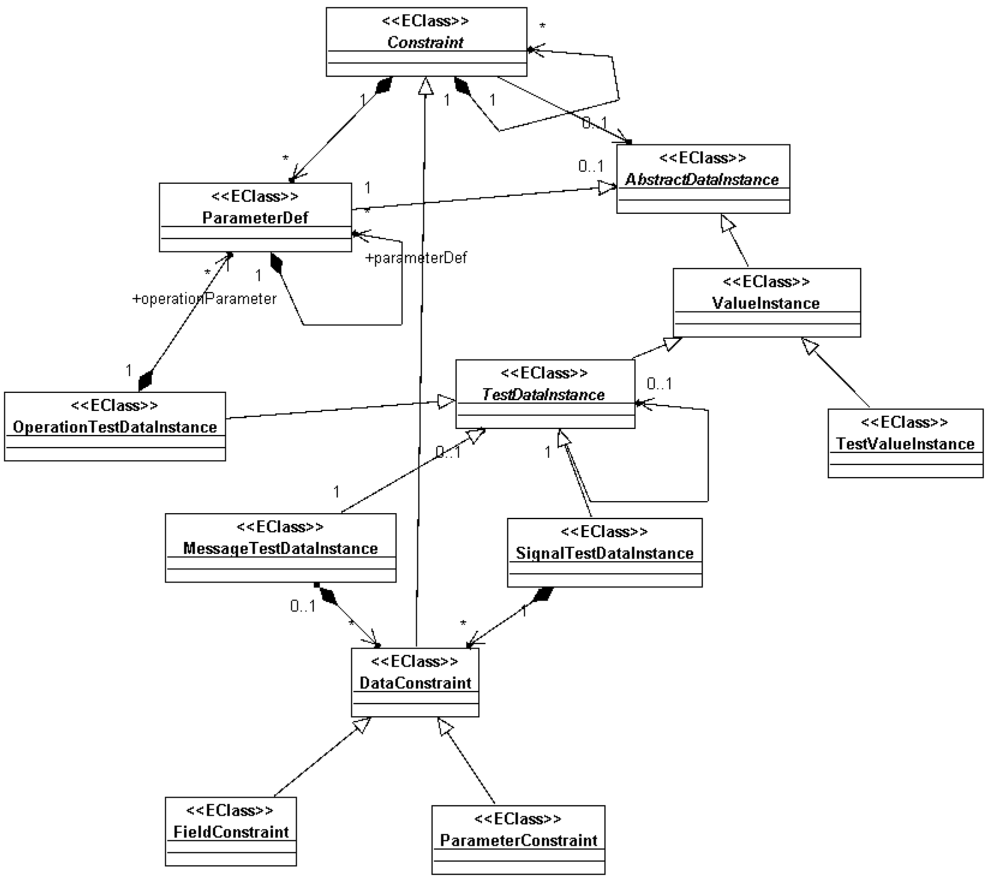
5.7.21 AbstractDataInstance

Description

The ***AbstractDataInstance*** element is an abstract classifier providing the base for modelling data instances in UTML.

Syntax

The ***AbstractDataInstance*** element extends *DescribedElement* (See Table 5.4)



<<EClass>> Constraint
<<EClass>> AbstractDataInstance
<<EClass>> ParameterDef
<<EClass>> ValueInstance
<<EClass>> OperationTestDataInstance
<<EClass>> TestDataInstance
<<EClass>> TestValueInstance
<<EClass>> MessageTestDataInstance
<<EClass>> SignalTestDataInstance
<<EClass>> DataConstraint
<<EClass>> FieldConstraint
<<EClass>> ParameterConstraint
+parameterDef
+operationParameter

Figure 5.21: Class Diagram: UTML Metamodel for Test Data Instances

5.7.22 ValueInstance

Description

The ***ValueInstance*** element models a value of a given data type in UTML.

Syntax

The ***ValueInstance*** element extends the following elements of the metamodel:

- *UniqueNamedElement* (See Table 5.7)
- *AbstractDataInstance* (See Section 5.7.21)

Table 5.53: Properties of the ValueInstance UTML element

Property	Description	Type	Occu-rence

type	Reference to the type definition for this value instance.	*TestDataType* (See Table 5.46)	1..1

5.7.23 TestDataInstance

Description

The ***TestDataInstance*** is an abstract classifier used as base for all test data instances in UTML. Test data instances represent concrete data resulting from instantiating previously defined ***TestDataType*** elements (See Section 5.7.13) or extensions thereof, e.g. ***MessageTestDataType*** (See Section 5.7.14), ***OperationTestDataType*** (See Section 5.7.17) and ***SignalTestDataType*** (See Section 5.7.17)

Syntax

The ***TestDataInstance*** element extends the following elements of the metamodel:

- *TestDataGroupItem* (See Section 5.7.8)
- *ValueInstance* (See Table 5.53)

The ***TestDataInstance*** element is an abstract classifier that is the base for all data instance definitions in UTML.

Table 5.54: Properties of the TestDataInstance UTML element

Property	**Description**	**Type**	**Occurence**
mirrorDataInstance	A test data instance that is suitable as response to this test data instance.	*TestDataInstance* (See Table 5.54)	0..1
direction	Direction in which the test data might be used.	*DataDirection* (See Table 5.39)	1..1

5.7.24 MessageTestDataInstance

Description

The ***MessageTestDataInstance*** element models a message that can be exchanged between components in a test architecture.

Semantics

MessageTestDataInstance elements represent instantiations of *MessageTestDataType* elements.

Constraints

Constraint If a *MessageTestDataInstance* element extends another one and that *MessageTestDataInstance* (base data instance) was designed for incoming data (i.e. with its *direction* property set to *IN*), then the extending data instance may not be used for outgoing test data. The motivation for this constraint is the fact that test data designed for checking incoming data may include constraints expressed using wildcards, which may not be enough to create concrete data to be sent to other entities.

```
(self.baseDataInstance.oclIsTypeOf(OclVoid) = false
and
self.baseDataInstance.direction
= utml::test_data::DataDirection::IN
)
implies
(self.direction=utml::test_data::DataDirection::IN)
```

Constraint The *type* property of a ***MessageTestDataInstance*** element must be a *MessageTestDataType* or a *BasicTestDataType*.

```
self.oclAsType(utml::test_data::ValueInstance)
.type.oclIsTypeOf(OclVoid) = false
implies
(self.oclAsType(utml::test_data::ValueInstance).type
.oclIsTypeOf(utml::test_data::MessageTestDataType) = true
or
self.oclAsType(utml::test_data::ValueInstance).type
.oclIsTypeOf(utml::test_data::BasicTestDataType) = true)
```

Syntax

The ***MessageTestDataInstance*** element extends the ***TestDataInstance*** element (See Table 5.54) of the metamodel.

Table 5.55: Properties of the MessageTestDataInstance UTML element

Property	**Description**	**Type**	**Occurrence**
dataConstraint	Constraints associated to this message data instance.	*DataConstraint* (See Table 5.57)	0..n
dataPatternKind	Test data pattern this message data instance is based on. This property can be used to design dynamic generation of test data based on the selected pattern. The implementation of the mechanism for generating test data dynamically is tool-specific and therefore out of the scope of the UTML notation.	*DataPatternKind* (Cf. Table 5.41)	0..1
baseDataInstance	Test data instance which this data instance extends or restricts.	*TestDataInstance* (See Table 5.54)	0..1
parameterDeclaration	Describes a parameter declaration for a message data instance. This functionality imported from the TTCN-3notation is very convenient for defining reusable test data for which the values or the constraints can be customized for a specific test case. It should be noted that concrete values must be provided for parameterized test data instances, whenever those are used to design test behaviour. Otherwise, the resulting test model is considered invalid.	*ParameterDeclaration* (See Table 5.49)	0..n

5.7.25 SignalTestDataInstance

Description

The ***SignalTestDataInstance*** element models a test data instance for communication based on continuous signals.

Constraint The *type* property of a ***SignalTestDataInstance*** element must be a *SignalTestDataType* or a *BasicTestDataType*.

```
self.oclAsType(utml::test_data::ValueInstance)
.type.oclIsTypeOf(OclVoid) = false
implies
(self.oclAsType(utml::test_data::ValueInstance).type
.oclIsTypeOf(utml::test_data::SignalTestDataType) = true
or
self.oclAsType(utml::test_data::ValueInstance).type
.oclIsTypeOf(utml::test_data::BasicTestDataType) = true)
```

Syntax

The ***SignalTestDataInstance*** element extends *TestDataInstance* (See Table 5.54)

5.7.26 OperationTestDataInstance

Description

The ***OperationTestDataInstance*** element models a test data instance for synchronous communication between components in a test model. It is equivalent to the invocation of a method or a function on an interface.

Semantics

OperationTestDataInstance elements represent instantiations of *OperationTestDataType* elements. This includes the specification of parameters, if any were defined in the associated *OperationTestDataType*.

Constraints

Constraint The *type* property of an ***OperationTestDataInstance*** element must be an OperationTestDataType.

```
self.oclAsType(utml::test_data::ValueInstance)
.type.oclIsTypeOf(OclVoid) = false
```

```
implies
self.oclAsType(utml::test_data::ValueInstance).type
.oclIsTypeOf(utml::test_data::OperationTestDataType) = true
```

Syntax

The ***OperationTestDataInstance*** element extends *TestDataInstance* (See Table 5.54)

Table 5.56: Properties of the OperationTestDataInstance UTML element

Property	Description	Type	Occurence
operation-Parameter	Parameter values for the operation data instance.	*Parameter-Def* (See Table 5.60)	0..n

5.7.27 DataConstraint

Description

The ***DataConstraint*** element provides a means for specifying constraints that have to be fullfiled by a given value instance to meet specific test requirements. Based on such constraints, instances of data can be generated automatically for sending stimuli to the SUT or responses from other (e.g. SUT) components can be validated against those constraints to assess correct behaviour.

Semantics

The ***DataConstraint*** element is used as a property of test data instances to indicate which constraints must be met by those data instances to serve a particular test purpose. Data constraints may be defined both for data designed to be sent to other components (e.g. to the SUT as stimulus) or to express requirements on incoming responses from the SUT. Based on those data constraints concrete values may be generated automatically in the target test environment to be used as stimuli. In the same manner oracles for checking SUT responses may be generated automatically as well for test execution.

Constraints

Constraint For constraints checking the value of a data instance or a field thereof, a reference value must be provided, either as string literal to be evaluated by the test environment or as a reference to another defined value instance of compatible type. The targetted element (i.e. the one for which the constraint is defined) is then compared with the reference value to assess whether the constraint is fullfiled or not. This constraint does not apply to UTML data constraints used for checking whether a value is present or not. If a reference value is provided in those cases, it will simply be ignored.

```
(self.constraintKind
<> utml::test_data::ConstraintKind::IS_PRESENT
and
self.constraintKind
<> utml::test_data::ConstraintKind::IS_NOT_PRESENT
)
implies
(
((self.referenceValueLiteral.oclIsTypeOf(OclVoid) = true
 or
 self.referenceValueLiteral = '')
implies
        self.referenceData.oclIsTypeOf(OclVoid) = false)
        or
(self.referenceData.oclIsTypeOf(OclVoid) = true
implies
        (self.referenceValueLiteral.oclIsTypeOf(OclVoid) = false
         and
        self.referenceValueLiteral <> ''
         )))
```

Syntax

The ***DataConstraint*** element extends *DescribedElement* (See Table 5.4)

Table 5.57: Properties of the DataConstraint UTML element

Property	**Description**	**Type**	**Occurence**
constraintKind	Kind of constraint.	*ConstraintKind* (See Table 5.42)	1..1
referenceData	A defined test data instance which this constraint refers to. This means, data instances required to meet this constraint will be compared against the value pointed to by this property.	*TestDataInstance* (See Table 5.54)	0..1

reference-ValueLi-teral	a character string representing the value which this constraint refers to. This field is to give the test designer a more flexible mean for defining the value of the reference for the constraint. It is then left to the lower-level test infrastructure to handle the mechanism for verifying that a test data instance fullfils the constraint or not. If the ***Reference-Data*** field has been provided, then this field can be omitted.	xsd:string	0..1

5.7.28 FieldConstraint

Description

The ***FieldConstraint*** element provides a mean for modelling a constraint on a field in a UTML structured test data type.

Constraints

Constraint The *field* property of a *FieldConstraint* element must refer to a field effectively supported by the type definition for the associated message data instance. The field may also be inherited from an extended data type.

```
(self.field.oclIsTypeOf(OclVoid)=false
and
 self.field.fieldDataType.oclIsTypeOf(OclVoid)=false
and
 self.theContainerMessageDataInstance
 .oclIsTypeOf(OclVoid)=false
)
implies
(self.theContainerMessageDataInstance
.oclAsType(utml::test_data::ValueInstance).type
.oclAsType(utml::test_data::MessageTestDataType)
.dataTypeField -> exists
(fieldDataType.name = self.field.fieldDataType.name)
or (self.theContainerMessageDataInstance.
oclAsType(utml::test_data::ValueInstance)
.type.oclAsType(utml::test_data::MessageTestDataType)
.dataTypeRelationship -> forAll(oclIsTypeOf(OclVoid) = false)
and
self.theContainerMessageDataInstance
.oclAsType(utml::test_data::ValueInstance).type
.oclAsType(utml::test_data::MessageTestDataType)
.dataTypeRelationship -> forAll(baseDataType
.oclIsTypeOf(OclVoid) = false)
and
self.theContainerMessageDataInstance
.oclAsType(utml::test_data::ValueInstance).type
.oclAsType(utml::test_data::MessageTestDataType)
.dataTypeRelationship -> exists(baseDataType
.oclAsType(utml::test_data::MessageTestDataType)
.dataTypeField -> exists(fieldDataType.name
 = self.field.fieldDataType.name))
)
)
```

Syntax

The ***FieldConstraint*** element extends *DataConstraint* (See Table 5.57)

Table 5.58: Properties of the FieldConstraint UTML element

Property	**Description**	**Type**	**Occurence**
field	Reference to the field to which the constraint applies.	*DataType-Field* (See Table 5.48)	1..1

5.7.29 ParameterConstraint

Description

The ***ParameterConstraint*** element provides a mean for modelling a constraint on a declared parameter in a UTML operation test data type.

Syntax

The ***ParameterConstraint*** element extends *DataConstraint* (See Table 5.57)

Table 5.59: Properties of the ParameterConstraint UTML element

Property	**Description**	**Type**	**Occurence**
param	Reference to the parameter declaration to which the constraint applies.	*Parameter-Declaration* (See Table 5.49)	1..1

5.7.30 ParameterDef

Description

The ***ParameterDef*** element models a parameter definition in UTML. Parameter definitions can be used to set the values of parameters to a any parameterizable UTML element. Parameterizable UTML elements include execution and thus can be used to customized a test suite to a given platform, a specific SUT or ensure that certain preconditions are fullfiled for a test case.

Semantics

The ***ParameterDef*** elements are used in combination with parameterizable elements to model the values to be used for the parameters declared by those elements when they are instantiated or used in test actions and test events.

Constraints

Constraint The value for the parameter must be provided either as a character string literal to be evaluated by the target test environment or as a reference to a previously defined value instance.

```
(self.paramValue.oclIsTypeOf(OclVoid) = true
implies
(self.paramValueLiteral.oclIsTypeOf(OclVoid) = false)
and
self.paramValueLiteral <> '')
or
((self.paramValueLiteral.oclIsTypeOf(OclVoid) = true
or
self.paramValueLiteral = '')
implies
self.paramValue.oclIsTypeOf(OclVoid) = false)
```

Constraint The value instance provided as parameter for a *ParameterDef* element may also be parameterizable. If so, the parameters required must be provided to match the definition of the value instance.

```
(self.paramValue.
oclIsTypeOf(utml::test_data::MessageTestDataInstance) = true)
implies
(self.parameterDef -> size()
 = self.paramValue
  .oclAsType(utml::test_data::MessageTestDataInstance)
  .parameterDeclaration -> size())
```

Syntax

The ***ParameterDef*** element extends *AbstractDataInstance* (See Section 5.7.21)

Table 5.60: Properties of the ParameterDef UTML element

Property	**Description**	**Type**	**Occurence**
parameterDeclaration	The parameter declaration for which a value is being set.	*ParameterDeclaration* (See Table 5.49)	1..1
paramValue	A reference to a previously defined value instance that will be attributed to the parameter.	*ValueInstance* (See Table 5.53)	0..1
paramValueLiteral	Alternatively to providing a reference to a defined value instance, a character string representing the value to be attributed to the parameter. At least one of those two alternatives must be provided for any parameter definition. If both the *paramValue* and *paramValueLiteral* are provided, then the *paramValue* field has priority.	xsd:string	0..1

5.7.31 TestParameter

Description

The ***TestParameter*** element models a test parameter definition in UTML. Test parameters are constant values that can be set prior to test execution and thus can be used to customize a test suite for a given platform, a specific SUT or to ensure that certain preconditions are fullfiled for a given test case.

Semantics

Syntax

The ***TestParameter*** element extends *ValueInstance* (See Table 5.53)

Table 5.61: Properties of the TestParameter UTML element

Property	**Description**	**Type**	**Occurence**
value	A reference to a previously defined *ValueInstance* with which the test paramater is initialized.	*Value-Instance* (See Table 5.53)	0..1
value-Literal	A character string literal describing a value to be assigned to the parameter. This is an alternative to providing a *ValueInstance* as described above. If both the ***value*** and the ***valueLiteral*** fields are provided, than the *ValueInstance* field has priority.	xsd:string	0..1

5.7.32 TestParameterSet

Description

The ***TestParameterSet*** element models a set of parameters, which can be attached to a test case.

Semantics

Syntax

The ***TestParameterSet*** element extends the following elements of the metamodel:

- *UniqueNamedElement* (See Table 5.7)
- *DescribedElement* (See Table 5.4)

Table 5.62: Fields and attributes of the TestParameterSet UTML element

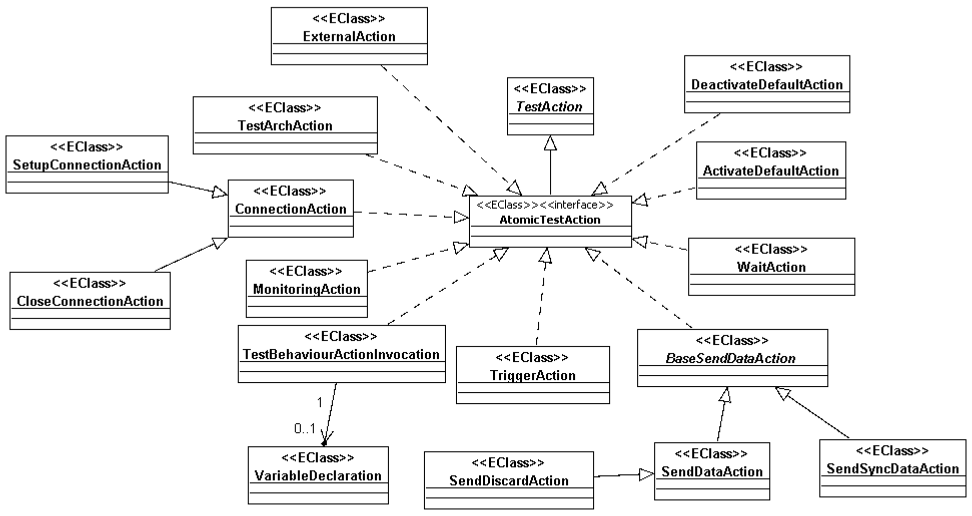
Property	**Description**	**Type**	**Occurence**
testPara-meter	A list of references to previously defined test parameters.	*TestPara-meter* (See Table 5.61)	1..n

5.8 Test Behaviour Design Concepts

As described in the overview to this chapter, test behaviour is designed with UTML through test sequence diagrams and test activity diagrams. Test sequence diagrams are used to describe test interactions between test components and SUT components via ports. To design more complex test behaviours, those interactions can be combined with each other in test activity diagrams.

5.8.1 Basic Principles of UTMLTest Behaviour Design

Action- and Event-based Semantics



<<EClass>> ExternalAction
<<EClass>> TestAction
<<EClass>> DeactivateDefaultAction
<<EClass>> TestArchAction
<<EClass>> SetupConnectionAction
<<EClass>> ActivateDefaultAction
<<EClass>> ConnectionAction
<<EClass>><<interface>> AtomicTestAction
<<EClass>> CloseConnectionAction
<<EClass>> MonitoringAction
<<EClass>> WaitAction
<<EClass>> TestBehaviourActionInvocation
<<EClass>> TriggerAction
<<EClass>> BaseSendDataAction
1
0..1
<<EClass>> VariableDeclaration
<<EClass>> SendDiscardAction
<<EClass>> SendDataAction
<<EClass>> SendSyncDataAction

Figure 5.22: Class Diagram: UTML Atomic Actions

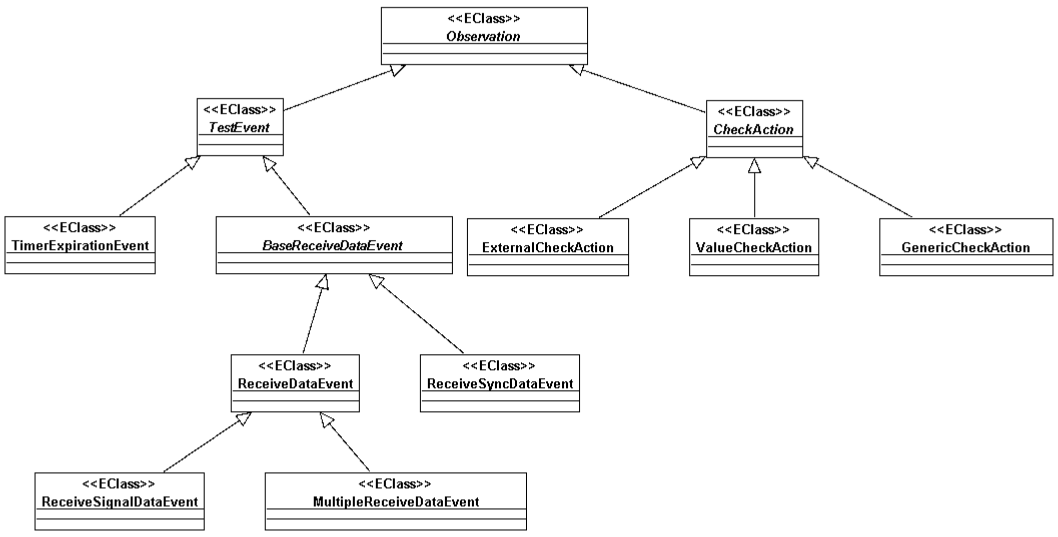
Test behaviour modelling in UTML is based on the same functional abstraction principles as those of the Action-Based or Action-Driven Testing (ABT) methodology introduced by Buwalda et al. [28] [29] and Li Feng et al. [52], respectively. The ABT methodology aims at optimizing the process of defining test cases by enabling test engineers to create executable test scripts from reusable *actions*. Those reusable actions encapsulate complex test scripting elements, so that the production of new test scripts is made easier and faster. The main difference between the ABT methodology and the one proposed in this work is that, instead of stating those actions informally using natural language, a clearly defined pattern-oriented test metamodel is used. This has the advantage that, the test models can remain as intuitive and concise as 'actions', while at the same time benefiting from the validation and transformation facilities that come with a model-driven engineering process.

Additionally to actions, UTML introduces the concept of *events* to design reactions expected to be observed on the SUT to assess that its behaviour is inline with the requirements of the test case.

Actions and Events are owned by test components on which they are run or to which they are attached. For interactive actions, the reference to the owning component can be derived automatically from the port instances involved in the interactive action. For local test actions, structured test actions and action blocks, the owning component is either set manually by selecting a reference to the owning test component from the action's/event's *theComponent* property's dialog or graphically by positioning the action's/event's figure on the hosting component's figure.

UTML Element	**Owning Test Component**
SendDataAction	Test component containing the source port.
ReceiveDataEvent	Test component containing the reception port.

Table 5.63: Rules for Test Action Ownership



<<EClass>> Observation
<<EClass>> TestEvent
<<EClass>> CheckAction
<<EClass>> TimerExpirationEvent
<<EClass>> BaseReceiveDataEvent
<<EClass>> ExternalCheckAction
<<EClass>> ValueCheckAction
<<EClass>> GenericCheckAction
<<EClass>> ReceiveDataEvent
<<EClass>> ReceiveSyncDataEvent
<<EClass>> ReceiveSignalDataEvent
<<EClass>> MultipleReceiveDataEvent

Figure 5.23: Class Diagram: UTML Observation Elements

Declarative Elements Declarative UTML behaviour elements are used to declare items that will be referenced by other elements in a test behaviour model. Declarative elements do not have any semantics in themselves, but their semantics will depend on the context in which they are refered to. The UTML notation defines the following declarative elements for test behaviour:

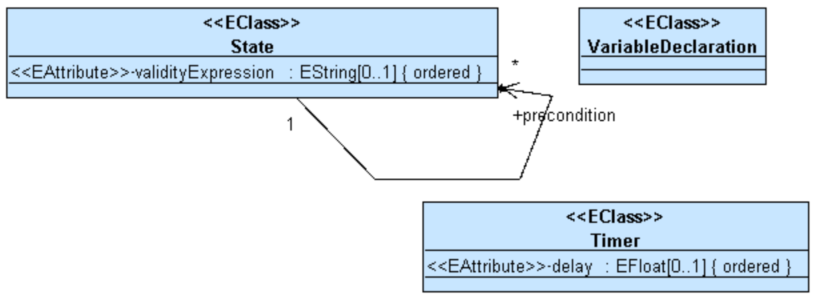

<<EClass>>
State
<<EAttribute>>-validityExpression : EString[0..1] { ordered }
<<EClass>>
VariableDeclaration
*
+precondition
1
<<EClass>>
Timer
<<EAttribute>>-delay : EFloat[0..1] { ordered }

Figure 5.24: Class Diagram: UTML Declarative Behaviour Elements

- *VariableDeclaration* (see Section 5.8.25).
- *Timer*(see Section 5.8.26).
- *State*(see Section 5.8.27).

Interactive actions Interactive actions are those in which more than one component are involved. Examples of interactive actions include UTML *SendDataAction*, *SendReceiveSequence*, *SetupConnectionAction*, etc. Those actions are represented graphically by different forms of lines connecting the port instances involved.

Local Test Actions Local UTML test behaviour actions are those that are associated to a single (owning) test component in the test architecture. Examples of local test actions include:

- *WaitAction*
- *CheckAction* and elements extending it (e.g. *ValueCheckAction*, *ExternalCheckAction*, etc.)
- *TestBehaviourActionInvocation*

Structured Test Actions and Action Blocks

Structured test actions are those that combine several other test actions, e.g. to model more complex test behaviour or a sequence of testing actions.

Architecture-Aware Test Design

The test architecture plays a central role for every UTML test design, as it does not only define the framework in which test behaviour is designed, but is also taken into account for determining which data types and values are relevant or

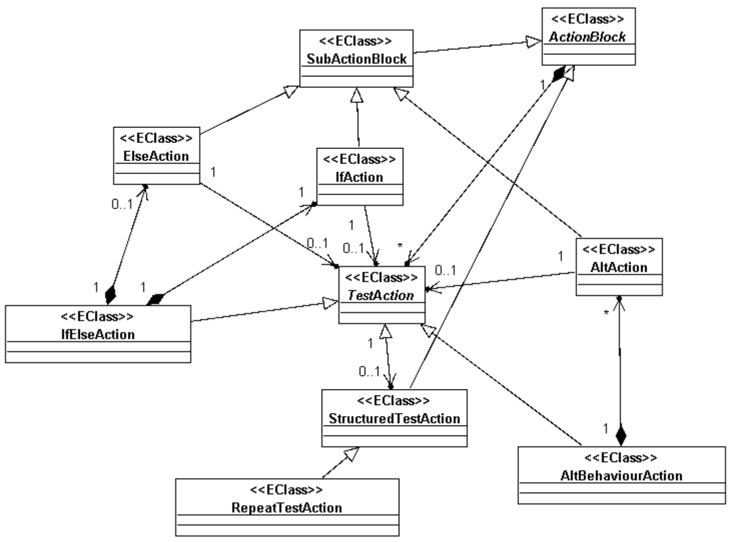


Figure 5.25: Class Diagram: UTML Structured Actions

not for certain test actions or events. For example, depending on the way a SUT is connected to other entities in the architecture and on the accessibility of those connection points with the SUT taken as black-box, the test designer would have to model certain data in the data model (e.g. those exchanged with the SUT via those points and those referenced as fields or values by them), while ignoring other data types and values (e.g. those exclusively used for internal communication between sub-entities of the SUT). Furthermore, while designing test behaviour, the constrained defined by the test architecture will have to be taken into account, to avoid that invalid test behaviour is designed. Examples of invalid test behaviour include:

- Establishing a connection between incompatible ports.
- Sending data via a port designed in the architecture as supporting only incoming communication (IN-Direction).
- Exchanging data over a port that does not support that type of communication.
- Designing an action to be hosted on a component that is inaccessible, given the selected test architecture.

Test Components Behaviour

Each test component is assumed to be a parallel test component in the same sense as in TTCN-3[58]. Therefore, concurrent behaviour can be designed by using more than one test component. First assessments indicate that this principle, combined with the possibility of describing synchronisation/coordination schemes among the test components is sufficient for expressing most test behaviour scenarios, including the more complex ones involving concurrency.

Verdicts Assignment

Following the *Assertion-Based Test Behaviour* design pattern described in Section A.5.1, test behaviour design with UTML focusses on correct test behaviour, while using assertions for checking SUT failures. Therefore the test behaviour model will not attempt to describe all possible branches of an SUT's behaviour tree, but rather describe the correct test behaviour between the test system and the SUT for the implemented test objective. The term "correct" here is used relative to requirements on the SUT and the resulting specification. Thus, it is sufficient for the UTML test behaviour model to describe the branch leading to the ***PASS*** verdict according to the designed test procedure.

The reasons for this principle are twofold:

Firstly, because with the purpose of testing being to uncover failures of the SUT, any test case terminating with a verdict different from ***PASS*** is an indication of a potential failure either on the side of the SUT or that of the test system and therefore, requires further analysis. Based on the selected predefined policy (Cf. Table 5.71), any deviation from the test behaviour specified in a UTML test model would lead to a ***FAIL*** or ***INCONCLUSIVE*** verdict. This means, there should be no such thing as a "positive" ***FAIL*** or ***INCONCLUSIVE*** verdict. Secondly, this principle enables UTML test behaviour models to remain concise, thus increasing their readability, understandability and maintainability. Obviously, this means that the test designer will have to analyse each requirement from the testing perspective rather than simply trying to emulate the SUT's behaviour.

For example, given the following requirement on an SUT:

"The system must respond to invalid input for parameter p of its operation f by throwing an exception e of type *ExceptionType*"

A possible test procedure would consist of the following test steps:

- Instanciate a value suitable v as invalid input for parameter p of operation f.
- Call operation f using v for parameter p.
- Check that an exception of type *ExceptionType* is thrown.

As displayed in the test procedure above, the SUT's exceptional behaviour, though unwished for, is considered to be "correct" with regard to the stated requirement.

Temporal abstraction

UTML defines concepts for supporting various forms of temporal abstraction [126]. The main temporal abstraction is that only the ordering of actions and events matters for test behaviour. Therefore, the notion of time used in UTML is an abstraction from physical time and its mapping to concrete values and associated clocks is left to the target test execution environment.

5.8.2 UTML Test Sequence Diagrams

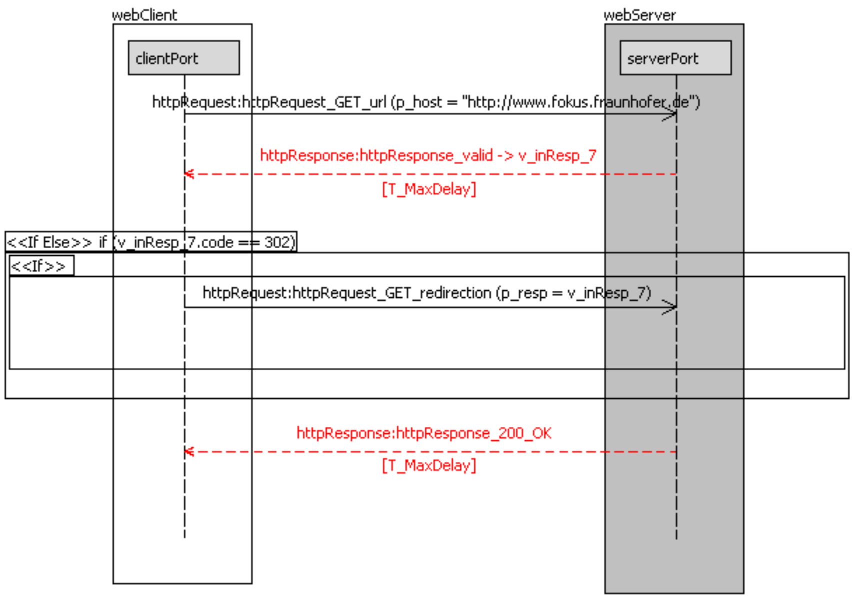

webClient
webServer
clientPort
serverPort
httpRequest:httpRequest_GET_url (p_host = "http://www.fokus.fraunhofer.de")
httpResponse:httpResponse_valid -> v_inResp_7
[T_MaxDelay]
<<If Else>> if (v_inResp_7.code == 302)
<<If>>
httpRequest:httpRequest_GET_redirection (p_resp = v_inResp_7)
httpResponse:httpResponse_200_OK
[T_MaxDelay]

Figure 5.26: Example UTML Test Behaviour Sequence Diagram

Figure 5.26 depicts an example UTML test sequence diagram which illustrates the similarities and the main differences to UML 1.4 sequence diagrams:

Lifelines

In basic UML sequence diagrams (as defined in version 1.4 of the notation), a lifeline is a representation of the lifecycle of an object involved in the interaction modelled in the diagram. Therefore, no distinction is made between the communication points actually used by the object to exchange messages with other entities. This limitation was found to be unsatisfactory for pattern-oriented test design, because the resulting sequence does not visually reflects the test architecture in which it is to occur. Therefore, UTML defines a lifeline as a combination of the lifecycle of a component involved in a test scenario and the port instances owned by that component and used for exchanging messages with other entities in the test architecture. While the port instances (represented in the same way as object lifelines in UML) can be used to attach the starting and end points of messages between the components, the components themselves can be used to attach actions and events that are internal to the test component and thus are not necessarily related to a particular port instance.

Another difference between lifelines in UTML and those in UML sequence diagrams is that the concept of activation boxes does not exist for UTML ports and components. A component is considered to be active over the whole duration of the test scenario, as long as it is not explicitly terminated.

Supported elements and operations

Table 5.64 lists the elements of UML sequence diagrams supported by UTML test sequences and their corresponding equivalents, where applicable.

Messages

Overview of messages supported by UTML test sequences

The first difference between UTML test sequence diagrams and UML sequence diagrams is the lifeline behaviour is designed with UTML along a series of principles that are based on the test design patterns described in Chapter 4. Those principles are presented in the next sections, before the elements of the UTML metamodel dedicated to test behaviour design are described in details.

5.8.3 UTML Test Activity Diagrams

Figure 5.27 displays an example UTML test activity diagram equivalent to the test sequence diagram displayed in Figure 5.26. As depicted in that figure, test activity diagrams are similar to UML activity diagrams, both syntactically and semantically. As one would have expected, the UML *signal* elements are used to represent interaction events such as *SendDataAction* and *ReceiveDataEvent*. Meanwhile, UML *activity* elements are used for all local test behaviour actions. Additionally, UTML test activity diagrams may be used to design complex test execution scenarios, with *activity* elements used to represent test cases or invocations of structured test behaviour elements. Finally, given the fact that concurrent behaviour is designed in UTML through parallel components, the UML activity diagrams' *fork* and *join* elements are not supported.

5.8.4 TestBehaviourModel

Description

The ***TestBehaviourModel*** element models a UTML test behaviour model. A ***TestBehaviourModel*** instance is equivalent to the behavioural part of a test

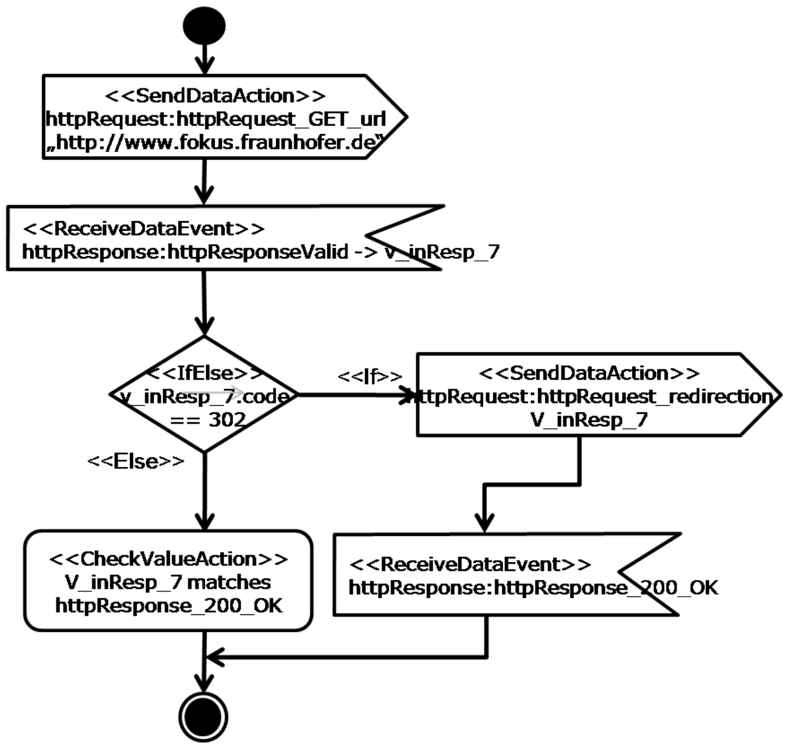

<<SendDataAction>>
httpRequest:httpRequest_GET_url „http://www.fokus.fraunhofer.de"
<<ReceiveDataEvent>>
httpResponse:httpResponseValid -> v_inResp_7
<<IfElse>>
v_inResp_7.code == 302
<<If>>
<<SendDataAction>>
httpRequest:httpRequest_redirection V_inResp_7
<<Else>>
<<CheckValueAction>>
V_inResp_7 matches httpResponse_200_OK
<<ReceiveDataEvent>>
httpResponse:httpResponse_200_OK

Figure 5.27: Example UTML Test Behaviour Activity Diagram

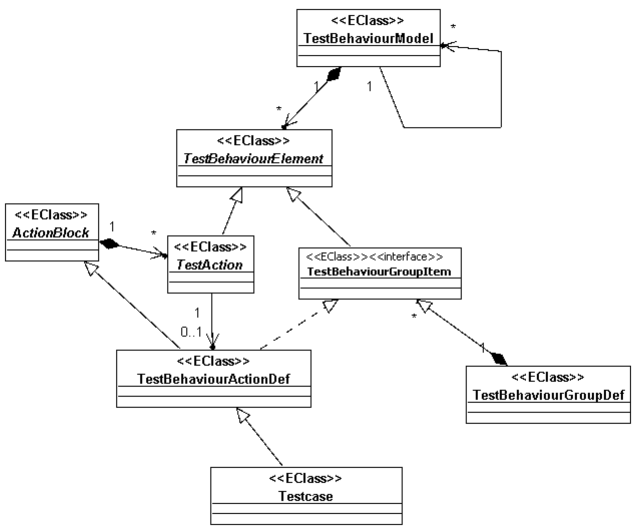

<<EClass>> TestBehaviourModel
*
1
1
*
<<EClass>> TestBehaviourElement
<<EClass>> ActionBlock
1
*
<<EClass>> TestAction
<<EClass>><<interface>> TestBehaviourGroupItem
1
0..1
*
1
<<EClass>> TestBehaviourActionDef
<<EClass>> TestBehaviourGroupDef
<<EClass>> Testcase

Figure 5.28: Class Diagram: UTML Main Containers for Test Behaviour

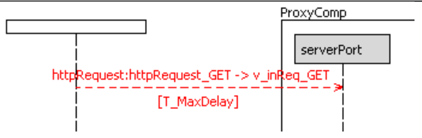
UML sequence diagram element	**UTML equivalent**	**Syntax**
Lifelines	Component instances and contained port instances	see Figure 5.26
Messages	See Table 5.65	
Self-Messages	none	N/A
Lost messages	none	N/A
Found Messages	*ReceiveDataEvent* or *ReceiveSyncDataEvent* from unspecified source	 ProxyComp serverPort httpRequest:httpRequest_GET -> v_inReq_GET [T_MaxDelay]
Life line start and end	none	N/A
Duration and time constraints	Duration and time constraints are expressed in UTML for expected events only through the association with timers.	N/A
Alternative fragment	*IfElseAction*(see Section 5.8.57)	See Figure 5.37
Loop fragment	*RepeatTestAction* (see Section 5.8.56)	See Figure 5.36
Option fragment	*AltBehaviourAction* (see Section 5.8.60)	See Figure 5.38
Gate	none	N/A

Table 5.64: Overview of UML sequence diagram elements supported by UTML test sequences

suite.

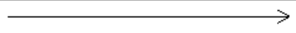
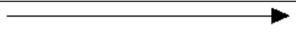
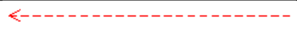
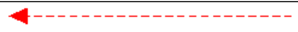
Message	**Concrete Syntax**	**Effect**
SendData-Action		Send asynchronous data (message or signal).
SendSync-DataAction		Send synchronous data (operation).
Receive-DataEvent		Expect asynchronous data (message or signal).
Receive-SyncData-Event		Expect synchronous data (operation).

Table 5.65: Overview of Messages supported by UTML Test Sequences

Constraints

Constraint (Optional) A test behaviour model should define a default timer to be used for actions and events to which no timer is explicitly associated.

```
self.defaultTimer.oclIsTypeOf(OclVoid) = false
```

Syntax

The *TestBehaviourModel* element extends the *BasicTestModel* (See Table 5.3.2) element defined previously

Table 5.66: Properties of the TestBehaviourModel UTML element

Property	**Description**	**Type**	**Occu-rence**
testArchi-tecture-Types-Model	References to test models containing basic test architecture elements.	*TestArchi-tecture-TypesModel* (See Table 5.20)	0..n

testDataModel	References to related test data models.	*TestDataModel* (See Table 5.37) from package *test_data*	0..n
testBehaviourModel	References to related test behaviour models.	*TestBehaviourModel* (See Table 5.66)	0..n
testBehaviourElement	Contained test behaviour model elements.	*TestBehaviourElement* (See Section 5.8.49)	0..n
testProceduresModel	Reference to related test procedures models.	*TestProceduresModel* (See Table 5.17)	0..n
timer	Reference to a timer to serve as default timer for all elements contained in the test behaviour model, for which though a timer is recommended, none was explicitly specified. For example, if a *ReceiveDataEvent* element is designed in a test behaviour model and the *timer* property of that element is left unspecified, then, if provided, the containing ***TestBehaviourModel*** element's default timer applies.	*Timer* (See Table 5.83)	0..1

5.8.5 TestBehaviourGroupItem

Description

The ***TestBehaviourGroupItem*** element is an abstract classifier that serves as base for members of groups in a UTML test behaviour model. Therefore, it provides the base for the partitioning mechanism in test behaviour models.

5.8.6 TestBehaviourGroupDef

Description

The ***TestBehaviourGroupDef*** element models a group in a UTML test behaviour model. A ***TestBehaviourGroupDef*** element is a mean for partitioning a test model in a manner similar to UML packages. It may contain other ***TestBehaviourGroupDef*** elements as sub-groups, as well as other test behaviour model elements extending the *TestBehaviourGroupItem* abstract classifier as children.

Semantics

The ***TestBehaviourGroupDef*** element has no associated semantic and is just a container for organising elements contained in a test behaviour model

Syntax

The *TestBehaviourGroupDef* element extends the following elements of the metamodel:

- *DescribedElement* (See Table 5.4)
- *ElementWithUniqueID* (See Section 5.9)
- TestBehaviourGroupItem (See Section 5.8.5)

Table 5.67: Properties of the TestBehaviourGroupDef UTML element

Property	**Description**	**Type**	**Occurence**
testBehaviourGroupItem	Test behaviour elements contained in the group.	*TestBehaviourGroupItem* (See Section 5.8.5)	0..n

5.8.7 Verdict

Description

The ***Verdict*** UTML element is a classifier for the possible kinds of verdict assignable in a test model.

Semantics

Following the *Assertion-Driven Test Behaviour Design* pattern described in Section A.5.1, the ***Verdict*** has no semantics in itself, but simply defines a value that can be used as an attribute for other elements. In fact, at the moment, the only UTML element requiring a verdict as attribute is the ***StopAction*** element described in Table 5.88. For all other behaviour elements the resulting verdict is derived from the assertions made on the SUT's responses.

The verdict assignment mechanism in UTML follows the same principles as defined for the TTCN-3 notation. It is therefore assumed that each component instance maintains a local verdict that will be influenced by the actions and events defined for that component. The overall verdict for a test case will be calculated by applying the verdict overwriting rule defined in the TTCN-3 standard specification [58], which stipulates that once a verdict different from *PASS* has been set in a test case, then that verdict cannot be overwritten back to *PASS* again.

Syntax

Table 5.68: The Verdict UTML element

Literal	**Description**
ERROR	An ERROR verdict.
NONE	The NONE verdict is the default value for instances of the *Verdict* UTML element.
INCONC	Indicates that no clear verdict could be assigned.
FAIL	Indicates that the SUT did not meet the requirements implemented by the test case.
PASS	Indicates that the SUT completed the test successfully and thus meets the assessed test objective.

5.8.8 BehaviourPatternKind

Description

The ***BehaviourPatternKind*** UTML element is an enumeration used to classify test behaviour patterns.

Syntax

Table 5.69: Properties of the BehaviourPatternKind UTML element

Property	**Description**
SEND_RECEIVE	Indicates a send-receive test sequence.
TRIGGER_RE-CEIVE	Indicates a trigger-receive test sequence, i.e. one whereby the SUT is triggered to send data which the test system then evaluates.
SEND_DISCARD	Indicates a send-discard test sequence. A send-discard test sequence is one whereby the test system sends some data to the SUT and expects those data to be discarded. I.e. the SUT is expected to ignore the data and not to respond to it.
CUSTOM	This literal is for user-defined test behaviour patterns.

5.8.9 BehaviourActionKind

Description

The ***BehaviourActionKind*** element is an *enumeration* used to classify test behaviour actions in categories, so that the main purpose of each action would be easily recognizable, without having to look at the details of the code. Table 5.70 lists the literals of that enumeration representing categories currently defined in UTML and describes each of them.

