A Tool–based Methodology for Model–driven System Testing of Service–centric Systems

Dissertation

by

Michael Felderer

submitted to the Faculty of Mathematics, Computer Science and Physics of the University of Innsbruck

in partial fulfillment of the requirements for the degree of doctor of technical sciences (Dr. techn.)

supervisor:

Prof. Dr. Ruth Breu

Institute of Computer Science

University of Innsbruck

April 2011

"Quality is not an act, it is a habit."

Aristotle

Abstract

The number and complexity of service–centric systems for implementing flexible and distributed IT–based processes is steadily increasing. Many application scenarios such as the cross–linking of traffic participants have demonstrated the power of service–centric systems. Elaborated standards, technologies, and frameworks for building service–centric systems have been developed, but system testing aspects have been neglected so far.

Appropriate system testing methods have to consider general issues such as automation or test–drivenness and specific issues of service–centric systems, e.g., the integration of various service technologies, the evolution of such systems, the strong linkage between domain and technology, the lack of service code observability, and the importance of non–functional properties such as security.

Model–driven testing approaches are particularly suitable for system testing of service–centric systems because they provide an abstract technology and implementation independent view on tests, are flexible in considering changes, and support high–quality test design in a very early phase of system development even before or simultaneous with system modeling. This raises the agile practice of test–driven development to the model level.

In this thesis we define a novel model–driven system testing methodology for service–centric systems called Telling TestStories. Telling TestStories is based on separated requirements, system, and test models validated by consistency, completeness, and coverage checks. Additionally, our approach guarantees full traceability between the requirements model, the system model, the test model, and the executable services of the system which is crucial for efficient test evaluation and test evolution management. As test models integrate test data in tabular form, our approach is model–driven and tabular supporting test design by domain experts. The methodology comes with a tool implementation and is applied in two case studies from the telecommunication and the home networking domain. Besides functional requirements testing, we employ the methodology on security requirements testing and regression testing based on evolving models.

Acknowledgements

First of all I want to thank my supervisor Prof. Dr. Ruth Breu. She gave me the opportunity to work in her research group and raised my interest in the topic of model–driven testing. She provided me a very motivating working environment within challenging research projects and a great team.

Most parts of this thesis have been developed as part of the research project Telling TestStories supported by the Trans-IT and the Softmethod GmbH. Frank Fiedler and Felix Schupp of Softmethod provided a suitable industrial case study to apply the concepts developed in this thesis.

Additionally, I have been involved in the MATE, Secure Change, and QE LaB projects which resulted in the chapters on security testing and test model evolution.

Many people supported me in writing this thesis. Dr. Joanna Chimiak–Opoka provided many helpful ideas and supported my in writing my first research papers. With Berthold Agreiter I had many fruitful discussions on security testing and test model evolution. Philipp Zech, Lukas Aichbauer, Alexander Catulli, and Michael Catulli supported me to implement a tool for the Telling TestStories methodology. Dr. Michael Breu provided helpful input for the design of the Telling TestStories tool. Andrea Jungmann and Thomas Schrettl provided a perfect infrastructure for minimizing the administrative and technical overhead unavoidable in research projects.

Generally, I want to thank all my colleagues of the Quality Engineering research group at the University of Innsbruck who have provided a creative environment for research. Thanks a lot – without all of you, this thesis would not have been possible!

I also want to thank all my long–standing friends, my sisters Esther and Karoline, and my girlfriend Simone for their mental support.

Finally and immensely, I am indebted to my parents Peter and Theresia who always believed in me and always supported my manifold interests – Thank you so much!

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

The beginning is the most important part of the work.

Plato

This chapter provides an introduction to this thesis. Based on the motivation for the problems addressed in this thesis in Section 1.1, we formulate a problem statement in Section 1.2. In Section 1.3 we discuss solutions to the stated problems and the contribution of this thesis. The chapter closes with an overview of the structure of the thesis in Section 1.4.