Syntax

Table 5.70: Properties of the BehaviourActionKind UTML element

Property	**Description**

GENERIC	The GENERIC literal is the default value for the ***BehaviourActionKind*** element. It should be used for any test behaviour action that cannot be classified as belonging to any of the other categories.
VALUE_COMPUTATION	Should be used for a test action whose main purpose is to compute a value required for testing. For example, a function for calculating a Cyclic Redundancy Check () for a given message in protocol testing.
STATE_CHANGE	Used for a test action whose purpose is to trigger a state transition at the SUT.
STATE_CHECK	Used for an action whose purpose is to check that an SUT is in a given state.
ARCHITECTURE_SETUP	For a test action used to setup a test architecture.
ARCHITECTURE_TEARDOWN	For a test action used to tear down an existing (or running) test architecture.
EXPECT_MESSAGE	For an action used to expect a message from an SUT.
SEND_MESSAGE	For an action used to send data to an SUT.
EXTERNAL	For an action which is executed beyond the borders of the test system, but which has an impact on it or on the SUT.
MONITOR	Used to monitor a certain state on the SUT.

5.8.10 PolicyKind

Description

The ***PolicyKind*** element is an *enumeration* used to classify the kinds of policy to follow for assigning a test verdict following assertions and observation of expected events from the SUT. Therefore the ***PolicyKind*** element is used as property of UTML event elements such as the *ReceiveDataEvent* (See Table 5.99), the *TimerExpirationEvent* elements .

Syntax

Table 5.71: Properties of the PolicyKind UTML element

Property	**Description**
STRICT	STRICT policy means any deviation from the expected behaviour from the SUT would lead to a *fail* verdict.
RELAX	In case of RELAX policy, whenever the SUT's behaviour does not match the described behaviour, then a verdict of *inconclusive* is applied.

5.8.11 TestBehaviourActionDef

Description

The ***TestBehaviourActionDef*** element is a key element of the UTML metamodel. It is equivalent to a function definition in a functional programming language. Therefore it might contain several types of other elements to represent a complete test behaviour.

Semantics

The ***TestBehaviourActionDef*** models test behaviour that can be reused in different context by invocation with the *TestBehaviourInvocationAction* element. For a ***TestBehaviourActionDef*** to be invoked in another one or in a test case, the test architectures associated to both elements must be equal or compatible with each other. The rule for test architecture compatibility for behaviour invocation is as follows:

If a behaviour f is defined on an architecture A_f and another behaviour g is defined on a test architecture A_g, then an invocation of g may only contain an invocation of f if A_f equals A_g or A_f is a subset A_g. With A_f being a subset of A_g if it contains the same component instances and features the same connections as A_g. ***TestBehaviourActionDef*** element may be associated to a component type definition. This implies that all elements contained in the component's type definition (e.g. port instance, declared variables, timers, etc.) are accessible from the behaviour definition.

Constraints

Constraint (Optional) A ***TestBehaviourActionDef*** should be associated to static test architecture.

```
self.testArchitecture.oclIsTypeOf(OclVoid) = false
```

Constraint (Optional) A ***TestBehaviourActionDef*** should contain at least one component belonging to the SUT.

```
self.componentInstance
-> exists(kind = utml::test_architecture::ComponentKind::SUT)
```

Syntax

The *TestBehaviourActionDef* element extends the following elements of the meta-model:

- *DescribedElement* (See Table 5.4)
- *TestBehaviourGroupItem* (See Section 5.8.5)

Table 5.4)

Table 5.72 list the various elements possibly contained in a ***TestBehaviour-ActionDef*** element.

Table 5.72: Properties of the TestBehaviourActionDef UTML element

Property	**Description**	**Type**	**Occurrence**
testArchitecture	A reference to the test architecture required for this function definition.	*TestArchitecture* (See Table 5.31)	0..1
componentInstance	A test component instance on which the ***TestBehaviourActionDef*** will be running. This concept of attaching a behaviour to a test component has been borrowed from the TTCN-3 language [58].	*TestComponentInstance* (See Table 5.29)	1..1
connections	Connections between test component instances involved in this TestBehaviourActionDef	*Connection* (See Table 5.30)	1..1

componentType	Type of component on which this TestBehaviourActionDef might be run (See TTCN-3 [58])	*ComponentType* (See Table 5.23) from package *test_architecture*	1..1
beginState	If a given SUT state is required as precondition for the TestBehaviourActionDef to be executed, it should be selected here.	*State* (See Table 5.84)	0..1
testAction	Test actions contained in the behaviour action definition.	*TestAction* (See Table 5.75) from package *test_behaviour*	0..n
endState	Final state of the SUT afte r the execution.	*State* (See Table 5.84) *test_behaviour*	0..1
variableDeclaration	Contained local variable declarations.	*VariableDeclaration* (See Section 5.8.25)	0..n
parameterDeclaration	A list of parameters for the ***TestBehaviourActionDef***. This is equivalent to function parameters in generic purpose programming languages.	*ParameterDeclaration* (See Table 5.49)	0..n
name	An identifier for the TestBehaviourActionDef.	xsd:string	1..1
actionKind	Classifier representing the main purpose of this ***TestBehaviourActionDef***	*BehaviourActionKind* (See Table 5.70)	0..1

response-Def	A definition of the response returned when this *TestBehaviouActionDef* is invoked.	*Operation-Response-Def* (See Table 5.51)	0..1

5.8.12 TestBehaviourActionInvocation

Description

The ***TestBehaviourActionInvocation*** element models the invocation of a previously defined test behaviour action element. It can be viewed as the equivalent to the invocation of a function in a classical programming language.

Constraints

Constraint A test behaviour definition may only be invoked on a test component if the component's type is compatible to the component type for which the test behaviour was designed.

```
(self.testBehaviourActionDef.oclIsTypeOf(OclVoid) = false
and
self.testBehaviourActionDef.componentType
.oclIsTypeOf(OclVoid) = false)
implies
((self.theComponent.oclIsTypeOf(OclVoid) = false
and self.theComponent.type.name
= self.testBehaviourActionDef.componentType.name)
    or
(self.theComponent.oclIsTypeOf(OclVoid) = false
and self.theComponent.type.baseComponentType -> isEmpty() = false
and self.theComponent.type.baseComponentType ->
exists(name=self.testBehaviourActionDef.componentType.name) ))
```

Syntax

The *TestBehaviourActionInvocation* element extends *AtomicTestAction* (See Section 5.8.15)

Table 5.73: Properties of the TestBehaviourActionInvocation UTML element

Property	**Description**	**Type**	**Occu-rence**

testBehaviour-Action-Def	A reference to the testBehaviour-ActionDef element to be invoked.	*TestBehaviour-ActionDef* (See Table 5.72)	1..1
parameterDef	A set of values to be passed as parameters for the invocation.	*Parameter-Def* (See Table 5.60)	0..n

5.8.13 Testcase

Description

The ***Testcase*** element models a test case in the UTML metamodel.

Semantics

The *Testcase* element extends the *test_behaviour:TestBehaviourActionDef* element(See Table 5.72) and just like that element, the ***Testcase*** element defines a scope, in the sense that it may contain declarative elements reachable only within the testcase.

There is no requirement for a ***Testcase*** element to explicitly assign a verdict.

Constraints

Constraint (Optional) A test case should define a component type as main component type. This constraint is adopted from the TTCN-3 notation.

```
self.componentType.oclIsTypeOf(OclVoid) = false
```

Constraint (Optional) A test case should define a component type as system component type to label which of the components in a test architecture represent the SUT.

```
self.systemComponentType.oclIsTypeOf(OclVoid) = false
```

Constraint (Optional) The main test component type and the system component type should be different from each other.

```
self.componentType <> self.systemComponentType
```

Constraint (Optional) A description should be provided for every test case.

```
self.description.oclIsTypeOf(OclVoid) = false and self.description <>
'TODO: Add description'
```

Constraint (Optional) Each test case should contain at least one test component instance belonging to the SUT.

```
self.oclAsType(utml::test_behaviour::TestBehaviourActionDef)
.componentInstance ->
exists(kind = utml::test_architecture::ComponentKind::SUT)
```

Constraint (Optional) Each test case should be associated to at least one test objective or test procedure.

```
self.testObjective -> size() > 0
or self.testProcedure.oclIsTypeOf(OclVoid) = false
```

Constraint (Optional) A test case should not be directly associated to a test procedure and test objectives at the same time. Otherwise, that may lead to conflicts with regard to traceability, because the test objectives referred to by the test procedure may be different from the ones directly associated to the test case.

```
(self.testObjective -> size() > 0
implies (self.testProcedure.oclIsTypeOf(OclVoid) = true))
and
(self.testProcedure.oclIsTypeOf(OclVoid) = false
implies ( self.testObjective -> size() = 0))
```

Constraint (Optional) Each test case should contain at least one test action with the *passCriterium* property set to *true*.

```
self.testAction -> exists(
(oclIsTypeOf(utml::test_behaviour::SendDataAction)
and
 oclAsType(utml::test_behaviour::SendDataAction)
 .passCriterium = true
 )
 or
(oclIsTypeOf(utml::test_behaviour::CheckAction)
and
 oclAsType(utml::test_behaviour::CheckAction)
 .passCriterium = true
 )
 or
(oclIsTypeOf(utml::test_behaviour::ValueCheckAction)
and
 oclAsType(utml::test_behaviour::ValueCheckAction)
 .passCriterium = true
 )
 or
(oclIsTypeOf(utml::test_behaviour::ExternalCheckAction)
and
 oclAsType(utml::test_behaviour::ExternalCheckAction)
 .passCriterium = true
 )
 or
(oclIsTypeOf(utml::test_behaviour::GenericCheckAction)
and
 oclAsType(utml::test_behaviour::GenericCheckAction)
 .passCriterium = true
 )
or
(oclIsTypeOf(utml::test_behaviour::ReceiveDataEvent)
and
 oclAsType(utml::test_behaviour::ReceiveDataEvent)
 .passCriterium = true
 )
 or
(oclIsTypeOf(utml::test_behaviour::TestEvent)
and
 oclAsType(utml::test_behaviour::TestEvent)
 .passCriterium = true
 )
)
```

Syntax

Table 5.74: Properties of the Testcase UTML element

Property	**Description**	**Type**	**Occu-rence**

systemComponentType	The type of system component for which this test case is applicable.	*ComponentType* (See Table 5.23)	0..1
testProcedure	A reference to a test procedure describing the testcase's scenario. This field is a mean for ensuring traceability of test cases to test objectives through the reference to those in the test procedure.	*TestProcedure* (See Table 5.19) from package *test_procedures*	0..1 [1]
testObjective	A reference to a test objective or a collection thereof covered by this test case. This is an alternative to the *TestProcedure* field for ensuring traceability between test cases and requirements.	*TestObjective* (See Table 5.15)	0..1[2]
testParameterSet	A reference to a set of parameters which is associated to this test case. A parameter set defines a series of static preconditions that must be fullfiled for the test case to be executed.	*TestParameterSet* (See Table 5.62)	0..1
notes	Free textual notes.	xsd:string	0..1

5.8.14 TestAction

Description

The ***TestBehaviourAction*** element is an abstract classifier providing the base for the test action concept of UTML. UTML test behaviour models consist of test actions being executed on test component instances interconnected within a given test architecture. Therefore, the UTML meta-model defines ***TestBehaviourAction*** as an abstract classifier from which all test actions will inherit.

Syntax

The *TestAction* element extends ***TestBehaviourElement*** type(See Section 5.8.49).

[1]Mandatory if no reference to covered test objectives was provided.
[2]Mandatory if no reference to test procedure was provided.

Table 5.75: Properties of the TestAction UTML element

parentActionDef	If the test action is part of a test action definition element, then this field contains a reference to the owning test action definition.	*TestBehaviourActionDef* (See Table 5.72)	0..1
theTestComponent	A reference to the owning test component. As described in Section 5.8.1, each test action is owned by a test component.	*ComponentInstance* (See Table 5.29)	1..1
parentAction	If the test action is a subaction, then this field contains a reference to the parent test action.	*StructuredTestAction* (See Section 5.8.55)	0..1

5.8.15 AtomicTestAction

A ***AtomicTestAction*** element is an abstract classifier modelling an atomic test action in UTML, i.e. one that cannot be decomposed into many sub actions.

Constraints

```
(self.theComponent.oclIsTypeOf(OclVoid) = false and
self.theComponent.type.oclIsTypeOf(OclVoid) = false and
self.parentAction.oclIsTypeOf(OclVoid) = false
and self.parentAction.theComponent.type.oclIsTypeOf(OclVoid))
self.theComponent.type.name = self.parentAction
.theComponent.type.name
```

5.8.16 ConnectionAction

Description

The ***ConnectionAction*** is an abstract classifier providing the base for modelling actions that have an impact on connections in a test architecture.

Syntax

The *ConnectionAction* element extends *test_behaviour:AtomicTestAction* (See Section 5.8.15)

5.8.17 SetupConnectionAction

Description

The ***SetupConnectionAction*** element models an action for setting up a connection between two ports.

Semantics

A ***SetupConnectionAction*** element may only be created between two ports, if the directions and data types supported by those ports allow them to be connected with each other.

Syntax

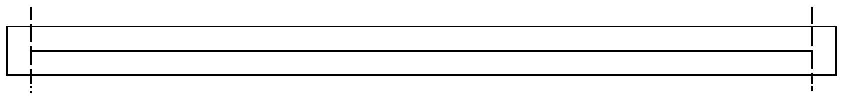


Figure 5.29: SetupConnectionAction in UTML Test Behaviour Sequence Diagram

UTML *SetupConnectionAction* elements are represented graphically by the symbol displayed on figure 5.29 and drawn as a link between the source port instance and the target port instance of the connection being setup.

The *SetupConnectionAction* element extends the following elements of the metamodel:

- *ConnectionAction* (See Section 5.8.16)
- *Connection* (See Section 5.30)

Table 5.76: Fields and attributes of the SetupConnectionAction UTML element

Property	**Description**	**Type**	**Occurrence**
sourcePort	A reference to the source port instance.	*PortInstance* (See Table 5.28)	1..1

destPort	A reference to the destination port instance.	*PortInstance* (See Table 5.28) from package *test_architecture*	1..1
architecture	A reference to the test architecture in which the connection is created.	*TestArchitecture* (See Table 5.31)	1..1 [3]

5.8.18 CloseConnectionAction

Description

The ***CloseConnectionAction*** element models an action for closing a connection between two ports.

Syntax

The *CloseConnectionAction* element extends the *ConnectionAction* (See Section 5.8.16) element of the metamodel.

Table 5.77: Fields and attributes of the CloseConnectionAction UTML element

Property	**Description**	**Type**	**Occurence**
connection	A reference to the connection to be closed.	*Connection* (See Table 5.30)	1..1

5.8.19 DefaultBehaviourAction

The *DefaultBehaviourAction* element extends *test_behaviour:TestAction* (See Table 5.75)

A ***DefaultBehaviourAction*** element models an alternative branch in a default test behaviour in UTML. The default test behaviour mechanism used in

[3] Implicit (Implicit properties like this one can be derived implicitly from other related properties of the same element and thus will not have to be explicitly specified by the test designer. For example, in this particular case, the reference to the containing test architecture can be retrieved automatically while creating the connection between two ports.)

UTML is a translation of the *altstep-default behaviour* concept introduced in TTCN-3.

Table 5.78: Properties of the DefaultBehaviourAction UTML element

Property	**Description**	**Type**	**Occurence**
triggeringEvent	The event triggering the alternative branch to be chosen.	*TestEvent* (See Section 5.8.21)	1..1
subAction	A set of test actions to be performed if the branch is selected.	*AtomicTestAction* (See Section 5.8.15)	0..n

5.8.20 Observation

Description

A ***Observation*** element is an abstract classifier defined in the UTML metamodel as base classifier for observable test behaviour elements.

Syntax

Table 5.79: Fields and attributes of the Observation UTML element

Property	**Description**	**Type**	**Occurence**

passCriterium	A boolean value indicating whether or not this event should be considered as a criterium for assigning a PASS verdict or not. The motivation for this property stems from the fact that it is sometimes required to label some test steps in a test procedure as critical for the overall test case verdict. The concrete interpretation of this property will depend on the target testing infrastructure and on the chosen test strategy. A possible interpretation would consist in setting the test case verdit to PASS, if this property was set to true and the event's assertions were successful. Otherwise, test execution will simply proceed without any verdict being set. However the successful assertion should be documented for traceability, e.g. in the form of corresponding log traces.	*PortInstance* (See Table 5.28)	1..1
policyKind	Policy for setting the verdict after this event's assertions are checked.	*PolicyKind* (See Table 5.71)	1..1

5.8.21 TestEvent

Description

A ***TestEvent*** element is an abstract classifier defined in the UTML metamodel as base classifier for observable test events.

Syntax

The *TestEvent* element extends *test_behaviour:Observation* (See Section 5.8.20)

5.8.22 DataReceptionEvent

Description

The ***DataReceptionEvent*** element models the reception of asynchronous data at a component from another (Test-/SUT-) component.

Semantics

Syntax

The *DataReceptionEvent* element extends *test_behaviour:TestEvent* (See Section 5.8.21)

Table 5.80: Properties of the DataReceptionEvent UTML element

Property	**Description**	**Type**	**Occurence**
portInstance	A reference to the reception port instance.	*PortInstance* (See Table 5.28)	1..1
incomingData	A reference to a test data instance as expected incoming data.	*TestDataInstance* (See Table 5.54)	1..1

5.8.23 TimerExpirationEvent

Description

The ***TimerExpirationEvent*** element models an event indicating the expiration of a timer in a UTML test behaviour model.

Syntax

The *TimerExpirationEvent* element extends *test_behaviour:TestEvent* (See Section 5.8.21)

Table 5.81: Fields and attributes of the TimerExpirationEvent UTML element

Property	**Description**	**Type**	**Occurence**
timer	A reference to the timer to expire.	*Timer* (See Table 5.83)	1..1

5.8.24 DefaultBehaviourDef

Description

The ***DefaultBehaviourDef*** element has been borrowed from the TTCN-3 notation, in which it is used to define a behaviour that may be checked implicitly on the SUT if the explicitly defined behaviour does not match expectations.

Semantics

Syntax

The *DefaultBehaviourDef* element extends *DescribedElement* (See Table 5.4)

Table 5.82: Properties of the DefaultBehaviourDef UTML element

Property	**Description**	**Type**	**Occurence**
defaultAction	The default actions representing the test behaviour alternatives.	*DefaultBehaviourAction* (See Table 5.78)	
componentType	A reference to the type of test component for which the default behaviour is applicable.	*ComponentType* (See Table 5.23)	1..1
id	An identifier for the default behaviour definition.	xsd:string	1..1

5.8.25 VariableDeclaration

Description

The ***VariableDeclaration*** element models the declaration of a variable in a UTML test behaviour model.

Semantics

VariableDeclaration elements can be used to store data received through *ReceiveDataEvent* or resulting from a *TestBehaviourInvocation* element. The value stored in a ***VariableDeclaration*** can be used for any test behaviour in which a value may be required.

Syntax

The *VariableDeclaration* element extends the following elements of the metamodel:

- *TestBehaviourElement* (See Section 5.8.49)
- *ValueInstance* (See Section 5.53)

5.8.26 Timer

Description

The ***Timer*** element models a timer declaration in the UTML metamodel

Semantics

Timers may be declared as default global timers for a whole test behaviour model, as local timers inside a component type or instance thereof or as local timers for a given *TestBehaviourActionDef* or *Testcase* element.

Constraints

```
((self.delay.oclIsTypeOf(OclVoid) = true or self.delay  <= 0)
and self.delayValue.oclIsTypeOf(OclVoid) = true) = false
```

Syntax

The *Constraints* element extends *DescribedElement* (See Table 5.4)

Table 5.83: Properties of the Timer UTML element

Property	**Description**	**Type**	**Occu-rence**
delay	A value for the timer's duration.	xsd:float. For more flexibility, the unit for the timer delay values is left open and can be decided at the later stage.	1..1

name	An identifier for the timer.	xsd:string	1..1

5.8.27 State

Description

A ***State*** in UTML is a declarative element describing a state in which the SUT may find itself at a given point in time of its behaviour. Based on those information, preambles can be executed to put the SUT in the required state, if that was defined as a precondition for the test case.

Syntax

The *State* element extends the following elements of the metamodel:

- *DescribedElement* (See Table 5.4)
- *UniqueNamedElement* (See Table 5.7)

Table 5.84: Properties of the State UTML element

Property	**Description**	**Type**	**Occurence**
testArchitecture	A reference to a test architecture for which this state is applicable.	*TestArchitecture* (See Table 5.31)	0..1
componentType	A reference to the type of component for which this state is applicable.	*ComponentType* (See Table 5.23)	1..1
triggeringActions	A sequence of actions that trigger the component to enter the state. The actions listed here are to be provided in their chronological order of occurence.	*TestBehaviour-ActionInvocation* (See Table 5.73)	0..n
precondition	A list of references to other state definitions that are preconditions to this state.	*State* (See Table 5.84)	0..n
validityExpression	A character string representation of an expression that can be used to verify that a given component is in this state.	xsd:string	0..1

5.8.28 StartTimerAction

Description

The ***StartTimerAction*** element models an action for starting a timer.

Syntax

The *StartTimerAction* element extends *test_behaviour:AtomicTestAction* (See Section 5.8.15)

Table 5.85: Fields and attributes of the StartTimerAction UTML element

Property	**Description**	**Type**	**Occurrence**
timer	A reference to the timer to be started.	*Timer* (See Table 5.83)	1..1

5.8.29 StopTimerAction

Description

The ***StopTimerAction*** element models an action for stopping a timer.

Syntax

The *StopTimerAction* element extends *test_behaviour:AtomicTestAction* (See Section 5.8.15)

Table 5.86: Properties of the StopTimerAction UTML element

Property	**Description**	**Type**	**Occurrence**
timer	A reference to the timer to be stopped.	*Timer* (See Table 5.83)	1..1

5.8.30 WaitAction

Description

The ***WaitAction*** element models a behaviour action whereby the associated test component is requested to suspend its behaviour for a certain delay before resuming to its other actions.

Semantics

The suspension mechanism for the ***WaitAction*** element is based either on a delay provided by the user or on a timer referrence. At the moment the UTML provides no notion of absolute time, however that could be added at a later stage, if required. Then, a corresponding property would be added to the current structure and the constraints will be extended accordingly.

Constraints

```
((self.delay.oclIsTypeOf(OclVoid) = true or self.delay <= 0)
and self.timer.oclIsTypeOf(OclVoid) = true) = false
```

Syntax

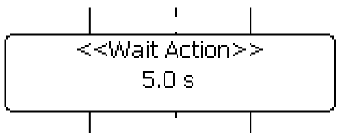


Figure 5.30: WaitAction in UTML Test Behaviour Sequence Diagram

The *WaitAction* element extends *test_behaviour:AtomicTestAction* (See Section 5.8.15)

Table 5.87: Properties of the WaitAction UTML element

Property	Description	Type	Occurence
delay	A value indicating the duration of the delay.	xsd:float	0..1
timer	A reference to a declared *Timer*. If the ***delay*** property has not been set, then the associated test component will suspend its behaviour until the referred timer expires.	***Timer*** (See Table 5.83)	0..1

5.8.31 StopAction

Description

The ***StopAction*** element models an action for terminating test execution immediately.

Semantics

The ***StopAction*** element terminates test execution immediately, assigning it the selected verdict. Therefore, the ***StopAction*** element provides the only mechanism for setting a test verdict explicitly. However, the overall test verdict is calculated according to the rule defined in TTCN-3 for verdict assignment.

Syntax

The *StopAction* element extends *test_behaviour:AtomicTestAction* (See Section 5.8.15)

Table 5.88: Properties of the StopAction UTML element

Property	**Description**	**Type**	**Occurence**
verdict	The verdict to assign.	***Verdict*** (See Table 5.68)	1..1

5.8.32 ExternalAction

Description

The ***ExternalAction*** element models an external test action. I.e. one that is to happen outside of the test system.

Syntax

The *ExternalAction* element extends *test_behaviour:AtomicTestAction* (See Section 5.8.15)

Table 5.89: Fields and attributes of the ExternalAction UTML element

Property	**Description**	**Type**	**Occurence**
instructions	A message describing instructions to be transmitted to the entity running the tests.	xsd:string	1..1

5.8.33 MonitoringAction

Description

The ***MonitoringAction*** element models an action for monitoring a given component until a certain state is reached.

Syntax

The *MonitoringAction* element extends *test_behaviour:AtomicTestAction* (See Section 5.8.15)

Table 5.90: Fields and attributes of the MonitoringAction UTML element

Property	**Description**	**Type**	**Occurence**
breakingState	A reference to this state that, if entered, will cause the monitoring to stop.	*State* (See Table 5.84)	1..1

5.8.34 SendDiscardAction

Description

The ***SendDiscardAction*** element models an action whereby the a component sends data to another one and expects those data to be discarded without notice.

Semantics

The ***SendDiscardAction*** element inherits all the constraints defined for the *SendDataAction* element.

Syntax

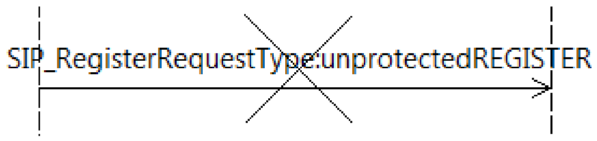


Figure 5.31: SendDiscardAction in UTML Test Behaviour Sequence Diagram

UTML *SendDiscardAction* elements are represented graphically by a crossed arrow linking the source port instance to the target port instance.

Figure 5.31 shows an example *SendDiscardAction* in its graphical representation. As displayed on that figure, the label associated to a *SendDiscardAction* follows the same format as for a *SendDataAction* discussed previously.

The *SendDiscardAction* element extends *test_behaviour:SendDataAction* (See Table 5.96)

Table 5.91: Properties of the SendDiscardAction UTML element

Property	**Description**	**Type**	**Occurrence**
timer	A reference to the timer, which if it expires, implies that the message has been discarded.	*Timer* (See Table 5.83)	1..1
allowed-Response	References to test responses that might be allowed from the other party, with the message still being considered discarded.	*Response* (See Table 5.92)	0..n

5.8.35 Response

Description

The ***Response*** element is a helper classifier modelling a potential response from an SUT after a stimulus.

Semantics

The ***Response*** element has no semantics in itself and is only used in combination with structured test behaviour actions to design a response from an SUT, e.g. following a stimulus by a test component.

Syntax

Table 5.92: Properties of the Response UTML element

Property	**Description**	**Type**	**Occurrence**
port	A reference to a port instance via which the response is expected.	*PortInstance* (See Table 5.28)	1..1

expectedData	A reference to a test data instance designing the expected data.	*MessageTestDataInstance* (See Table 5.55)	1..1
unexpectedData	Reference to data instances that are disallowed as response.	*MessageTestDataInstance* (See Table 5.55)	0..n

5.8.36 OperationOutput

Description

The ***OperationOutput*** element is a helper element used to design the expected output for an operation call.

Syntax

Table 5.93: Properties of the OperationOutput UTML element

Property	**Description**	**Type**	**Occurence**
value	A reference to a previously defined *ValueInstance* representing the value against which the operation's output will be compared to assess its validity.	*ValueInstance* (See Table 5.53)	0..1
valueLiteral	A character string literal describing a value which the operation output will be compared against. This is an alternative to providing a *ValueInstance* as described above. If both the ***value*** and the ***valueLiteral*** fields are provided, than the *ValueInstance* field has priority.	xsd:string	0..1

dataCons-traint	A collection of constraints that have to be met by the operation output to be valid.	*DataCons-traint* (See Table 5.57)	0..n

5.8.37 TriggerAction

Description

A ***TriggerAction*** element models an action for triggering an action on an SUT component.

Semantics

Syntax

The *TriggerAction* element extends *test_behaviour:AtomicTestAction* (See Section 5.8.15)

Table 5.94: Properties of the TriggerAction UTML element

Property	**Description**	**Type**	**Occu-rence**
trigger-Notifi-cation	A notification textual message to describe the triggering action.	xsd:string	1..1

5.8.38 BaseSendDataAction

Description

A ***BaseSendDataAction*** element is an abstract classifier that models an action for sending data from one component to another one through a test port instance.

Semantics

BaseSendDataAction elements may only be created between ports that are connected with each other. The connection between ports may have been defined in a static test architecture associated to the contained test behaviour or may result from a precedent dynamic connection using the *SetupConnection-Action* element. It should be noted that the dynamic connection may occur in a separate *TestBehaviourActionDef* element invoked earlier using the *Test-BehaviourInvocationAction.*

The data to be sent by the *BaseSendDataAction* element may be defined either by providing a reference to a previously designed test data instance (see the *transmittedDataInstance* property) or by a combination of a data type and a collection of constraints based on which a concrete value would be generated by the target test environment (see the *transmittedDataInstance*).

Constraints

Constraint Data must not be sent from one port to the same port.

```
self.sourcePort <> self.destPort
```

Constraint Data designed as incoming data (i.e. with their *direction* property set to *IN*) must not be sent out through a *BaseSendDataAction* element.

```
(self.transmittedDataInstance
.oclIsTypeOf(utml::test_data::TestDataInstance) = true
 or self.transmittedDataInstance
 .oclIsTypeOf(utml::test_data::MessageTestDataInstance) = true
 or self.transmittedDataInstance
 .oclIsTypeOf(utml::test_data::OperationTestDataInstance) = true
 or self.transmittedDataInstance
 .oclIsTypeOf(utml::test_data::SignalTestDataInstance) = true)
  implies
 (self.transmittedDataInstance
 .oclAsType(utml::test_data::TestDataInstance).direction
<> utml::test_data::DataDirection::IN)
```

Constraint According to the black-box testing paradigm, data must be sent out only from components belonging to the test system and not from those belonging to the SUT.

```
(self.theComponent.oclIsTypeOf(OclVoid) = false)
        implies
(self.theComponent.kind
= utml::test_architecture::ComponentKind::TEST_COMPONENT)
```

Constraint Data must not be sent from a port, if that port was assigned the direction *IN*. Only ports defined as *INOUT* or *OUT* may be used for that purpose.

```
(self.sourcePort.oclIsTypeOf(OclVoid) = false)
implies
(self.sourcePort.direction
<> utml::test_data::DataDirection::IN)
```

Constraint If a paramaterizable test data instance is used in a *SendDataAction* element, then the required parameters must be provided to complete the test data instance.

```
(self.transmittedDataInstance.oclIsTypeOf
(utml::test_data::MessageTestDataInstance) = true)
implies
(self.transmittedDataInstanceParameter -> size() =
self.transmittedDataInstance.oclAsType
(utml::test_data::MessageTestDataInstance)
.parameterDeclaration -> size())
```

```
((self.transmittedDataInstance
.oclIsTypeOf(OclVoid) = true)
  implies
 (self.transmittedDataType
 .oclIsTypeOf(OclVoid) = false
 and
 self.dataConstraint ->  isEmpty() = false))
 and
 (
 (self.transmittedDataType
 .oclIsTypeOf(OclVoid) = true
 or
 self.dataConstraint ->  isEmpty() = true)
  implies
 (self.transmittedDataInstance
 .oclIsTypeOf(OclVoid) = false)
 )
```

Constraint

Syntax

The *BaseSendDataAction* element extends *test_behaviour:AtomicTestAction* (See Section 5.8.15)

Table 5.95: Fields and attributes of the BaseSendDataAction UTML element

Property	Description	Type	Occurrence
connection	A reference to the connection to use for sending the data.	*Connection* (See Table 5.30)	1..1
sourcePort	A reference to the source port instance for transmitting the data.	*PortInstance* (See Table 5.28)	1..1
expected-Operation-Output	A reference to a data instance expected to be sent as response for this *SendDataAction.*	*MessageTest-Data-Instance* (See Table 5.55)	0..1
trans-mitted-DataIns-tance	A reference to the test data instance to be transmitted.	*Abstract-Data-Instance* (See Section 5.7.21)	0..1
trans-mitted-Data-Instance-Para-meter	Optional parameter values for the transmitted test data instance.	*Parameter-Def* (See Table 5.60)	0..n
trans-mitted-DataType	For asynchronous communication, this property is a reference to a test data type based on which a concrete value will be generated for transmission. For synchronous communication, the parameters required for the referred *Operation-TestDataType* will have to be provided.	*TestData-Type* (See Table 5.46)	0..1

destPort	Reference to the destination port.	*Port-Instance* (See Table 5.28)	1..1 [4]

5.8.39 SendDataAction

Description

A ***SendDataAction*** element models an action for sending asynchronous data from one component to another one through a test port instance.

Semantics

The *SendDataAction* element extends the *test_behaviour:BaseSendDataAction* element (See Section 5.8.38). Therefore it shares the same semantics as defined in that section, with the only difference being that it is used for sending data in an asynchronous communication scheme.

Constraints

The ***SendDataAction*** element inherits all the constraints defined for the *BaseSendDataAction*.

Syntax

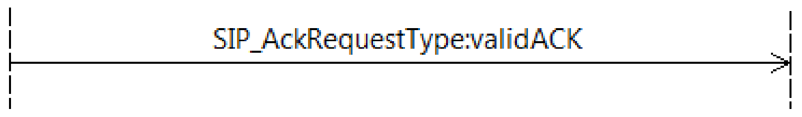


Figure 5.32: SendDataAction in UTML Test Behaviour Sequence Diagram

UTML *SendDataAction* elements are represented graphically by an arrow linking the source port instance to the target port instance.

As displayed in Figure 5.32, the label associated to a *SendDataAction* follows the format <*DataTypeId*>:<*DataInstanceId*>, whereby <*DataType*> denotes the identifier for the type of be transmitted and <*DataInstanceId*> the identifier of the data instance itself.

Table 5.96: Fields and attributes of the SendDataAction UTML element

[4] Implicit

Property	**Description**	**Type**	**Occurrence**
transmittedDataInstance	A reference to the test data instance to be transmitted.	*ValueInstance* (See Table 5.53)	0..1
dataConstraint	A collection of constraints which in combination with the ***transmittedDataType*** property will be used generate a concrete value for transmission.	*DataConstraint* (See Table 5.57)	1..1

5.8.40 SendSyncDataAction

Description

A ***SendSyncDataAction*** element models an action for sending synchronous data from one component to another one through a test port instance.

Semantics

The *SendSyncDataAction* element extends the *test_behaviour:BaseSendDataAction* element (See Section 5.8.38). Therefore it shares the same semantics as defined in that section, with the only difference being that it is used for sending data in a synchronous communication scheme.

Constraints

The ***SendSyncDataAction*** element inherits all the constraints defined for the *BaseSendDataAction*.

Syntax

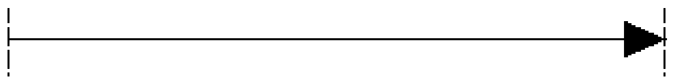


Figure 5.33: SendSyncDataAction in UTML Test Behaviour Sequence Diagram

SendSyncDataAction elements are represented graphically by an arrow similar to the one used for *SendDataAction* elements with the difference that the decorator at the end of the arrow is a filled triangle similar to those used for synchronous messages in UML sequence interaction diagrams.

Table 5.97: Fields and attributes of the SendSyncDataAction UTML element

Property	**Description**	**Type**	**Occurence**
expectedOperationOutput	A reference to a data instance expected to be sent as response for this *SendSyncDataAction*. It should be noted that exceptions supported by the called operation are included among possible responses for this property.	*OperationOutput* (See Table 5.93)	0..1
transmittedDataInstance	A reference to the test data instance to be transmitted.	*OperationTestDataInstance* (See Table 5.56)	0..1
transmittedDataType	For asynchronous communication, this property is a reference to a test data type based on which a concrete value will be generated for transmission. For synchronous communication, the parameters required for the referred *OperationTestDataElement* will have to be provided.	*TestDataType* (See Table 5.46)	0..1
outputParameterConstraint	A collection of constraints which the *INOUT* and *OUT* parameters provided by the *SendDataAction* must fullfil after the called operation returns.	*ParameterConstraint* (See Table 5.59)	0..n
destPort	Reference to the destination port.	*PortInstance* (See Table 5.28)	1..1 [5]

[5] Implicit

return-Timer	This property is for synchronous communication and indicates the maximum delay for expecting the operation call modeled by this ***SendSyncDataAction*** to return.	*Timer* (See Table 5.83)	

5.8.41 BaseReceiveDataEvent

Description

A ***BaseReceiveDataEvent*** element defines an abstract classifier modelling the expection of incoming data from another source.

Semantics

In accordance to the black-box testing paradigm, the receiving port i.e. the port instance at which the data is expected must not belong to an SUT component but to a test component. Also in a similar manner than for actions for sending data, a *BaseReceiveDataEvent* element can only be created between ports that are connected.

Constraints

Constraint (Optional) A guard timer should be provided for every *ReceiveDataEvent* element to ensure that if the expected data is not received, then the timer's expiration would be used to stop waiting for the data and thus avoid a livelock situation. If no timer is provided, then the default timer defined for the whole test behaviour model (see Table 5.66) will be used instead.

[language=OCL]

```
self.timer.oclIsTypeOf(OclVoid) = false
```

Constraint Data may only be received from components belonging to the test system and not to the SUT.

```
(self.theComponent.oclIsTypeOf(OclVoid) = false)
        implies
(self.theComponent.kind
= utml::test_architecture::ComponentKind::TEST_COMPONENT)
```

```
```

Constraint Data reception must not be defined through ports designed to be used exclusively for outwards communication, i.e. ports having their *direction* property set to *OUT*.

```
(self.receptionPort.oclIsTypeOf(OclVoid) = false)
implies
(self.receptionPort.direction
<> utml::test_data::DataDirection::OUT)
```

Constraint The port from which the expected data is expected to originate from must be different from the port at which it is expected.

```
self.receptionPort <> self.sourcePort
```

Constraint If a parameterized value is used to model the expected data, then the parameter values for those parameters must be provided to complete the definition.

```
(self.expectedDataInstance.oclIsTypeOf
(utml::test_data::MessageTestDataInstance) = true) implies
(self.expectedDataInstanceParameter -> size() =
self.expectedDataInstance.oclAsType
(utml::test_data::MessageTestDataInstance)
.parameterDeclaration -> size())
```

Constraint The expected data must be specified either using a combination of the data type and a series of constraints (for checking) or through a predefined reusable value instance in which the required constraints would have been stated.

```
((self.expectedDataInstance.oclIsTypeOf(OclVoid) = true)
 implies
(self.expectedDataType.oclIsTypeOf(OclVoid) = false
and
self.dataConstraint -> isEmpty() = false))
and
(
(self.expectedDataType.oclIsTypeOf(OclVoid) = true
or
self.dataConstraint -> isEmpty() = true)
 implies
(self.expectedDataInstance.oclIsTypeOf(OclVoid) = false)
)
```

Syntax

The *BaseReceiveDataEvent* element extends *test_behaviour:AtomicTestAction* (See Section 5.8.15)

Table 5.98: Properties of the BaseReceiveDataEvent UTML element

Property	Description	Type	Occurence
timer	A reference to a timer to use for avoiding deadlock while expecting the incoming data instance.	*Timer* (See Table 5.83)	0..1
receptionPort	The reception port at which the component will listen to check for the incoming data.	*PortInstance* (See Table 5.28) from package *test_architecture*	1..1
connection	Connection via which data will be transmitted.	*Connection* (See Table 5.30)	1..1
expectedDataInstanceParameter	Optional Reference to parameter values for the expected test data instance.	*ParameterDef* (See Table 5.60)	0..n
storage	A reference to a variable declaration in which the received data instance will be storaged for possible later usage.	*VariableDeclaration* (See Section 5.8.25)	0..1

sourcePort	A reference to a port instance from which the data is expected.	*PortInstance* (See Table 5.28)	1..1 [6]

5.8.42 ReceiveDataEvent

Description

A ***ReceiveDataEvent*** element models an action for expressing that a test component is expecting another component to send some data.

Semantics

The *ReceiveDataEvent* element extends the *test_behaviour:BaseReceiveDataEvent* element(See Section 5.8.41). Therefore it shares the same basic semantics, with the specificity that the data is being expected in a synchronous communication scheme.

Constraints

```
(self.expectedDataInstance
.oclIsTypeOf(utml::test_data::TestDataInstance) = true
or self.expectedDataInstance
.oclIsTypeOf(utml::test_data::MessageTestDataInstance) = true
or self.expectedDataInstance
.oclIsTypeOf(utml::test_data::OperationTestDataInstance) = true
or self.expectedDataInstance
.oclIsTypeOf(utml::test_data::SignalTestDataInstance) = true)
  implies
 (self.expectedDataInstance
 .oclAsType(utml::test_data::TestDataInstance)
 .direction <> utml::test_data::DataDirection::OUT)
```

Syntax

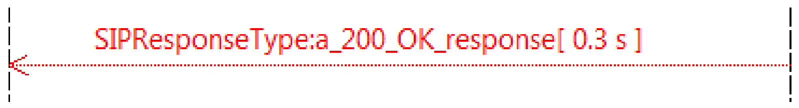


Figure 5.34: ReceiveDataEvent in UTML Test Behaviour Sequence Diagram

[6]Implicit

UTML *ReceiveDataEvent* elements are represented graphically by an arrow linking the port instance at which data is expected to the originating port instance.

As depicted on figure 5.35, the label associated to a *ReceiveDataEvent* follows the format *<DataTypeId>:<DataInstanceId>[<TimerDelay>]*, whereby *<Datatype>* denotes the identifier for the type of test data expected, *<DataInstanceId>* the identifier of the data instance expected and *<TimerDelay>* the maximal delay of the timer associated to the *ReceiveDataEvent*.

Table 5.99: Properties of the ReceiveDataEvent UTML element

Property	**Description**	**Type**	**Occurence**
expectedDataInstance	A reference to the test data instance expected.	*AbstractDataInstance* (See Section 5.7.21)	1..1
storage	A reference to a variable declaration in which the received data instance will be storaged for possible later usage.	*VariableDeclaration* (See Section 5.8.25)	0..1

5.8.43 ReceiveSyncDataEvent

Description

A ***ReceiveSyncDataEvent*** element models an action for expressing that a test component is expecting another component to send some data.

Semantics

In accordance to the black-box testing paradigm, the receiving port i.e. the port instance at which the data is expected must not belong to an SUT component but to a test component.

Constraints

```
 self.expectedDataInstance
.oclIsTypeOf(utml::test_data::OperationTestDataInstance) = true
   implies
 (self.expectedDataInstance
    .direction <> utml::test_data::DataDirection::OUT)
```

```
((self.expectedDataInstance.oclIsTypeOf(OclVoid) = true)
 implies
(self.expectedDataType.oclIsTypeOf(OclVoid) = false
and
self.dataConstraint -> isEmpty() = false))
and
(
(self.expectedDataType.oclIsTypeOf(OclVoid) = true
or
self.dataConstraint -> isEmpty() = true)
 implies
(self.expectedDataInstance.oclIsTypeOf(OclVoid) = false)
)
```

Syntax

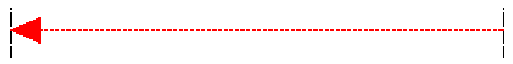


Figure 5.35: ReceiveSyncDataEvent in UTML Test Behaviour Sequence Diagram

UTML *ReceiveSyncDataEvent* elements are represented graphically by the same kind of arrows as *ReceiveDataEvent* with the difference that the decorator at the end of the arrow is a filled triangle similar to the one used for synchronous messages in UML sequence interaction diagrams.

The *ReceiveSyncDataEvent* element extends *test_behaviour:AtomicTestAction* (See Section 5.8.15)

Table 5.100: Properties of the ReceiveSyncDataEvent UTML element

Property	**Description**	**Type**	**Occurence**
expectedDataInstance	A reference to the test data instance expected.	*OperationTestDataInstance* (See Table 5.56)	1..1

operation-Output	A reference to a test data instance expected to be returned by the test component receiving the data.	*Operation-Output* (See Table 5.93)	0..1

5.8.44 MultipleReceiveDataEvent

Description

The ***MultipleReceiveDataEvent*** element models the awaiting of successive messages from another component. The associated test component instance starts a loop, checking everytime that the incoming message is received. If the indicated break-expression evaluates to ***true*** or the maximal number of expected data instances is reached, then the component stops looping.

Semantics

The *MultipleReceiveDataEvent* element extends the *test_behaviour:ReceiveDataEvent* element (See Table 5.99). Therefore, it inherits the same basic semantics and associated constraints.

Syntax

Table 5.101: Fields and attributes of the MultipleReceive-DataEvent UTML element

Property	**Description**	**Type**	**Occu-rence**
break-Expres-sion	A character string representing an expression which evaluates as a boolean value.	xsd:string	0..1 [7]
max-Instances-Expression	A character string representing an expression which evaluates as the maximal number of expected incoming data instances.	xsd:string	0..1 [8]

[7]Mandatory, if *maxInstancesExpression* omitted.
[8]Mandatory, if *breakExpression* omitted.

5.8.45 TestArchitectureActionKind

Description

The ***TestConfigActionKind*** element is an *enumeration* used to classify possible kinds of test behaviour actions on test architectures elements. Table 5.102 lists the literals of that enumeration and their meaning.

Syntax

Table 5.102: Properties of the TestArchitectureActionKind UTML element

Property	**Description**
ARCHI-TECTURE-_SETUP	For a test action used to setup a test architecture.
ARCHI-TECTURE-_TEARDOWN	For a test action used to tear down an existing (or running) test architecture.

5.8.46 TestSequence

Description

The ***TestSequence*** element is an abstract classifier modelling a sequence of test actions in the UTML metamodel.

Semantics

Syntax

The *TestSequence* element extends test_behaviour:TestAction (See Table 5.75)

5.8.47 SendReceiveSequence

Description

The ***SendReceiveSequence*** element models a test sequence whereby the owner test component sends some data to another one and immediately expects the other component to respond by sending data.

Semantics

Syntax

The *SendReceiveSequence* element extends *test_behaviour:TestSequence* (See Section 5.8.46)

Table 5.103: Fields and attributes of the SendReceiveSequence UTML element

Property	Description	Type	Occurence
sendDataAction	The description of the action for sending out data.	*SendDataAction* (See Table 5.96)	1..1
receiveDataEvent	A description of the action for receiving incoming response data.	*ReceiveDataEvent* (See Table 5.99)	1..1

5.8.48 TriggerReceiveSequence

Description

The ***TriggerReceiveSequence*** element models a test sequence whereby the owner testcomponent expects another component to send some data, after it has been triggered through some external means for doing so.

Semantics

Syntax

The *TriggerReceiveSequence* element extends *test_behaviour:TestSequence* (See Section 5.8.46)

Table 5.104: Properties of the TriggerReceiveSequence UTML element

Property	Description	Type	Occurence
triggerAction	Details on the action for triggering the other component.	*TriggerAction* (See Table 5.94)	1..1

receiveDataEvent	Details on the action for receiving incoming data.	*ReceiveDataEvent* (See Table 5.99)	1..1

5.8.49 TestBehaviourElement

Description

The ***TestBehaviourElement*** element is an abstract classifier that serves as base classifier for all other elements in a UTML test behaviour model.

Semantics

Syntax

The *TestBehaviourElement* element extends *UtmlElement* (See Section 5.3.1)

5.8.50 CheckAction

Description

The ***CheckAction*** is an abstract classifier that serves as base for test behaviour actions designed to perform some verifications in the test behaviour.

Syntax

The *CheckAction* element extends *test_behaviour:Observation* (See Section 5.8.20)

5.8.51 ExternalCheckAction

Description

The ***ExternalCheckAction*** element models an action whereby a manual check is performed externally during test execution. In the lower-level test infrastructure, the execution of an ***ExternalCheckAction*** should return an OK or NOK value indicating whether the check was successful or not.

Syntax

The *ExternalCheckAction* element extends *test_behaviour:CheckAction* (See Section 5.8.50)

Table 5.105: Fields and attributes of the ExternalCheckAction UTML element

Property	Description	Type	Occu-rence
condition	A textual description of a condition to check on the component.	xsd:string	1..1

5.8.52 ValueCheckAction

Description

The ***ValueCheckAction*** element models an action for verifying that the value of a given variable meets certain constraints.

Semantics

The ***ValueCheckAction*** may be associated to a variable declaration designed using the *VariableDeclaration* element or contain a behaviour invocation designed using the *BehaviourActionInvocation* element. In the later case, the value returned by the behaviour invocation will be checked against the provided constraints.

Constraints

Constraint If a reference to a defined variable is omitted in the *ValueCheckAction* element, then a test behaviour invocation must be provided instead and vice-versa.

```
(self.variable.oclIsTypeOf(OclVoid) = true
and self.testBehaviourActionInvocation
.oclIsTypeOf(OclVoid) = true) = false
```

Constraint The invoked test behaviour type definition must define a return value to be used in a *ValueCheckAction* element.

```
self.testBehaviourActionInvocation.oclIsTypeOf(OclVoid) = false
implies
(self.testBehaviourActionInvocation.testBehaviourActionDef
.oclIsTypeOf(OclVoid) = false
and
self.testBehaviourActionInvocation.testBehaviourActionDef
.responseDef.oclIsTypeOf(OclVoid)
= false
)
```

Syntax

The *ValueCheckAction* element extends *test_behaviour:CheckAction* (See Section 5.8.50)

Table 5.106: Fields and attributes of the ValueCheckAction UTML element

Property	**Description**	**Type**	**Occurence**
variable	The variable whose value will be checked.	*VariableDeclaration* (See Section 5.8.25)	0..1
testBehaviourActionInvocation	A definition of a behaviour invocation whose return value will be checked against the provided constraints.	*VariableDeclaration* (See Section 5.8.25)	0..1
dataConstraint	The set of constraints the data will be checked against.	*DataConstraint* (See Table 5.57)	1..n

5.8.53 ActionBlock

Description

The *ActionBlock* element is an abstract classifier used for designing blocks of complex behaviours in a UTML test model. Therefore, as it defines a container for other kinds of test behaviour actions, all other elements of the UTML metamodel requiring that functionality extend this classifier.

Syntax

Table 5.107: Properties of the SubActionBlock UTML element

Property	**Description**	**Type**	**Occurence**
testAction	The sub-actions composing the action block.	*TestAction* (See Table 5.75)	0..n

5.8.54 SubActionBlock

Description

A ***SubActionBlock*** element models an action block that may be contained in another action block.

Syntax

The *SubActionBlock* element extends *test_behaviour:ActionBlock* (See Section 5.8.53)

Table 5.108: Properties of the SubActionBlock UTML element

Property	Description	Type	Occurence
theComponent	A reference to the owning test component.	*ComponentInstance* (See Table 5.29)	1..1

5.8.55 StructuredTestAction

Description

The ***StructuredTestAction*** element is an abstract classifier modelling a structured test action in UTML, i.e. an action that can be decomposed in several other actions called sub-actions.

Semantics

The *StructuredTestAction* element extends the *test_behaviour:TestAction* (See Table 5.75) and the *test_behaviour:ActionBlock* (See Table 5.107) elements. Therefore, it models an action that may not only contain other actions refered to as sub-actions, but also be contained itself in other action blocks.

5.8.56 RepeatTestAction

Description

The ***RepeatTestAction*** element models a loop in a UTML test model. Figure 5.36 depicts an example UTML *RepeatAction* in its graphical representation.

Semantics

The *RepeatTestAction* element defines a test behaviour block used to model a loop in UTML. All sub-actions contained in the *RepeatTestAction* will be repeated sequentially following one of the following schemes:

- If the *breakExpression* property is provided, then repeat the contained actions until the defined breaking expression is evaluated to *true*.
- If the *continueConditionExpression* property is provided, then repeat the contained actions as long as the provided expression evaluates to *true*.
- If the *maxNumberOfTimes* property is provided, then repeat the contained actions for the specified number of times.
- If the *timer* property is provided, then repeat the contained actions until the referenced timer expires.

Syntax

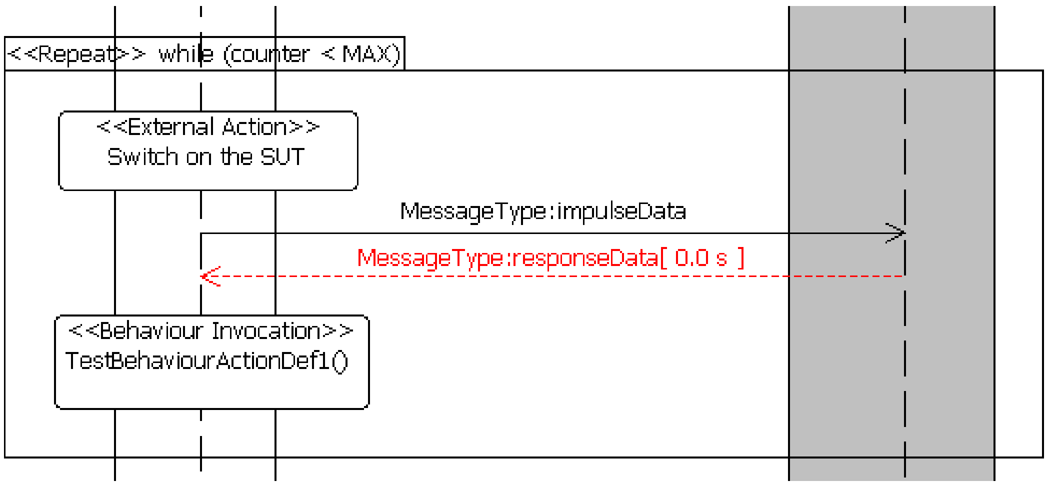

<<Repeat>> while (counter < MAX)
<<External Action>>
Switch on the SUT
MessageType:impulseData
MessageType:responseData[0.0 s]
<<Behaviour Invocation>>
TestBehaviourActionDef1()

Figure 5.36: RepeatAction in UTML Test Behaviour Sequence Diagram

The *RepeatTestAction* element extends *test_behaviour:StructuredTestAction* (See Section 5.8.55)

Table 5.109: Properties of the RepeatTestAction UTML element

Property	**Description**	**Type**	**Occurence**
breakExpression	A character string describing the expression, which if true would stop the loop.	xsd:string	0..1 [9]

[9]Mandatory, if ***maxRepeatTimesExpression*** and ***continueConditionExpression*** are omitted.

maxRepeat-Times-Expression	A String literal which evaluates to the maximal number of times to go through the loop.	xsd:string	0..1 [10]
continue-Condition-Expression	A String literal which expresses the condition required by the loop to continue execution.	xsd:string	0..1 [11]
timer	Reference to a timer whose expiration will stop the loop	*Timer* (See Table 5.83)	0..1 [12]

5.8.57 IfElseAction

Description

The ***IfElseAction*** element models an If-Else block in a UTML behaviour model.

Constraints

Constraint The condition for an ***IfElseAction*** element must be provided, either as a character string literal or as a reference to a previously defined value instance of type *boolean* or extension thereof.

```
((self.conditionLiteral.oclIsTypeOf(OclVoid) = true
or self.conditionLiteral='')
 and self.conditionReference
 .oclIsTypeOf(OclVoid) = true) = false
```

Syntax

Figure 5.37 depicts an example *IfElseAction* element in a UTML test sequence diagram, which illustrate the associated concrete syntax. The graphical element consists of a mandatory If-block and an optional Else-Block. In each of those blocks test-actions can be designed using the same toolset as for other UTML elements. Table 5.110 displays the structure of the *IfElse* element.

Table 5.110: Properties of the IfElseAction UTML element

[10]Mandatory, if ***breakExpression*** and ***timer*** are omitted.
[11]Mandatory, if ***breakExpression*** and ***timer*** are omitted.
[12]Mandatory, if ***breakExpression*** and ***continueConditionExpression*** are omitted.

Property	**Description**	**Type**	**Occurrence**
conditionLiteral	A character string describing the expression to be used as condition for the if-statement.	xsd:string	0..1 [13]
conditionReference	A reference to value instance to be used as condition	*AbstractDataInstance* (See Section 5.7.21)	0..1 [14]
ifAction	A block containing actions to be executed if the condition is evaluated to true	*IfAction* (See Table 5.111)	1..1
ifAction	A block containing actions to be executed if the condition is evaluated to true	*ElseAction* (See Table 5.112)	0..1

5.8.58 IfAction

Description

The ***IfAction*** element models the part of an If-Else block that applies if the condition specified is evaluated to *true*.

Syntax

The *IfAction* element extends *test_behaviour:SubActionBlock* (See Section 5.8.54)

Table 5.111: Properties of the IfAction UTML element

Property	**Description**	**Type**	**Occurrence**
parentAction	The parent action in which this action is contained.	*TestAction* (See Table 5.75)	1..1

[13]Mandatory, if ***conditionReference*** is omitted.
[14]Mandatory, if ***conditionLiteral*** is absent.

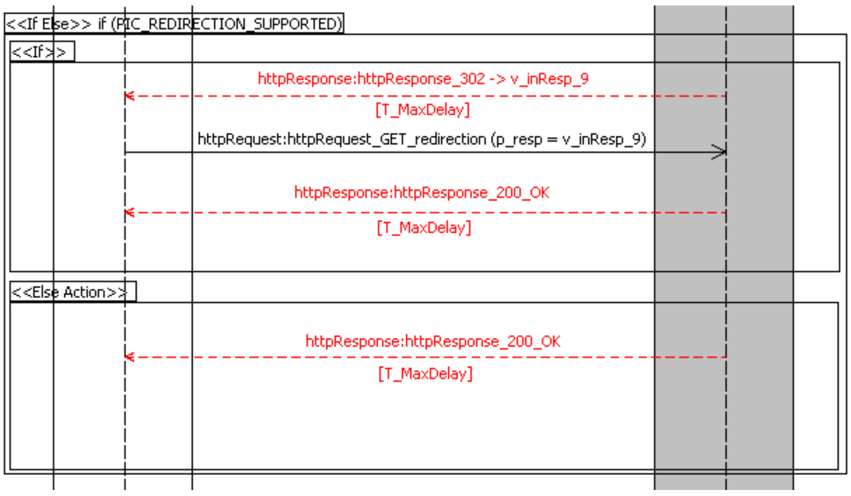

<<If Else>> if (PIC_REDIRECTION_SUPPORTED)
<<If>>
httpResponse:httpResponse_302 -> v_inResp_9
[T_MaxDelay]
httpRequest:httpRequest_GET_redirection (p_resp = v_inResp_9)
httpResponse:httpResponse_200_OK
[T_MaxDelay]
<<Else Action>>
httpResponse:httpResponse_200_OK
[T_MaxDelay]

Figure 5.37: IfElseAction in UTML Test Behaviour Sequence Diagram

5.8.59 ElseAction

Description

The ***ElseAction*** element models the part of an If-Else block that applies if the condition specified is evaluated to *false*.

Syntax

The *ElseAction* element extends *test_behaviour:SubActionBlock* (See Section 5.8.54)

Table 5.112: Properties of the ElseAction UTML element

Property	Description	Type	Occurence
parent-Action	The parent action in which this action is contained.	*TestAction* (See Table 5.75)	1..1

5.8.60 AltBehaviourAction

Description

The *AltBehaviourAction* element models a block of alternative behaviours in a UTML model.

Constraints

```
((self.conditionLiteral.oclIsTypeOf(OclVoid) = true
or self.conditionLiteral='')
 and self.conditionReference
 .oclIsTypeOf(OclVoid) = true) = false
```

Constraint

Syntax

Figure 5.38 depicts an example *AltBehaviourAction* element as represented in

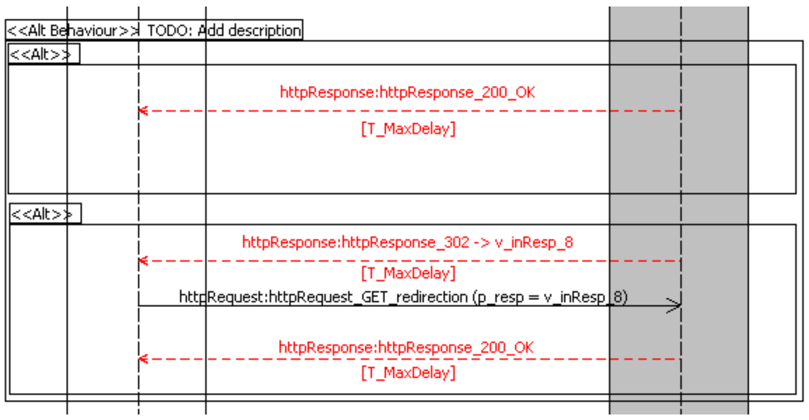

<<Alt Behaviour>> TODO: Add description
<<Alt>>
httpResponse:httpResponse_200_OK
[T_MaxDelay]
<<Alt>>
httpResponse:httpResponse_302 -> v_inResp_8
[T_MaxDelay]
httpRequest:httpRequest_GET_redirection (p_resp = v_inResp_8)
httpResponse:httpResponse_200_OK
[T_MaxDelay]

Figure 5.38: AltBehaviourAction in UTML Test Behaviour Sequence Diagram

Table 5.113: Properties of the AltBehaviourAction UTML element

Property	**Description**	**Type**	**Occurence**
conditionLiteral	A character string describing the expression to be used as condition for evaluating the alternative behaviour action.	xsd:string	0..1 [15]
conditionReference	A reference to value instance to be used as condition	*AbstractDataInstance* (See Section 5.7.21)	0..1 [16]
altAction	Blocks containing alternative actions to be executed if their associated triggering event is observed	*AltAction* (See Table 5.114)	0..n
interleave	A flag indicating whether interleave operation should apply for the alternatives or not. If this property is set to true, interleave behaviour will apply.	*Boolean*	0..1

[15] Mandatory, if ***conditionReference*** is omitted.

[16] Mandatory, if ***conditionLiteral*** is absent.

5.8.61 AltAction

Description

The *AltAction* element models an alternative sub-block within an *AltBehaviourAction* element block.

Syntax

The *AltAction* element extends *test_behaviour:SubActionBlock* (See Section 5.8.54)

Table 5.114: Properties of the AltAction UTML element

parent-Action	The parent action in which this action is contained.	*TestAction* (See Table 5.75)	1..1
trig-gering-Event	.	*TestEvent* (See Section 5.8.21)	1..1

5.8.62 ActivateDefaultAction

Description

The ***ActivateDefaultAction*** element models the activation of a default behaviour to be checked implicitly, if explicitly specified test behaviour options do not apply.

Syntax

The *ActivateDefaultAction* element extends *test_behaviour:AtomicTestAction* (See Section 5.8.15)

Table 5.115: Properties of the ActivateDefaultAction UTML element

Property	**Description**	**Type**	**Occurence**
default-Behaviour-Def	A reference to the default behaviour definition, that needs do be activated.	*Default-Behaviour-Def* (See Table 5.82)	1..1

5.8.63 DeactivateDefaultAction

Description

The ***DeactivateDefaultAction*** element models the activation of a default behaviour to be checked implicitly, if explicitly specified test behaviour options do not apply.

Syntax

The *DeactivateDefaultAction* element extends *test_behaviour:AtomicTestAction* (See Section 5.8.15)

Table 5.116: Properties of the DeactivateDefaultAction UTML element

Property	**Description**	**Type**	**Occurence**
activateDefaultAction	A reference to the default activation to which this deactivation will apply.	*ActivateDefaultAction* (See Table 5.115)	1..1

5.9 Mapping UTML Concepts to Existing (Test Scripting) Languages

Model transformation is an essential aspect of MDE. The transformation from PIM to PSM is a model-to-model (M2M) transformation, but eventually models will mostly be transformed into lower level textual notations through a model-to-text (M2T) transformation. As a language defining concepts at a higher level of abstraction, UTML can be mapped to any lower-level notation used for implementing executable test scripts. This transformation can be performed independently of whether the notation is an intermediary scripting notation (e.g. TTCN-3) to be executed in a particular test execution environment, or a generic purpose programming language instrumented for test automation (e.g. JAVA, Python, C, etc.).

Given the importance of model-transformation to the MDE-process, the Object Management Group (OMG) has introduced a collection of standard notations for specifying such transformations in a tool-independent manner. While the MOF Model-to-Text(MOFM2T) Transformation Language (MTL) [73] can be used for transforming MOF metamodel instantiations into textual notations, the QueryViewsTransformation (QVT) [71] language is more appropriate for specifying transformations from one metamodel into another one. Alternatively, the ATLAS Transformation Language (ATL) [92] may be used for that purpose, although it is not an OMG standard.

In the next sections, possible mapping approaches from UTML into selected notations are described. It is worth noting that for the sake of conciseness, these are just proposals for mapping covering a selection of elements from the UTML notation, because introducting a complete mapping would have widely exceeded the scope of this thesis. Also, the proposed mapping cannot be viewed as normative, because depending on the intended purpose, different mapping schemes can be developed and applied to fit the constraints of existing testing infrastructure or test equipments.

5.9.1 Mapping to TTCN-3

The mapping to TTCN-3 proposed in this section is based on a M2T transformation. That approach was chosen, mainly because the TTCN-3 metamodel is not part of the standard for that language. Moreover, a transformation via that metamodel would have just unnecessarily introduced an additional step in the process, because the end target notation for TTCN-3 is its textual form and not its metamodel form. In fact, no tool support for working directly with instances of the TTCN-3 metamodel was known to the author as those lines were written.

The mapping defines a transformation rule for each of the selected UTML elements it addresses. The transformation rules are expressed using the OMG's MTL standard language, which allows the specification of complex transformation rules through a syntax adopted from the OCL langage.

Table 5.117: Example UTML to TTCN-3 Mapping

UTML Element	TTCN-3 Mapping
TestModel, ***TestArchitectureTypesModel***, ***TestDataModel***, ***TestArchitectureModel***, ***TestBehaviourModel***	module
MessageTestDataType	record, union, enumerated (depending on type selector in UTML)
MessageTestDataInstance	template
OperationTestDataType	operation
OperationTestDataInstance	template
TestBehaviourActionDef	function
Testcase	See Section B.1.1 for details.
SendDataAction	See Section B.1.2 for details.
ReceiveDataEvent	See Section B.1.3 for details.
SendDiscardAction	See Section B.1.4 for details.
WaitAction	See Section B.1.5 for details.
SetupConnectionAction	See Section B.1.6 for details.
CloseConnectionAction	See Section B.1.7 for details.

Timer	timer
State	N/A
DefaultBehaviourDef	See Section B.1.8 for details.
StopTimerAction	See Section B.1.9 for details.
StartTimerAction	See Section B.1.10 for details.
ValueCheckAction	See Section B.1.11 for details.

5.9.2 Mapping to JUnit

The mapping for JUnit is provided using the same approach as for TTCN-3. Again, the mapping rules are specified as MTL transformation rules, taking as input the UTML metamodel element to be transformed and generating JAVA code suitable for execution via the JUnit testing engine.

Table 5.118: Example UTML to TTCN-3 Mapping

UTML Element	**JUnit Mapping**
TestModel, ***TestArchitectureTypesModel***, ***TestDataModel***, ***TestArchitectureModel***, ***TestBehaviourModel***	Testsuite class
MessageTestDataType	context-specific (e.g. Class definition)
MessageTestDataInstance	context-specific (e.g. Object Instantiation)
OperationTestDataType	method declaration
OperationTestDataInstance	method invocation (in connection with a SendDataAction)
TestBehaviourActionDef	function
Testcase	JAVA class extending JUnit's Testcase class (See Section B.2.1 for details)
SendDataAction	context-specific (e.g. method invocation, Remote Procedure Call (RPC), etc.)
ReceiveDataEvent	context-specific
SendDiscardAction	context-specific
WaitAction	See Section B.2.2
SetupConnectionAction	context-specific, depending on the SUT
CloseConnectionAction	Context-specific (SUT-dependent)
Timer	JAVA Object emulating a timer

State	N/A
DefaultBehaviour-ActionDef	N/A
StopTimerAction	Timer object stop
StartTimerAction	Timer object start
ValueCheckAction	assertTrue statement

5.10 Summary

This chapter has presented the concepts of the UTML notation and their graphical representation in various forms of diagrams. A metamodel approach was chosen to express those concepts to ensure that they are completely defined and thus applicable in a practical sense. The work presented in this chapter is closely related to the UML Testing Profile proposed by the OMG. However most of the concepts defined by that profile remain vague and hardly applicable for solving real-life test engineering problems. Therefore, conciseness, preciseness and practicability were the main driving forces in defining the UTML notation.

Section 5.2.1 has presented graphical visualisation elements for UTML. However, it is worth noting that some technical constraints had to be taken into account while selecting the graphical elements. The decision on which figure to use for each of the visual UTML elements had to ensure that the selected figure can also be implemented and used in the framework used for prototyping. Therefore, those figures leave room for further improvements at a later stage.

Section 5.9 has described how UTML concepts map to existing notations used for implementing executable test scripts. Example mappings were provided for TTCN-3 and JUnit, clearly demonstrating that similar mappings could be added for any other language, depending on the targetted test environment.

However, defining a notation for pattern-oriented test engineering also implies providing an appropriate tool set for using that notation in real-life case studies. In that process, new ideas emerge on potential improvements to the notation itself and the associated tool set towards higher usability, robustness and expressivity. In the next section, an extensible architecture for such a tool chain is described, including a prototype implementation aimed at supporting the evaluation of the proposed methodology through the case studies.

Chapter 6

Evaluation: Implementation and Case Studies

6.1 Introduction

This work started with the intuitive assumption that model-driven development techniques bear the potential of significantly improving the test development process, both quantitatively and qualitatively. That assumption originates from claims of similar gains from applying MDE to software system development. However, although it may sound plausible and even obvious, providing scientific evidence to support it is less trivial than it might appear. This has lead some authors to even question the real benefits of MDE as a whole. E.g. Mohagheghi et al [112] ask a bit provocatively:

Where is The Proof?

The task of verifying that assumption appears to be even more difficult in the context of model-driven test development, as it is advocated in this work. One of the reasons for that difficulty is the quasi non-existence of published practical experiences of applying the methodology described in this work. This might find its explanation in the scarcity of software available on the market for supporting such a methodology [135], combined with the reticence from the industry to publish results related to product quality assessment. In fact, compared to the large number of MBT tools featuring automated generation of tests from system models [14, 157], the number of existing tools to support MDT is insignificantly low. The only list of available MDT tools similar to those provided by [14, 157] that could be found for this thesis is the one provided by Torres et al [154]. Although some of the tools described in that work claim to provide support for model-based manual test case construction, the methodology they use was found to be inappropriate for the type of experimentation required to verify the hypothesis of this thesis. Therefore a collection of prototype tools were developed during this work to assess our methodology and the process it supports. Although this required a significant effort, those software design and implementation activities were also beneficial for the work, because they helped identify several requirements and issues through practical experimentation, that would have been difficult to anticipate otherwise.

Therefore, an important part of the work in this thesis consisted in designing and implementing an appropriate toolset to support the pattern-oriented model-driven test development methodology, so that it could then be evaluated in some real-life case studies.

This chapter describes the architecture designed for that tool set, the implementation approach and the prototype implementation resulting from that process.

After the prototype tool has been presented, a small scale example is presented to illustrate how pattern-oriented test engineering can be applied in a real-life test development project. Also, a case study featuring a conformance test suite for the IP Multimedia Subsystem (IMS) is presented, that was used to measure the impact of the approach on the test development process in terms of productivity gain, to verify the assumption made intuitively at the beginning of the work.

6.2 Implementation: The UTML Eclipse Plug-in Tool chain

6.2.1 Requirements on The Model-Driven Test Engineering Toolset

As in any software development project, the first development phase for the UTML MDT toolset consisted in gathering user requirements.

Jennitra [5] lists a selection of requirements on functional tests to address the growing challenges faced with in testing todays, in particular in the context of extreme programming and agile methods. Those requirements are:

- Ease to write: Writing functional tests should be an activity that remains accessible to staff with little technical background and that can produce results quickly to ensure that it does not become a bottleneck for other activities of the software system development process.

- Readability: Functional tests are shared artifacts between stakeholders in the software product's business process. Therefore, they must remain readable, so that all parties can easily understand what each test case verifies.

- Correctness: Despite being readable, functional tests must be correct to ensure that products are not deployed with failures that are generally more costly and difficult to identify and to correct, once the software is deployed.

- Maintainability: Tests are essential to ensure the quality of software products. However, it cannot be assigned too much resources. Therefore the effort in maintaining tests should not be main cost factor in the development process and afterwards.

- Locatability: The tests should be organised in such a way that finding a given test should be possible quickly and without too much effort.

For these requirements to be fulfilled by tests scripts and the test models out of which they are generated, the test modelling toolset must also take them into account.

6.2.2 The Proposed Architecture

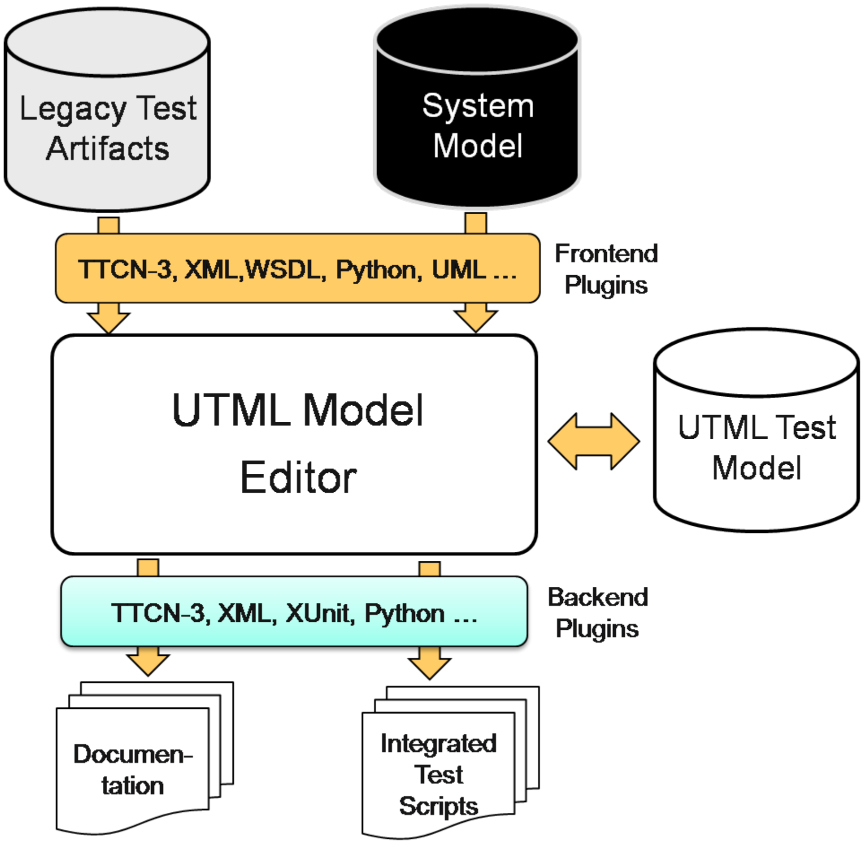


Figure 6.1: Architecture of the UTML Prototype Toolchain

Figure 6.1 displays an architecture designed to build a prototype application to meet the requirements listed in section 6.2.1. As depicted in that figure, the prototype application is built around a UTML editor that forms the core of the architecture. The UTML editor will provide a graphical user interface, through which users will be able to perform all types of operations on test models (e.g. creating, modifying, transforming, analysing, etc.). To ensure that those test models can be shared among distributed users, they will be stored in a common repository, as depicted on the right-hand side of the figure.

The need to incorporate legacy tests in any new testing approach has already been highlighted and is viewed as an important issue by the industry [123]. Therefore, additionally, to support the reuse of legacy test artifacts designed using other notations, the application will provide a flexible programming interface (API) through which transformators from those notations into UTML will be plugged-in at runtime. Those user-defined transformators will add to a default set of standard transformators or front-end plugins that will be provided by the application to support automatic transformation from established test notations (e.g. TTCN-3) to UTML.

Following the same principle, another API will be provided to support the integration of transformators from UTML abstract test models to concrete executable test scripts source code.

The presence of front-ends and back-ends enable the usage of the tool to perform round trip engineering. Using the front-ends, legacy test artifacts will be imported into UTML so that they can visualized and analysed more easily to be used as a base for new test model elements. Then, using the back-ends the new test model will be transformed back to the lower-level test notation for further processing, leading eventually to test execution.

6.2.3 Prototype Implementation

Implementation Approach

Interestingly, an MDE approach was chosen for the prototype toolset itself. Which means, this work provided a unique opportunity for not just evaluating the application of the MDE approach to test development, but also to product software development in general.

The implementation approach is based on a specification of the UTML metamodel as an EMF (Eclipse Modelling Framework) ECore model. EMF is one of the most popular MDE frameworks available on the market. It is integrated in the Eclipse framework and provide a series of tools to support modelling and model transformation into numerous programming languages (JAVA, C++, PHP, etc.) and modelling notations (UML, SysML, etc.). ECore is an implementation of the OMG's Meta-Object-Facility (MOF) concept for the Eclipse framework. Using the facilities provided by EMF an automatic generation of a toolset from the metamodel can be performed, producing an editor for the notation represented by the metamodel. However the automatically generated editor is exclusively based on the information provided in the metamodel. Therefore, it lacks some essen-

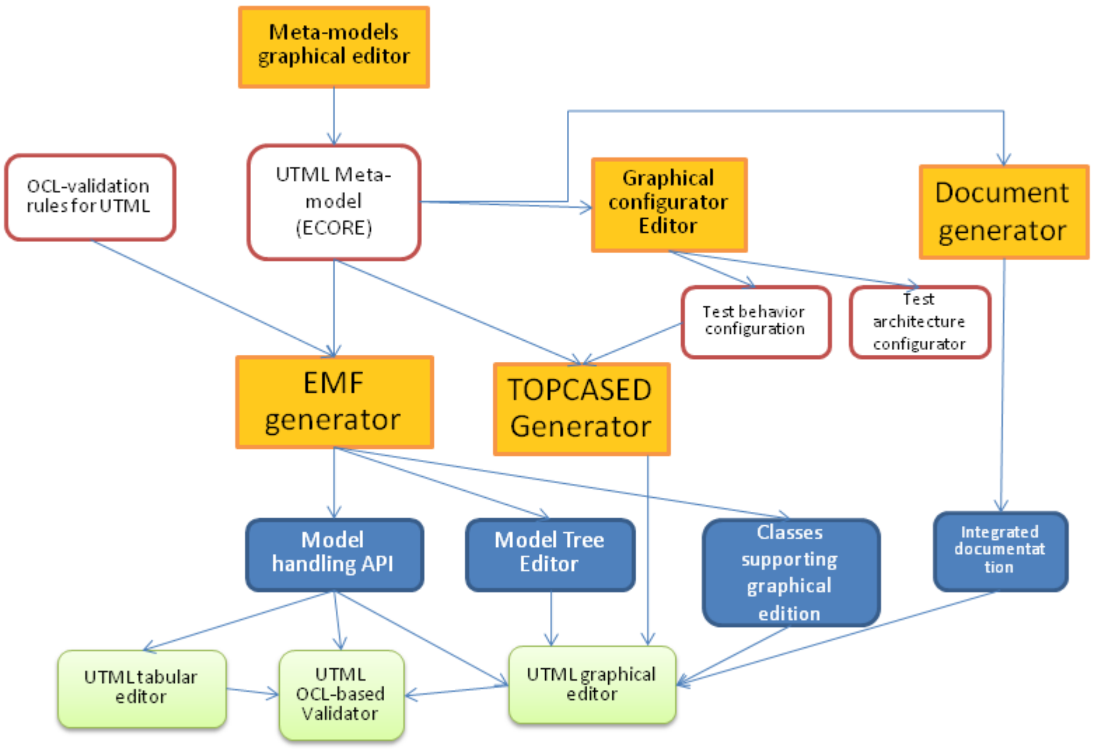


Figure 6.2: UTML Prototype Toolchain's Implementation Approach

tial context information required to improve the usability of the resulting tools. Those context information can be added manually to the generated JAVA code. Using appropriate annotations in Javadoc comments, it can be ensured that the manually modified code is not destroyed by following generation processes.

To provide a graphical editor implementing the visualisation of UTML elements discussed in Chapter 5 an MDE approach was applied again. The Open-Source Toolkit for Critical Systems (TOPCASED) [149] is a framework based on EMF and allowing the definition of graphical representation for elements of an ECore metamodel. The mapping between elements of the ECore metamodel and their graphical representation is described in so-called diagram configurator files. For each of the diagram types defined in Section 5.2.1, a diagram configurator file was specified in XML, based on which source code was generated automatically, using the facilities of TOPCASED.

Implementation of Test Patterns As discussed in Section 4.3.3 an approach combining the *generative* approach with the *tool environment support* approach was chosen to specify test patterns in this thesis. To implement that approach, three groups of features are provided with the prototype tool, additionally to the constraints already embodied in the UTML metamodel.

Policies on actions One way of implementing patterns in a modelling tool consists in defining policies, based on which operations on graphical elements of the notation would be allowed or disallowed, depending on the current context. The prototype tool implemented in this thesis provides the possibility for activating or deactivating the enforcement of those policies, depending on the main purpose of the test modelling activity. If the test modelling activity is performed, just to provide a visual documentation to a test suite and it is not intended to generate any test scripts out of the test model, then the policies should be deactivated, with the logical consequence of a higher probability that the resulting test models may be syntactically and semantically faulty. On the other hand, if emphasis is laid on the correctness of the test model with the aim of transforming it into executable test scripts for a target test execution environment, then the policies should be activated and will make it impossible to perform disallowed actions on graphical elements of the test model.

OCL-Constraints OCL-constraints enable an online or offline validation of the test model or elements thereof to ensure that the semantical requirements w.r.t. the underlying patterns are met. *Online validation* refers to the fact that the test models are validated automatically every time a modification has occurred, whereas *offline validation* refers to on-demand validation triggered by a corresponding request. The prototype tool implements a total of 40 OCL-constraints, that are used to validate the test models. Listing 6.1 shows an example OCL-constraint used to ensure that connections between ports belonging to the same component are identified and disallowed.

```
inv different_components_for_port_connection:
 (self.sourcePort.oclIsTypeOf(OclVoid) = false and self.destPort.oclIsTypeOf(OclVoid) = false)
 implies
 (self.sourcePort.theComponent \<\> self.destPort.theComponent)
```

Listing 6.1: Example OCL-Constraint

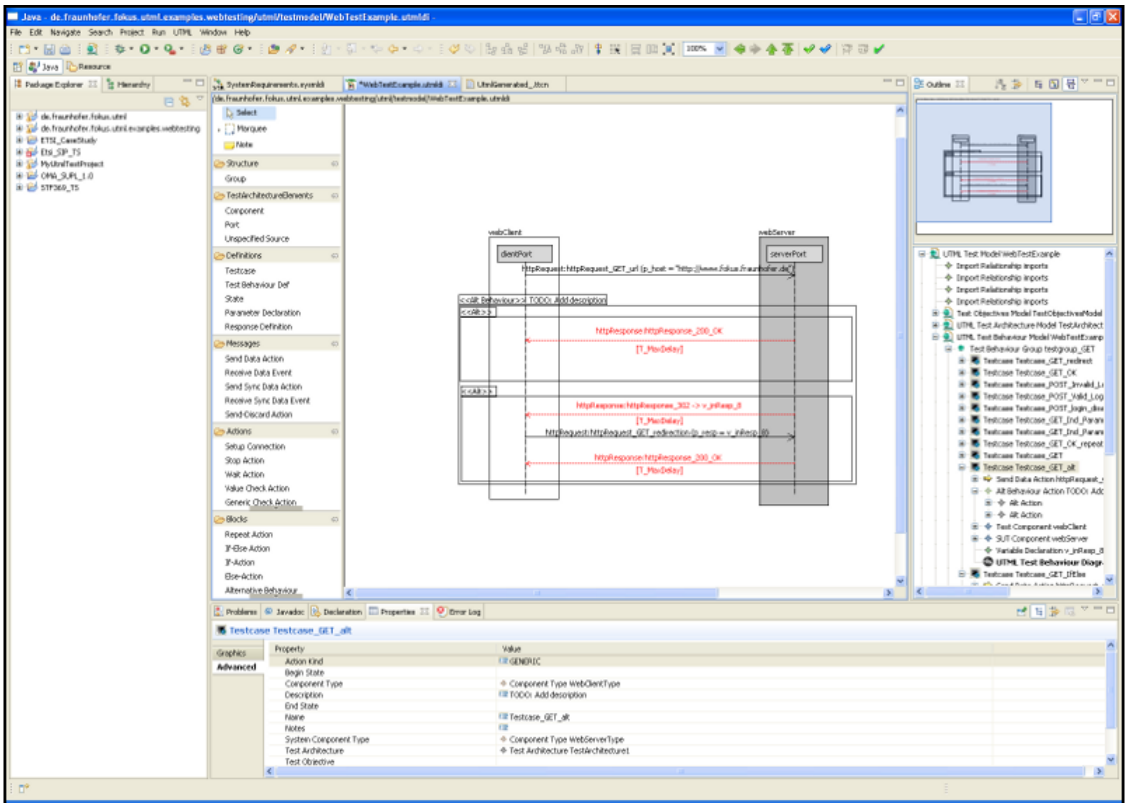
Wizards Wizards provide support to test modelling activities by guiding the process and ensuring that the test expert is provided the right set of available tools and possible choices at each step of that process. As a proof of concept, the prototype tool implemented in this thesis provides two main categories of wizards: *creation wizards* and *transformation wizards*. While *creation wizards* provide guidance for the instantiation of new elements to existing or newly created test models, *transformation wizards* guide the test designer through the process of creating another view to a test model from an existing view. An example of transformation wizard allows the creation of a test behaviour diagram (i.e. the behaviour view on a test model) from an existing test architecture diagram (i.e.

the architectural view). Also, to illustrate the potential of wizards for pattern-oriented test modelling, the prototype tool implements a wizard for creating new test architectures based on the architectural test patterns described in Section A.3 of Appendix A.

Technical Challenges

Several technical challenges were faced while developing the prototype tools. One of the main difficulties originated from poor documentation of the many features present in the TOPCASED platform. However, once this hurdle was crossed and the mechanisms of the platform were understood, it proved a very efficient tool for implementing a visual DSML like UTML. Customisation of the source code generated with TOPCASED also worked smoothly. Even whenever the code needed to be newly generated (e.g. after a modification to the meta-model or to the graphical editor's model), the manually modified code would be left untouched, provided it was annotated properly beforehand.

The MDTester Prototype Tool



Property	Value
Action Kind	GENERIC
Begin State	
Component Type	Component Type WebClientType
Description	TODO: Add description
End State	
Name	Testcase_GET_alt
Notes	
System Component Type	Component Type WebServerType
Test Architecture	Test Architecture TestArchitecture1
Test Objective	

Figure 6.3: Screenshot of UTML Prototype Tool

Figure 6.3 displays a screenshot of the MDTester (Model Driven Test EngineeRing) application developed as a proof of concept for this thesis. MDTester is a set of Eclipse plugins, which can installed on applications based on the Eclipse platform to provide an Integrated Development Environment (IDE) for modelling test systems, transforming the latter into executable test scripts, execute them and analyse the results.

For that purpose, MDTester provides the following features:

- Graphical Editor for all types of test diagrams defined by the UTML metamodel
- Tabular editor for all types of UTML test models.
- Test modelling policies based on black-box test patterns (e.g. filtering of selection choices, allowance/proscription of actions on test model elements)
- Automatic validation of test models against the UTML metamodel
- Automatic validation of test models against test patterns constraints expressed in the OMG's OCL notation
- Integration of externally defined OCL constraints for automatic validation of test models
- Automatic transformation of UTML test models into test scripting notations to obtain executable test scripts. Currently supported: TTCN-3, JUnit, XML.
- Plug-in API to support the seamless integration of additional external transformators

Please refer to MDTester's complete installation and user guide [160] for details on how to install and to use the prototype tools.

6.3 Evaluation: Example and Case Studies

6.3.1 An Example: Pattern Oriented MDT for a Web Application

Introduction

This section describes an example usage of MDTester, the prototype implementation developed in this work, to design functional tests for a web application, following the pattern-oriented test engineering approach. Firstly, the UTML test model designed for the test suite is described, then the process of transforming that test model into executable tests is explained.

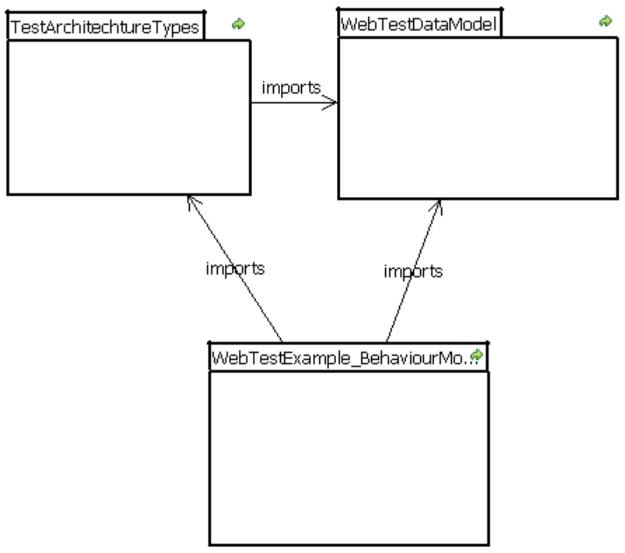


Figure 6.4: Overview of UTML Test Model for HTTP example

The Test Model

Figure 6.4 displays an overview of the UTML test model for the web application example. As depicted in that figure, the test model comprises three submodels, with the test behaviour model in a central role. The test behaviour model refers to the test architecture types model and the test data model for accessing test architecture type definitions and test data model elements respectively. Also visible in that figure is the fact that the test objectives and the test procedures model have been omitted from the root test model. Furthermore, a separate test architecture model (e.g. to define static test architectures) was omitted as well. This is an illustration of how optional test models might be skipped in the process towards executable test cases. For example, in case of harsh time constraints for the test project under development or for small-scale projects where no benefits are expected from such formalism in gathering test requirements and describing test procedures.

Modelling Test Data Figure 6.5 shows a view on the test data model, displaying the test data type definition for a HTTP request message. As depicted in that figure, the HTTP request message type is modeled as ***MessageTestDataType*** UTML element containing three fields. The figure also displays the type definitions associated to that data type, as well as the links between type fields and their associated type definitions (dashed lines between fields and type definitions).

After the types have been defined, modelling data instances, i.e. more or less concrete values to be used for sending impulses to the SUT or verify its response,

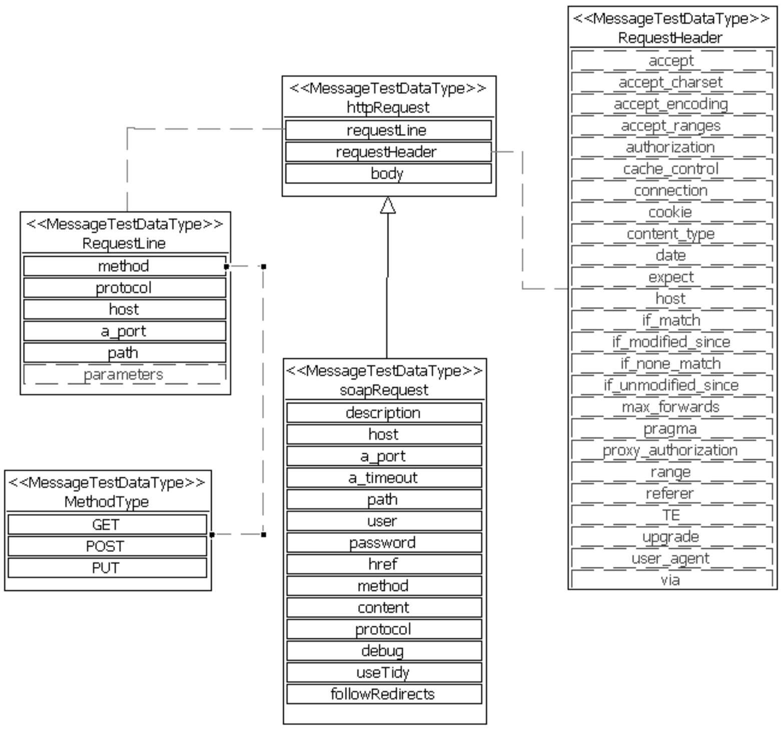


Figure 6.5: Test Data Type Definitions for HTTP example

is the next step. Figure 6.6 and figure 6.7 show examples of test data instances designed for this example. The model elements displayed on Figure 6.6 represent data instances suitable to be used to describe impulses on the SUT, while those on Figure 6.7 represent data instances for modelling expectations on the SUT's responses.

Modelling Test Behaviour Figure 6.8 displays an overview of the test behaviour model's tree structure for the web testing example. The behaviour model consists of a single group of testcases containing two testcases. The first test case (*Testcase_GET_redirect*) checks that the SUT (Web server) performs HTTP redirection correctly, when submitted with a given URL as input, while the second test case (*Testcase_GET_OK*) verifies that the SUT responds with a normal 200 OK HTTP response, if provided a valid URL as input. Figure 6.9 displays the UTML test sequence diagram for the *Testcase_GET_redirect* test case, which reflects the behaviour expected from a web server performing HTTP redirection.

Test Execution

To execute the tests modeled for this case study, it was chosen to use the JUnit test framework. That choice was mainly motivated by the existence and avail-

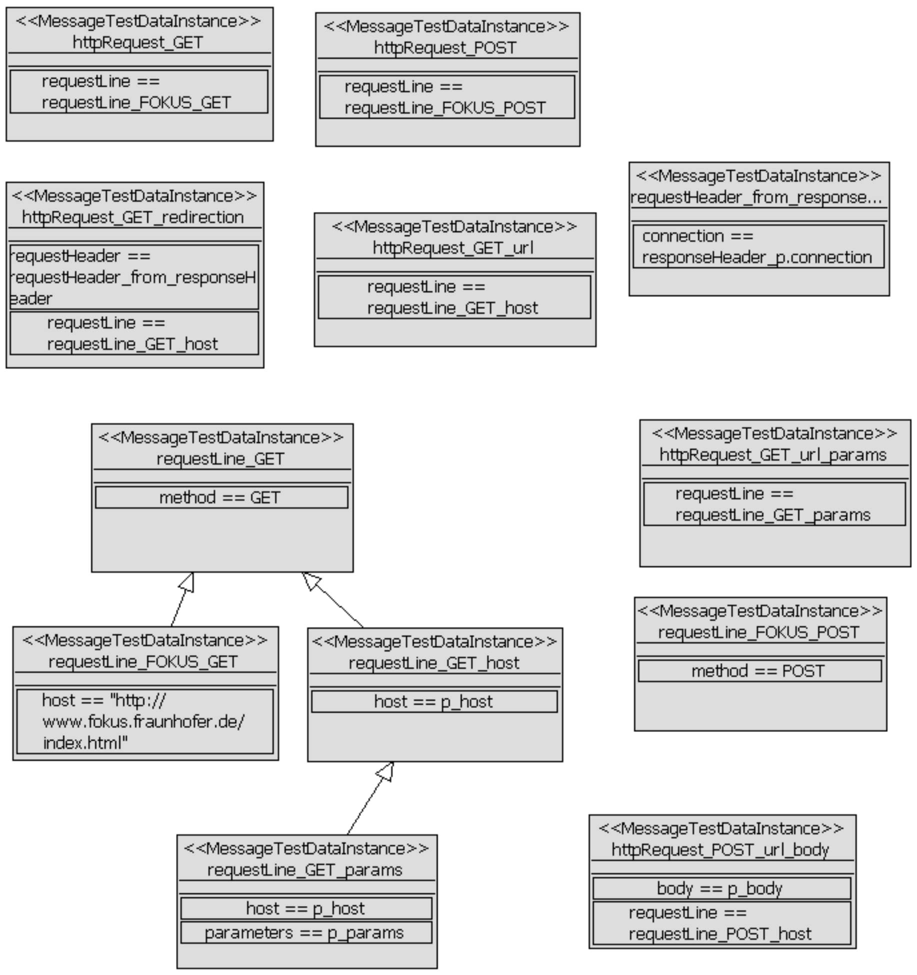

<<MessageTestDataInstance>>
httpRequest_GET
requestLine == requestLine_FOKUS_GET
<<MessageTestDataInstance>>
httpRequest_POST
requestLine == requestLine_FOKUS_POST
<<MessageTestDataInstance>>
requestHeader_from_response...
connection == responseHeader_p.connection
<<MessageTestDataInstance>>
httpRequest_GET_redirection
requestHeader == requestHeader_from_responseHeader
requestLine == requestLine_GET_host
<<MessageTestDataInstance>>
httpRequest_GET_url
requestLine == requestLine_GET_host
<<MessageTestDataInstance>>
requestLine_GET
method == GET
<<MessageTestDataInstance>>
httpRequest_GET_url_params
requestLine == requestLine_GET_params
<<MessageTestDataInstance>>
requestLine_FOKUS_GET
host == "http://www.fokus.fraunhofer.de/index.html"
<<MessageTestDataInstance>>
requestLine_GET_host
host == p_host
<<MessageTestDataInstance>>
requestLine_FOKUS_POST
method == POST
<<MessageTestDataInstance>>
requestLine_GET_params
host == p_host
parameters == p_params
<<MessageTestDataInstance>>
httpRequest_POST_url_body
body == p_body
requestLine == requestLine_POST_host

Figure 6.6: Elements of UTML Test Data Model for HTTP example: Impulses

ability of the HTTPUnit framework that relies on JUnit to provide a convenient API for performing all types test operations using the HTTP protocol. The aim was to avoid the additional burden of designing and implementing yet another test execution environment or of implementing the complex adaptation layer for one of the existing test execution environments.

However, a prerequisite to test execution is the transformation of the test model into a notation that can be handled by the target test framework (JAVA-JUnit in this case). The transformation was achieved automatically, using the MDTester tool's JUnit export-plugin. Listing 6.2 and listing 6.3 display excerpts from the source code automatically generated from the UTML test model. While listing 6.2 displays the JUnit test suite mapping the *WebTestExample_BehaviourModel* test behaviour model, listing 6.3 displays the source code mapping the *Test-case_GET_redirect* test case modeled on figure 6.9

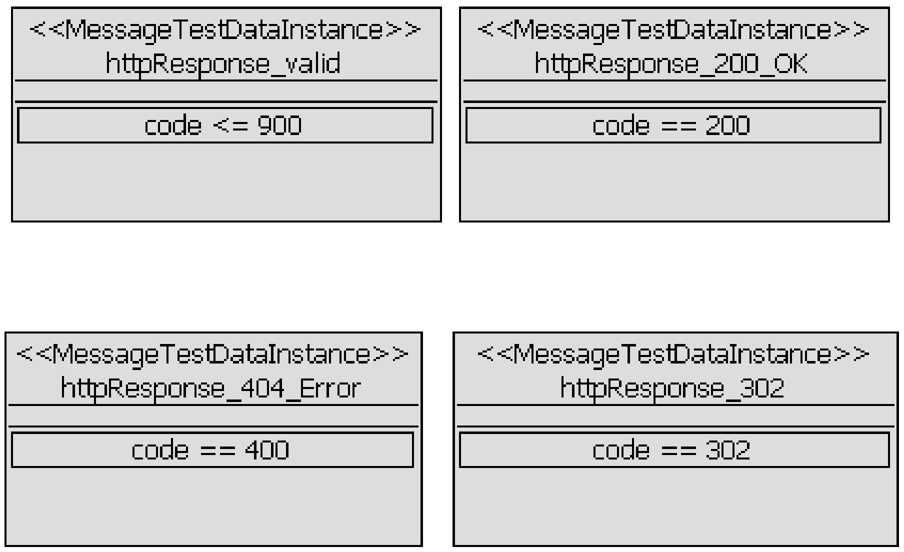
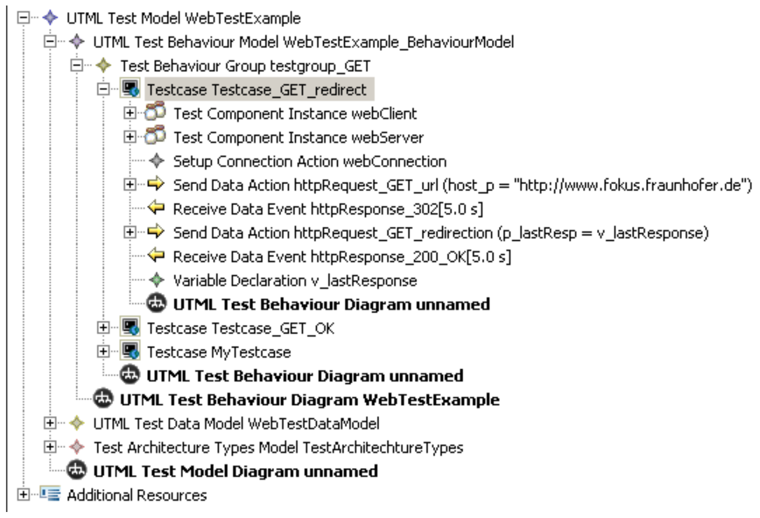


Figure 6.7: Elements of UTML Test Data Model for HTTP example: Responses



```
UTML Test Model WebTestExample
  UTML Test Behaviour Model WebTestExample_BehaviourModel
    Test Behaviour Group testgroup_GET
      Testcase Testcase_GET_redirect
        Test Component Instance webClient
        Test Component Instance webServer
        Setup Connection Action webConnection
        Send Data Action httpRequest_GET_url (host_p = "http://www.fokus.fraunhofer.de")
        Receive Data Event httpResponse_302[5.0 s]
        Send Data Action httpRequest_GET_redirection (p_lastResp = v_lastResponse)
        Receive Data Event httpResponse_200_OK[5.0 s]
        Variable Declaration v_lastResponse
        UTML Test Behaviour Diagram unnamed
      Testcase Testcase_GET_OK
      Testcase MyTestcase
      UTML Test Behaviour Diagram unnamed
    UTML Test Behaviour Diagram WebTestExample
  UTML Test Data Model WebTestDataModel
  Test Architecture Types Model TestArchitechtureTypes
  UTML Test Model Diagram unnamed
Additional Resources
```

Figure 6.8: Structure of Test Behaviour Model for HTTP example

```
import de.fraunhofer.fokus.testing.web.http.*;

import junit.framework.Test;
import junit.framework.TestSuite;

/**
 * This JUnit test suite has been automatically generated from a UTML test
 * model. Modifications on this source code will not
 * be taken into account by the generator in subsequent
 * operations Please make sure you keep a copy of
```

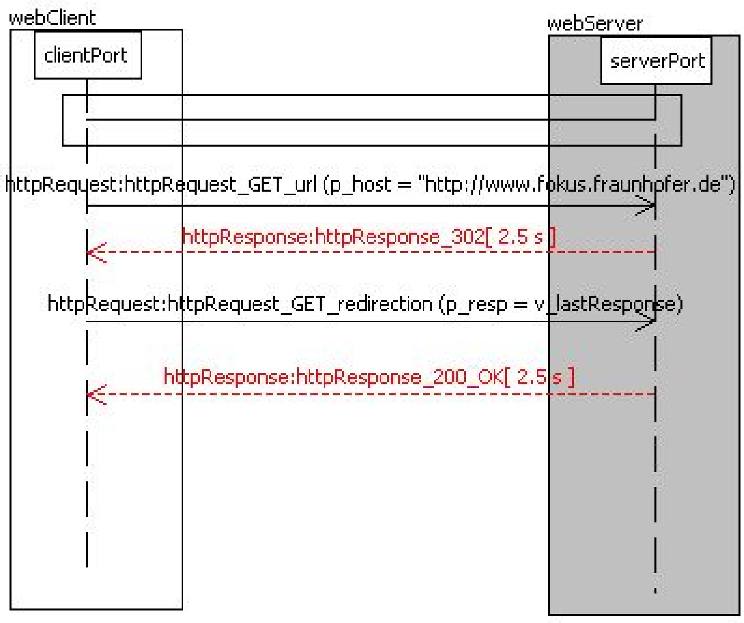

webClient
clientPort
webServer
serverPort
httpRequest:httpRequest_GET_url (p_host = "http://www.fokus.fraunhofer.de")
httpResponse:httpResponse_302[2.5 s]
httpRequest:httpRequest_GET_redirection (p_resp = v_lastResponse)
httpResponse:httpResponse_200_OK[2.5 s]

Figure 6.9: Test Behaviour Diagram for HTTP redirecting scenario

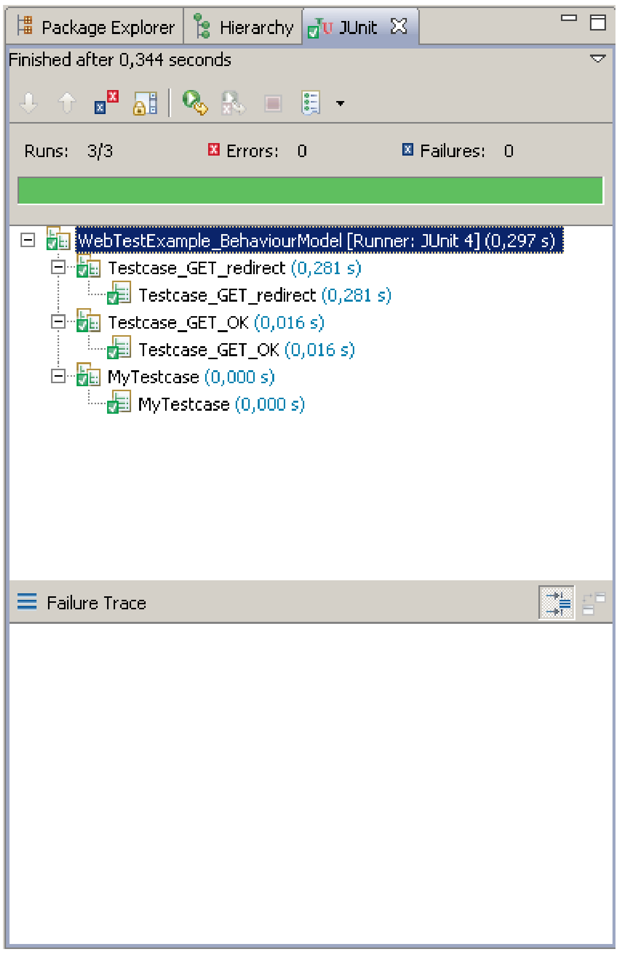
```
 * the file , before re-starting the transformation process.
 */
public class WebTestExample_BehaviourModel {

 public static Test suite() {
  TestSuite suite = new TestSuite("WebTestExample_BehaviourModel");
  suite.addTestSuite(Testcase_GET_redirect.class);
  suite.addTestSuite(Testcase_GET_OK.class);
  suite.addTestSuite(MyTestcase.class);

  return suite;
 }
}
```

Listing 6.2: Generated JUnit Code for the HTTP example

```
/**
 * @purpose TP version:
 * @desc: A testcase featuring the HTTP GET command and its usage
 * to retrieve a web page content
 * Test procedures:
 *
 */
public class Testcase_GET_redirect extends HttpTestcase {

 public Testcase_GET_redirect() {
  super("Testcase_GET_redirect", "Automatically generated test case");
 }

 public void testTestcase_GET_redirect() throws Exception {
  // Setup architecture

  // Preamble

  // Test body

  createHttpRequest("http://www.fokus.fraunhofer.de");
  setURL("http://www.fokus.fraunhofer.de");

  setMethod(MethodKind.GET());
  sendHttpRequest(5);
  HTTP_Response v_lastResponse = getHttpResponse();
  checkHttpResponseDelay(5);
  checkHttpResponse_code("EQUALS", "302");

  createHttpRequest(v_lastResponse.getHeader("location"));
  setURL(v_lastResponse.getHeader("location"));

  setMethod(MethodKind.GET());
  sendHttpRequest(5);
  checkHttpResponseDelay(5);
  checkHttpResponse_code("EQUALS", "200");
  // Postamble
 }

}// end Testcase_GET_redirect
```

Listing 6.3: Generated JUnit source code for the Testcase_GET_redirect test case displayed in figure 6.9

One of the biggest benefits of generating JUnit tests in combination with Eclipse-based (test) development environments, is their ease of use, additionally to the fact that they can be executed without any further implementation effort required. Figure 6.10 shows a screen capture of the test execution window, which is integrated in the development environment. However, it is worth noting that in case of a JUnit assertion failing, thus leading to a *FAIL* verdict for the test case, an analysis of the reasons for failure appears to be less convenient.



Finished after 0,344 seconds

Runs: 3/3 Errors: 0 Failures: 0

```
WebTestExample_BehaviourModel [Runner: JUnit 4] (0,297 s)
    Testcase_GET_redirect (0,281 s)
        Testcase_GET_redirect (0,281 s)
    Testcase_GET_OK (0,016 s)
        Testcase_GET_OK (0,016 s)
    MyTestcase (0,000 s)
        MyTestcase (0,000 s)
```

Figure 6.10: Screenshot of JUnit test execution for HTTP example

6.3.2 The IMS Case Study

Introduction

This section describes a case study featuring the usage of pattern-oriented test engineering to design and implement an IMS conformance test system.

The Test Model

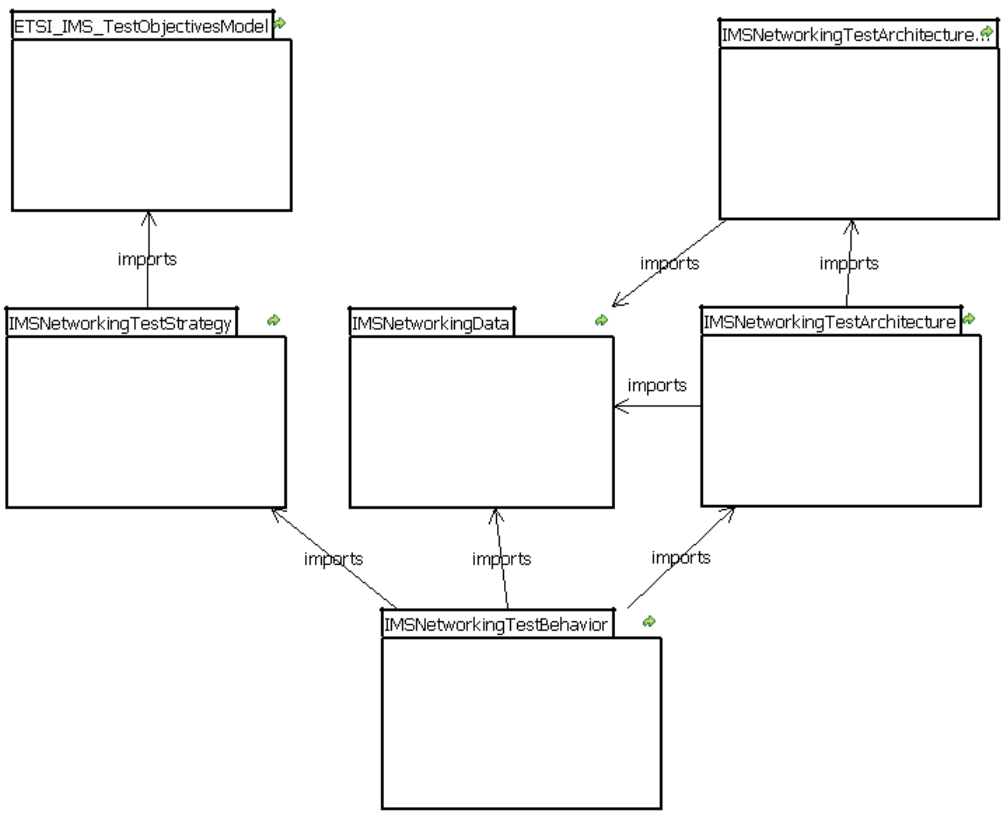

ETSI_IMS_TestObjectivesModel
IMSNetworkingTestArchitecture...
imports
imports
imports
IMSNetworkingTestStrategy
IMSNetworkingData
IMSNetworkingTestArchitecture
imports
imports
imports
imports
IMSNetworkingTestBehavior

Figure 6.11: Overview of UTML Test Model for IMS case study

The aim of the test model for the IMS case study was to cover all aspects of the test development process, starting from designing a test plan based on conformance requirements specified in the various IMS standards, through to executable test cases in the form of TTCN-3 scripts. Figure 6.11 depicts a graphical view on the test model's root element and illustrates that process, reflected in the structure of the test model. As depicted in that figure, the test model consists of six sub-models, each of them covering a specific aspect of test design, according to the separation-of-concerns pattern described in section A.1.1. The next sections provide a detailed description of each of those sub-models.

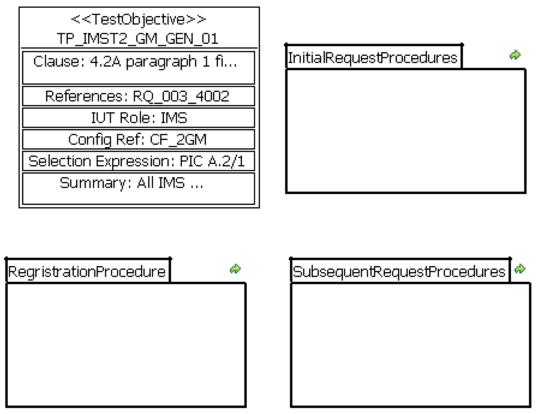


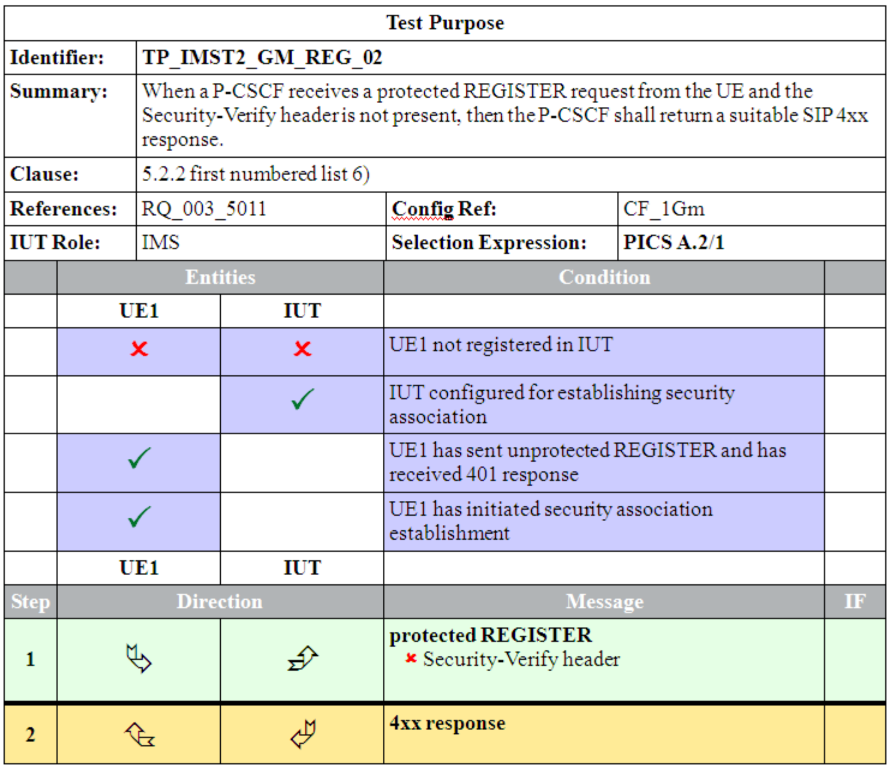
Figure 6.12: Overview of Test Objectives Diagram for IMS case study

Designing the Test Plan Following the process illustrated in Figure 4.1 and described in Section 4.2, the test engineering process starts with the design of a test plan in the form of a test objectives model. Figure 6.12 displays a view on the *ETSI_IMS_TestObjectivesModel* test model, which contains three groups of test objectives. The structure of the test objectives model corresponds to that of the original test suite structure (TSS) document defined by the European Telecommunication Standardization Institue (ETSI). The TSS document was provided as a set of text format files (MS-Word), containg test purposes written in the TPLan notation. Figure 6.13 shows a sample test purpose for IMS conformance testing specified with TPLan. Each TPLan test purpose consists of two parts:

- The declaration part comprises the first five upper rows of the table. It contains an identifier, a summary description of the test purpose and several other information on the test purpose.
- The behavioural part comprises the lower part of the table and describes a sequence of actions and observations to be performed for the test case.

In accordance to the methodology proposed in this thesis, the declaration part of the TPLan test purposes maps to UTML test objectives. Therefore, the essential part of the test objectives modelling activity consisted in transforming those TPLan test purposes into UTML test objectives model elements, following that mapping. Figure 6.12 also displays the visualisation of a test objective element resulting from that transformation, while Figure 6.14 shows a tree view on the whole test objectives model with the associated diagrams and the other related test models.

Designing the Test Procedures The test procedures model for the case study was also obtained by transforming the TPLan test purposes into UTML test proce-

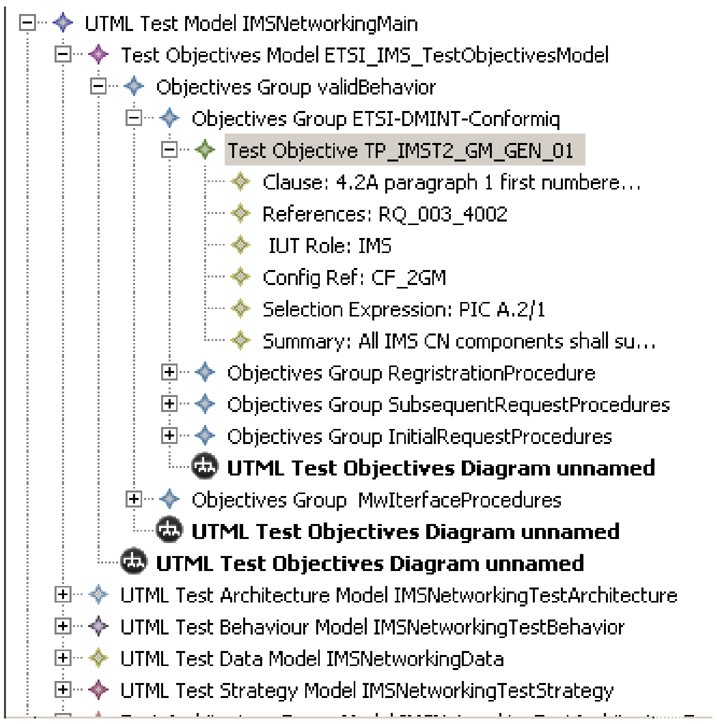


Test Purpose				
Identifier:	**TP_IMST2_GM_REG_02**			
Summary:	When a P-CSCF receives a protected REGISTER request from the UE and the Security-Verify header is not present, then the P-CSCF shall return a suitable SIP 4xx response.			
Clause:	5.2.2 first numbered list 6)			
References:	RQ_003_5011	**Config Ref:**	CF_1Gm	
IUT Role:	IMS	**Selection Expression:**	**PICS A.2/1**	
	Entities		**Condition**	
	UE1	**IUT**		
	✗	✗	UE1 not registered in IUT	
		✓	IUT configured for establishing security association	
	✓		UE1 has sent unprotected REGISTER and has received 401 response	
	✓		UE1 has initiated security association establishment	
	UE1	**IUT**		
Step	**Direction**		**Message**	**IF**
1	→	→	**protected REGISTER** ✗ Security-Verify header	
2	←	←	**4xx response**	

Figure 6.13: Example of TPLan Test Purpose for IMS Conformance Testing

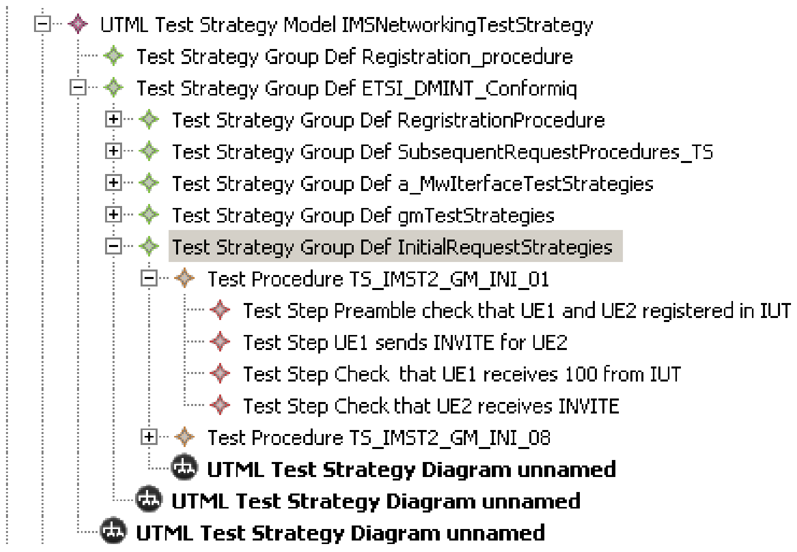
dures. However, to obtain the test procedure, the behavioural part of the test purpose (cf. figure 6.13) was taken as input. Figure 6.15 displays an overview of the test procedures model for the IMS case study in its tree representation. As depicted in that figure, the test procedures model has the same structure as the test objectives model, since both are derived from the same TSS document. Further, figure 6.16 displays a graphical representation of two selected test procedures resulting from the manual transformation process.

Designing Test Data After the test procedures have been defined, the test design process moved to the next phase of design the test data required for the IP Multimedia Subsystem (IMS) conformance test suite. Figure 6.17 displays the root test data diagram for the IMS case study and at the same time illustrates the structure of that model, which comprises eight groups of test data modelling elements. As depicted in that figure, some of the groups contain data type definitions for a given protocol used in the IMS context, while others contain data instances (i.e. concrete values) to be used for modelling test behaviour. E.g. the *SipDataTypes* group contains data type definitions for the Session Initiation Protocol (SIP) protocol [137], while the *SipDataInstances* group contains concrete test values for those data types. Additionally, more groups might be created for



```
UTML Test Model IMSNetworkingMain
  Test Objectives Model ETSI_IMS_TestObjectivesModel
    Objectives Group validBehavior
      Objectives Group ETSI-DMINT-Conformiq
        Test Objective TP_IMST2_GM_GEN_01
          Clause: 4.2A paragraph 1 first numbere...
          References: RQ_003_4002
           IUT Role: IMS
          Config Ref: CF_2GM
          Selection Expression: PIC A.2/1
          Summary: All IMS CN components shall su...
        Objectives Group RegristrationProcedure
        Objectives Group SubsequentRequestProcedures
        Objectives Group InitialRequestProcedures
        UTML Test Objectives Diagram unnamed
      Objectives Group  MwIterfaceProcedures
      UTML Test Objectives Diagram unnamed
    UTML Test Objectives Diagram unnamed
  UTML Test Architecture Model IMSNetworkingTestArchitecture
  UTML Test Behaviour Model IMSNetworkingTestBehavior
  UTML Test Data Model IMSNetworkingData
  UTML Test Strategy Model IMSNetworkingTestStrategy
```

Figure 6.14: Overview of Test Objectives Model for IMS case study



```
UTML Test Strategy Model IMSNetworkingTestStrategy
  Test Strategy Group Def Registration_procedure
  Test Strategy Group Def ETSI_DMINT_Conformiq
    Test Strategy Group Def RegristrationProcedure
    Test Strategy Group Def SubsequentRequestProcedures_TS
    Test Strategy Group Def a_MwIterfaceTestStrategies
    Test Strategy Group Def gmTestStrategies
    Test Strategy Group Def InitialRequestStrategies
      Test Procedure TS_IMST2_GM_INI_01
        Test Step Preamble check that UE1 and UE2 registered in IUT
        Test Step UE1 sends INVITE for UE2
        Test Step Check  that UE1 receives 100 from IUT
        Test Step Check that UE2 receives INVITE
      Test Procedure TS_IMST2_GM_INI_08
      UTML Test Strategy Diagram unnamed
    UTML Test Strategy Diagram unnamed
  UTML Test Strategy Diagram unnamed
```

Figure 6.15: Overview of Test Procedures Diagram for IMS case study

generic data types or data instances, e.g. for global test parameters. Figure 6.18 shows a view on the test data model displaying sample test data type definitions for the SIP protocol. As depicted in that figure, most SIP request types are based on a generic type definition (*SIPRequestType*), which they extend or

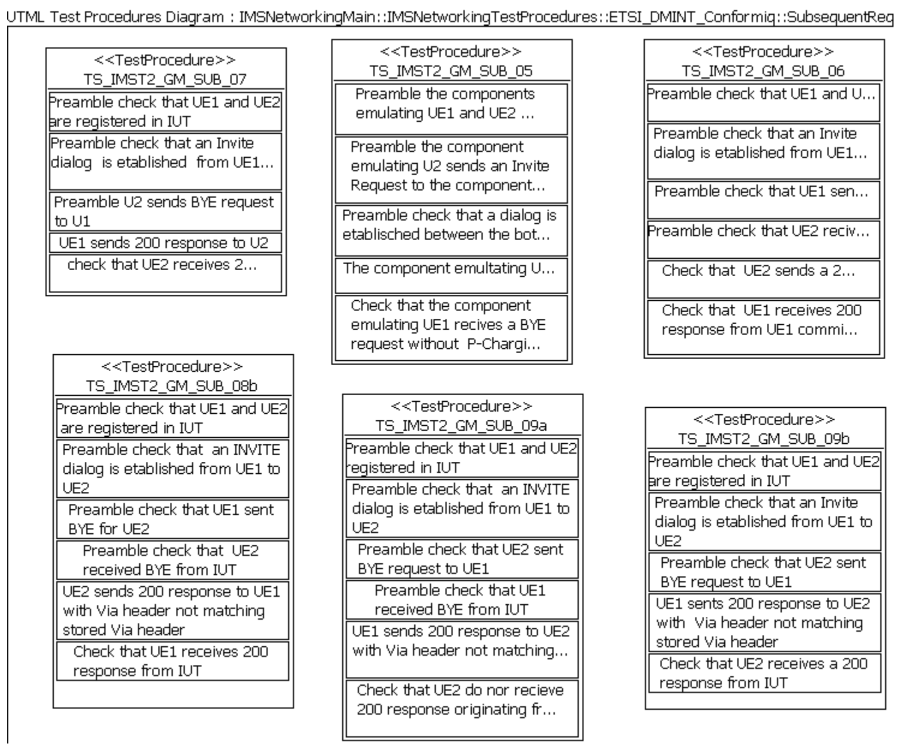

UTML Test Procedures Diagram : IMSNetworkingMain::IMSNetworkingTestProcedures::ETSI_DMINT_Conformiq::SubsequentReq
<<TestProcedure>>
TS_IMST2_GM_SUB_07
Preamble check that UE1 and UE2 are registered in IUT
Preamble check that an Invite dialog is established from UE1...
Preamble U2 sends BYE request to U1
UE1 sends 200 response to U2
check that UE2 receives 2...
<<TestProcedure>>
TS_IMST2_GM_SUB_05
Preamble the components emulating UE1 and UE2 ...
Preamble the component emulating U2 sends an Invite Request to the component...
Preamble check that a dialog is etablisched between the bot...
The component emultating U...
Check that the component emulating UE1 recives a BYE request without P-Chargi...
<<TestProcedure>>
TS_IMST2_GM_SUB_06
Preamble check that UE1 and U...
Preamble check that an Invite dialog is etablished from UE1...
Preamble check that UE1 sen...
Preamble check that UE2 reciv...
Check that UE2 sends a 2...
Check that UE1 receives 200 response from UE1 commi...
<<TestProcedure>>
TS_IMST2_GM_SUB_08b
Preamble check that UE1 and UE2 are registered in IUT
Preamble check that an INVITE dialog is etablished from UE1 to UE2
Preamble check that UE1 sent BYE for UE2
Preamble check that UE2 received BYE from IUT
UE2 sends 200 response to UE1 with Via header not matching stored Via header
Check that UE1 receives 200 response from IUT
<<TestProcedure>>
TS_IMST2_GM_SUB_09a
Preamble check that UE1 and UE2 registered in IUT
Preamble check that an INVITE dialog is etablished from UE1 to UE2
Preamble check that UE2 sent BYE request to UE1
Preamble check that UE1 received BYE from IUT
UE1 sends 200 response to UE2 with Via header not matching...
Check that UE2 do nor recieve 200 response originating fr...
<<TestProcedure>>
TS_IMST2_GM_SUB_09b
Preamble check that UE1 and UE2 are registered in IUT
Preamble check that an Invite dialog is etablished from UE1 to UE2
Preamble check that UE2 sent BYE request to UE1
UE1 sents 200 response to UE2 with Via header not matching stored Via header
Check that UE2 receives a 200 response from IUT

Figure 6.16: Example Test Procedures for IMS case study

restrict using additional constraints on the contained fields.

Finally, figure 6.19 displays sample test data instances from the test model, illustrating the extension mechanism allowing the reuse of previously defined test data instances to define new ones.

Designing the Test Architecture The design of the test architecture is divided in two phases. The first phase consists in defining types for the architectural elements that are required for building test architectures. Then, in a second phase, the test architectures are modeled, based on instances of the types defined in the first step.

Defining Types for the Test Architecture Figure 6.20 displays an overview of the test architecture types diagram for the IMS case study, which contains a series of test component types and a group containing port type definitions. A more detailed view on the test architecture types model is displayed on figure 6.21 in a tree representation of that model. The definition of port types and component types require access to test data information (e.g. data type definitions). Therefore, as already depicted on figure 6.11, the test architecture types model refers to the test data model to achieve that purpose.

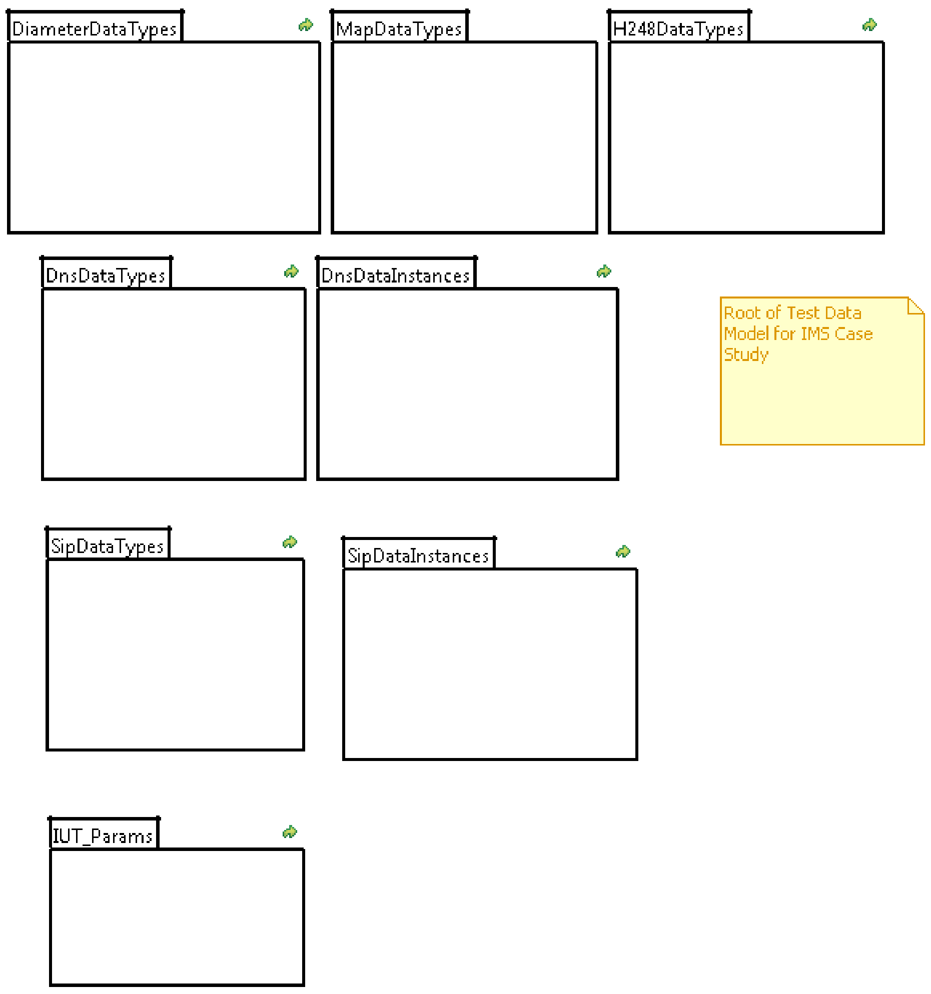


Figure 6.17: Root Test Data Diagram for IMS case study

Designing Static Test Architectures Based on the type definitions for test architecture elements modeled in the previous step the static test architectures could be modeled as well. Figure 6.22 displays the root test architecture diagram for the IMS case study. As depicted in that figure, the test architecture model contains four different static test architecture, each of which is represented as a cloud in its graphical form. Those test architectures were selected among the 11 defined in the TSS document mentioned previously, because the test cases selected for the case study required them. A test architecture diagram for a sample test architecture used in the case study is shown in Figure 6.23.

Modelling Test Behaviour The test behaviour model has the most dependencies to other elements of the test model. Therefore it can only be designed, once those test models are finished and ready to be refered to. Figure 6.24 displays an

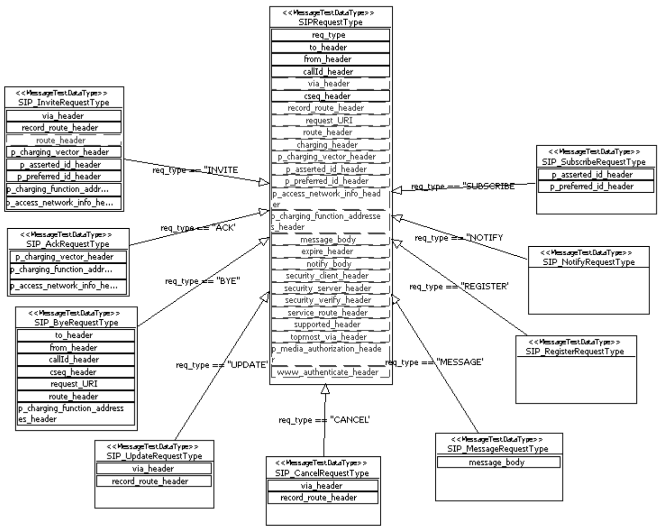


Figure 6.18: Test Data Type Definitions for IMS case study

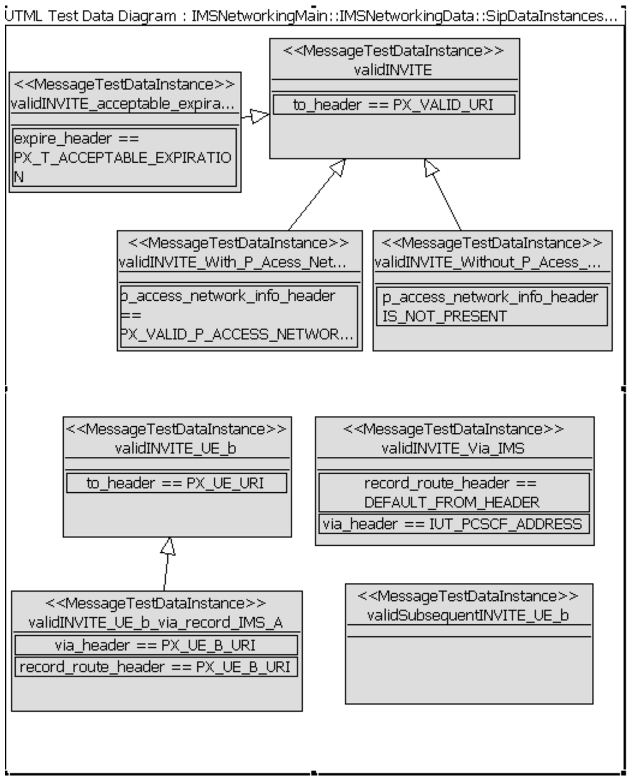

UTML Test Data Diagram : IMSNetworkingMain::IMSNetworkingData::SipDataInstances...
<<MessageTestDataInstance>>
validINVITE
to_header == PX_VALID_URI
<<MessageTestDataInstance>>
validINVITE_acceptable_expira...
expire_header == PX_T_ACCEPTABLE_EXPIRATION
<<MessageTestDataInstance>>
validINVITE_With_P_Acess_Net...
p_access_network_info_header == PX_VALID_P_ACCESS_NETWOR...
<<MessageTestDataInstance>>
validINVITE_Without_P_Acess_...
p_access_network_info_header IS_NOT_PRESENT
<<MessageTestDataInstance>>
validINVITE_UE_b
to_header == PX_UE_URI
<<MessageTestDataInstance>>
validINVITE_Via_IMS
record_route_header == DEFAULT_FROM_HEADER
via_header == IUT_PCSCF_ADDRESS
<<MessageTestDataInstance>>
validINVITE_UE_b_via_record_IMS_A
via_header == PX_UE_B_URI
record_route_header == PX_UE_B_URI
<<MessageTestDataInstance>>
validSubsequentINVITE_UE_b

Figure 6.19: Test Data Instances for IMS case study

overview of the test behaviour model for the IMS case study. The test behaviour models consists of five groups represented each as package symbol:

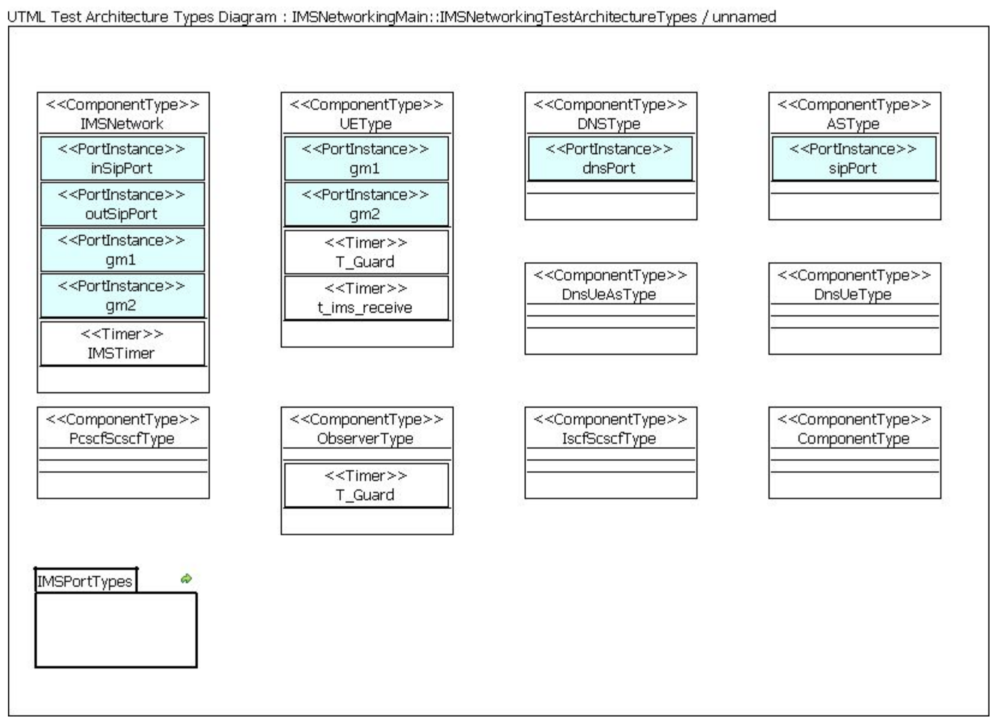
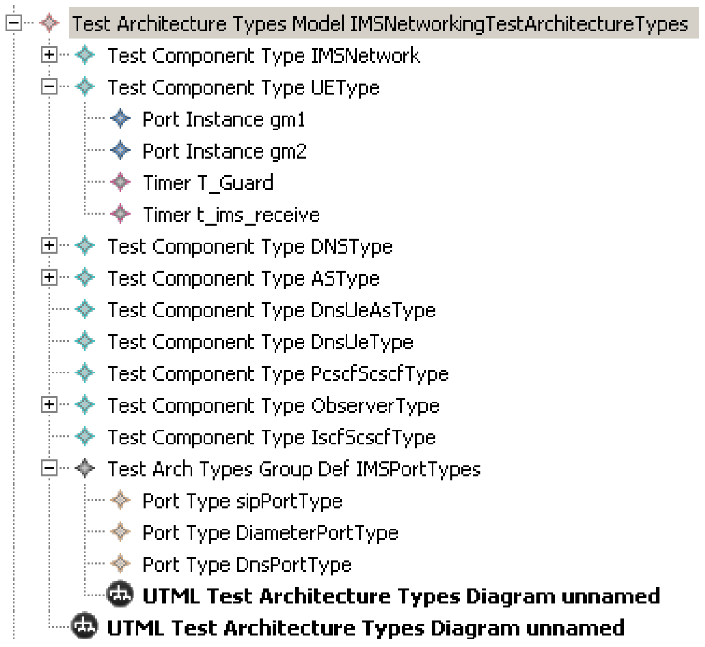

UTML Test Architecture Types Diagram : IMSNetworkingMain::IMSNetworkingTestArchitectureTypes / unnamed
<<ComponentType>> IMSNetwork
<<PortInstance>> inSipPort
<<PortInstance>> outSipPort
<<PortInstance>> gm1
<<PortInstance>> gm2
<<Timer>> IMSTimer
<<ComponentType>> UEType
<<PortInstance>> gm1
<<PortInstance>> gm2
<<Timer>> T_Guard
<<Timer>> t_ims_receive
<<ComponentType>> DNSType
<<PortInstance>> dnsPort
<<ComponentType>> ASType
<<PortInstance>> sipPort
<<ComponentType>> DnsUeAsType
<<ComponentType>> DnsUeType
<<ComponentType>> PcscfScscfType
<<ComponentType>> ObserverType
<<Timer>> T_Guard
<<ComponentType>> IscfScscfType
<<ComponentType>> ComponentType
IMSPortTypes

Figure 6.20: Overview of Test Architecture Types Diagram for IMS case study



```
Test Architecture Types Model IMSNetworkingTestArchitectureTypes
    Test Component Type IMSNetwork
    Test Component Type UEType
        Port Instance gm1
        Port Instance gm2
        Timer T_Guard
        Timer t_ims_receive
    Test Component Type DNSType
    Test Component Type ASType
    Test Component Type DnsUeAsType
    Test Component Type DnsUeType
    Test Component Type PcscfScscfType
    Test Component Type ObserverType
    Test Component Type IscfScscfType
    Test Arch Types Group Def IMSPortTypes
        Port Type sipPortType
        Port Type DiameterPortType
        Port Type DnsPortType
        UTML Test Architecture Types Diagram unnamed
    UTML Test Architecture Types Diagram unnamed
```

Figure 6.21: Overview of Test Architecture Types Model for IMS case study

- The *timers* group contains timer declarations.

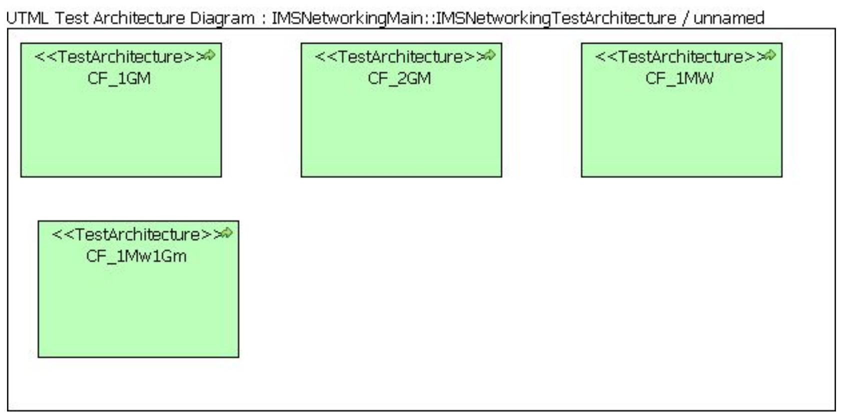


Figure 6.22: Root Test Architecture Diagram for IMS

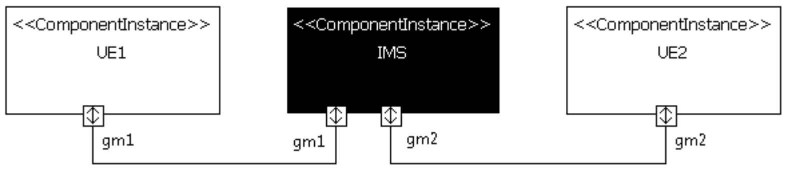


Figure 6.23: Test Architecture Diagram for a static IMS test architecture

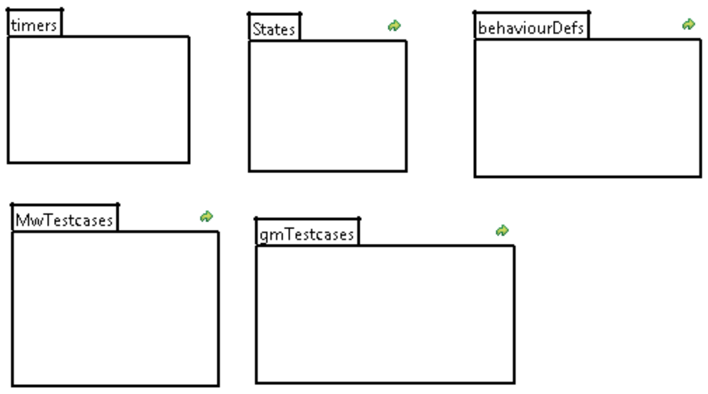
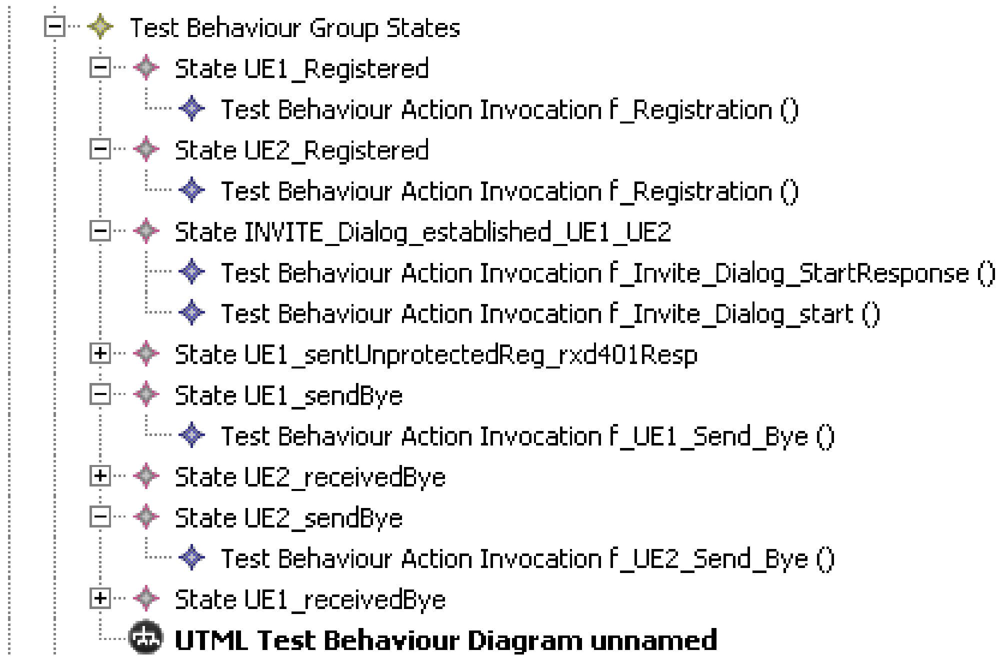


Figure 6.24: Overview of Test Behaviour diagram for IMS case study

- The *behaviourDefs* group contains reusable behaviour definition elements similar to functions in functional programming languages that can be invoked in test cases or other behaviour definitions.

- The *States* group contains elements modelling the possible state in which components of the test architecture can find themselves in. Those states can then be used as pre-/post conditions for test cases and other test behaviours. Figure 6.25 shows the structure of the *States* group with further details on some of the state definitions it contains. As displayed in that figure, each state contains a series of triggering function invocations that it requires to be entered.

- Finally, the *MwTestcases* and the *gmTestcases* groups contain the test cases. An example of one test sequence diagram for a test case is displayed at figure 6.26.



```
Test Behaviour Group States
    State UE1_Registered
        Test Behaviour Action Invocation f_Registration ()
    State UE2_Registered
        Test Behaviour Action Invocation f_Registration ()
    State INVITE_Dialog_established_UE1_UE2
        Test Behaviour Action Invocation f_Invite_Dialog_StartResponse ()
        Test Behaviour Action Invocation f_Invite_Dialog_start ()
    State UE1_sentUnprotectedReg_rxd401Resp
    State UE1_sendBye
        Test Behaviour Action Invocation f_UE1_Send_Bye ()
    State UE2_receivedBye
    State UE2_sendBye
        Test Behaviour Action Invocation f_UE2_Send_Bye ()
    State UE1_receivedBye
    UTML Test Behaviour Diagram unnamed
```

Figure 6.25: Modelling of states for the IMS test model

Test Execution

Listing 6.4 displays an extract from the TTCN-3 source code resulting from the automated transformation of the UTML test model into TTCN-3, using the TTCN-3 backend developed with the prototype tool. The transformation is performed according to the mapping rules defined in Section B and illustrates how parallel test components are designed with UTML and how they may be translated into a test specification language such as TTCN-3. As displayed in the listing, the behaviour of each test component is translated into a TTCN-3 function, which is then invoked in the test case when the component instance is started. Logically, passive test components (i.e. those for which the first action consists of waiting for an incoming message or for user interaction) are started first, before active components are started in their turn.

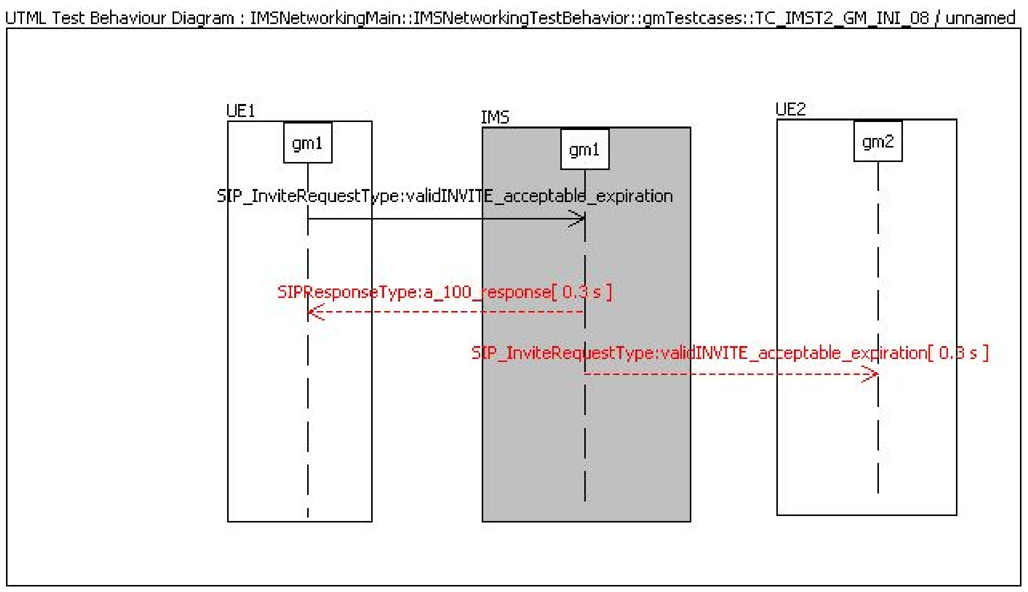

UTML Test Behaviour Diagram : IMSNetworkingMain::IMSNetworkingTestBehavior::gmTestcases::TC_IMST2_GM_INI_08 / unnamed
UE1
IMS
UE2
gm1
gm1
gm2
SIP_InviteRequestType:validINVITE_acceptable_expiration
SIPResponseType:a_100_response[0.3 s]
SIP_InviteRequestType:validINVITE_acceptable_expiration[0.3 s]

Figure 6.26: Test Behaviour diagram for a sample IMS test case

It is worth noting that the TTCN-3 code displayed in Listing 6.4 was automatically generated in its entirety and was compilable right away, without any additional manual editing.

```
  * Functions for test component behaviours
  */
 group TC_IMST2_GM_INI_08_functions {
  function f_TC_IMST2_GM_INI_08_UE2_behaviour() runs on UEType {
   T_Guard.start;
   alt {
    [] gm2.receive(validINVITE_acceptable_expiration) {
     T_Guard.stop;
     setverdict(
      pass,
     "*** F_TC_IMST2_GM_INI_08_UE2_BEHAVIOUR(): "
     &"SIP_InviteRequestType message "
     &"received as expeced ***" );
    }
    [] T_Guard.timeout {
     setverdict(
      fail,
     "*** F_TC_IMST2_GM_INI_08_UE2_BEHAVIOUR(): "
     &"Time out while expecting "
     &"SIP_InviteRequestType message ***" );
    }
   }
  } // end f_TC_IMST2_GM_INI_08_UE2_behaviour
  function f_TC_IMST2_GM_INI_08_UE1_behaviour() runs on UEType {
   gm1.send(validINVITE_acceptable_expiration);
   T_Guard.start;
   alt {
    [] gm1.receive(a_100_response) {
     T_Guard.stop;
```

```
      setverdict(
       pass,
      "*** F_TC_IMST2_GM_INI_08_UE1_BEHAVIOUR(): "
      &"SIPResponseType message "&
      &"received as expected ***" );
     }
     [] T_Guard.timeout {
      setverdict(
       fail,
      "*** F_TC_IMST2_GM_INI_08_UE1_BEHAVIOUR(): "
      &"Time out while expecting "
      &"SIPResponseType message ***" );
     }
    }
  } // end f_TC_IMST2_GM_INI_08_UE1_behaviour
 } // end TC_IMST2_GM_INI_08_functions
/**
 * @purpose
 *      TP version: Clause : 5.2.7.2, 5.2.8.3, RFC4028
 * References :
 *      RQ_003_5064, RQ_003_5068, RQ_003_5065
 *  IUT Role : IMS
 * Config Ref :
 *      CF_2GM
 * Selection Expression : PICS A.2/1, A.3/12.1.1
 * Summary :
 *  When a
 *      P-CSCF requires periodic refreshment of a session established after
 *      receiving a SIP INVITE request from a UE and the Session-Expires
 *      header of the INVITE request indicates acceptable refresh frequency
 *      then it forwards the request to the destination UE and returns a 100
 *      (Trying) to the originating UE.
 *@desc: TODO: Add description
 *Test procedure:
 *  1: Preamble check that UE1 and UE2 registered in IUT
 *  2: UE1 sends INVITE for UE2
 *  3: Check that UE1 receives 100 response from IUT
 *  4: Check that UE2 receives INVITE with a valid Session-Expires
 *      header
 */
testcase TC_IMST2_GM_INI_08() runs on ComponentType system IMSNetwork {
 // Test execution
 // Setup configuration: CF_2GM
 // Instanciate test components
 var UEType UE2 := UEType.create;
 var UEType UE1 := UEType.create;
 map(UE1:gm1, system:gm1);
 map(UE2:gm2, system:gm2);

 // Preamble
 // Test body
 // First start passive components
 UE2.start(f_TC_IMST2_GM_INI_08_UE2_behaviour());
 // Then, start active components
 UE1.start(f_TC_IMST2_GM_INI_08_UE1_behaviour());
 // Wait until components complete their job
 UE2.done;
 UE1.done;
```

```
  // Postamble
  // Teardown configuration: CF_2GM
  unmap(UE1:gm1, system:gm1);
  unmap(UE2:gm2, system:gm2);
} // end TC_IMST2_GM_INI_08
```

Listing 6.4: Generated TTCN-3 Code for the IMS case study

Evaluation

Quantitative Analysis A key metric for quantitative analysis of any development process is productivity gain. Evaluating the productivity of pattern oriented test development is a relatively straightforward task. For that purpose, the output (e.g. number of implemented test cases) would simply have to be correlated with the invested effort (e.g. number of person-days/person-months involved) for a project or a series of projects. However, to measure the impact of introducing a new approach on that productivity is a less trivial task, because productivity data before and after the introduction of the new approach need to be compared with each other. Ideally, to ensure a fair comparison, at least the following conditions need to be fulfilled:

- Both methodologies should be applied on the same case study: The starting point for both test development approaches should be the same system specification or test plan, targeting the same SUT
- Separate teams should apply the methodology, each on its side in a separate project.
- The same time frame will apply to both projects and results will be collected at the end for evaluation.
- Both teams should have comparable level of expertise in their respective field.

However, such an ideal setup could not be provided for this IMS case study. Therefore the quantitative comparison in this work had to be based on assumptions resulting from statistical analysis of past TTCN-3 test development projects. Table 6.1 summarizes the results obtained, after applying the pattern-oriented test development methodology on the case study. Taking into account that the project duration was set to 5 person-days and that a total result of 19 test cases were implemented at its end, productivity factor is 19/5 = 3.8 test cases/day. It should be pointed that, this result was obtained with team of designers with a rather low level of testing and modelling expertise. Therefore, it can be assumed that slightly higher results would be obtained with experienced test designers.

Project Duration(Days)	Produced Test cases	Productivity (Test cases/Day)
5	19	3.8

Table 6.1: Results of Applying Pattern-Oriented Test Engineering to IMS Case Study

To measure the productivity gain generated by this work's approach, the results obtained with pattern-oriented test modelling are compared with those generally obtained through "traditional" test development approaches. Figure 6.27

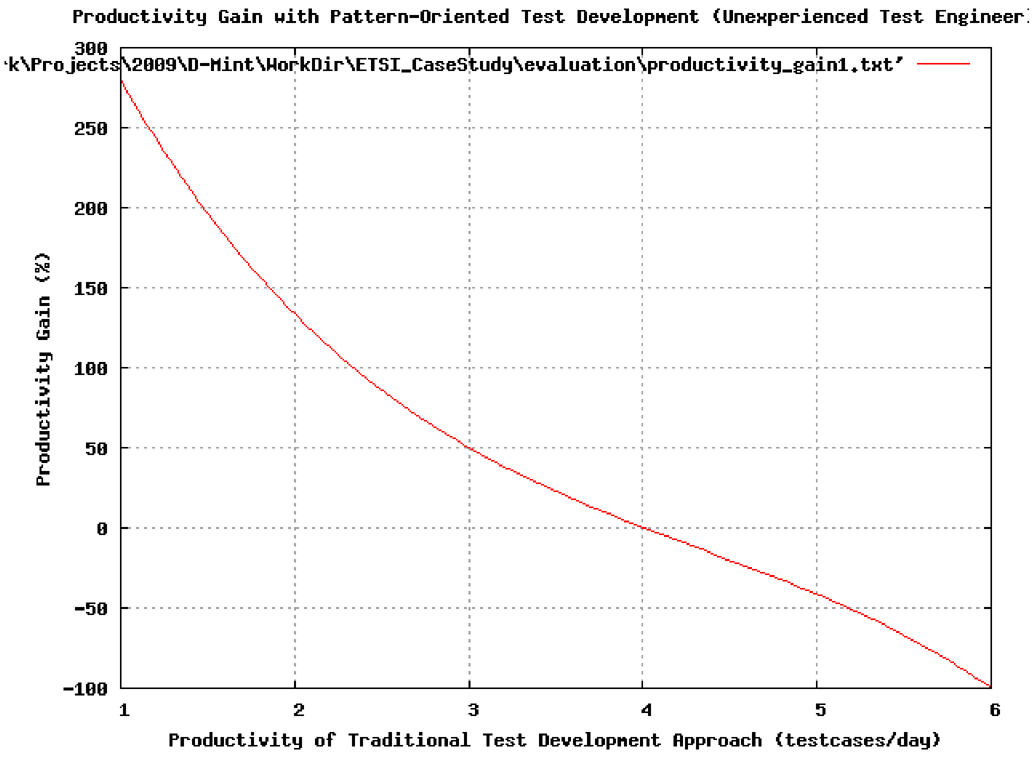


Figure 6.27: Productivity Gain From Pattern-Oriented Test Development, without taking into account the impact of Test Objectives and Test Procedures

depicts the evolution of productivity gain, depending on the productivity obtained without pattern-oriented test development. Generally, for TTCN-3 test development, realistic estimations of productivity range between 2 and 5 test cases/day. Therefore,the plot in Figure 6.27 indicates that, if the existing process allows a production rate of more than 4 test cases/day (including test objectives definition, test procedure design and documentation), then applying the methodology proposed in this thesis would rather cause a productivity loss. On the other hand, the productivity could be significantly improved (30 to 90%), when the production rate of the existing methodology is between 2 and 4 test cases/day. Moreover, if it is assumed that, the specification of a test plan (test

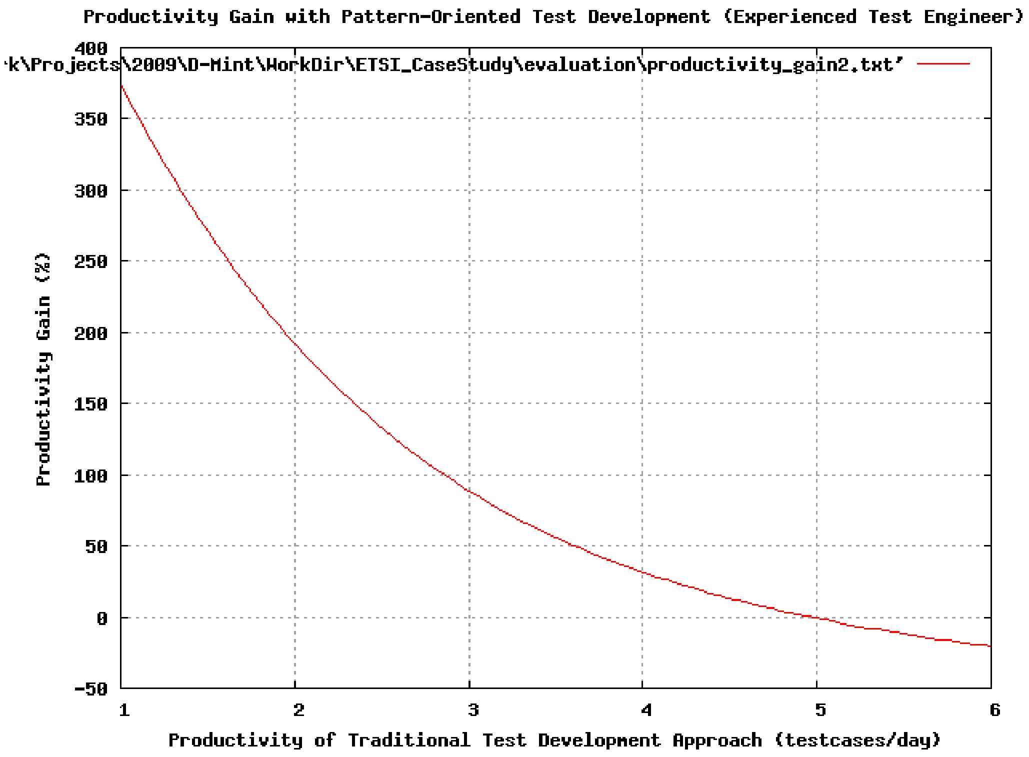


Figure 6.28: Productivity Gain From Pattern-Oriented Test Development Based on Pure Test System Design

objectives) and of test procedures consumes 20% of the effort in pattern-oriented test development and are generally not taken into account, when estimating the productivity of the test development process, then the productivity gain is even higher, as depicted on Figure 6.28.

Qualitative Analysis Using model-driven approach to test development offers a wide range of qualitative benefits, compared to traditional development approach. Test models offer a higher level of readability, maintainability, documentation and flexibility that plain test scripts and non-formal notations. Furthermore, existing MDE frameworks (e.g. Eclipse EMF, TOPCASED) provide a wide range of functionalities for creating, managing, validating and transforming models that can be used to provide powerful tool chains to support the process. However, a source of general concern is the quality of the test scripts generated automatically from the process. For this IMS case study, the TRex [164] tool was used measure the quality of the generated TTCN-3 test scripts. The authors of TRex define a metric called *Template coupling* (ranging between 1 and 3) to measure the maintainability of TTCN-3 scripts. The automatically generated IMS test scripts scored 1.015 on that metrics, indicating the high level of maintainability of those scripts (1.0 is best).

6.3.3 The OMA SUPL Case Study

Introduction

The Open Mobile Alliance[1] (OMA) is an international body which defines open standards for the application layer in fixed and mobile communications networks. Location Based Services (LBS) are one of the categories of services addressed by OMA through various protocols such as MLP (Mobile Location Protocol), RLP (Roaming Location Protocol) and SUPL (Secure User Plane Location Protocol). With the release of its version 2.0, new functionalities were added to the initial SUPL v1.0 specification. Therefore, the existing conformance tests developed for version 1.0 with TTCN-3 needed to be upgraded to cover version 2.0.

This case study describes how the methodology proposed in this thesis was used to perform round-trip engineering, firstly to visualize and analyze the existing test scripts, then to reuse elements thereof to design new test cases at a higher level of abstraction, before finally transforming those back into executable TTCN-3 test cases.

The Test Model

Although reuse is known to be a potentially highly rewarding task, putting it in practice is by no means trivial. In fact, a pre-condition for reusing legacy source code or any sort, is to figure out how it is structured and how its composing elements are related to each other. Given that a (good) picture is said to be worth thousand words, visualizing the source code can be a good starting point for analysing it in terms of potential reuse.

Therefore the first step in this case study consisted in using the TTCN-3 frontend developed with the prototype tool to transform the TTCN-3 source code for SUPL v1.0 into UTML models. The TTCN-3 frontend for UTML transforms elements defined in TTCN-3 into their UTML equivalents according to the mapping described in Section B.1 of Appendix B. However, it worth mentioning that to limit the size of the resulting test model, the transformation does not cover the whole depth of the TTCN-3 abstract syntax tree(AST). For example, for functions defined in TTCN-3, only their signature is transformed to allow their reuse, while the behaviour they contain is left out. In a similar manner, for TTCN-3 templates, only their key caracteristics are extracted (e.g. name, direction, parameters etc.).

[1] http://www.openmobilealliance.org

```
type record ULP_PDU {
    _0ULP_PDU length_ ,
    Version version ,
    SessionID sessionID ,
    UlpMessage message_
  }

  type integer _0ULP_PDU (0 .. 65535);

  type union UlpMessage {
    SUPLINIT msSUPLINIT,
    SUPLSTART msSUPLSTART,
    SUPLRESPONSE msSUPLRESPONSE,
    SUPLPOSINIT msSUPLPOSINIT,
    SUPLPOS msSUPLPOS,
    SUPLEND msSUPLEND,
    SUPLAUTHREQ msSUPLAUTHREQ,
    SUPLAUTHRESP msSUPLAUTHRESP,
    Ver2_SUPLTRIGGEREDSTART msSUPLTRIGGEREDSTART,
    Ver2_SUPLTRIGGEREDRESPONSE msSUPLTRIGGEREDRESPONSE,
    Ver2_SUPLTRIGGEREDSTOP msSUPLTRIGGEREDSTOP,
    Ver2_SUPLNOTIFY msSUPLNOTIFY,
    Ver2_SUPLNOTIFYRESPONSE msSUPLNOTIFYRESPONSE,
    Ver2_SUPLSETINIT msSUPLSETINIT,
    Ver2_SUPLREPORT msSUPLREPORT
  }
```

Listing 6.5: Example TTCN-3 Source Code for OMA SUPL Test Data Type

Listing 6.5 shows a code snippet from the legacy TTCN-3 test specification for the OMA SUPL protocol, while Figure 6.29 depicts the UTML test data diagram resulting from the transformation of that test specification into UTML.

Figure 6.29 provides a good illustration of the power of visualisation for understanding and reusing existing TTCN-3 test automation scripts. An example of such reuse is displayed in Figure 6.30, which features an existing SUPL v.1.0 test data instance (*s_ulpPdu*), initially defined in as a TTCN-3 template, beeing extended to design a new test data instance for SUPL v2.0 (*m_ulpPduVersion*). Finally, Figure 6.31 displays the test sequence diagram for one of the new OMA SUPL v2.0 test cases designed in the case study.

Test Execution

Using the TTCN-3 backend for UTML, the test models were transformed automatically into TTCN-3 test skeletons which were then completed manually into fully executable test cases. The manual effort for completing the test cases could be estimated to approximately 10% of the total effort. A sample TTCN-3 code generated from the testcase depicted in Figure 6.31 can be found in Appendix C.

Evaluation

Some tooling issues needed to be addressed during this case study. Those issues were mostly related to the ability of the EMF tools to handle large size models

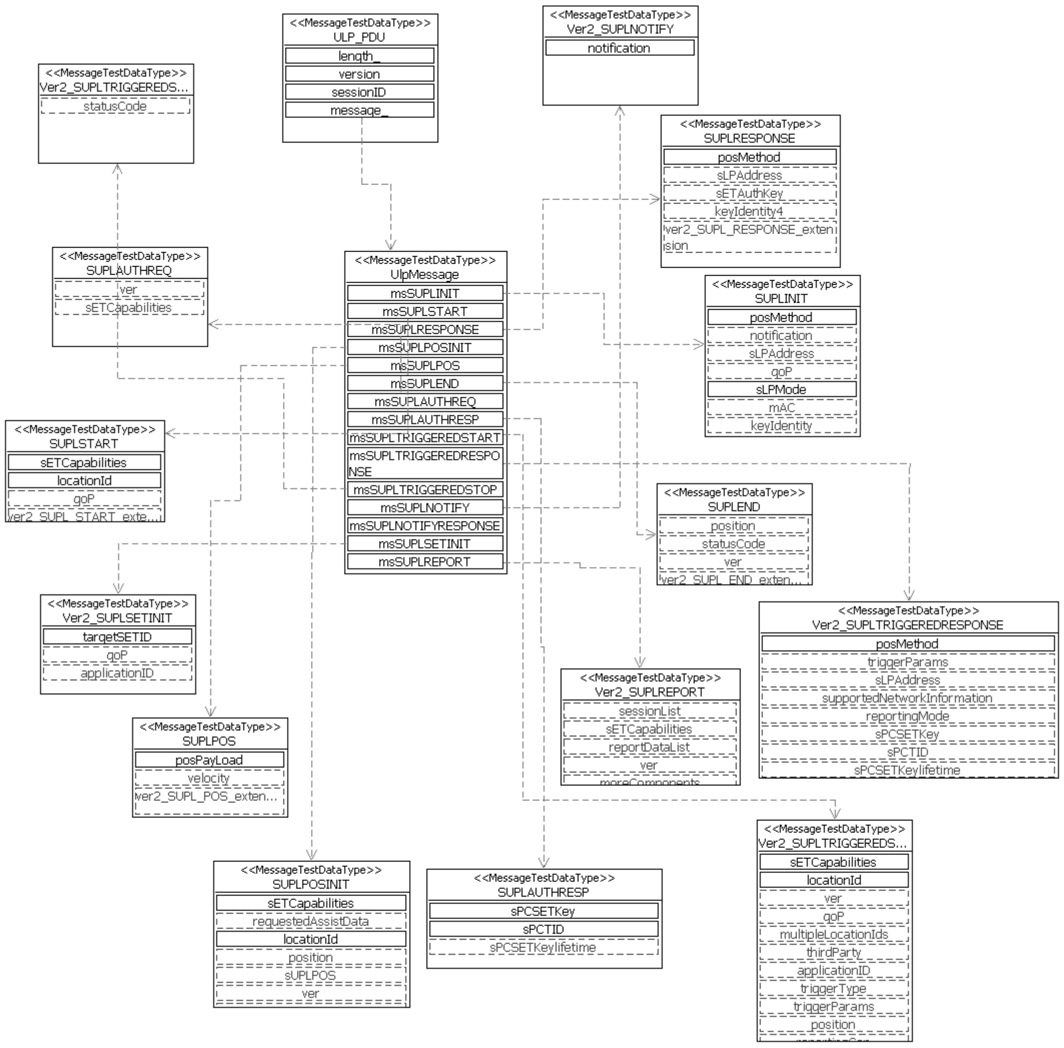


Figure 6.29: Examples of UTML Test Data Diagram resulting from automated Transformation from TTCN-3 OMA SUPL v1.0

resulting from reverse engineering of the existing TTCN-3 code. The consequence of this was that the delay for loading the test model was too long and thus was affecting productivity.

This case study also underlined the need for supporting functionalities that are essential for any development or modelling activity, such as tools for searching for certain elements in the artefacts or for tracking modifications between different versions. The latter is even more important if several designers/developers work on the same model. Eventually, those functionalities were implemented in the prototype tool using mechanisms provided by the EMF tool chain. However, it was later discovered that those functionalities would fail because of the large

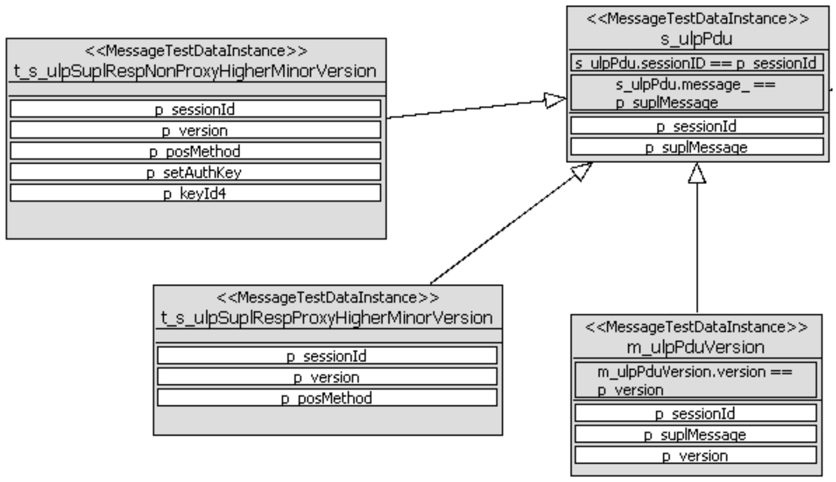

<<MessageTestDataInstance>>
t_s_ulpSuplRespNonProxyHigherMinorVersion
p_sessionId
p_version
p_posMethod
p_setAuthKey
p_keyId4
<<MessageTestDataInstance>>
s_ulpPdu
s_ulpPdu.sessionID == p_sessionId
s_ulpPdu.message_ == p_suplMessage
p_sessionId
p_suplMessage
<<MessageTestDataInstance>>
t_s_ulpSuplRespProxyHigherMinorVersion
p_sessionId
p_version
p_posMethod
<<MessageTestDataInstance>>
m_ulpPduVersion
m_ulpPduVersion.version == p_version
p_sessionId
p_suplMessage
p_version

Figure 6.30: Reuse of Legacy Test Data in UTML Test Data Model for OMA SUPL Testing

size of the model. Therefore, after all test cases had been designed completely in the test model, an algorithm was developed and applied to reduce the size of the test model, by deleting all elements that were not actually referred to by the test cases. This lead to a reduction of the test model's size by 1/3, namely from approximately 40kLOC to 12kLOC.

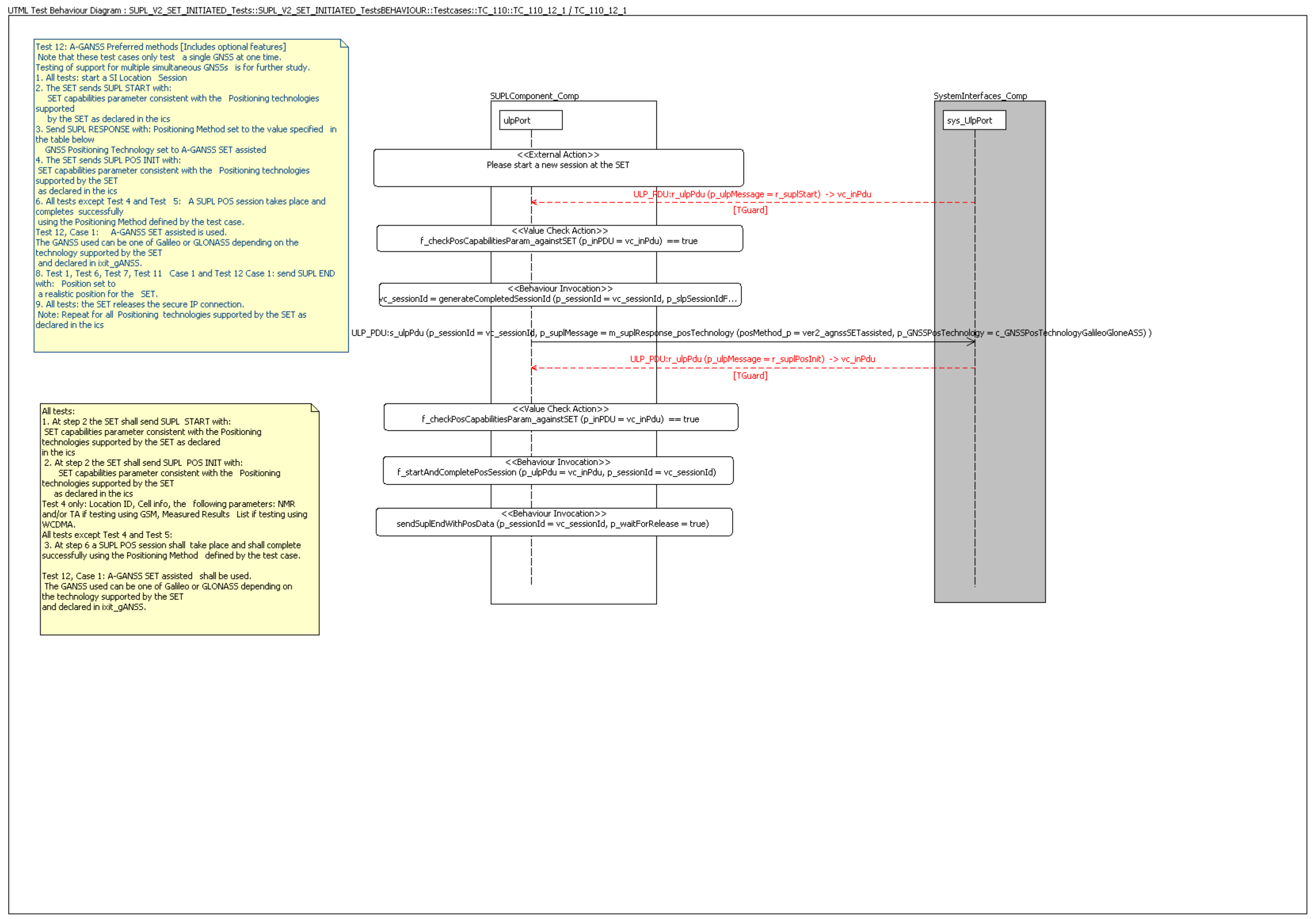
Eventually at the end of this case study, a total of 29 test cases were developed using the approach proposed in this thesis, while at the same time 35 test cases were developed in parallel using a more traditional TTCN-3 test development approach. This means that, the efforts were required to fix the previously mentioned tooling issues, neearly the same level of productivity was achieved with model-driven test engineering as with traditional test engineering. This indicates that, if the prototype tool had reached a higher level of maturity at the beginning of the case study, the MDTE approach would have lead to even better results.

Furthermore, the fact that the documentation for the test cases could be generated automatically from the UTML model helped not only for internal communication within the project, but also for providing the final project report including a graphical description of the behaviour in each test case.

6.3.4 The Parlay-X Case Study

Introduction

This section describes a case study featuring the usage of pattern-oriented test engineering to design and implement a test automation solution for web services


UTML Test Behaviour Diagram : SUPL_V2_SET_INITIATED_Tests::SUPL_V2_SET_INITIATED_TestsBEHAVIOUR::Testcases::TC_110::TC_110_12_1 / TC_110_12_1
SUPLComponent_Comp
ulpPort
SystemInterfaces_Comp
sys_UlpPort
<<External Action>>
Please start a new session at the SET
ULP_PDU:r_ulpPdu (p_ulpMessage = r_suplStart) -> vc_inPdu
[TGuard]
<<Value Check Action>>
f_checkPosCapabilitiesParam_againstSET (p_inPDU = vc_inPdu) == true
<<Behaviour Invocation>>
vc_sessionId = generateCompletedSessionId (p_sessionId = vc_sessionId, p_slpSessionIdF...
ULP_PDU:s_ulpPdu (p_sessionId = vc_sessionId, p_suplMessage = m_suplResponse_posTechnology (posMethod_p = ver2_agnssSETassisted, p_GNSSPosTechnology = c_GNSSPosTechnologyGalileoGloneASS))
ULP_PDU:r_ulpPdu (p_ulpMessage = r_suplPosInit) -> vc_inPdu
[TGuard]
<<Value Check Action>>
f_checkPosCapabilitiesParam_againstSET (p_inPDU = vc_inPdu) == true
<<Behaviour Invocation>>
f_startAndCompletePosSession (p_ulpPdu = vc_inPdu, p_sessionId = vc_sessionId)
<<Behaviour Invocation>>
sendSuplEndWithPosData (p_sessionId = vc_sessionId, p_waitForRelease = true)

Test 12: A-GANSS Preferred methods [Includes optional features]
Note that these test cases only test a single GNSS at one time.
Testing of support for multiple simultaneous GNSSs is for further study.
1. All tests: start a SI Location Session
2. The SET sends SUPL START with:
SET capabilities parameter consistent with the Positioning technologies supported
by the SET as declared in the ics
3. Send SUPL RESPONSE with: Positioning Method set to the value specified in the table below
GNSS Positioning Technology set to A-GANSS SET assisted
4. The SET sends SUPL POS INIT with:
SET capabilities parameter consistent with the Positioning technologies supported by the SET
as declared in the ics
6. All tests except Test 4 and Test 5: A SUPL POS session takes place and completes successfully
using the Positioning Method defined by the test case.
Test 12, Case 1: A-GANSS SET assisted is used.
The GANSS used can be one of Galileo or GLONASS depending on the technology supported by the SET
and declared in ixit_gANSS.
8. Test 1, Test 6, Test 7, Test 11 Case 1 and Test 12 Case 1: send SUPL END with: Position set to
a realistic position for the SET.
9. All tests: the SET releases the secure IP connection.
Note: Repeat for all Positioning technologies supported by the SET as declared in the ics

All tests:
1. At step 2 the SET shall send SUPL START with:
SET capabilities parameter consistent with the Positioning technologies supported by the SET as declared
in the ics
2. At step 2 the SET shall send SUPL POS INIT with:
SET capabilities parameter consistent with the Positioning technologies supported by the SET
as declared in the ics
Test 4 only: Location ID, Cell info, the following parameters: NMR and/or TA if testing using GSM, Measured Results List if testing using WCDMA.
All tests except Test 4 and Test 5:
3. At step 6 a SUPL POS session shall take place and shall complete successfully using the Positioning Method defined by the test case.

Test 12, Case 1: A-GANSS SET assisted shall be used.
The GANSS used can be one of Galileo or GLONASS depending on the technology supported by the SET
and declared in ixit_gANSS.

Figure 6.31: UTML Test Sequence Diagram for an OMA SUPL Test Case

specified through the Parlay-XAPIs. The purpose of the Parlay APIs[2] is to facilitate access to services provided by telecommunication networks, so that new IT and telephony applications using those services can be developed more rapidly, even by IT-developers who are not necessarily experts in telecommunication systems. With the growing importance of web services, the Parlay APIs have been specified since 2004 as a collection of web services gathered under the Parlay-X label.

This idea of applying the method proposed in this thesis for test automation in the context of Parlay-X was found attractive, because it provided the opportunity for assessing the application of the approach to a new domain, namely that of web services, while at the same time for checking whether the approach is suitable for systems using synchronous (Request-Response) communication scheme.

The Parlay-X web service APIs are specified with the Web Service Definition Language (WSDL) and cover several categories of services such as *Call Control*, *User Interaction*, *Messaging*, *Mobility* etc. For the case study, the *Send SMS* interface that belongs to the *Messaging* category and provides a gateway to Short Messaging System usually available in mobile GSM networks.

The Test Model

The test model for this case study was designed by transforming the system model (provided as a WSDL file) into UTML and to extend the automatically generated test model manually to obtain a complete test model. This manual step was required because the WSDL system model does not contain a description of the system's behaviour, but only its structure and the associated data types.

The Test Data Model Listing 6.6 displays an extract from WSDL specification for the Parlay-X *SendSMS* interface, containing definitions of data types used by that web service.

```
type="parlayx_sms_send_local_xsd:sendSms"/> <xsd:complexType name="sendSms">
   <xsd:sequence>
      <xsd:element name="addresses" type="xsd:anyURI" minOccurs="1"
      maxOccurs="unbounded"/> <xsd:element name="senderName" type="xsd:string"
      minOccurs="0" maxOccurs="1"/> <xsd:element name="charging"
      type="parlayx_common_xsd:ChargingInformation"
       minOccurs="0" maxOccurs="1"/>
      <xsd:element name="message" type="xsd:string"/> <xsd:element
      name="receiptRequest" type="parlayx_common_xsd:SimpleReference"
      minOccurs="0" maxOccurs="1"/>
   </xsd:sequence>
</xsd:complexType>

<xsd:element name="sendSmsResponse"
type="parlayx_sms_send_local_xsd:sendSmsResponse"/>
<xsd:complexType name="sendSmsResponse">
```

[2]The Parlay APIs are defined by the Parlay Group (http:\www.parlay.org)

```
   <xsd:sequence>
      <xsd:element name="result" type="xsd:string"/>
   </xsd:sequence>
</xsd:complexType>
<wsdl:message name="SendSms_sendSmsRequest">
   <wsdl:part name="parameters"
   element="parlayx_sms_send_local_xsd:sendSms"/>
</wsdl:message>

<wsdl:message name="SendSms_sendSmsResponse">
   <wsdl:part name="result"
   element="parlayx_sms_send_local_xsd:sendSmsResponse"/>
</wsdl:message>
```

Listing 6.6: Extract from the Parlay-X *SendSMS* WSDL Service Specification of Data Types

Figure 6.32 displays an extract of the UTML test data diagram for the test data model resulting from transforming the WSDL elements displayed in Listing 6.6 to UTML, using the WSDL frontend provided by the prototype tool.

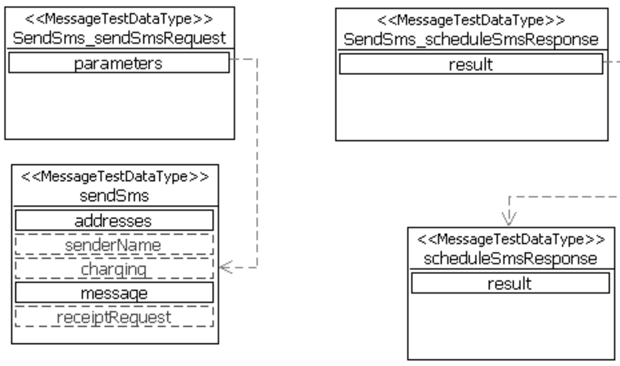


Figure 6.32: Extract of UTML Test Data Diagram displaying Elements imported from Parlay-X System Model (WSDL)

The Test Architecture Model As displayed in Listing 6.7, besides the system's data object model, the SUT's WSDL file also contains a specification of ports through which the *SendSMS* service can be accessed and the operations supported by those ports. Those information can be transformed automatically into UTML, so that they can be reused to design the test architecture model.

```
<wsdl:portType name="SendSms">
   <wsdl:operation name="sendSms">
      <wsdl:input message="parlayx_sms_send:SendSms_sendSmsRequest"/>
      <wsdl:output message="parlayx_sms_send:SendSms_sendSmsResponse"/>
      <wsdl:fault name="ServiceException"
      message="parlayx_common_faults:ServiceException"/>
      <wsdl:fault name="PolicyException"
      message="parlayx_common_faults:PolicyException"/>
```

```
    </wsdl:operation>
...
</wsdl:portType>
```

Listing 6.7: Extract from the Parlay-X *SendSMS* WSDL Service Specification Operation Types

Figure 6.33 displays the test architecture resulting from that transformation. As displayed in that figure, a simple P2P test architecture pattern has been applied to derive the test architecture, which consists of one test components connected to the SUT via the *sendSMSPort*.

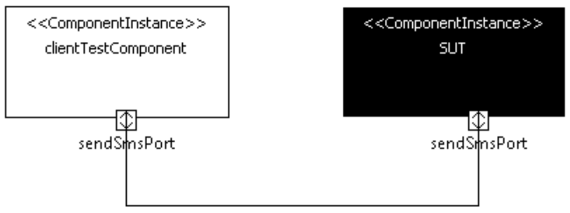


Figure 6.33: Automatically Generated Test Architecture for the Parlay-X *SendSMS* Web Service

The Test Behaviour Model The behaviour of the client and the application server involved in a Parlay-X scenario is specified by the Parlay-X standard in the form of natural language, sometimes illustrated with UML sequence diagrams showing the expected interactions between those parties. An example of one such UML sequence diagram is displayed in Figure 6.34, which depicts how a client may invoke the Parlay-X *SendSMS* web service to send a short message, then after a short while, query the web service to get the delivery status of the sent short message. Based on the Parlay-X specification and on UML sequence diagrams such as the one displayed in Figure 6.34, test sequence diagrams were designed for the *SendSMS* service, taking into account and referring to the test data and the test architecture models described in Paragraph 6.3.4 and Paragraph 6.3.4 respectively.

Figure 6.35 displays a UTML test sequence diagram for a test case targetting the Parlay-X *SendSMS* web service. The objective of the test case is to check that the web service meets the requirements on the delivery delays for SMSs sent through it.

Test Execution

To illustrate the transformation of the UTML test model designed in this case study towards executable test cases, the TTCN-3 backend was used again, leading to the source code displayed in Listing 6.8.

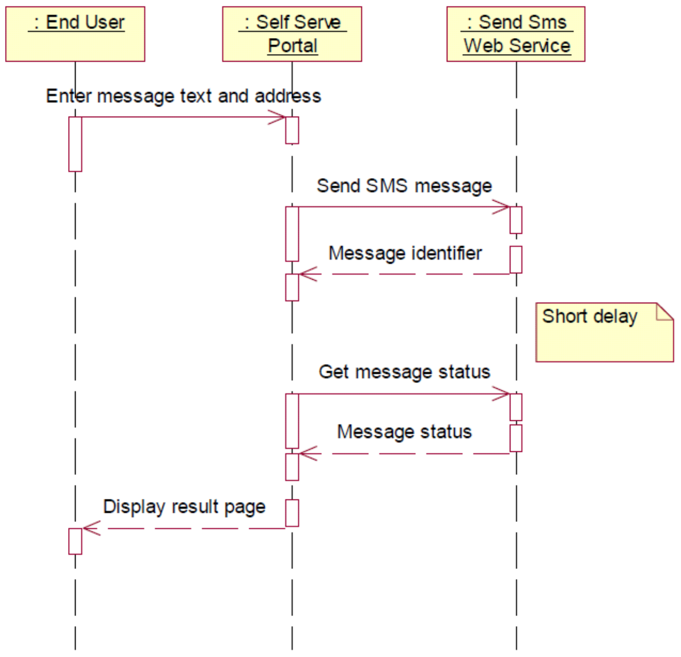


Figure 6.34: UML Sequence Diagram for Parlay-X *SendSMS* Web Service

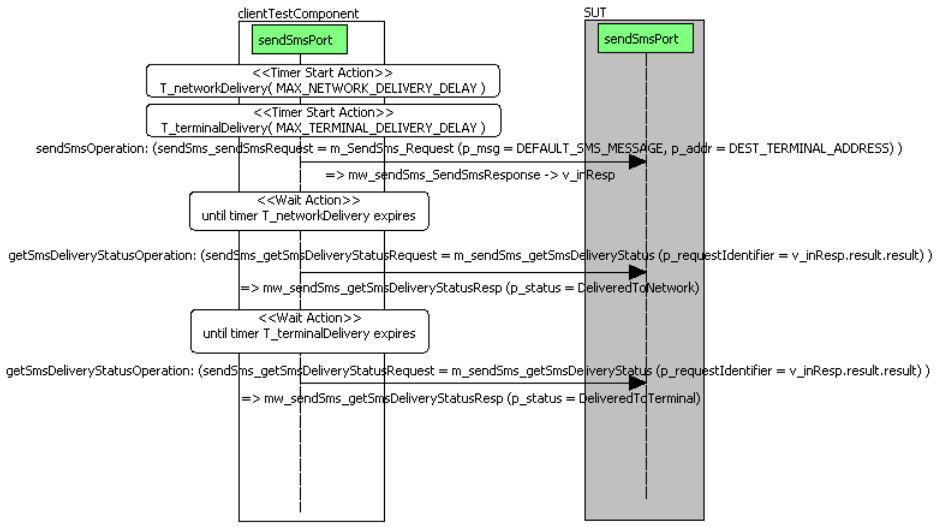

clientTestComponent
sendSmsPort
SUT
sendSmsPort
<<Timer Start Action>>
T_networkDelivery(MAX_NETWORK_DELIVERY_DELAY)
<<Timer Start Action>>
T_terminalDelivery(MAX_TERMINAL_DELIVERY_DELAY)
sendSmsOperation: (sendSms_sendSmsRequest = m_SendSms_Request (p_msg = DEFAULT_SMS_MESSAGE, p_addr = DEST_TERMINAL_ADDRESS))
=> mw_sendSms_SendSmsResponse -> v_inResp
<<Wait Action>>
until timer T_networkDelivery expires
getSmsDeliveryStatusOperation: (sendSms_getSmsDeliveryStatusRequest = m_sendSms_getSmsDeliveryStatus (p_requestIdentifier = v_inResp.result.result))
=> mw_sendSms_getSmsDeliveryStatusResp (p_status = DeliveredToNetwork)
<<Wait Action>>
until timer T_terminalDelivery expires
getSmsDeliveryStatusOperation: (sendSms_getSmsDeliveryStatusRequest = m_sendSms_getSmsDeliveryStatus (p_requestIdentifier = v_inResp.result.result))
=> mw_sendSms_getSmsDeliveryStatusResp (p_status = DeliveredToTerminal)

Figure 6.35: UTML Test Sequence Diagram of Test Case for Parlay-X *SendSMS* Web Service

```
testcase TC_SmsDeliveryDelay ()
runs on SendSmsComponentType
system SendSmsComponentType {
    //Local variables and timers
    var string v_requestId;
    var SendSms_sendSmsResponse v_inResp;
```

```
timer T_networkDelivery := MAX_NETWORK_DELIVERY_DELAY;
timer T_terminalDelivery := MAX_TERMINAL_DELIVERY_DELAY;
//Test execution
//Setup configuration: MyP2PTestArchitecture
map ( self:sendSmsPort, system:sendSmsPort );
//Preamble
//Test body
T_networkDelivery.start;
T_terminalDelivery.start;
sendSmsPort.call ( sendSmsOperation:
{m_SendSms_Request ( DEFAULT_SMS_MESSAGE, DEST_TERMINAL_ADDRESS )} ) {
    [] sendSmsPort.getreply
    ( sendSmsOperation:? value mw_sendSms_SendSmsResponse ) -> value v_inResp {
    log ( "*** Got reply mw_sendSms_SendSmsResponse for sendSmsOperation
     call ***" );
    }
    [] sendSmsPort.getreply {
        setverdict ( fail );
    }
    [] sendSmsPort.catch {
        setverdict ( fail );
    }
    }
    log ( "*** TC_SMSDELIVERYDELAY: start waiting
    until T_networkDelivery expires. ***" );
    wait ( MAX_NETWORK_DELIVERY_DELAY );
    sendSmsPort.call ( getSmsDeliveryStatusOperation:
    {m_sendSms_getSmsDeliveryStatus ( v_inResp.result.result )} ) {
    [] sendSmsPort.getreply ( getSmsDeliveryStatusOperation:?
    value mw_sendSms_getSmsDeliveryStatusResp ) {
        log ( "*** Got reply mw_sendSms_getSmsDeliveryStatusResp
        for getSmsDeliveryStatusOperation call ***" );
    }
    [] sendSmsPort.getreply {
        setverdict ( fail );
    }
    [] sendSmsPort.catch {
        setverdict ( fail );
    }
    }
    log ( "*** TC_SMSDELIVERYDELAY: start waiting until
    T_terminalDelivery expires. ***" );
    wait ( MAX_TERMINAL_DELIVERY_DELAY );
    sendSmsPort.call ( getSmsDeliveryStatusOperation:
    {m_sendSms_getSmsDeliveryStatus ( v_inResp.status.status )} ) {
    [] sendSmsPort.getreply ( getSmsDeliveryStatusOperation:?
    value mw_sendSms_getSmsDeliveryStatusResp ) {
        setverdict (pass, "*** Got reply mw_sendSms_getSmsDeliveryStatusResp
        for getSmsDeliveryStatusOperation call ***" ); }
    [] sendSmsPort.getreply {
        setverdict ( fail );
    }
    [] sendSmsPort.catch {
        setverdict ( fail );
    }
}
//Postamble
unmap ( self:sendSmsPort, system:sendSmsPort );
```

```
} //end TC_SmsDeliveryDelay
```

Listing 6.8: TTCN-3 Source Code generated from the UTML test behaviour model for the Parlay-X *SendSMS* Testcase displayed in Figure 6.35

Although the generated TTCN-3 source code was not effectively executed against an application server providing the Parlay-X services, the fact that it was successfully validated with a TTCN-3 compiler is a clear indication, that its quality can at least be considered as acceptable.

Evaluation

This case study has demonstrated how the approach proposed in this thesis can be used to develop test automation for systems and services that use a synchronous communication scheme following a request-response scenario. The case study also provided the opportunity to evaluate the WSDL-frontend developed with the prototype tool. The WSDL frontend transforms system models specified with WSDL automatically into UTML test model , using the mapping rules described in Section B.2 of Appendix B. The combined usage of that frontend together with the TTCN-3 backend made reduced the test development effort for such systems drastically, while at same time ensuring a higher quality of the resulting test cases.

6.3.5 The Digital Watch Case Study

Introduction

The digital watch case study is an interesting application of the methodology proposed in this thesis for combining MBT and MDT techniques to facilitate test automation. Also, this study demonstrate how the proposed approach can be used in the embedded systems domain, besides the services and communication domains covered by the other case studies.

The Test Model

The system model used for this case study is an example SysML model for a digital watch, provided by the TOPCASED modelling tool to demonstrate SysML support[3]. The test model for the digital watch results from a combination of automated generation from the SUT's model (SysML). Firstly, the requirements on the SUT are transformed into UTML test objectives, using a model-to-model transformation implemented via the MDTester SysML frontend. The transformation of SysML requirements into UTML test objectives can either be performed on individual requirements or on packages containing several requirements or

[3]The complete model is available for download at http://www.topcased.org

subpackages. Figure 6.36 depicts an example requirements package that was added to the original example model for demonstration purpose. As depicted in that figure, the designed requirements are not more than illustrating examples without any real semantical relationship to the digital watch model itself. To ensure traceability between system and test model, there's a need to create

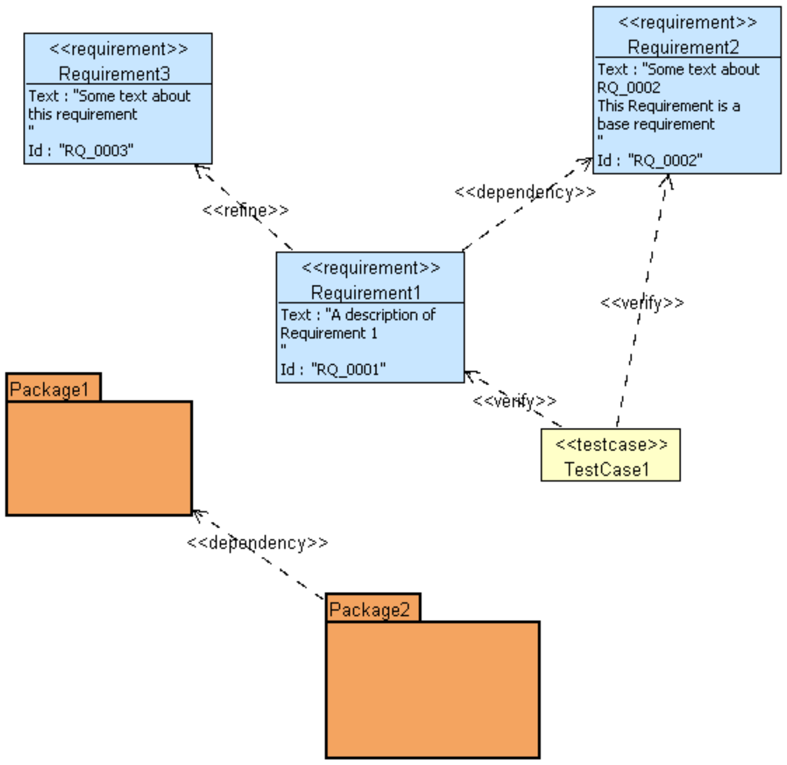


Figure 6.36: SysML Requirements Diagram for the digital watch

references between the test model and the requirements specified in the system model. Therefore, those requirements had to be transformed from SysML to UTML, using the SysML frontend developed with the prototype tool in this thesis. The SysML to UTML transformation was performed based on the mapping rules defined in Table B.1 of Appendix B. Figure 6.37 displays the result of that transformation process. As depicted in that figure, the transformation not only creates a structure in the test objectives model that is equivalent to the original structure of the SysML, but also keeps track of the dependency relationships existing between elements of the system model.

Then, the SUT's logical architecture (including associated data model) is transformed in a similar manner into a test architecture using one of the architectural test design patterns described in Section A.3 of Appendix A. Figure 6.38 displays the logical architecture of the digital watch, represented as a SysML internal block diagram. As depicted in that figure, the digital watch consists of a

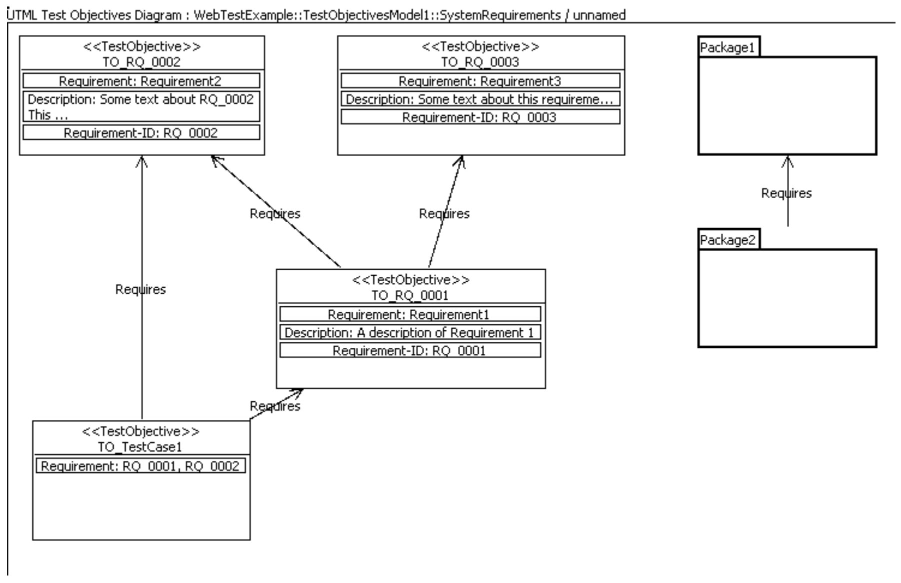

UTML Test Objectives Diagram : WebTestExample::TestObjectivesModel1::SystemRequirements / unnamed
<<TestObjective>> TO_RQ_0002
Requirement: Requirement2
Description: Some text about RQ_0002 This ...
Requirement-ID: RQ_0002
<<TestObjective>> TO_RQ_0003
Requirement: Requirement3
Description: Some text about this requireme...
Requirement-ID: RQ_0003
Package1
Package2
Requires
<<TestObjective>> TO_RQ_0001
Requirement: Requirement1
Description: A description of Requirement 1
Requirement-ID: RQ_0001
<<TestObjective>> TO_TestCase1
Requirement: RQ_0001, RQ_0002

Figure 6.37: UTML Test Objectives Diagram resulting from transformation of SysML Requirements

processor block and of a series of blocks building together the watch's display. For this case study, the processor (*watchProcessor*) block was taken as the SUT for which test cases were to be developed. The transformation of the *watchProcessor* block from SysML to UTML consisted, not just of generating the equivalent SUT component in UTML, but also a test architecture on which test cases could be designed in a following step. The generation of a test architecture is done based on the test architecture pattern selected by the user through the wizard provided by the prototype tool. Figure 6.39 displays the result of the transformation operation, in case the *One-on-One* test architecture pattern described in Section A.3.2 was selected. As expected, the resulting test architecture features one test component providing the same ports as the SUT, but with inverted directions (mirror ports). Alternatively, the *Sandwich* test architecture pattern described in Section A.3.6 could have been chosen instead, leading to the test architecture displayed in Figure 6.40. As expected, the resulting test architecture splits test behaviour between two parallel test components, whereby one of those components (*Stimulating_TC*) will be used for send stimuli to the SUT, while the other one (*Observing_TC*) will be used to assess that the SUT reacts as expected to those stimuli.

Test Execution

For this case study, no test behaviour was designed, because the SysML system model only contains architectural elements and does not address behaviour.

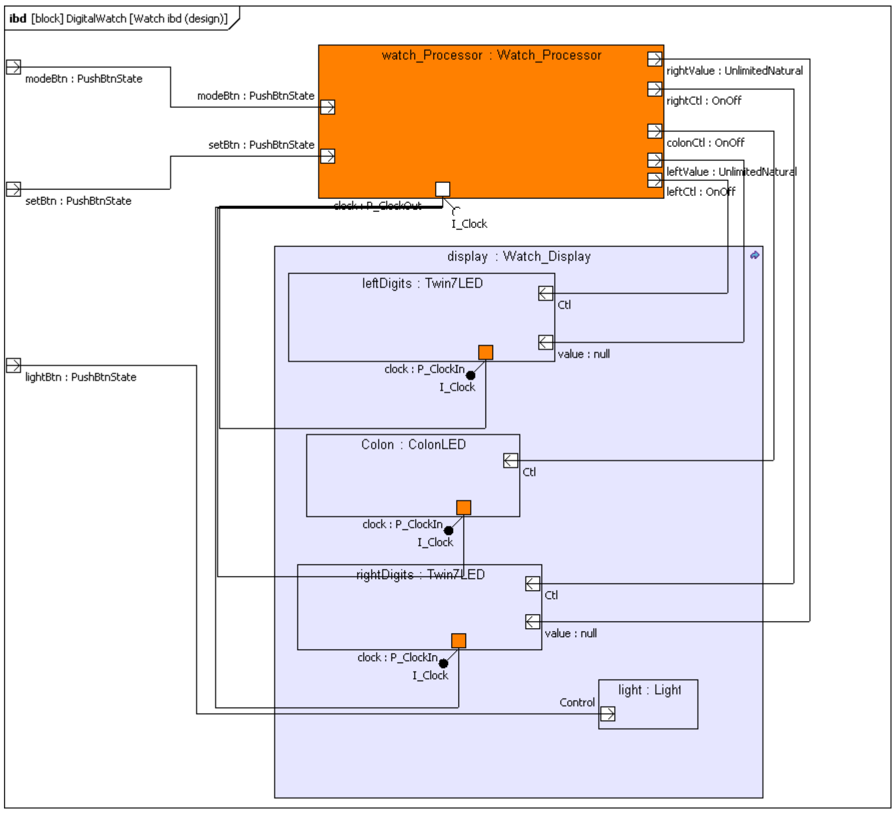


Figure 6.38: SysML Block Diagram Displaying the Logical Architecture of the digital watch

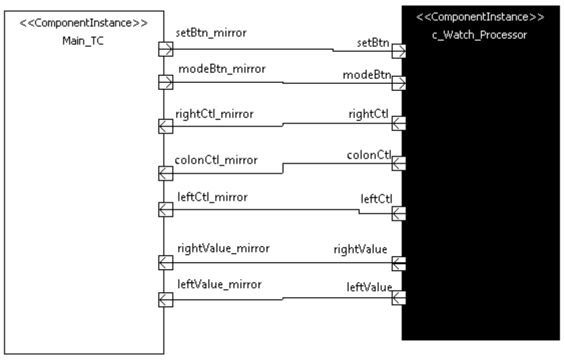


Figure 6.39: Test Architecture derived from the SysML Block Diagram for the *watchProcessor* Block (*One-on-One* Test Architecture Pattern)

Furthermore, adding such behaviour elements to the system model from scratch required significantly more resources and deeper domain-specific knowledge, both of which were unavailable at that moment.

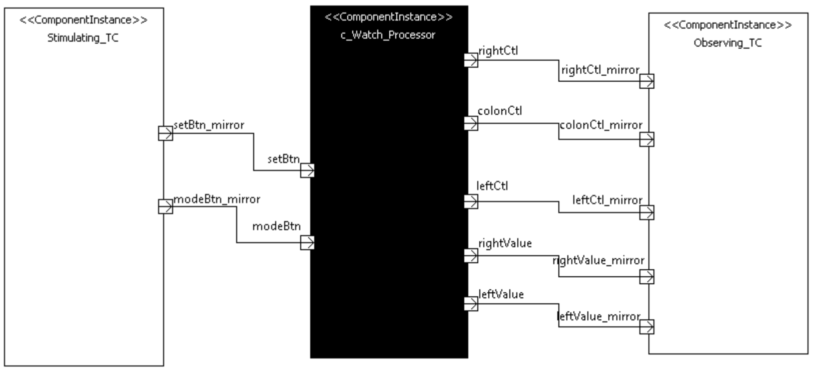


Figure 6.40: Test Architecture derived from the SysML Block Diagram for the *watchProcessor* System Component (*Sandwich* Test Architecture Pattern)

Evaluation

This case study has demonstrated how model-to-model transformation can be used to implement traceability between system and test model. Additionally, the usage of the SysML notation in the case study demonstrates that the approach proposed in this thesis can also be applied to domains in which that notation is used for system design (e.g. Embedded systems, Automotive, Avionics).

6.4 Summary

This chapter has proposed an application's architecture for pattern-oriented model-driven testing. As a proof-of-concept, a prototype application based on the proposed architecture has been implemented. Then, that prototype tool chain was used to apply the pattern-oriented test development approach on a selection of case studies to evaluate it and identify future potential improvements. A total of five case studies have been presented, describing how the approach proposed in this thesis was applied successfully to improve the test automation process for various kinds of SUTs from different application domains. A comparison of the output obtained with pattern-oriented MDT with current state of the art indicates improvements, both in terms of the productivity and of the quality of the resulting test suites. The evaluation done in this thesis has mainly been of technical nature based on collected case study data and statistical figures. However, further analysis will be required to evaluate to which extends the approach meets its potential end user requirements, e.g. with regard to usability. Furthermore, as already indicated in a publication of first results of this work [53], it should be interesting to evaluate the impact of the method for test development in other domains beyond the communication domain that has been the focus of the described case studies. However, the results obtained in the case studies presented in this chapter clearly demonstrate that the proposed approach can truely be considered as a promising way ahead for further research. Possible domains to be considered include automotive, railway and transformation systems, as well as service-based systems.

Chapter 7

Conclusions And Outlook

7.1 Summary and Conclusion

Combining models and software testing has always been a tempting idea. With the increasing popularity of models and model-driven software engineering, that idea has been gaining even more momentum expressed in various forms of model-based or model-driven testing.

This thesis has analyzed the model driven testing problem from the test automation perspective and after identifying the strengths and weaknesses of existing approaches, has proposed a new methodology for a more efficient use of design pattern in test automation. Firstly a collection of design patterns in test automation has been presented, resulting from experience gathered by test engineers in past successful test automation projects. Then the concepts identified by those patterns provided the base for the UTML notation, a DSML dedicated to test design following a MDT process with the aim of facilitating test design through usage of patterns. The definition of the UTML notation presented in Chapter 6 of this thesis has addressed not just both the abstract and the concrete syntax of that new language, but also its semantics and the constraints associated to each of its elements.

However, the fact that patterns come from practical experience implies that any methodology related to patterns must be checked against real case studies. Therefore a prototype test design tool was developed to validate the initial assumption made at the beginning of the work. Using that prototype tool, two case studies were conducted as part of this thesis and could demonstrate the positive impact of applying pattern-oriented model-driven testing. The result is a test development process that takes full advantage of the benefits that come with MDE and were introduced in Section 1.1 of this thesis. Those benefits include higher quality test artifacts (e.g. with regard to readability, understandability, reusability) obtained through automated model transformations and round-trip engineering.

Although the concepts presented in this thesis, both for pattern-oriented model-driven testing and the supporting UTML notation clearly aim at a broader application spectrum, the selected case studies are located in the domain of communication protocols and reactive software systems. This has undeniably influenced some of the design decisions with regard to the notation and can be explained by the fact that practical experiences gathered during the thesis were mainly located in that domain. Nevertheless, the definition of the UTML notation provided in this thesis offers a good base for further improvements to support other communication paradigms more common in other application domains (e.g. embedded systems, continuous signals etc.).

7.2 Outlook

The topics discussed in this thesis obviously cover a too broad spectrum to be covered with the same level of detail in a single thesis. Therefore, some issues were left for further research in future work, because they required a deeper analysis and resource beyond the scope a thesis like this one. Some of those issues include:

7.2.1 Usage of state machines for test behaviour modelling

Integration of state machines for test behaviour modelling to generate the sequences automatically from those: Using a state machine to describe behaviour, though less intuitive and more difficult than with sequence diagrams, undoubtedly has some advantages, such as a higher level of conciseness and the possibility of generating the sequence diagrams automatically from the state machine. In this thesis the possibility of using state machines to describe test behaviour has not been analysed further in detail, although it represents an interesting way of combining different approaches of model-based testing to achieve more efficiency.

7.2.2 Implementation of further templates for test patterns instantiation

The test patterns described in this thesis from an analysis of a large number of test suites mostly specified with the TTCN-3 notation. Therefore, although the identified test patterns can rightfully be considered to reflect accurately the practical experiences in that area, a further analysis with other black-box testing techniques and notations may reveal new patterns that have not been considered in this thesis.

7.2.3 Better modelling support for continuous signals and case studies thereof

Because of the lack of practical case studies dealing with continuous signals, out of which testing patterns specific for that domain could be gathered, this thesis focused mainly on an application to the testing of asynchronous and synchronous communication protocols. Further works and case studies are required to evaluate to what extend the approach, at its current stage, is applicable for that domain as well and potentially to provide the modifications necessary for it to better address requirements specific to that domain or beyond.

7.2.4 Automated Analysis of Test Script Code based on Patterns

Another interesting work area for the future is the instrumentation of patterns to facilitate the analysis of legacy test scripts through automated recognition of patterns they may contain. The method consists of walking through an abstract syntax tree (AST) of the test script code to be analysed, searching for code snippets that meet the definition of the patterns. Once identified, a visualisation of the patterns found can quickly provide a more abstract and clearer picture of the test script code, thus facilitating further operations such as reuse, refactoring etc.

During the work of this thesis, that approach has been applied on some existing TTCN-3 test suites to visualise the test data defined in those test suites with very satisfying results. Doing the same for test behaviour appears to be a tempting and promising idea that is certainly worth exploring further.

7.2.5 Empirical evaluation of the approach based on feedback from test experts

The evaluation of the approach proposed in this thesis, as described in Chapter 7 has been essentially of technical nature, based on statistics and using quantifiable metrics. However, some key factors for a successful adoption of a new technology as the one proposed in this thesis are more difficult to quantify, because they are determined by the way the new technology is perceived by the domain experts supposed to use it. In its book *Diffusion of Innovations* [136], Rogers lists the following five key characteristics for the adoption of innovations:

- Relative advantage: is your innovation better than the existing method?
- Compatibility: does your innovation integrate with the existing method?
- Complexity: is your innovation difficult to understand?
- Trialability: is it easy for people to experiment with your innovation?

- Observability: are the benefits of your innovation easily visible?

Although those characteristics have been carefully taken into account with the approach presented in this thesis, only a series of case studies including a usage of the associated tools by test experts and a survey of their feedback will provide an accurate picture, as to what extent those goals have effectively been met.

Appendix A

A Collection of Test Design Patterns

This appendix describes a collection of patterns identified in various test automation projects during this work. Each pattern is described based on the template provided in Section 4.3.2.

A.1 Generic Test Design Patterns

A.1.1 Pattern: Separation of Test Design Concerns

Context

This pattern is a generic organisational test design pattern and is applicable at any test scope for large size test projects. It is assumed that test development is process running in parallel to the development of the SUT or integrated to it, with both of them having the requirements as a common starting point.

Problem

How to organise the file structure of test artifacts. Test artifacts are resources used for storing the design and implementation of a test automation solution. They include high level design models, documentation artifacts through to source code of executable test scripts. The size and the complexity of those test artifacts can grow considerably, raising questions as to how to organise properly to keep a good overview and facilitate collaborative work.

Forces

- To avoid test design activity becoming a bottleneck to the development process, having different teams working in collaboration on the will speed up that process.
- Synchronisation and version control conflicts between the actors involved in test design may cause resources being wasted to address them.
- Large compilation units increase the risk of potential version control conflicts among parallel developers/designers.

Solution

Divide the various tasks over several test designers, by organising modules accordingly. Each task is addressed separately to allow parallel processing. Applying this pattern requires that the technologies involved (e.g. the notation used for designing the tests) provide such mechanisms. Modules may be organised based on the aspect they cover(e.g. Test data, test architecture) or based on the SUT feature they target.

Known Uses

Instantiations of this test pattern can be observed in numerous test automation solutions. The code snippet below from the IPv6 conformance test suite [145] displays an example in TTCN-3 of a test script importing elements of other test modules to design test behaviour.

```
module AtsIpv6_Common_Functions {
 // Importing Generic Libraries
 //LibCommon
 import from LibCommon_BasicTypesAndValues all;
 import from LibCommon_DataStrings all;
    . . .
 // Importing test data modules
 //LibIpv6
 import from LibIpv6_Interface_Templates all;
 import from LibIpv6_CommonRfcs_TypesAndValues all ;
    . . .
 // Importing test architecture modules
 //AtsIpv6
 import from AtsIpv6_TestSystem all;
 import from AtsIpv6_TestConfiguration_TypesAndValues all ;
    . . .
} //end module AtsIpv6_Common_Functions
```

Discussion

A difficulty in applying this pattern consists in ensuring that the number of separate modules remains within sensible limits. Otherwise, the effort of managing all parallel activities can reduce the positive impact of the pattern and even lead to less productivity. However a small number of modules will inevitably lead to more version controlling conflicts, with several people potentially working in parallel on the same modules. In such cases the usage of an appropriate version controlling system, along with clearly defined policies is highly recommended.

Related Patterns

This pattern is an application of the Separation of Concern, a.k.a *Divide and Conquer* design pattern known both in generic software engineering, as well as in test design [47].

A.1.2 Pattern: Grouping of Test Design Concerns

Context

This pattern is a generic organisational test design pattern and is applicable at any test scope for test projects of any size.

Problem

The size of test models can grow considerably in the development process. How to organise tests within a module to enhance reuse, maintainability and readability? To be able to manage the test model conveniently, elements added to it should be easy to localize to check their definition, modify them or even refer to them.

Solution

Just as each test model should be divided in several different modules which can be concatenated using an import mechanism, each of those modules should have a clear structure using a grouping mechanism to organise tests artifacts. The grouping mechanism allows for elements of the test model to be organised in groups which can be used to keep a clear overview of the elements contained in a test model file or module. The criteria for grouping can be defined based on:

- SUT Features: e.g. in a test behaviour model different groups of test cases can be defined, each of them covering a feature of the SUT targeted by the contained test cases. This will facilitate selecting or disabling that group of test cases, depending on whether the feature is effectively provided in the end product or not.

- Category of elements: e.g. in a test data model, a group can be defined to contain data type definitions, while another one can be defined containing data instance definitions. In a similar manner, a test behaviour model may be organised in separate groups, e.g. containing respectively test cases, function definitions or any other elements of test behaviour.

For more expressiveness, the grouping mechanism should allow groups to contain subgroups

Known Uses

- The UTML notation introduced in this work provides a grouping mechanism for all kinds of test model elements it supports
- The TTCN-3 language [58] also provides a grouping mechanism using its **group** keyword.
- The TPLan [57, 146] also defines a **group** keyword in its syntax for the same purpose.
- Although, xUnit [109] (junit, HTTPUnit, XML-Unit, etc.) do not explicitly define a grouping mechanism for organising tests, test cases can be grouped together, using an xUnit **TestSuite** class, which can contain several test cases, but can be executed at once.

Discussion

While this pattern may not be very useful for small size test models, it is nearly indispensable for any bigger ones. There is practically no alternative to grouping as such. One potential pitfall to be avoided is, when the tree structure created by the groups and their contained subgroups is too deep, to the extent that the contained model elements become too difficult to access.

Related Patterns

This pattern is an extension to the *Aspect Driven Test Design* pattern defined in Section A.1.1

References

[67, 47]

A.2 Patterns in Test Objectives Design

A.2.1 Pattern: Prioritization of test objectives

Context

This pattern is an organisational test design pattern that aims at optimizing the planning of testing activities and is applicable to any test scope.

Problem

Due to resource limitations, it is often the case in testing projects that not all test cases can be developed and/or executed at a time. How to design tests, so that key decisions can be taken confidently in the testing process. Key decisions include:

- When can test activities be considered sufficient to provide a level of confidence in the SUT that is high enough to allow its release?
- Which test cases need to be implemented and executed first and which ones can be left aside for later stage in the testing process?

Solution

As recommended by IEEE 829 [83], introduce a prioritization scheme for test objectives in the test model. Prioritization can be provided for a test objective taken individually or for a group of test objectives. Test implementation and test execution can then be planned based on the priority level of the test objectives, to ensure that test cases with highest priority are available on time before product delivery.

Test case prioritization aims at ordering test cases according to some criterion to schedule their implementation and/or execution. The choice of a test case prioritization among the numerous ones described in the literature [50] depends not only on the applied test design strategy, but also on the objectives of that prioritization. Possible objectives include a higher fault detection rate and costs reduction in system and regression testing. Rothermel et al. [139] define the test case prioritization problem as follows:

Given: T, a test suite; PT, the set of possible orderings (prioritizations) of T; and f, a function from PT to the set of real numbers.

Problem: Find $T' \in PT$ such that $(\forall T'')(T'' \in PT)(T'' \neq T')[f(T') \geq f(T'')]$ The test objectives are ordered based on the real number value ($award$) returned by the function f. Obviously that value depends on the prioritization objective and thus on the factors taken into account by function f.

Discussion

Again, similar to other organisational patterns mentioned before, the size of the testing project and the resource constraints it faces shall be taken into account, whenever the application of this pattern is considered.

Known Uses

Besides code-coverage based approaches such as the one proposed by Rothermel et al. for regression testing [138, 139, 49, 50, 44], Srikanth et al. [150, 151] propose an approach consisting in prioritizing requirements for tests, based on a series of factors, e.g. *customer-assigned priority*, *requirements volatility* or *implementation complexity*. Other uses include Srivatsva et al. [152] who propose a prioritization technique that adds risk factors to the equation and Qu et al. [130] who propose a test case prioritization technique suitable for black-box testing. Finally, as suggested by many authors [151, 166], economic factors may also be considered as prioritization factors for test cases.

As shown by the numerous examples mentioned above, prioritization of test cases is used implicitly in several instances, even though it may not always be explicitely supported by the test design notation itself. Generally a separate tool is used to manage that aspect of the test process. However, it would be highly beneficial to integrate it into the test design process, so that appropriate tool support can be used to calculate the priority value for each test case automatically, based on the predefined factors.

A.2.2 Pattern: Traceability of Requirements to Test Artifacts

Context

This pattern is an organisational test design pattern that aims at facilitating the coupling of testing activities to the rest of the software development process. It is mainly applicable to integration and system tests.

Problem

How to achieve (bi-directional) traceability between test design artifacts and system artifacts to enable automatic coverage analysis, monitor progress of testing activities and assess overall quality?

Solution

Each test objective should be linked to a (set of) requirements or features of the SUT. Those requirements could be functional or non-functional. The test objectives could represent a risk in relation to the feature or a mean for verifying that the SUT meets the requirements

Known Uses

Known uses of this pattern include:

- The UTML meta-model's *TestObjective* element defines a reference to a series of requirements the specified test objective covers
- The TPLan [57, 146] notation also provides a similar concept in its syntax definition

- Some model-based testing tools generate test objective descriptions from state diagrams of the system under the test (e.g Conformiq's QTronic tool [82]). Since, the test objectives are a result of a transformation process from the system model, a link to system requirements is also possible, provided those requirements can be mapped to certain paths in the state automaton.

Discussion

One key difficulty in applying this pattern is to ensure that changes to the test model are propagated in both directions of the link to avoid dead links and keep the test model consistent. The test design tool should take care of that and update a test objective element accordingly, if one of the covered system requirements is altered (e.g. deleted, moved to another location, renamed, etc.). Such a propagation of changes could be facilitated by the usage of the same notation or of the same design technology (e.g. EMF, MOF) for those aspects being linked with each other. Otherwise, some serious maintainability issues will emerge.

A.2.3 Pattern: Selection criteria for test objectives

Context

This test design pattern aims at optimizing testing activities by making testing more efficient and is applicable to any test scope.

Problem

Shorter test development life cycle to address more complex SUTs means not all tests can always be developed and executed in time. How to allow a selection of which tests should be treated with higher priority, while ensuring that a minimum number of failures are still present in the delivered product?

Solution

Let each test objectives model define selection criteria for the applicability of test objectives at individual or at group level. Such selection criteria will be used to make decisions on planing test design, test implementation and test execution. Also, according to whether a given feature is supported by a product line test cases applying to that product line could be selected automatically and prioritized accordingly for development and execution.

Known Uses

The ISO 9646 Conformance Test Methodology Framework (CTMF) [87] defines the concept of an *Implementation Conformance Statement (ICS)* as

> A statement made by the supplier of an implementation or system claimed to conform to a given specification, stating which capabilities have been implemented.The ICS can take several forms: protocol ICS, profile ICS, profile specific ICS, and information object ICS.

ICSs are commonly used in conformance testing to define selection criteria for test objectives and the test cases (or groups thereof) implementing them. That approach is implemented in several TTCN-3 test suites, in which, test execution is controlled using the values set for the ICS in the control part.

Related Patterns

This pattern can be combined with the *Prioritization of Test Objectives* pattern described in Section A.2.1.

A.2.4 Pattern: Traceability of Test Objectives to Fault Management

Context

This organisational test design pattern aims at facilitating the coupling of testing activities to other activities in the software development process. Although it may also be used for unit-level testing, it mainly targets system and integration tests.

Problem

In spite of all testing efforts, errors in software are inevitable and will eventually occur. How can it be ensured that the information gathered in analysing and fixing those errors can be exploited for the benefit of future testing activities and for improving the overall quality of the software product under test?

Solution

Every time a failure is (inadvertently or deliberately) discovered on a version of the SUT, a test objective should be created in the test objectives model to cover that defect and ensure that it will be checked in subsequent regression tests automatically

Known Uses

- Testopia [114] is a test case management extension to the well-known bug management tool Bugzilla. However, due to time constraints, we have not used Testopia yet, to analyse to what extent it applies this pattern. It is likely that similar other tools exist on the market, but we have not got the opportunity to look into those for further analysis yet.

Discussion

The same type of potential issues identified for the *linkage of test objectives to system requirements* pattern (section A.2.2) also apply for this pattern.

A.3 Test Architecture Design Patterns

A.3.1 Pattern: Extensibility/Restriction of Test Architecture Elements

Context

This test design pattern aims at enhancing reuse of test design artifacts and is applicable for (sub-)system level and integration testing. The potential benefits are lower for unit-testing at the class level, because those rarely require complex test architectures.

Problem

Reuse in test development can help in avoiding redundancy and save time, as well as costly resources. Therefore, wherever applicable, means should be provided to reuse already defined test elements to create new ones. Test architecture is one area, where this can be done, with potential high benefits. How to enhance reusability of test architecture artifacts?

Solution

Provide the ability to extend or restrict existing test architecture artifacts. Modifiable test artifacts include. Test architecture artifacts

Known Uses

- The TTCN-3 language provides a concept of component type reuse through the **extends** keyword
- UTML notation implements this pattern by providing the capability to specify a base component type for any new component type being defined, thus, introducing a mechanism similar to inheritance in OO-programming.

Related Patterns

A similar pattern for test data elements is described in section A.4.3

A.3.2 Pattern: One-on-One Test Architecture

Context

This pattern is applicable for system testing. For integration testing, it has limited impact, because more than one component might need to be emulated by the test system. While this might be feasible in some cases, it would be more difficult to achieve if the behaviour of the components to be emulated are required to follow parallel and concurrent threads.

Problem

How to model a test architecture for a system providing a limited set of entry points and interacting with its environment following a sequential non-concurrent behaviour?

Solution

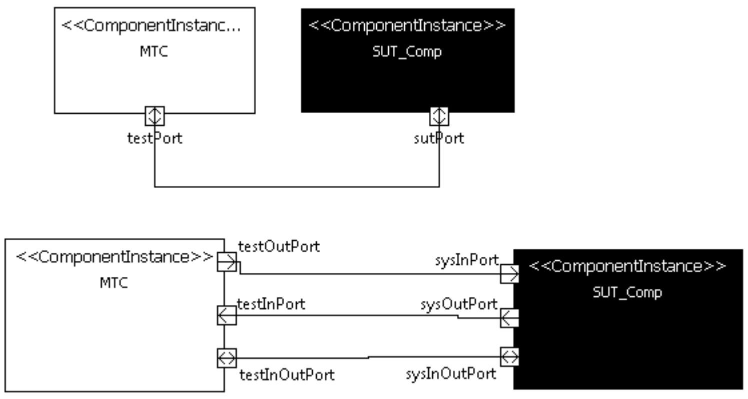


Figure A.1: Test architecture Diagram for One-on-One Pattern

This pattern is applicable to all those SUTs where the SUT interfaces are directly controllable and observable. Benefits: Having a single test component implies that synchronisation mechanisms based on message exchange or other RPC-like mechanism do not have to be implemented at the testing side. Variables defined in the test component can be used to describe states based on which decisions can be made on the test verdict. Shortcomings: The test component has to emulate the complete behaviour of system component it replaces. Depending on the level of complexity of that behaviour, this might be more or less difficult to achieve. Furthermore, having a single component makes it difficult to deal with concurrency at the testing side, if required.

Known Uses

This pattern is applied in numerous conformance test suites. E.g.:

- the collection of IPv6 test suites [145] used e.g. for the IPv6 logo brand
- the IP Multimedia Subsystem (IMS) benchmark test suite [43] used for performance testing IMS server equipment
- the CORBA component test suite [13] used for integration testing of CORBA components

Discussion

Potential difficulties in handling concurrent behaviour from the SUT and to emulate similar behaviour to stimulate the SUT.

Related Patterns

This pattern is the logical opposite to the *Central Test Coordinator* test design pattern described in Section A.3.8. It is also referred to as the *Central tester* test design pattern [51].

A.3.3 Pattern: Point-to-Multi Point(PMP) Test Architecture

Context

This test design pattern is applicable to system or integration-level testing of distributed systems exchanging data through communication protocols.

Problem

How to design a test architecture suitable for an SUT under the following requirements/constraints:

- The test system shall not be distributed, i.e. all its components will be running on a single host.
- The SUT will be exchanging data through several different ports and may consist of several distributed SUT components.

Solution

A PMP test architecture consists of one test component hosting a single port, which is connected to each of the ports provided or required by the SUT components to send impulses and check responses.

Discussion

The advantage of the PMP test architecture design pattern is that it helps the test system in avoiding concurrency issues, because a single port is used in the same test component. However, there are also some drawbacks to be taken into account. For example, given the fact that the test component's port must support communication with more than one SUT component at the same time, the state of each of those communication will have to be handled by the test component. Furthermore, if the SUT ports support different communication protocols, then the test component ports will have to support all of those communication protocols at the same time and provide multiplexing capability to handle each of the communication channels separately.

A.3.4 Pattern: Flexibility of the test architecture model

Context

This pattern is applicable to integration- and system testing.

Problem

How to facilitate transformation of SUT architecture into test architecture?

Solution

The test architecture model should allow any component within a test architecture to be marked either as parallel test component (PTC) i.e. as part of the test system or as SUT component.

Known Uses

- The TTCN-3 *system* keyword can be used to mark a selected test component in a test case as a representation of the SUT's interfaces.

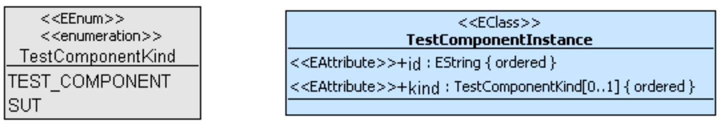
- The UTP standard defines a concept of *SUT* as

  > a part, the system, subsystem, or component being tested [70].

  Further, [70] states that

  > A 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.

- UTML has adopted the SUT concept defined by UTP in the form of the *ComponentKind* attribute of each *ComponentInstance* element of its meta-model, which allows to specify the nature of an entity in a test architecture either as part of the test system or of the SUT. Figure A.2 depicts the



<<EEnum>>
<<enumeration>>
TestComponentKind
TEST_COMPONENT
SUT
<<EClass>>
TestComponentInstance
<<EAttribute>>+id : EString { ordered }
<<EAttribute>>+kind : TestComponentKind[0..1] { ordered }

Figure A.2: UML Class Diagram for UTML ComponentInstance Element

UML class diagram for the UTML *ComponentInstance* element, along with its *ComponentKind* attribute.

A.3.5 Pattern: Proxy Test Component

Context

This test architecture is applicable to (sub-)system- and integration testing

Problem

How to verify that two SUT components behave correctly without interfering in their logic and without having to emulate their behaviour in the test system? How to observe (and evaluate) the communication exchange between SUT components or to observe (and evaluate) the communication exchange at an SUT interface.

Solution

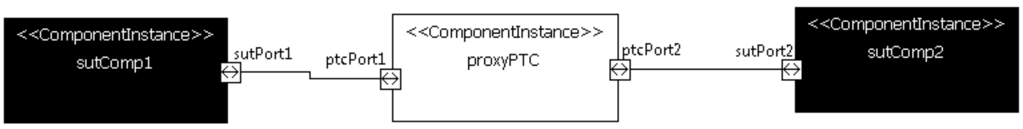


Figure A.3: Test architecture Diagram for Proxy Test Component Pattern

A proxy test component is connected between two SUT components. Each message that is received by the proxy test component is evaluated, then forwarded to its actual recipient, i.e. the other SUT component. A proxy test component can operate in duplex mode and forward messages in both directions if required by the SUT components' design.

Known Uses

Known uses of this pattern include

- Conformance testing of CORBA Components Model (CCM) entities [13]

Discussion

Performance issues: Delay created by the proxy test component might alter the communication between SUT components. If time constraints for that communication are too tight, it might be impossible to apply the pattern, because of the SUT components' inability to deal with such unexpected behaviour. Additionally no online evaluation should be performed by the proxy test component to avoid additional flaws.

Related Patterns

The *Monitoring Test Component* test design pattern described in Section A.3.7 is an extension of this pattern [51]

A.3.6 Pattern: Sandwich Test Architecture

Context

The *sandwich* test architecture design pattern is applicable to subsystem, system- and integration testing.

Problem

How to design a test architecture for an SUT that uses more than one communication channel to exchange data with its environment?

Solution

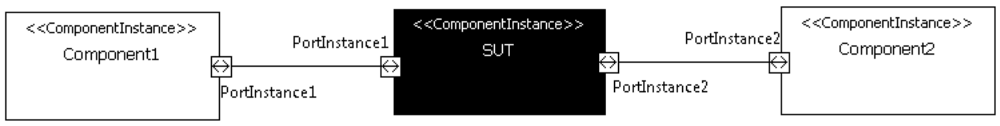


Figure A.4: Test architecture Diagram for Sandwich Test Architecture Pattern

As depicted in figure A.4, the *sandwich* test architecture design pattern features two parallel test components, each of which emulate an entity that interact with the SUT and each of which is connected to the SUT via one or several of its ports.

Known Uses

The *sandwich* test architecture design pattern defines an architecture that is similar to the one defined by ISO 9646 [87] for conformance testing and featuring an upper tester and a lower tester, with the IUT between the two. That kind of architecture is widely used in conformance testing of communication protocols.

Discussion

A *sandwich* test architecture makes more sense, if the behaviour of both test components involved is required to run concurrently in parallel, with no relation to each other. Otherwise the *one-on-one* test architecture may be more appropriate to provide the same functionality, while avoiding the computational and implementation costs of parallel test components.

A.3.7 Pattern: Monitoring Test Component

Context

The *monitoring test component* test architecture design pattern is applicable to subsystem-, system- and integration testing.

Problem

How to observe (and evaluate) the communication exchange between SUT components or to observe (and evaluate) the communication exchange at an SUT interface.

Solution

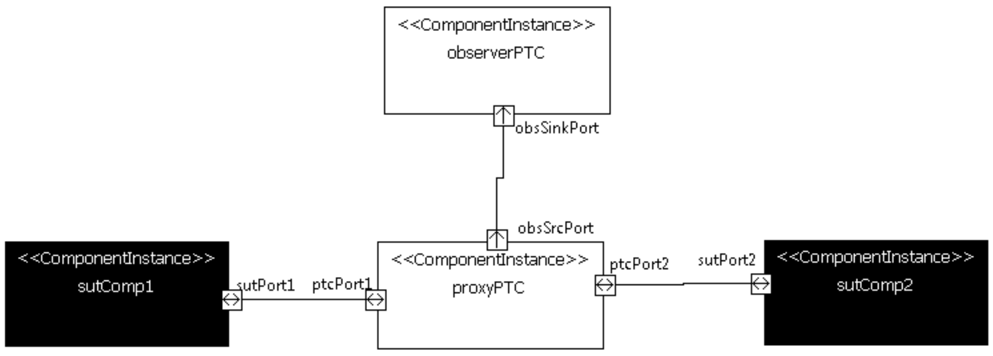


Figure A.5: Test architecture Diagram for Monitor Pattern

One test component per monitored connection or one test component per monitored port is involved in this pattern. It observes at a special monitoring port being attached to a connection between SUT components or the communication at the port to which it is attached to. The monitoring component is in a passive role and is simply a data sink for messages being sent from the SUT component. By defining constraints (time, data) on the incoming data, the monitoring component can check that the SUT component behaves according to the system's requirements.

Known Uses

CCM-Testing [13]

Related Patterns

This pattern is a specialization of the *Proxy Test Component* test design pattern described in Section A.3.5.

References

[51]

A.3.8 Pattern: Central Test Coordinator

Context

This pattern is more applicable to integration and system testing. It is less the case for unit testing at the class level. However, it can be applied for system testing, whereby a unit testing framework is instrumented for that purpose.

Problem

How to model a test architecture that is suitable for load- , performance- or conformance testing on an SUT requiring parallel and possibly distributed processing.

Solution

As depicted in Figure A.6, this pattern features a test component acting as test coordinator and thus controlling the life cycle other components it controls. Each of the controlled test components is connected to the controlling component via a connection through which coordination messages can be exchanged to control the components' behaviour. To keep the overhead of processing those coordination messages as low as possible, to not affect the proper test behaviour, coordination messages should be kept as simple as possible in their structure. The real testing activities are performed by the controlled test components, which are directly connected to the SUT.

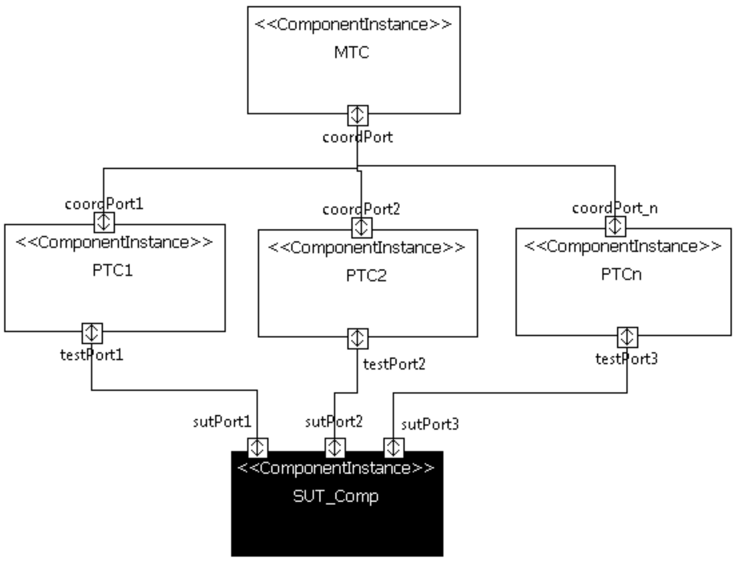


Figure A.6: Test architecture Diagram for Central Test Coordinator Pattern

Known Uses

Several TTCN-3 projects such as [124] involving UTML protocol testing (Siemens) and [42] involving BCMP protocol performance testing.

Discussion

An intelligent coordination pattern is required between the main test component and the parallel test components. The additional load and delays created by that communication should be taken into account while evaluating the SUT component's test results.

Related Patterns

This pattern is the opposite of the *One on One test architecture* pattern defined in Section A.3.2

References

[51]

A.4 Test Data Design Patterns

In this section patterns for designing test data are presented, as well as patterns for automatically generating test data

A.4.1 Pattern: Purpose-Driven Test Data Design

Context

This pattern is more applicable to integration and system testing. For unit testing, the efforts implied would outweigh the potential benefits of applying the pattern.

Problem

How to ensure that all the test data model elements required for a test suite have been defined ? In large test development projects involving more than one test engineer working on the same test model, there is a need to ensure that redundant data is not defined at many instances in the same test suite, thus negatively affecting readability and maintainability. For example, in TTCN-3 test suites, too many templates might be defined under different identifiers, although they represent the same test data instances, in terms of functionality. Such redundancy in a test system can affect its understandability and maintainability.

Solution

Assign each defined test data a rule specifying what makes the test data unique and what purpose it fulfills in the test suite. Additional benefit might be obtained by using a machine processable notation for specifying the rule associated to the data instance. For that purpose, an assertion language such as OCL can be used. Based on that rule, before a new test data would be added to the test model, it can be checked automatically, if another test data meeting the associated criteria does not exist yet in the test model and a warning issued accordingly.

Known Uses

- The UTML associates each test data instance with a set of constraints it meets

- The Classification Tree Method (CTM) and similar class partitioning approaches for data generation are applications of this pattern.

Discussion

A post-analysis of the test model [117] can also help identifying and addressing this problem.

A.4.2 Pattern: Basic Static Test Data Pool

Context

This test design pattern is applicable for any test scope.

Problem

Defining test data from scratch is time costly and inefficient. How to reduce the effort for that activity and save costly time and resources?

Solution

Providing a basic data pool from which further data instances can be specified, using extension/restriction schemes can help in reducing the test data specification efforts. The initial set of static test data definitions can be specified or automatically generated, using one the following techniques:

- Boundary Value Analysis(BVA) [142, 131]
- Random Value Analysis(RVA)
- Default Values Analysis(DVA)
- (Domain) Equivalence Partitioning(EP)

Detailed descriptions of each of these techniques can be found in various publications on testing (e.g. The British Computing Society's (BCS) Standard for Software Component Testing [84]).

Once the basic set of static test data has been specified or generated, new instances of test data can be created by changing some of the properties of the appropriate basic test data, or by adding further rules, based on which the instances of the test data can be created at test execution or be used to validate the EUT's responses.

Known Uses

The UTML notation allows the application of this pattern by providing the capability to link a specified data instance with a data pattern kind describing a mechanism through which a concrete instance of the specified test data can be created. Data pattern kinds correspond to the testing techniques listed above. The implementation of the mechanism is left to the test environment.

Related Patterns

This test design pattern can be combined with the *Reusable test data definitions* pattern described in Section A.4.3.

References

A.4.3 Pattern: Reusable Test Data Definitions

Context

This test design pattern is applicable to unit, integration and system testing

Problem

How to facilitate reuse of already defined test data artifacts?

Solution

To facilitate reuse of already defined test data artifacts (i.e. both types and instances) the test design notation should provide a mean for referring to existing test data elements in the test model. The relationship between the original test data artifact and the new one can be based on extension or restriction. An extension means, all the rules of the original remain valid, but are extended with new additional rules. For example, for a data type definition an extension may consist in adding an additional field to the existing structure of the type. On the other hand, a restriction maintains the structure of the original data artifact as-is, but adds new constraints to it. An example of constraint would consist in making mandatory a field that was previously defined as optional in a data type definition.

Known Uses

Mechanisms for extending/restricting existing test data artifacts are provided by several test notations (e.g. the XML Schema Descriptor language (XSD), TTCN-3, UTML). Classical object inheritance, as supported by several object-oriented programming languages can also be instrumented to implement a similar result, in situations whereby they are used for test scripting.

Discussion

If the test notation supports a form of inheritance, it will facilitate the application of this pattern.

Related Patterns

This test design pattern can use the *Default values*, the *Boundary values* and the *Domain partitioning* techniques described in section A.4.2, which provide a base for the *Basic static test data pool* pattern.

A.4.4 Pattern: Dynamic Test Data Pool

Context

This Pattern is applicable to any test scope

Problem

Statically defined test data restrict the coverage of the tests, because they increase the risk of ignoring certain areas of the testing domain. Certain tests require for test-data to be generated dynamically, at test execution time, on an "on-demand" basis. This can be very useful in situations whereby it would be too costly to define all test data statically. Furthermore that enhances the quality of the tests, as each set of test will be created very specifically for the objective to be addressed by that test case.

Solution

A dynamic test data pool is an entity which can be called via a predefined API to generate test data dynamically, i.e. during test execution. For that purpose, the data pool is provided a set of criteria, which the generated data is supposed to fulfill, and based on which an appropriate test data instance will be selected or generated to be returned to the calling entity. For expressing the criteria on the test data, a constraint notation such as the OMG Object Constraint Language (OCL) or any other similar notation is recommended to allow automated processing.

Known Uses

- This pattern is applied by IBM's Rational Functional Tester to generate test data based on equivalence class partitioning.

- The Classification Tree Method (CTM) [68] follows a strategy similar to this pattern, with each branch of the classification tree representing a constraint fulfilled by the associated data.

- This pattern is also applied in OO-Programming (e.g. C# [129]) as a mean to provide test data on an on-demand base. The approach proposed there consists in using the *Builder* design pattern [60] and to create a builder class for every class to be tested. The builder class provides a set of methods, each creating a different flavor of an instance of the class, matching certain requirements for the purpose of testing.

Discussion

The mean for defining the selection criteria of test data instances is a critical aspect of this pattern. The chosen notation should base on a clearly defined syntax to allow the criteria to be processed automatically while selecting matching data instances. Using natural language instead would significantly reduce the impact of the pattern.

Related Patterns

This pattern is sometime combined with the *class partitioning* pattern, whereby equivalent classes are defined and test data are dynamically generated for each class, based on its defining criteria.

To instantiate this test design pattern, the *Builder* design pattern, which is similar to the *Factory* pattern [60], can be used. In particular in the case of unit-level testing.

A.5 Test Behaviour Design Patterns

Patterns in test behaviour modelling

A.5.1 Pattern: Assertion-Driven Test Behaviour Design

Context

This pattern can be applied to unit, integration and system testing

Problem

How to ensure that the intent of each test case can be quickly understood, without having to navigate too deeply into the test script's source code?

Solution

While modelling each test case, the focus should always be laid on what behaviour is expected from the SUT for that particular test case. Even if an erroneous behaviour is expected, then the *positive path* for the test case is the one to be visible from the test case's design and implementation. Here, the positive path in a test script for a test case is defined as the one leading to a *PASS* verdict. That means, there should be no "positive" *FAIL* verdict. Furthermore, unexpected behaviour in testing should not be modeled explicitly in test behaviour model, but should rather be handled implicitly by some exception handling or similar mechanism, based on the expected behaviour's model. Otherwise the test model loses in readability and maintainability.

Known Uses

- The concept of
- Several TTCN-3 test suites define functions for key actions in the test scenarios and invoke those functions in the test cases, instead of putting all the details of those actions at the highest level of the source code, i.e. the test case level. Key actions include sending an impulse to the SUT, receiving a given response from the SUT, checking that SUT's response meets some defined constraints and assertions, etc.
- The TTCN-3 *altstep-default* mechanism can also be viewed as an application of this pattern. A TTCN-3 *default* behaviour is one that is used as alternative whenever an explicitly specified behaviour does not occur. Activating/deactivating a TTCN-3 *altstep-default* switches it on/off as possible alternative behaviour.
- xUnit (JUnit, HTTP-Unit . . .) use this test design pattern. In JUnit and frameworks based on JUnit, the test cases mainly consist of assertions to be verified on objects and values from returning methods. If any exception is thrown in the process a *FAIL* or an *ERROR* verdict is set for the test case.

Discussion

Related Test Patterns

This pattern provide the base for all xUnit Test Patterns [109]. Also it is widely used as *Assertion-Based Verification* for various software domains ranging from UML to embedded systems.

References

xUnit Test Patterns [109]

A.5.2 Pattern: Context-Aware Test Behaviour Design

Context

This pattern is applicable to any test scope

Problem

While designing a test model, there is always a risk for conceptual flaws finding their way into the model, thus making it inconsistent and more difficult to exploit for automatic processing. Such conceptual flaws include for instance, the specification of actions that are effectively impossible to implement in black-box testing scenarios. For example, specifying a test impulse from an entity marked as being part of the SUT or selecting an abstract data instance to be used as test impulse. How could such errors be avoided right away to ensure high quality of the resulting test models and save costs?

Solution

While modelling test behaviour, the current test context should constantly be taken into account. For example, it should be ensured that the test modeler designs the test behaviour from the tester's perspective in a black-box testing approach. Therefore, the test design tool should use those context information to filter the choices presented to the user for selection in a pro-active approach, to avoid that, conceptual flaws are introduced in the test model. Furthermore, the test design tool should provide facilities to verify a test model partly or entirely, to identify such flaws and provide guidance for their correction.

Known Uses

- The TTCN-3 applies this pattern in the semantics of its **port** concept, by providing the possibility of specifying a test component for a given test case as the **system** component. TTCN-3 compilers then check that the semantics match the defined rules. However, a pro-active approach, consisting in appropriate type completion and wizards, while writing TTCN-3 code is not yet supported by most of the existing tools.

- Model-driven test engineering approaches like the one proposed in this work with the UTML notation offer better opportunities to implement this test pattern, since model-driven development environments provide the technical means for attaching rules to a metamodel in form OCL constraints, which can be evaluated on-line (i.e. as the test model is being designed) or offline (after the test model has completely been designed).

Discussion

The application of this pattern requires a good documentation and technical support on the part of the test design tools, as it might not always be clear to users why certain operations, they try to perform would be disallowed. Also, the error and warning messages resulting from validation should be clear enough to inform the person doing the test design on the issue identified and potentially on ways to address them.

A.5.3 Pattern: Test Component Factory

Context

Integration-, System Testing

Problem

How to model dynamically scalable, but yet maintainable tests?

Solution

One test component, generally called main test component (MTC) serves as the generator without any further test functionality. The actual testing of the SUT is performed by the generated parallel test components (PTCs). The behaviour of all PTCs is specified only once and linked to the type definition of the component type.

Known Uses

This pattern is used in several test suites. In particular to measure the latency of servers e.g. IMS serving entities [43], web services. The test component factory can generate parallel test components at runtime, with each of those emulating a client. The ability to instantiate test components dynamically makes it possible to generate load on the server under test according to any predefined scenario or to reflect a given distribution (Poisson, Normal, Exponential, etc.).

Discussion

The architecture of the PTCs, i.e. the connections among themselves and between them and the EUTs is not considered by this pattern and should be addressed by the use of the appropriate architectural patterns. It should be taken into account that a mechanism for controlling the lifecycle of the test components will also be required, along with a mean to coordinate the behaviour of the created PTCs.

Related Patterns

This pattern is the testing pendant to the abstract factory design pattern known in generic software design [90, 62]. It has also been referred to as the generator pattern [51] or the *Give me an army* pattern [51]

References

[51]

A.5.4 Pattern: Central Coordination of Test Components

Context

Integration- and System Testing

Problem

How to coordinate multiple parallel test components to perform a given test behaviour?

Solution

Let one of test components play a central role as main coordination points for the remaining test components in the test architecture.

Known Uses

In TTCN-3, the Main Test Component (MTC) can be used as central coordination point for the parallel test components. For that purpose, it must be connected to each of them and exchange coordination messages over those connections to control the behaviour of the PTCs to achieve the required behaviour for the whole test system.

Related Patterns

This pattern is related to the *centralized test coordinator* architectural test design pattern mentioned in section A.3.8 above. Furthermore, this pattern extends the *test component factory* pattern mentioned previously. Finally, this pattern is the logical opposite to the *distributed coordination of test design patterns* pattern.

References

[51]

A.5.5 Pattern: Distributed Coordination of Test Components

Context

Problem

How to coordinate multiple parallel test components to perform a given test behaviour?

Solution

Define and implement a coordination scheme, whereby the behaviour of each test components depends of the behaviour of the other test components involved in the test scenario.

APartioning of Test Artifacts

Known Uses

This test design pattern is applied in several TTCN-3 test suites, using a combination of that notation's **stop** and **stop all** keywords.

Discussion

Related Patterns

This pattern is the logical opposite of the *Central Coordination of Test Components* test design pattern described in Section A.5.4. The *Follow the Leader* test behaviour pattern described in the ETSI's collection of test design patterns is an extension of this test design pattern, which defines how test components should terminate their life cycle, depending on the termination of one of the others test components involved in the test architecture. The initially terminating test component is referred to as the "leader", which other test components follow, by stopping their test execution and terminating as well.

References

[51]

A.5.6 Pattern: Time Constraints in Test Behaviour

Context

This pattern is applicable to any test scope

Problem

How to handle exceptional situations in test scenarios involving interactions between test components among themselves or with SUT components

Solution

Define timing constraints on test actions involving more than one component. E.g. for each action representing an impulse to an SUT component or an expected response, provide a timing constraint to allow the test system to recover, if the action does not complete smoothly. The timing constraint can be defined via a timer which is started shortly before the action is started and which would trigger an event, if it expires before the action has completed as expected.

Known Uses

- In TTCN-3 a so-called guard-timer can be used to define a timing constraint for an expected signal on a test component. The guard timer's expiration, while waiting for a reaction from the SUT, triggers an event that can be handled to set the test verdict accordingly.
- Real-Time TTCN-3 [37] proposes to extend the TTCN-3 notation with the concept of this pattern.
- The UTML notation applies this pattern by attaching a timer specification to every specification of an event expected as response from an EUT.

Related Patterns

This pattern is equivalent to the *latency* test design pattern mentioned in [37] and used in performance testing of various kinds of servers (e.g. web, application, etc.).

Appendix B

UTML Mapping Examples

This appendix provides some details on the mapping of UTML elements to TTCN-3 and JUnit.

B.1 UTML to TTCN-3 Mapping Rules

B.1.1 Testcase

```
[template public processTestcase (testcase_p:Testcase)]
/**
[if (testcase_p.description.oclIsTypeOf(OclVoid))]
* @desc:
[printAsComment(testcase_p.description)/]
[/if]
* @purpose [if (testcase_p.testObjective.oclIsTypeOf(OclVoid))]
[for (t_obj:TestObjective | testcase_p.testObjective)] [t_obj.id/][/for]
[/if]
* TP version:
[if (testcase_p.testProcedure.oclIsTypeOf(OclVoid))]
[if (testcase_p.testProcedure.testObjective.oclIsTypeOf(OclVoid))]
  [for (t_obj: TestObjective | testcase_p.testProcedure.testObjective)]
   [for (descElt: DescriptionElement | t_obj.objectiveDescElement)]
    [if (descElt.value<>"")]
*
* [mapKeyword(descElt.name)/]
[printAsComment(descElt.value)/]
   [/if]
  [/for]
 [/for]
[/if]
[/if]
 [for (t_obj: TestObjective | testcase_p.testObjective)]
  [for (descElt: DescriptionElement | t_obj.objectiveDescElement)]
   [if (descElt.value<>"")]
*
* [mapKeyword(descElt.name)/]
```

```
[printAsComment(descElt.value)/][/if][/for]
[/for]
[if (testcase_p.testProcedure.oclIsTypeOf(OclVoid))]
* Test procedure:
[let cnter:Integer = 1]
[for (t_step: TestStep | testcase_p.testProcedure.testStep)]
[printAsComment(cnter.toString().concat(": ").concat(t_step.content))/]
[let cnter:Integer = cnter+1]
[/for]
[/if]
*
*/
[let i_nrOfComps:Integer = testcase_p.componentInstance->size()]
[let testCompType:String = "ComponentType"/]
[let sysCompType:String = "SystemComponentType"/]
[if (i_nrOfComps > 2)]
 [if (testcase_p.componentType.oclIsTypeOf(OclVoid))]
  [let testCompType:String = testcase_p.componentType.name/]
 [/if]
 [if (testcase_p.systemComponentType.oclIsTypeOf(OclVoid))]
  [let sysCompType:String = testcase_p.systemComponentType.name/]
 [/if]
[else]
 [if (testcase_p.localTestComponent.oclIsTypeOf(OclVoid))]
 [if (testcase_p.localTestComponent.type.oclIsTypeOf(OclVoid))]
 [let testCompType:String = testcase_p.localTestComponent.type.name/]
 [/if]
 [/if]
[/if]
testcase [testcase_p.name/]()
runs on [testCompType/]
system [if (testcase_p.systemComponentType.oclIsTypeOf(OclVoid))]
[testcase_p.systemComponentType.name/] [else] SystemComponentType [/if]
{

 [if (testcase_p.variableDeclaration->size() > 0
 || testcase_p.timerDeclaration->size() > 0)]
 //Local variables and timers
  timer T_WAIT;
 [for (vd:VariableDeclaration | testcase_p.variableDeclaration)]
 [processVariableDeclaration(vd)/]
 [/for]
 [for (td:Timer | testcase_p.timerDeclaration)]
  [processTimerDeclaration(td)/]
 [/for]
 [/if]

 [if (testcase_p.testProcedure.oclIsTypeOf(OclVoid))]
 [if (testcase_p.testProcedure.testObjective.oclIsTypeOf(OclVoid))]
  //Test execution
  [processPreconditions(testcase_p.testProcedure.testObjective, testcase_p.name)/]
 [/if]
 [elseif (testcase_p.testObjective.oclIsTypeOf(OclVoid))]
  //Test execution
  [processPreconditions(testcase_p.testObjective, testcase_p.name)/]
 [/if]


 //Setup configuration
```

```
[if (testcase_p.testArchitecture.oclIsTypeOf(OclVoid))]:
 [testcase_p.testArchitecture.id/][/if]
[if (i_nrOfComps > 2)]
//Instanciate test components
 [for (comp:ComponentInstance | testcase_p.componentInstance)]
  [if (comp.kind.literal <> "SUT")]
   var [comp.type.name/] [comp.id/] :=  [comp.type.name/].create;
  [/if]
 [/for]
[/if]
[if (testcase_p.testArchitecture.oclIsTypeOf(OclVoid))]
 [if (testcase_p.testArchitecture.setupFunction->size()>0)]
  [for (setupFunction:TestBehaviourActionDef |
   testcase_p.testArchitecture.setupFunction)]
  [processTestAction(setupFunction, testcase_p.name)/]
  [/for]
 [else]
 [for (connection:Connection | testcase_p.testArchitecture.connections)]
  [if (i_nrOfComps > 2)]
  [processConnection(connection)/]
  [else]
  [processSingleComponentConnection(connection)/]
  [/if]
 [/for]
 [/if]
 [if (testcase_p.testArchitecture.associatedDefault.oclIsTypeOf(OclVoid))]
 [for (default: DefaultBehaviourDef | testcase_p.testArchitecture.associatedDefault)]
activate([default.id/]());
 [/for]
[else]
 //WARNING: No configuration for testcase
[/if]
[/if]

//Preamble
[if (testcase_p.beginState.oclIsTypeOf(OclVoid))]
 [for (st:State | testcase_p.beginState)]
 [for (t_act:TestAction | st.triggeringActions)]
  [if (t_act.theComponent.oclIsTypeOf(OclVoid))]
  [if (i_nrOfComps > 2)][t_act.theComponent.id/].start([/if]
    [if (t_act.oclIsTypeOf(TestBehaviourActionInvocation))]
  [if (t_act.testBehaviourActionDef.oclIsTypeOf(OclVoid))]
[if (t_act.storage.oclIsTypeOf(OclVoid))][t_act.storage.name/]
:= [/if] [t_act.testBehaviourActionDef.name/]([processParams(t_act)/])
  [else]
  // Warning: Missing TestBehaviourActionDef in test behaviour action invocation.
  // No code generated
  [/if]
  [/if]
  [if (i_nrOfComps > 2)])[/if];
  [if (i_nrOfComps > 2)][t_act.theComponent.id/].done;[/if]
  [else]
  //WARNING: No test component for invocation. No code generated
  [/if]
 [/for]
 [/for]
[/if]

//Test body
```

```
[if (i_nrOfComps <= 2)]
[for (ta: TestAction | testcase_p.testAction)]
 [processTestAction(ta, testcase_p.name)/]
[/for]
[else]
 //First start passive components
 [for (comp:ComponentInstance | testcase_p.passiveComponentInstance)]
  [if (comp.kind.literal <> "SUT")]
[comp.id/].start( f_[testcase_p.name/]_[comp.id/]_behaviour());
  [/if]
 [/for]
 //Then, start active components
 [for (comp:ComponentInstance | testcase_p.activeComponentInstance)]
  [if (comp.kind.literal <> "SUT")]
[comp.id/].start( f_[testcase_p.name/]_[comp.id/]_behaviour());
  [/if]
 [/for]
 //Wait until components complete their job
 [for (comp:ComponentInstance | testcase_p.componentInstance)]
  [if (comp.kind.literal <> "SUT")]
  [comp.id/].done;
  [/if]
 [/for]

[/if]

//Postamble
[if (testcase_p.endState.oclIsTypeOf(OclVoid))]
 [for (st:State | testcase_p.endState)]
 [for (t_act:TestAction | st.triggeringActions)]
  [if (t_act.theComponent.oclIsTypeOf(OclVoid))]
  [if (i_nrOfComps > 2)][t_act.theComponent.id/].start([/if]
    [if (t_act.oclIsTypeOf(TestBehaviourActionInvocation))]
  [if (t_act.testBehaviourActionDef.oclIsTypeOf(OclVoid))]
[if (t_act.storage.oclIsTypeOf(OclVoid))][t_act.storage.name/]
:= [/if][t_act.testBehaviourActionDef.name/]([processParams(t_act)/])
  [else]
  // Warning: Missing TestBehaviourActionDef in test behaviour
  // action invocation. No code generated
  [/if]
  [/if]
  [if (i_nrOfComps > 2)])[/if];
  [if (i_nrOfComps > 2)][t_act.theComponent.id/].done;[/if]

  [else]
  //WARNING: No test component for invocation. No code generated
  [/if]
 [/for]
 [/for]
[/if]

[if (testcase_p.testArchitecture.oclIsTypeOf(OclVoid))]
[if (testcase_p.testArchitecture.teardownFunction->size() > 0)]
//Teardown configuration: [testcase_p.testArchitecture.id/]
[for (teardownFunction:TestBehaviourActionDef
| testcase_p.testArchitecture.teardownFunction)]
  [processTestAction(teardownFunction, testcase_p.name)/]
[/for]
[else]
```

```
 [for (connection:Connection | testcase_p.testArchitecture.connections)]
  [if (i_nrOfComps > 2)]
  [processDisconnection(connection)/]
  [else]
  [processSingleComponentDisconnection(connection)/]
  [/if]
 [/for]
 [/if]
 [/if]
}//end [testcase_p.name/]
[if (testcase_p.testObjective.oclIsTypeOf(OclVoid))]
  [for (t_obj: TestObjective | testcase_p.testObjective.)]
   [for (descElt:DescriptionElement | t_obj.objectiveDescElement)]
    [if (descElt.name.toLower().contains("description"))]
    with {extension "Description: [descElt.value/]"}
   [/if]
  [/for]
 [/for]
[/if]

[/template]
```

B.1.2 SendDataAction

```
[template public processSendDataAction (action_p:SendDataAction, mirror_p:Boolean)]
 [if (action_p.connection.oclIsTypeOf(OclVoid)
 && action_p.sourcePort.oclIsTypeOf(OclVoid)
 && action_p.transmittedDataInstance.oclIsTypeOf(OclVoid))]
  [if (!mirror_p)]
  [action_p.sourcePort.name/].send([action_p.transmittedDataInstance.name/]
  [if (action_p.transmittedDataInstanceParameter->size()>0)]
  ([processParamsList(action_p.transmittedDataInstanceParameter)/])[/if]);
  [else]
  [if (action_p.destPort.oclIsTypeOf(OclVoid))]
  timer t_default := 100.0;
  alt{
   [][action_p.destPort.name/].receive([action_p.transmittedDataInstance.type.name/]:?)
   {
    [if (action_p.passCriterium)]
    setverdict(pass,"*** received expected
    [action_p.transmittedDataInstance.type.name/] message ***");
    [else]
    log("*** received expected
    [action_p.transmittedDataInstance.type.name/] message ***");
    [/if]
   }
   [][action_p.destPort.name/].receive{
    setverdict(fail,"*** received unexpected message ***");
    stop;
   }
   []t_default.timeout{
    setverdict(fail,"*** timer t_default timed out =>
     message [action_p.transmittedDataInstance.name/] not received ***");
    stop;
   }
  }
   [else]
   // Warning: Test model incomplete:
```

```
   // Unspecified Destination Port for Send Data Action. No code generated
   [/if]
  [/if]
 [else]
  // Warning: Test model was incomplete. No code generated for send data action
 [/if]
[/template]
```

B.1.3 ReceiveDataEvent

```
 [template public processReceiveDataEvent
 (action_p:ReceiveDataEvent, context_p:String, mirror_p:Boolean)]
 [if (action_p.connection.oclIsTypeOf(OclVoid)
&& (action_p.timer.oclIsTypeOf(OclVoid) || action_p.defaultTimer.oclIsTypeOf(OclVoid))
&& action_p.receptionPort.oclIsTypeOf(OclVoid)
&& action_p.expectedDataInstance.oclIsTypeOf(OclVoid))]
 [if (action_p.timer.oclIsTypeOf(OclVoid))]
 [let timer:Timer = action_p.timer]
 [elseif (action_p.defaultTimer.oclIsTypeOf(OclVoid))]
 [let timer:Timer = action_p.defaultTimer][/if]
 [if (!mirror_p)]
  [let b_storage:Boolean = false/]
  [if (action_p.storage.oclIsTypeOf(OclVoid))]
   [let b_storage:Boolean = true/]
   //@processVariableDeclaration varDecl_p = action_p.storage/
  [/if]
  [if (action_p.timerRestart)]
  [timer.name/].start;
  [/if]
  alt{
   [][action_p.receptionPort.name/]
   .receive([action_p.expectedDataInstance.name/]
   [if (action_p.expectedDataInstanceParameter->size()>0)]
   ([processParamsList(action_p.expectedDataInstanceParameter)/])
   [/if])[if (b_storage)] -> value [action_p.storage.name/][/if]{
    [timer.name/].stop;
    [if (action_p.passCriterium)]
    setverdict(pass,"*** [context_p.toUpper()/]:
     [action_p.expectedDataInstance.type.name/] message received as expected ***");
    [else]
    log("*** [context_p.toUpper()/]:
     [action_p.expectedDataInstance.type.name/] message received as expected ***");
    [/if]
   }
   [if (wrapper.properties.generate_timeout_branches)]
   [if (timer.oclIsTypeOf(OclVoid))]
   [][timer.name/].timeout{
    setverdict(fail,"*** [context_p.toUpper()/]:
    Time out while expecting [action_p.expectedDataInstance.type.name/] message ***");
    stop;
   }
   [else]
   //Warning: Code generation skipped for ReceiveDataEvent: timer missing
   [/if]
   [/if]
  }
 [else]
  [if (action_p.sourcePort.oclIsTypeOf(OclVoid))]
```

```
   [if (action_p.expectedDataInstance.mirrorDataInstance.oclIsTypeOf(OclVoid))]
   [action_p.sourcePort.name/]
   .send([action_p.expectedDataInstance.mirrorDataInstance.name/]);
   [else]
   [action_p.sourcePort.name/].send([action_p.expectedDataInstance.name/]);
   [/if]
   [else]
   //Warning: Code generation skipped for ReceiveDataEvent: source port missing
   [/if]
  [/if]
  [else]
   // Warning: Test model was incomplete. No code generated for receive data action
  [/if]
[/template]
```

B.1.4 SendDiscardAction

```
 [template public processSendDiscardAction (action_p: SendDiscardAction,
 context_p:String, mirror_p:Boolean)]
 [if (action_p.connection.oclIsTypeOf(OclVoid)
&& action_p.sourcePort.oclIsTypeOf(OclVoid)
&& action_p.transmittedDataInstance.oclIsTypeOf(OclVoid)
&& action_p.timer.oclIsTypeOf(OclVoid)
&& action_p.allowedResponse.oclIsTypeOf(OclVoid))]
  [processSendDataAction(action_p, mirror_p)/]
  [if (action_p.timerRestart)]
  [action_p.timer.name/].start;
  [/if]
  alt{
  [for (resp: Response | action_p.allowedResponse)]
   [for (data: MessageTestDataInstance | resp.expectedData)]
   [][resp.port.name/].receive([data.name/]){
    log("*** [context_p.toUpper()/]: received allowed [data.type.name/] message ***");
    repeat;
   }
   [/for]
   [for (data: MessageTestDataInstance | resp.unexpectedData)]
   [][resp.port.name/].receive([data.name/]){
    [action_p.timer.name/].stop;
    setverdict(fail,"*** [context_p.toUpper()/]:
    received unexpected [data.type.name/] message ***");
    stop;
   }
   [/for]
   [][resp.port.name/].receive{
    setverdict(fail,"*** [context_p.toUpper()/]:
    received disallowed message for a message to be discarded ***");
    stop;
   }
  [/for]
   [][action_p.timer.name/].timeout{
    [if (action_p.passCriterium)]
    setverdict(pass,"*** [context_p.toUpper()/]:
    timer [action_p.timer.name/] timed out => message
    [action_p.transmittedDataInstance.name/] discarded as expected ***");
    [else]
    log("*** [context_p.toUpper()/]: timer [action_p.timer.name/] timed out
    => message [action_p.transmittedDataInstance.name/] discarded as expected ***");
    [/if]
   }
  }
 [else]
  // Warning: Test model was incomplete. No code generated for send-discard sequence
 [/if]
[/template]
```

B.1.5 WaitAction

```
[template public processWaitAction (action_p: WaitAction, context_p:String)]
 [let bTimer:Boolean = false/]
 [let bDelay:Boolean = false/]
 [if (action_p.delay > 0)]
 log("*** [context_p.toUpper()/]:
 start waiting for [action_p.delay_formatted/] seconds. ***");
 [let timerName:String = "T_WAIT"/]
 [let bDelay:Boolean = true/]
 [elseif (action_p.timer.oclIsTypeOf(OclVoid))]
 [let timerName:String = action_p.timer.name/]
 [let bTimer:Boolean = true/]
 log("*** [context_p.toUpper()/]: start waiting until [action_p.timer.name/] expires. ***");
 [/if]
 [if (wrapper.properties.mapping__wait_action == "")]
 [if (bTimer || bDelay)]
 [if (bDelay)]
 [timerName/].start([action_p.delay_formatted/]);
 [/if]
 alt{
  [][timerName/].timeout{
   [if (bTimer)]
   log("*** [context_p.toUpper()/]:
   finished waiting for timer [timerName/] to expire. ***");
   [else]
   log("*** [context_p.toUpper()/]:
   finished waiting for [action_p.delay_formatted/] seconds. ***");
   [/if]
  }
 }
 [else]
//WARNING: Timer and delay missing for WaitAction model element. No code will be generated
 [/if]
 [else]// Customized Mapping
 [if (bTimer || bDelay)]
  [if (bDelay)]
  [wrapper.properties.mapping__wait_action/]([action_p.delay_formatted/]);
  [elseif (bTimer)]
  [wrapper.properties.mapping__wait_action/]([action_p.timer.delay_formatted/]);
  [/if]
 [else]
//WARNING: Timer and delay missing for WaitAction model element. No code will be generated
 [/if]
 [/if]
[/template]
```

B.1.6 SetupConnectionAction

```
[template public processSetupConnectionAction (action_p: SetupConnectionAction)]
 [if (action_p.sourcePort.
 oclIsTypeOf(OclVoid) && action_p.destPort.oclIsTypeOf(OclVoid))]
  [if (action_p.destPort.theComponent.oclIsTypeOf(OclVoid))]
   [if (action_p.destPort.theComponent.kind.literal=="SUT")]
   map(self:[action_p.sourcePort.name/]
     ,system:[action_p.destPort.name/]);
   [else]
   connect([action_p.connection.destPort.theComponent.id/]:[action_p.sourcePort.name/]
     ,system:[action_p.destPort.name/]);
   [/if]
  [else]
   // Warning: Test model was incomplete. No code generated for SetupConnection action
  [/if]
 [elseif (action_p.connection.oclIsTypeOf(OclVoid))]
  [if (action_p.connection.destPort.theComponent.oclIsTypeOf(OclVoid))]
   [if (action_p.connection.destPort.theComponent.kind.literal=="SUT")]
   map(self:[action_p.connection.sourcePort.name/]
     ,system:[action_p.connection.destPort.name/]);
   [else]
   connect(self:[action_p.connection.sourcePort.name/]
     ,[action_p.connection.destPort.theComponent.id/]
     :[action_p.connection.destPort.name/]);
   [/if]
  [else]
   // Warning: Test model was incomplete. No code generated for SetupConnection action
  [/if]
 [else]
  // Warning: Test model was incomplete. No code generated for SetupConnection action
 [/if]
[/template]
```

B.1.7 CloseConnectionAction

```
[template public processCloseConnectionAction
(action_p: CloseConnectionAction, context_p:String)]
  [if (action_p.connection.oclIsTypeOf(OclVoid))]
   [if (action_p.connection.destPort.theComponent.kind.literal=="SUT")]
  unmap(self:[action_p.connection.sourcePort.name/]
    ,system:[action_p.connection.destPort.name/]);
   [else]
  disconnect(self:[action_p.connection.sourcePort.name/]
    ,[action_p.connection.destPort.theComponent.id/]
    :[action_p.connection.destPort.name/]);
   [/if]
 [else]
  // Warning: Test model was incomplete. No code generated for CloseConnection action
 [/if]
[/template]
```

B.1.8 DefaultBehaviourDef

```
[template public processDefaultBehaviourDef (defaultDef_p: DefaultBehaviourDef)]
altstep [defaultDef_p.id/]()
[if (defaultDef_p.componentType.oclIsTypeOf(OclVoid))]
runs on [defaultDef_p.componentType.name/][/if]
{
[for (act: TestAction | defaultDef_p.defaultAction)]
[if (act.triggeringEvent.oclIsTypeOf(OclVoid))]
[][processTriggeringEvent(act.triggeringEvent)/]{
 [for (subAct: TestAction | act.testAction)]
```

```
  [processTestAction(subAct, defaultDef_p.id)/]
 [/for]
}
[else]
//WARNING: Triggering event missing for default action
[/if]
[/for]
}
[/template]
```

B.1.9 StopTimerAction

```
[template public processStopTimerAction (action_p: StopTimerAction, context_p:String)]
[if (action_p.timer.oclIsTypeOf(OclVoid))]
[action_p.timer.name/].stop;
[else]
//WARNING: Timer missing for StopTimerAction model element. No code will be generated
[/if]
[/template]
```

B.1.10 StartTimerAction

```
[template public processStartTimerAction (action_p: StartTimerAction, context_p:String)]
[if (action_p.timer.oclIsTypeOf(OclVoid))]
[action_p.timer.name/].start[if (action_p.delay.oclIsTypeOf(OclVoid)
&& action_p.delay > 0)]([action_p.delay_formatted/])[/if];
[else]
//WARNING: Timer missing for StartTimerAction model element. No code will be generated
[/if]
[/template]
```

B.1.11 ValueCheckAction

```
[template public processValueCheckAction (action_p:ValueCheckAction, context_p: String]
 [if (action_p.dataConstraint->size() > 0
 && (action_p.variable.oclIsTypeOf(OclVoid)
  || action_p.testBehaviourInvocationAction.oclIsTypeOf(OclVoid)))]
  [if (action_p.variable.oclIsTypeOf(OclVoid)
  && action_p.variable.name.oclIsTypeOf(OclVoid))]
   [let variable:String = action_p.variable.name/]
  [else]
   [if (action_p.testBehaviourActionInvocation
   .testBehaviourActionDef.oclIsTypeOf(OclVoid)
   && action_p.testBehaviourActionInvocation
   .testBehaviourActionDef.name.oclIsTypeOf(OclVoid))]
   [let variable:String]
   [action_p.testBehaviourActionInvocation.testBehaviourActionDef.name/]
   ([processParams(action_p.testBehaviourActionInvocation)/])[/let]
   [else]
   [let variable:String = "UNSPECIFIED_FUNCTION"/]
   [/if]
  [/if]
  [for (constr:DataConstraint | action_p.dataConstraint)]
  [processConstraint(constr, variable, action_p, context_p)/]
  [/for]
 [else]
  // Warning: Test model was incomplete. No code generated for value check action.
  [if (action_p.dataConstraint->size() <= 0)]
```

```
    //No constraint specified
   [/if]
   [if (!action_p.variable.oclIsTypeOf(OclVoid))]
   //NO variable specified
   [/if]
   [if (!action_p.testBehaviourActionInvocation.oclIsTypeOf(OclVoid))]
   //NO behaviour invocation specified
   [/if]
 [/if]
[/template]
```

B.2 UTML to JUnit Mapping Rules

B.2.1 Testcase

```
[template public processTestcase (testcase_p: Testcase)]
[if (selectedObjects->size()==1)]
 import de.fraunhofer.fokus.testing.web.http.*;
 import de.fraunhofer.fokus.utml.generated.*;

[/if]

/**
* @purpose [if (testcase_p.testObjective.oclIsTypeOf(OclVoid))]
[for (t_obj:TestObjective | testcase_p.testObjective)] [t_obj.id/][/for]
[/if]
* TP version: [if (testspec.oclIsTypeOf(OclVoid))]
[if (testspec.version.oclIsTypeOf(OclVoid))][testspec.version/][/if][/if]
[if (testcase_p.testProcedure.oclIsTypeOf(OclVoid))]
[if (testcase_p.testProcedure.testObjective.oclIsTypeOf(OclVoid))]
  [for (t_obj:TestObjective | testcase_p.testProcedure.testObjective)]
   [for (descElt:DescriptionElement| t_obj.objectiveDescElement)]
    [if (descElt.value<>"")]
*
* [mapKeyword(descElt.name)/]
[printAsComment(descElt.value)/]
   [/if]
  [/for]
 [/for]
[/if]
[/if]
* @desc:
[if (testcase_p.description.oclIsTypeOf(OclVoid))]
[printAsComment(testcase_p.description)/]
[/if]
* Test procedure:
[if (testcase_p.testProcedure.oclIsTypeOf(OclVoid))]
[let cnter:Integer = 1]
[for (t_step:TestStep | testcase_p.testProcedure.testStep)]
[printAsComment text_p=cnter+": "+t_step.content/]
[let cnter:Integer = cnter+1]
[/for]
[/if]
*
*/
public class [testcase_p.name/] extends HttpTestcase {

 public [testcase_p.name/](){
```

```
 super("[testcase_p.name/]"
 ,"This test case has been automatically generated from a UTML test model.
 [testcase_p.description?js_string/]");
}

public void test[testcase_p.name/]() throws Exception{

 [if (testcase_p.testProcedure.oclIsTypeOf(OclVoid))]
 [if (testcase_p.testProcedure.testObjective.oclIsTypeOf(OclVoid))]
 [let t_obj_list:Set(TestObjective) = testcase_p.testProcedure.testObjective]
 [/if]
 [elseif (testcase_p.testObjective.oclIsTypeOf(OclVoid))]
 [let t_obj_list:Set(TestObjective) = testcase_p.testObjective]
 [/if]

 //Test execution
 [if (t_obj_list.oclIsTypeOf(OclVoid))]
  [for (t_obj:TestObjective | t_obj_list)]
  [for (descElt:DescriptionElement| t_obj.objectiveDescElement)]
  [if (descElt.name.toLower()=="pics item")]
   [if (descElt.value.contains(PICS_SEPARATOR))]
    [let sep:String = PICS_SEPARATOR]
   [else]
    [let sep:String = " "]
   [/if]
   [for (pic:String | descElt.value?split(sep))]
    [if (!(pic==""))]
  if (!([pic/]))
  {
   printOut("**** [testcase_p.name/]: Info :
    TC needs [pic.replace("-", "_")/]  to be supported ****");
   stop();
  }
    [/if]
   [/for]
  [/if]
  [/for]
 [/for]
 [/if]

 //Setup configuration[if (testcase_p.testArchitecture.oclIsTypeOf(OclVoid))]
 : [testcase_p.testArchitecture.id/][/if]
 [if (testcase_p.testArchitecture.oclIsTypeOf(OclVoid))]
  [if (testcase_p.testArchitecture.setupFunction->size()>0)]
   [for (setupFunction:TestBehaviourActionDef |
    testcase_p.testArchitecture.setupFunction)]
 [processTestAction(setupFunction, testcase_p.name)/]
   [/for]
  [else]
  [for (connection:Connection | testcase_p.testArchitecture.connections)]
   [processConnection(connection)/]
  [/for]
  [/if]
  [if (testcase_p.testArchitecture.associatedDefault.oclIsTypeOf(OclVoid))]
  [for (default: DefaultBehaviourDef | testcase_p.testArchitecture.associatedDefault)]
 activate([default.id/]());
  [/for]
 [else]
  //WARNING: No configuration for testcase
```

```
[/if]
[/if]

//Preamble
[for (bs:State | testcase_p.beginState)]
 [for (t_act:Action | bs.triggeringActions)]
[t_act.name/]([processDefaultParams(t_act)/]);
 [/for]
[/for]

//Test body
[for (var:VariableDeclaration | testcase_p.variableDeclaration)]
 [processVariableDeclaration(var)/]
[/for]

[if (i_nrOfComps <= 2)]
[for (ta:TestAction | testcase_p.testAction)]
 [processTestAction(ta, testcase_p.name)/]
[/for]
[else]
 //First start passive components
 [for (comp:ComponentInstance | testcase_p
 .passiveComponentInstance)]
  [if (comp.kind.literal <> "SUT")]
f_[testcase_p.name/]_[comp.id/]_behaviour();
  [/if]
 [/for]
 //Then, start active components
 [for (comp:ComponentInstance | testcase_p.activeComponentInstance)]
  [if (comp.kind.literal <> "SUT")]
f_[testcase_p.name/]_[comp.id/]_behaviour());
  [/if]
 [/for]
[/if]
[if (testcase_p.testArchitecture.oclIsTypeOf(OclVoid))]
[if (testcase_p.testArchitecture.teardownFunction->size() > 0)]
//Teardown configuration: [testcase_p.testArchitecture.id/]
[for (teardownFunction:TestBehaviourActionDef |
 testcase_p.testArchitecture.teardownFunction)]
[processTestAction(teardownFunction, testcase_p.name)/]
[/for]
[else]
[for (connection:Connection | testcase_p.testArchitecture.connections)]
 [processDisconnection(connection)/]
[/for]
[/if]
[/if]
//Postamble
[for (es:State | testcase_p.endState)]
 [for (t_act:TestAction | es.triggeringActions)]
[t_act.name/]([processDefaultParams(t_act)/]);
 [/for]
[/for]
}

}//end [testcase_p.name/]
[/template]
```

B.2.2 WaitAction

```
[template public processWaitAction (action_p:WaitAction, context_p:String)]
 printOut("*** [context_p.toUpper()/]:
 start waiting for [action_p.delay_formatted/] ms. ***");
 sleep([action_p.delay_formatted/]);
[/template]
```

B.3 SysML to UTML Mapping

Table B.1: SysML to UTML Mapping

SysML Element	**UTML Equivalent**
Requirements Concepts	
Package in requirements diagram	Test objectives group definition (*TestObjectivesGroupDef*)
Requirement	*TestObjective*
Dependency Relationship	Requirement relationship
Verify Relationship	Requirement relationship
Testcase	*TestObjective*
Architecture Concepts	
Package in block diagram	Test architecture group definition (*TestArchitectureGroupDef*)
Block	Component instance (*ComponentInstance*)
Flowport	Port Instance (*PortInstance*)

B.4 WSDL to UTML Mapping

Table B.2: WSDL to UTML Mapping

WSDL Element	**UTML Equivalent**
Data Concepts	
definitions-/wsdl:definitions	*TestDataModel*

complexType-/wsdl:complex-Type	*MessageTestDataType*
simpleType-/wsdl:simpleType	*MessageTestDataType*
part/wsdl:part	*ParameterDeclaration*, if parent XML node is of type *operation* or *DataTypeField* otherwise
fault/wsdl:fault	*OperationExceptionDef*
operation	*OperationTestDataType*
output-/wsdl:output	*OperationResponseDef*
input/wsdl:input	*ParameterDeclaration*
Architecture Concepts	
definitions-/wsdl:definitions	*TestArchitectureTypesModel*
portType-/wsdl:portType	*PortType*

Appendix C

UTML Model Transformation Examples

C.1 Example of Model Transformation: UTML to TTCN-3

```
/**
* @desc:
*
* @purpose SUPL-2.0-con-110-12-1
* TP version:
*
* Test Case Id :
*
*  SUPL-2.0-con-110
*
* Test Object :
*
*   Client
*
* Test Case Description :
*
*  To test SET correctly actions single session Positioning method
*
* Specification Reference :
*
*  ULP TS 5.2.1, 8, 9
*
* SCR Reference :
*
*  ULP-PRO-C-009-O, ULP-PRO-C-011-M, ULP-PRO-C-012-O,
*  ULP-PRO-C-013-O, ULP-PRO-C-014-O, ULP-PRO-C-015-O, ULP-PRO-C-016-O,
*  ULP-PRO-C-018-O *
*
* Tool :
*
```

```
*  SUPL Client Conformance Test Tool
*
* Test code :
*
*  Validated test code for this test case
*
* Preconditions :
*
*  ( (ics_AGANSSSETassisted_Galileo_SET_initiated
*  AND ics_AGANSSSETbased_Galileo_SET_initiated) OR
*  (ics_AGANSSSETassisted_GLONASS_SET_initiated AND
*       ics_AGANSSSETbased_GLONASS_SET_initiated)) AND
*  (ixit_gANSS.galileo or
*  ixit_gANSS.gloneass)
*  Test Procedure :
*
*  Test 12: A-GANSS Preferred methods [Includes optional features]
*   Note that these test cases only test a single GNSS at one time.
*  Testing of support for multiple simultaneous GNSSs is for further study.
*
*  1. All tests: start a SI Location Session
*  2. The SET sends SUPL START with:
*    SET capabilities parameter consistent with the Positioning technologies
*   supported
*
*    by the SET as declared in the ics
*  3. Send SUPL RESPONSE with: Positioning Method set to the value
*  specified in the table below
*
*   GNSS Positioning Technology set to the value specified in the table below
*
*  4. The SET sends SUPL POS INIT with:
*   SET capabilities parameter consistent with the Positioning
*  technologies supported by the SET
*
*   as declared in the ics
*  6. All tests except Test 4 and Test 5: A SUPL POS session
*   takes place and completes successfully
*
*   using the Positioning Method defined by the test case.
*  Test 12, Case 1:  A-GANSS SET assisted is used.
*  The GANSS used can be one of Galileo or GLONASS depending on
*  the technology supported by the SET
*
*   and declared in ixit_gANSS.
*  8. Test 1, Test 6, Test 7, Test 11 Case 1 and Test 12 Case 1:
*  send SUPL END with: Position set to
*
*   a realistic position for the SET.
*  9. All tests: the SET releases the secure IP connection.
*   Note: Repeat for all  Positioning technologies supported
*  by the SET as declared in the ics
*
*
* Pass-Criteria :
*
*  All tests:
*  1. At step 2 the SET shall send SUPL START with:
*   SET capabilities parameter consistent with the Positioning
*  technologies supported by the SET as declared
*
*  in the ics
*   2. At step 2 the SET shall send SUPL POS INIT with:
*    SET capabilities parameter consistent with the Positioning
*   technologies supported by the SET
*    as declared in the ics
*
*  All tests except Test 4 and Test 5:
*   3. At step 6 a SUPL POS session shall take place and shall complete
*  successfully using the Positioning Method defined by the test case.
*
*  Test 12, Case 1: A-GANSS SET assisted shall be used.
*   The GANSS used can be one of Galileo or GLONASS depending
*  on the technology supported by the SET
*
*  and declared in ixit_gANSS.
*
*
* @desc To test SET correctly actions single session Positioning method -
* Test 12: A-GANSS SET Preferred methods
*
*/
testcase TC_110_12_1()
runs on SUPLComponent system SystemInterfaces {

 //     Local variables and timers
 var
 GNSSPosTechnology
  v_posTechnology := c_GNSSPosTechnologyGalileoGloneASS;

 //     Test execution
 //     check preconditions
 if (not
   (((ics_AGANSSSETassisted_Galileo_SET_initiated and
      ics_AGANSSSETbased_Galileo_SET_initiated) or
     (ics_AGANSSSETassisted_GLONASS_SET_initiated and
      ics_AGANSSSETbased_GLONASS_SET_initiated)) and
    (ixit_gANSS.galileo or ixit_gANSS.gloneass))) {
  log(
   "**** TC_110_12_1: Info : TC needs
   ( (ics_AGANSSSETassisted_Galileo_SET_initiated
   AND ics_AGANSSSETbased_Galileo_SET_initiated) OR
   (ics_AGANSSSETassisted_GLONASS_SET_initiated AND
   ics_AGANSSSETbased_GLONASS_SET_initiated)) AND
   (ixit_gANSS.galileo or ixit_gANSS.gloneass) to be supported ****"
  );
  stop;
 }

 if (ixit_gANSS.galileo) {
  v_posTechnology := c_GNSSPosTechnologyGalileo;
 } else if (ixit_gANSS.gloneass) {
  v_posTechnology := c_GNSSPosTechnologyGloneASS;
 }

 //     Setup configuration: simpleTestArchitecture_no_sim
 activate(ts_DefaultDef());
```

```
//      Preamble
                f_init(staticdefaultGPS);
                f_start_SCC();

//      Test body
//      STEP 1: Start a SI Location Session
showMessage("Please start a new session at the SET");
TGuard.start;
alt {
 //      STEP 2: SET send SUPL START
 [] ulpPort.receive(r_ulpPdu(r_suplStart)) -> value vc_inPdu
 {
  TGuard.stop;
  log(
   "*** TC_110_12_1:  ULP_PDU ( SUPL START )
   message received as expected ***"
  );
 }
   }

//      STEP 2: check the consistents of SET capabilities with
//      the Positioning technology supported
if (f_checkPosCapabilitiesParam_againstSET(vc_inPdu)) {
 setverdict(
  pass,
  "*** TC_110_12_1 Received positioning capabilities
  are consistent with SET capabilities ***"
 );
   } else {
 setverdict(
  fail,
  "*** TC_110_12_1 Received positioning capabilities
  are NOT consistent with SET capabilities ***"
 );
 stop;
}
v_sessionId :=
 valueof(generateCompletedSessionId(v_sessionId,
            slpSessionIdIPv6));
//      STEP 3: send SUPL RESPONSE
ulpPort
.send(m_ulpPdu(m_Version, v_sessionId,
      m_ulpMessageSuplResponse_posTechnology
      (ver2_agnssSETassisted, v_posTechnology)));
TGuard.start;
alt {
 //      STEP 4: SET sends SUPL POS INIT
 [] ulpPort.receive(r_ulpPdu(r_suplPosInit)) ->
    value vc_inPdu {
  TGuard.stop;
  log(
   "*** TC_110_12_1:  ULP_PDU ( SUPL POS INIT )
   message received as expected ***"
  );
 }
}

//      STEP 4: check the consistents of SET capabilities with
//      the Positioning technology supported
```

```
 if (f_checkPosCapabilitiesParam_againstSET(vc_inPdu)) {
  setverdict(
   pass,
   "*** TC_110_12_1 Received positioning capabilities
    are consistent with SET capabilities ***"
  );
 } else {
  setverdict(
   fail,
   "*** TC_110_12_1 Received positioning capabilities
   are NOT consistent with SET capabilities ***"
  );
  stop;
 }

 //      STEP 6: A SUPL POS session takes place and completes
 //      successfully
 f_startAndCompletePosSession(vc_inPdu, v_sessionId,
          m_methodType_msAssisted
          (m_accuracyOpt_omitted));

 //      STEP 8: send SUPL END with Position set to a realistic
 //      position for SET
 sendSuplEndWithPosData(v_sessionId, true);

 //      Postamble
 f_Postamble();
} //      end TC_110_12_1
with {
 extension
 "Description: To test SET correctly actions single session Positioning method"
}
```

Bibliography

[1] Ellen Agerbo and Aino Cornils. How to preserve the benefits of design patterns. *SIGPLAN Not.*, 33(10):134–143, 1998. [cited at p. 43]

[2] Hervé Albin-Amiot and Yann-Gaël Guéhéneuc. Design patterns: A round-trip. In Gilles Ardourel, Michael Haupt, Jose Luis Herrero Agustin, Rainer Ruggaber, and Charles Suscheck, editors, *proceedings of the 11th ECOOP workshop for Ph.D. Students in Object-Oriented Systems*, June 2001. [cited at p. 62]

[3] Ch. Alexander. *The Timeless Way of Building.* Oxford University Press, 1979. [cited at p. 7, 29]

[4] Daniel Amyot, Luigi Logrippo, and Michael Weiss. Generation of test purposes from use case maps. *Comput. Netw.*, 49(5):643–660, 2005. [cited at p. 8]

[5] Jennittra Andrea. Envisioning the next-generation of functional testing tools. *Software, IEEE*, 24(3):58–66, 2007. [cited at p. 210]

[6] L. Apfelbaum and J. Doyle. Model-based testing. In *Proceedings of the 10th international software quality week (SQW97)*, May 1997. [cited at p. 23, 29]

[7] J. Bach. Test automation snake oil. In *Proceedings of the 14th International Conference and Exposition on Testing Computer Software*, 1999. [cited at p. 9]

[8] Janvi Badlaney, Rohit Ghatol, and Romit Jadhwani. An introduction to data-flow testing. Technical report, NC State University, Department of Computer Sciences, Sep 2006. [cited at p. 10]

[9] Stefan Baerisch. Model-driven test-case construction. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 587–590, New York, NY, USA, 2007. ACM. [cited at p. 40, 41]

[10] Paul Baker, Zhen Ru Dai, Jens Grabowski, Oystein Haugen, Ina Schieferdecker, and Clay Williams. *Model-Driven Testing: Using the UML Testing Profile.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007. [cited at p. 35]

[11] Paul Baker, Shiou Loh, and Frank Weil. Model-driven engineering in a large industrial context - motorola case study. In *MoDELS*, pages 476–491, 2005. [cited at p. 34, 35]

[12] Aline Lúcia Baroni, Yann-Gaël Guéhéneuc, and Hervé Albin-Amiot. Design patterns formalization. Technical Report 03/03/INFO, École des Mines de Nantes, June 2003. [cited at p. 61]

[13] Harold J. Batteram, Wim Hellenthal, Willem A. Romijn, Andreas Hoffmann, Axel Rennoch, and Alain Vouffo. Implementation of an open source toolset for ccm components and systems testing. In Roland Groz and Robert M. Hierons, editors, *TestCom*, volume 2978 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2004. [cited at p. 269, 272, 274]

[14] A. Belinfante, L. Frantzen, and C. Schallhart. *Model-Based Testing of Reactive Systems*, chapter Chapter, pages 391–438. Springer, 2005. [cited at p. 209]

[15] Matthias Beyer, Winfried Dulz, and Fenhua Zhen. Automated ttcn-3 test case generation by means of uml sequence diagrams and markov chains. *ats*, 0:102, 2003. [cited at p. 8, 28]

[16] Robert V. Binder. *Testing Object Oriented Systems: Models, Patterns and Tools*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1999. [cited at p. 21, 25, 40, 46, 50, 51]

[17] Sandrine Blazy, Frédéric Gervais, and Régine Laleau. Reuse of specification patterns with the b method. In *ZB*, pages 40–57, 2003. [cited at p. 7]

[18] Marc Born, Ina Schieferdecker, Olaf Kath, and Chiaki Hirai. Combining system development and system test in a model-centric approach. In *RISE*, pages 132–143, 2004. [cited at p. 28]

[19] Jan Bosch. Design patterns & frameworks: On the issue of language support. In *ECOOP Workshops*, pages 133–136. Springer, 1997. [cited at p. 52, 61]

[20] Lionel C. Briand and Yvan Labiche. A uml-based approach to system testing. In *'UML' '01: Proceedings of the 4th International Conference on The Unified Modelling Language, Modelling Languages, Concepts, and Tools*, pages 194–208, London, UK, 2001. Springer-Verlag. [cited at p. 25]

[21] Lionel C. Briand and Yvan Labiche. A UML-based approach to system testing. *Software and System Modelling*, 1(1):10–42, 2002. [cited at p. 8]

[22] Lionel C. Briand and Yvan Labiche. A uml-based approach to system testing. Technical Report Technical Report SCE-01-01, Carleton University, February 2002. [cited at p. 25]

[23] Ed Brinksma, Wolfgang Grieskamp, and Jan Tretmans. 04371 summary – perspectives of model-based testing. In Ed Brinksma, Wolfgang Grieskamp, and Jan Tretmans, editors, *Perspectives of Model-Based Testing*, number 04371 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2005. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany. [cited at p. 27, 28, 36]

[24] Trask Bruce, Paniscotti Dominick, Roman Angel, and Vikram Bhanot. Using model-driven engineering to complement software product line engineering in developing software defined radio components and applications. In *OOPSLA '06:*

Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications, pages 846–853, New York, NY, USA, 2006. ACM. [cited at p. 7]

[25] J. Bruck and K. Hussey. Customizing uml: Which technique is right for you? http://www.eclipse.org/modelling/mdt/uml2/docs/articles/Customizing_UML2_Which-_Technique_is_Right_For_You/article.html. [cited at p. 38, 39]

[26] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture, A System Of Patterns, Volume 1.* Wiley Series in Software Design Patterns, 2001. [cited at p. 29, 30, 43]

[27] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. Past, present, and future trends in software patterns. *IEEE Software*, 24(4):31–37, 2007. [cited at p. 7]

[28] H. Buwalda. Action figures. *Software Testing and Quality Engineering*, pages 42–47, 2003. [cited at p. 140]

[29] H. Buwalda and M. Kasdorp. Getting automated testing under control. *Software Testing and Quality Engineering*, pages 39–44, 1999. [cited at p. 140]

[30] Jens R. Calamé and Jaco van de Pol. Applying Model-based Testing to HTML Rendering Engines – A Case Study. In Kenji Suzuki, Teruo Higashino, Andreas Ulrich, and Toru Hasegawa, editors, *Proceedings of the 20th IFIP TC6/WG6.1 International Conference on Testing of Communicating Systems (TestCom 2008), 7th International Workshop on Formal Approaches to Testing of Software (FATES 2008)*, volume 5047 of *LNCS*, pages 250–265. Springer, 2008. [cited at p. 29]

[31] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Trans. Softw. Eng.*, 4(3):178–187, 1978. [cited at p. 23]

[32] J. M. Clarke. Automated test generation from a behavioral model. In *Proceedings of the 11th international software quality week (SQW98)*, May 1998. [cited at p. 23]

[33] Middleware Company. Model driven development for j2ee utilizing a model driven architecture (mda) approach. productivity analysis. Technical report, Report by the Middleware Company on behalf of Compuware, 2003. [cited at p. 7]

[34] Mirko Conrad. Systematic testing of embedded automotive software - the classification-tree method for embedded systems (ctm/es). In Ed Brinksma, Wolfgang Grieskamp, and Jan Tretmans, editors, *Perspectives of Model-Based Testing*, number 04371 in Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2005. <http://drops.dagstuhl.de/opus/volltexte/2005/325> [date of citation: 2005-01-01]. [cited at p. 6]

[35] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory.* Wiley-Interscience, August 1991. [cited at p. 107]

[36] I. Craggs, M. Sardis, and T. Heuillard. Agedis case studies: Model-based testing in industry. In *Proceedings of the 1st European Conference on Model Driven Software Engineering*, pages 106–117. imbus AG, December 2003. [cited at p. 29]

[37] Zhen Ru Dai, Jens Grabowski, and Helmut Neukirchen. Timedttcn-3 a real-time extension for ttcn-3. In *Testing of Communicating Systems*, pages 407–424. Kluwer, 2002. [cited at p. 286]

[38] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. *Software Engineering, International Conference on*, 0:285, 1999. [cited at p. 26, 27, 29]

[39] David E. Delano and Linda Rising. System test pattern language copyright 1996 ag communication systems corporation permission is granted to make copies for plop '96., 1996. [cited at p. 41]

[40] David E. DeLano and Linda Rising. *Pattern languages of program design 3*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1997. [cited at p. 45]

[41] Arilo C. Dias Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme H. Travassos. A survey on model-based testing approaches: a systematic review. In *WEASELTech '07: Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies*, pages 31–36, New York, NY, USA, 2007. ACM. [cited at p. 28]

[42] Sarolta Dibuz, Tibor Szabó, and Zsolt Torpis. Bcmp performance test with ttcn-3 mobile node emulator. In *TestCom*, pages 50–59, 2004. [cited at p. 275]

[43] George Din. An ims performance benchmark implementation based on the ttcn-3 language. *Int. J. Softw. Tools Technol. Transf.*, 10(4):359–370, 2008. [cited at p. 269, 283]

[44] Hyunsook Do, Gregg Rothermel, and Alex Kinneer. Prioritizing junit test cases: An empirical assessment and cost-benefits analysis. *Empirical Softw. Engg.*, 11(1):33–70, 2006. [cited at p. 263]

[45] J. Dong and Y. Zhao an T. Peng. A review of design pattern mining techniques. *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, 19(6):823–855, 2009. [cited at p. 49, 50]

[46] Jing Dong, Yajing Zhao, and Tu Peng. Architecture and design pattern discovery techniques - a review. In *Software Engineering Research and Practice*, pages 621–627, 2007. [cited at p. 49]

[47] E. Dustin. *Effective Software Testing. 50 Specific Way to Improve Your Testing.* Addison-Wesley, 2003. [cited at p. 41, 261, 262]

[48] Ibrahim K. El-Far and James A. Whittaker. *Encyclopedia of Software Engineering*, chapter Model-based software testing. John Wiley & Sons, Inc., 2002. [cited at p. 22, 23, 27]

[49] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Test case prioritization: A family of empirical studies. *IEEE Trans. Softw. Eng.*, 28(2):159–182, 2002. [cited at p. 263]

[50] Sebastian Elbaum, Gregg Rothermel, Satya Kanduri, and Alexey G. Malishevsky. Selecting a cost-effective test case prioritization technique. *Software Quality Control*, 12(3):185–210, 2004. [cited at p. 263]

[51] M. Frey et al. Etsi draft report: Methods for testing and specification (mts); patterns for test development (ptd). Technical report, European Telecommunications Standards Institute (ETSI), 2004. [cited at p. 47, 269, 272, 274, 275, 283, 284, 285]

[52] Li Feng and Sheng Zhuang. Action-driven automation test framework for graphical user interface (gui) software testing. *Autotestcon, 2007 IEEE*, pages 22–27, Sept. 2007. [cited at p. 140]

[53] Alain-Georges Vouffo Feudjio. Model-driven functional test engineering for service centric systems. In *Proceedings of The 5th International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TridentCom 2009)*, 2009. [cited at p. 253]

[54] D.G. Firesmith. Pattern language for testing object-oriented software. *Object Magazin*, 5(9):42–45, 1996. [cited at p. 40]

[55] Marcus Fontoura and Carlos José de Lucena. Extending UML to improve the represenation of design patterns. *Journal of Object Oriented Programming*, 13(11):12–19, March 2001. [cited at p. 52]

[56] Rüdiger Foos, Christian Bunse, Hagen Höpfner, and Torsten Zimmermann. Tml: an xml-based test modelling language. *SIGSOFT Softw. Eng. Notes*, 33(2):1–6, 2008. [cited at p. 39]

[57] ETSI ES 202 553: Methods for Testing and Specification (MTS). Tplan: A notation for expressing test purposes. Technical report, European Telecommunications Standards Institute, Sophia Antipolis, 2007. [cited at p. 262, 264]

[58] Methods for Testing and Specification (MTS). The testing and test control notation version 3; part1: Ttcn-3 core language. Technical report, European Telecommunications Standards Institute (ETSI), 2003. [cited at p. 16, 51, 143, 152, 156, 157, 262]

[59] Robert B. France, Dae-Kyoo Kim, Sudipto Ghosh, and Eunjee Song. A uml-based pattern specification technique. *IEEE Trans. Softw. Eng.*, 30(3):193–206, 2004. [cited at p. 7]

[60] Elisabeth Freeman, Eric Freeman, Bert Bates, and Kathy Sierra. *Head First Design Patterns*. O'Reilly, October 2004. [cited at p. 279, 280]

[61] S. Fujiwara, G. V. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17:591–603, 1991. [cited at p. 23]

[62] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1994. [cited at p. 7, 30, 43, 283]

[63] Shudi Gao, C. Michael Sperberg-McQueen, Henry S. Thompson, Noah Mendelsohn, David Beech, and Murray Maloney. W3c xml schema definition language (xsd) 1.1 part 1: Structures. World Wide Web Consortium, Working Draft WD-xmlschema11-1-20080620, June 2008. [cited at p. 116]

[64] Adam Geras, James Miller, Michael R. Smith, and James Love. A survey of test notations and tools for customer testing. In *XP*, pages 109–117, 2005. [cited at p. 17, 33]

[65] J. Grabowski. Sdl and msc based test case generation: An overall view of the samstag method, 1994. [cited at p. 28, 69]

[66] J. Grabowski, D. Hogrefe, and R. Nahm. Test case generation with test purpose specification by mscs, 1993. [cited at p. 28, 69]

[67] Dorothy Graham, Erik van Veenendaal, Isabel Evans, and Rex Black. *Foundations of Software Testing: ISTQB Certification.* Int. Thomson Business Press, 2006. [cited at p. 16, 262]

[68] Matthias Grochtmann and Klaus Grimm. Classification trees for partition testing. *Softw. Test., Verif. Reliab.*, 3(2):63–82, 1993. [cited at p. 279]

[69] Juergen Grossmann, Ines Fey, Alexander Krupp, Mirko Conrad, Christian Wewetzer, and Wolfgang Mueller. Testml - a test exchange language for model-based testing of embedded software. In Manfred Broy, Ingolf H. Krger, and Michael Meisinger, editors, *ASWSD*, volume 4922 of *Lecture Notes in Computer Science*, pages 98–117. Springer, 2006. [cited at p. 40]

[70] Object Management Group. Unified modelling language: Testing profile, finalized specification. Technical report, Object Management Group, April 2004. [cited at p. 8, 35, 270]

[71] Object Management Group. *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.1*, 2005. [cited at p. 204]

[72] Object Management Group. *Meta Object Facility (MOF) Core Specification Version 2.0*, 2006. [cited at p. 38]

[73] Object Management Group. *MOF Model to Text Transformation Language, Version 1.0*, 2008. [cited at p. 204]

[74] Object Management Group. *OMG Unified Modelling Language (OMG UML), Superstructure*, 2009. [cited at p. 37]

[75] Alain Le Guennec, Gerson Sunyé, and Jean marc Jézéquel. Precise modelling of design patterns. In *In Proceedings of UML00*, pages 482–496. Springer Verlag, 2000. [cited at p. 52]

[76] Mary Jean Harrold and Gregg Rothermel. Performing data flow testing on classes. pages 154–163. ACM Press, 1994. [cited at p. 10]

[77] A. Hartman and K. Nagin. The agedis tools for model based testing. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, volume 29, pages 129–132, New York, NY, USA, July 2004. ACM Press. [cited at p. 8, 27, 28]

[78] Görel Hedin. Language support for design patterns using attribute extension. In *ECOOP '97: Proceedings of the Workshops on Object-Oriented Technology*, pages 137–140, London, UK, 1998. Springer-Verlag. [cited at p. 52, 62]

[79] Mats P. E. Heimdahl. Model-based testing: Challenges ahead. In *COMPSAC '05: Proceedings of the 29th Annual International Computer Software and Applications Conference*, page 330, Washington, DC, USA, 2005. IEEE Computer Society. [cited at p. 26, 27]

[80] Brian Henderson-Sellers. Uml - the good, the bad or the ugly? perspectives from a panel of experts. *Software and System Modelling*, 4(1):4–13, 2005. [cited at p. 36]

[81] William E. Howden. Software test selection patterns and elusive bugs. In *COMPSAC '05: Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC'05) Volume 1*, pages 25–32, Washington, DC, USA, 2005. IEEE Computer Society. [cited at p. 40]

[82] Antti Huima. Implementing conformiq qtronic. In *TestCom/FATES*, pages 1–12, 2007. [cited at p. 265]

[83] IEEE. Draft ieee standard for software and system test documentation (revision of ieee 829-1998). Technical report, IEEE, 2008. [cited at p. 16, 19, 48, 263]

[84] Specialist Interest Group in Software Testing (BCS SIGIST). Standard for software component testing, working draft 3.4. Technical report, British Computer Society (BCS), apr 2001. [cited at p. 277]

[85] ISO/IEC. Iso/iec standard no. 9126: Software engineering product quality; parts 14. Technical report, Organization for Standardization (ISO) / International Electrotechnical Commission (IEC), Geneva, Switzerland, 2001-2004. [cited at p. 31, 44]

[86] ISTQB. Standard glossary of terms used in software testing. Technical report, International Software Testing Qualification Board ISTQB, 2006. [cited at p. 16, 19]

[87] ITU-T. Osi conformance testing methodology framework(ctmf), recommendation x-290. Technical report, ITU-T, 1995. [cited at p. 266, 273]

[88] Ivar Jacobson, M. Griss, and P. Jonsson. *Software Reuse: Architecture, Process and Organization for Business Success.* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1997. [cited at p. 30]

[89] A. Z. Javed, P. A. Strooper, and G. N. Watson. Automated generation of test cases using model-driven architecture. In *AST '07: Proceedings of the Second International Workshop on Automation of Software Test*, page 3, Washington, DC, USA, 2007. IEEE Computer Society. [cited at p. 28]

[90] Jean-Marc Jézéquel, Michel Train, and Christine Mingins. *Design Patterns and Contracts.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. [cited at p. 283]

[91] Paul C. Jorgensen. *Software Testing: A Craftsman's Approach.* CRC Press, Inc., Boca Raton, FL, USA, 1995. [cited at p. 22]

[92] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. Atl: A model transformation tool. *Science of Computer Programming*, 72(1-2):31–39, June 2008. [cited at p. 204]

[93] Cem Kaner. Architectures of test automation, 2000. [cited at p. 9]

[94] S.-K. Kim and D. Carrington. Using integrated metamodelling to define oo design patterns with object-z and uml. *Software Engineering Conference, 2004. 11th Asia-Pacific*, pages 257–264, Nov.-3 Dec. 2004. [cited at p. 52]

[95] S. Kleene. *Representation of Events in Nerve Nets and Finite Automata*, pages 3–42. Princeton University Press, Princeton, N.J., 1956. [cited at p. 22]

[96] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. [cited at p. 6]

[97] Thomas Kühne. What is a model? In Jean Bezivin and Reiko Heckel, editors, *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2005. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany. [cited at p. 21]

[98] Dae kyoo Kim, Robert France, Sudipto Ghosh, and Eunjee Song. A uml-based metamodelling language to specify design patterns. In *In Proceedings of WiSME, UML Conference*, 2003. [cited at p. 52]

[99] R. Lai. How could research on testing of communicating systems become more industrially relevant? In *Selected proceedings of the IFIP TC6 9th international workshop on Testing of communicating systems*, pages 3–13, London, UK, UK, 1996. Chapman & Hall, Ltd. [cited at p. 28, 29]

[100] R. Lai. A survey of communication protocol testing. *Journal of Systems and Software*, 62(1):21 – 46, 2002. [cited at p. 27, 28]

[101] Richard Lai and Wilfred Leung. Industrial and academic protocol testing: the gap and the means of convergence. *Computer Networks and ISDN Systems*, 27(4):537–547, 1995. [cited at p. 28]

[102] Chang Liu and Debra J. Richardson. Using application states in software testing. *Software Engineering, International Conference on*, 0:776, 2000. [cited at p. 23]

[103] Jochen Ludewig. Models in software engineering an introduction. *Software and Systems Modeling*, 2(1):5–14, March 2003. [cited at p. 21]

[104] Jeffrey K. H. Mak, Clifford S. T. Choy, and Daniel P. K. Lun. Precise modelling of design patterns in uml. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 252–261. IEEE Computer Society, 2004. [cited at p. 52]

[105] David Mapelsden, John Hosking, and John Grundy. Design pattern modelling and instantiation using dpml. In James Noble and John Potter, editors, *Fortieth International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, volume 10 of *CRPIT*, pages 3–11, Sydney, Australia, 2002. ACS. [cited at p. 62]

[106] John D. McGregor and David A. Sykes. *A Practical Guide to Testing Object-oriented Software (Object Technology Series)*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2001. [cited at p. 46]

[107] G. H. Mealy. A method for synthesizing sequential circuits. *Bell Systems Technical Journal*, 1955. [cited at p. 22]

[108] Gerard Meszaros. *XUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, 2007. [cited at p. 9, 40, 46, 262, 281]

[109] MODELWARE. Modelware d5.3-1 industrial roi, assessment, and feedback. master document. revision 2.2. Technical report, 2006. [cited at p. 7]

[110] MODELWARE. Modelware d5.3-4 france telecom roi, assessment, and feedback. revision 1.1. Technical report, 2006. [cited at p. 7]

[111] Parastoo Mohagheghi and Vegard Dehlen. Where is the proof? - a review of experiences from applying mde in industry. In *ECMDA-FA '08: Proceedings of the 4th European conference on Model Driven Architecture*, pages 432–443, Berlin, Heidelberg, 2008. Springer-Verlag. [cited at p. 6, 209]

[112] E. F. Moore. *Gedanken experiments on sequential machines*, pages 129–153. Princeton University Press, Princeton, N.J., 1956. [cited at p. 22]

[113] Mozilla.org. Testopia. Web Page, 2009. [cited at p. 266]

[114] Arilo Dias Neto, Rajesh Subramanyan, Marlon Vieira, Guilherme Horta Travassos, and Forrest Shull. Improving evidence about software technologies: A look at model-based testing. *IEEE Software*, 25(3):10–13, 2008. [cited at p. 29]

[115] Helmut Neukirchen. *Languages, Tools and Patterns for the Specification of Distributed Real-Time Tests.* PhD thesis, Dissertation, Universität Göttingen, November 2004 (electronically published on http://webdoc.sub.gwdg.de/diss/2004/neukirchen/index.html and archived on http://deposit.ddb.de/cgi-bin/dokserv?idn=974026611 . Persistent Identifier: urn:nbn:de:gbv:7-webdoc-300-2), November 2004. [cited at p. 9]

[116] Helmut Neukirchen and Martin Bisanz. Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites. In *Proceedings of the 19th IFIP International Conference on Testing of Communicating Systems and 7th International Workshop on Formal Approaches to Testing of Software (TestCom/FATES 2007)*, pages 228–243. Springer, Heidelberg, June 2007. [cited at p. 50, 276]

[117] *Model Driven Architecture (MDA)*, Juli 2001. [cited at p. 6]

[118] S Pickin. *Test des Composants Logiciels pour les Telecommunications.* PhD thesis, Universite de Rennes, France, 2003. [cited at p. 39, 71]

[119] S. Pickin, C. Jard, T. Heuillard, J.M. Jezequel, and P. Defray. A uml-integrated test desciption language for component testing. In A. Evans, R. France, A. Moreira, and B. Rumpe, editors, *Practical UML-Based Rigorous Development Methods*, volume P7 of *Lecture Notes in Informatics (GI Series)*. Kollen-Druck + Verlag, 2001. [cited at p. 39, 70, 71]

[120] Simon Pickin and Jean-Marc Jezequel. Using uml sequence diagrams as the basis for a formal test description language. In *Integrated Formal Methods*, pages 481–500. Springer, 2004. [cited at p. 35, 36, 39, 71]

[121] S. Pietsch and B. Stanca-Kaposta. Model-based testing with utp and ttcn-3 and its application to hl7. Technical report, Testing Technologies, Conquest, Potsdam, Germany, 2008. [cited at p. 22]

[122] Andrej Pietschker. Automating test automation. *Int. J. Softw. Tools Technol. Transf.*, 10(4):291–295, 2008. [cited at p. 212]

[123] Andrej Pietschker. Automating test automation. *Int. J. Softw. Tools Technol. Transf.*, 10(4):291–295, 2008. [cited at p. 275]

[124] L. Prechelt, B. Unger, W. F. Tichy, P. Brössler, and L. G. Votta. A controlled experiment in maintenance comparing design patterns to simpler solutions. *IEEE Trans. Softw. Eng.*, 27(12):1134–1144, 2001. [cited at p. 61]

[125] Wolfgang Prenninger and Alexander Pretschner. Abstractions for model-based testing. *Electronic Notes in Theoretical Computer Science*, 116:59 – 71, 2005. Proceedings of the International Workshop on Test and Analysis of Component Based Systems (TACoS 2004). [cited at p. 27, 68, 88, 111, 145]

[126] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner. One evaluation of model-based testing and its automation. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 392–401, New York, NY, USA, 2005. ACM. [cited at p. 29]

[127] A. Pretschner, O. Slotosch, E. Aiglstorfer, and S. Kriebel. Model-based testing for real: The inhouse card case study. *Int. J. Softw. Tools Technol. Transf.*, 5(2):140–157, 2004. [cited at p. 29]

[128] Nat Pryce. Test data builders: an alternative to the object mother pattern. http://www.natpryce.com/articles/000714.html, aug 2007. [cited at p. 279]

[129] Bo Qu, Changhai Nie, Baowen Xu, and Xiaofang Zhang. Test case prioritization for black box testing. *Computer Software and Applications Conference, Annual International*, 1:465–474, 2007. [cited at p. 264]

[130] Muthu Ramachandran. Testing software components using boundary value analysis. *EUROMICRO Conference*, 0:94, 2003. [cited at p. 277]

[131] I. Rauf, A. Nadeem, and M. Khokhar. Formalizing object oriented design patterns with object-z. *Multitopic Conference, 2006. INMIC '06. IEEE*, pages 269–274, Dec. 2006. [cited at p. 52]

[132] Stuart C. Reid. An empirical analysis of equivalence partitioning, boundary value analysis and random testing. *Software Metrics, IEEE International Symposium on*, 0:64, 1997. [cited at p. 115]

[133] Debra J. Richardson, Owen T. O'Malley, and C. Tittle. Approaches to specification-based testing. In *Symposium on Testing, Analysis, and Verification*, pages 86–96, 1989. [cited at p. 17]

[134] H. Robinson. Obstacles and opportunities for model-based testing in an industrial software environment. In *Proceedings of the 1st European Conference on Model Driven Software Engineering*, pages 118–127. imbus AG, December 2003. [cited at p. 27, 28, 209]

[135] Everett M. Rogers. *Diffusion of Innovations, 5th Edition*. Free Press, 5 edition, August 2003. [cited at p. 257]

[136] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. Rfc3261 - sip: Session initiation protocol. Technical report, IETF, 2002. [cited at p. 226]

[137] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. Test case prioritization: An empirical study. In *ICSM '99: Proceedings of the IEEE International Conference on Software Maintenance*, page 179, Washington, DC, USA, 1999. IEEE Computer Society. [cited at p. 263]

[138] Gregg Rothermel, Roland J. Untch, and Chengyun Chu. Prioritizing test cases for regression testing. *IEEE Trans. Softw. Eng.*, 27(10):929–948, 2001. [cited at p. 263]

[139] Ekkart Rudolph, Ina Schieferdecker, and Jens Grabowski. Hypermsc - a graphical representation of ttcn. In *SAM*, pages 76–, 2000. [cited at p. 69]

[140] M. Al Saad, N. Kamenzky, and J. Schiller. *Model Driven Engineering Languages and Systems*, chapter Visual ScatterUnit: A Visual Model-Driven Testing Framework of Wireless Sensor Networks Applications, pages 751–765. Springer, 2008. [cited at p. 40]

[141] Philip Samuel and Rajib Mall. Boundary value testing based on uml models. *Asian Test Symposium*, 0:94–99, 2005. [cited at p. 277]

[142] Pedro Santos-Neto, Rodolfo Resende, and Clarindo Pádua. Requirements for information systems model-based testing. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 1409–1415, New York, NY, USA, 2007. ACM. [cited at p. 8]

[143] Ina Schieferdecker and Jens Grabowski. The graphical format of ttcn-3 in the context of msc and uml. In *Proceedings of the 3rd Workshop of the SDL Forum Society on SDL and MSC (SAM'2002), Aberystwyth (UK*, pages 2–6. Springer Verlag, 2002. [cited at p. 69]

[144] Stephan Schulz. Test suite development with ttcn-3 libraries. *Int. J. Softw. Tools Technol. Transf.*, 10(4):327–336, 2008. [cited at p. 260, 269]

[145] Stephan Schulz, Anthony Wiles, and Steve Randall. Tplan-a notation for expressing test purposes. In Alexandre Petrenko, Margus Veanes, Jan Tretmans, and Wolfgang Grieskamp, editors, *TestCom/FATES*, volume 4581 of *Lecture Notes in Computer Science*, pages 292–304. Springer, 2007. [cited at p. 262, 264]

[146] Eclipse Open source Project. The eclipse modelling project. http://www.eclipse.org/modelling, 2008. [cited at p. 55]

[147] Eclipse Open source Project. The eclipse project. http://www.eclipse.org, 2008. [cited at p. 55]

[148] TOPCASED Open source Project. Toolkit in open source for critical applications & systems development. http://topcased.gforge.enseeiht.fr/, 2008. [cited at p. 213]

[149] Hema Srikanth. Requirements-based test case prioritization. In *Doctoral Symposium in International Conference of Software Engineering*, page 27695, 2005. [cited at p. 263]

[150] Hema Srikanth and Laurie Williams. On the economics of requirements-based test case prioritization. In *EDSER '05: Proceedings of the seventh international workshop on Economics-driven software engineering research*, pages 1–3, New York, NY, USA, 2005. ACM. [cited at p. 263, 264]

[151] Praveen Ranjan Srivastva, Krishan Kumar, and G Raghurama. Test case prioritization based on requirements and risk factors. *SIGSOFT Softw. Eng. Notes*, 33(4):1–5, 2008. [cited at p. 264]

[152] Gerson Sunyé, Alain Le Guennec, and Jean marc Jézéquel. Design pattern application in uml. In *In Proceedings of the 14 th European conference on Object Oriented programming, Springer LNCS 1850*, pages 44–62. Lecture, 2000. [cited at p. 7]

[153] Arturo H. Torres-Zenteno, Mariá J. Escalona, Manuel M., and Javier J. Gutiérrez. A mda-based testing - a comparative study. In Boris Shishkov, Jose Cordeïro, and Alpesh Ranchordas, editors, *ICSOFT (1)*, pages 269–274. INSTICC Press, 2009. [cited at p. 209]

[154] G. J. Tretmans and H. Brinksma. Torx: Automated model-based testing. In A. Hartman and K. Dussa-Ziegler, editors, *First European Conference on Model-Driven Software Engineering, Nuremberg, Germany*, pages 31–43, December 2003. [cited at p. 29]

[155] Edward R. Tufte. *The Visual Display of Quantitative Information.* Graphics Press, May 2001. [cited at p. 69]

[156] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing. In *A Taxonomy of model-based testing*, number 04/2006 in Working paper series. University of Waikato, Department of Computer Science, 2006. [cited at p. 8, 22, 209]

[157] Mark Utting and Bruno Legeard. *Practical Model-Based Testing. A Tools Approach.* Elsevier, 2006. [cited at p. 5, 6, 21, 22]

[158] Alain Vouffo-Feudjio and Ina Schieferdecker. Test patterns with ttcn-3. In Jens Grabowski and Brian Nielsen, editors, *FATES*, volume 3395 of *Lecture Notes in Computer Science*, pages 170–179. Springer, 2004. [cited at p. 9, 41]

[159] Alain-G. Vouffo Feudjio and Achille Kingni Nentedem. Mdtester user and installation guide. http://www.fokus.fraunhofer.de/distrib/motion/utml/MDTesterUserGuide.html, March 2009. [cited at p. 216]

[160] Ingo Weisemoeller and Andy Schuerr. A comparison of standard compliant ways to define domain specific languages. In *Models in Software Engineering*, Lecture Notes in Computer Sciences, pages 47–58. Springer, 2008. [cited at p. xiv, 62, 63]

[161] J. Zander, Zhen Ru Dai, I. Schieferdecker, and G. Din. From u2tp models to executable tests with ttcn-3. *Proceedings of Second European Workshop On Model Driven Architecture (MDA) EWMDA*, 2004. [cited at p. 23, 28, 35]

[162] J. Zander-Nowicka, A. Marrero Perez, I. Schieferdecker, and Z.R. Dai. Test design patterns for embedded systems. 2007. [cited at p. 9]

[163] Benjamin Zeiß, Helmut Neukirchen, Jens Grabowski, Dominic Evans, and Paul Baker. TRex - An Open-Source Tool for Quality Assurance of TTCN-3 Test Suites. In *Proceedings of CONQUEST 2006 – 9th International Conference on Quality Engineering in Software Technology, September 27-29, Berlin, Germany.* dpunkt.Verlag, Heidelberg, September 2006. [cited at p. 238]

[164] Benjamin Zeiß, Diana Vega, Ina Schieferdecker, Helmut Neukirchen, and Jens Grabowski. Applying the ISO 9126 Quality Model to Test Specifications – Exemplified for TTCN-3 Test Specifications. In *Software Engineering 2007 (SE 2007). Lecture Notes in Informatics (LNI) 105. Copyright Gesellschaft für Informatik*, pages 231–242. Köllen Verlag, Bonn, March 2007. [cited at p. 44]

[165] Xiaofang Zhang, Changhai Nie, Baowen Xu, and Bo Qu. Test case prioritization based on varying testing requirement priorities and test case costs. In *QSIC*, pages 15–24, 2007. [cited at p. 264]

Index