1.1 Motivation

Software testing is one of the core fields of software engineering and is the essential technique for quality assurance in industrial software development. Studies indicate that more than 50% of the cost of software development is devoted to testing [Har00, Bin99]. Surveys like a NIST study in 2002 [NIS02] reveal that the potential for improvements in the testing process is still enormous and can reach up to 30% of the costs of the testing process.

The potential for improvements is even greater for new technologies that have recently been adopted by the industry such as Service–oriented architecture (SOA) [EHH+08].

Service–oriented architectures, or more generally service–centric systems in most cases are *dynamically evolving systems*. For instance, in a networked health care scenario this means that information exchange will start with a number of hospitals plus practitioners and will be successively extended by new stakeholder instances, stakeholder types (e.g., pharmacies, laboratories), services, and workflows. Due to the open nature of these systems complex quality properties like *security and privacy* of processed information play a major role for design and operation. In the health care scenario, the correctness and integrity of patient–related data is

of uttermost importance for the safety of human lives. In addition, many security requirements are imposed by legal regulations. In the health care scenario this includes complex access rules to patient related data and the ownership of the patient with respect to his/her data which have to be tested.

While major international efforts in industry and academia so far have focused on the development of standards, technologies and frameworks for realizing service–centric systems only a minority of approaches deal with testing aspects [CDP06]. Testing has been investigated for other types of systems like object oriented systems intensively [Bin99] but testing service–centric systems is a novel area of research.

Especially testing the overall service–centric system and their acceptance is of high importance. Novel system testing methods are therefore needed that consider general issues such as automatization or test–drivenness and specific issues of service–centric systems, such as the integration of various service technologies, the evolution of such systems, the strong linkage between domain and technology, the lack of service code observability, and the importance of non–functional properties such as security.

Acceptance testing is a means to integrate the knowledge and experience of the end user into the software development process and to clarify requirements [RTDP+09]. It therefore becomes more and more important due to the increasing dependency of everyday life on computer–aided support. Acceptance testing or system testing for a long time has been primarily performed manually, e.g., by the owner or user of a system under test (SUT). However, in the era of incremental software development non–automated tests are expensive and error–prone. Available data indicates that on average 85% of the defects are estimated to come from inadequate requirements [WRH+00] and that debugging is up to the factor 100 more expensive after deployment than in the early design and implementation phase of a system [Boe81]. In the recent years some testing tools and techniques have been developed to overcome this situation and to support automatic and early system testing. These approaches even improve the requirements elicitation and system design itself, and integrate the design of test cases on an abstract, business–oriented level.

The *Framework for Integrated Test* (FIT) [Mug05], introduced by Ward Cunningham in 2002, is a framework to define acceptance tests in the form of tables. A test designer writing a FIT table focuses exclusively on defining input data and expected results. FIT then presents test results by simply highlighting places where the subjects behavior deviates from that expected. Specific test drivers, so called fixtures, are applied for automatically executing the tabular test definitions and to interpret the business concepts used in the tabular tests by linking them to the system under test. FIT therefore improves the communication between developers and analysts. Additionally, it allows the latter to specify test cases in an easy way. FIT supports test–driven development and is widely used in agile software development [BA04].

Model–driven testing uses models for test modeling and for generating test cases. In the last few years model-driven testing has received a great deal of attention [GNRS09] and has started to find use in mainstream software development projects. Just as the analysis and design of the functional code in working software applications can be significantly enhanced by the use of formal, yet human-friendly, models so can the analysis, design, and execution of tests. In fact, in the early stages of system specification the two go hand–in–hand, because the functionality of the tests as well as the application code are both driven by the required system behavior. Model–driven testing supports the abstract, technology–independent design of tests in an early phase of the software development process. It improves traceability and maintenance of tests and according to recent results also the quality of tests [RBGW10].

Both, tabular test definition languages and model–driven testing are established system testing methods. The combination of these methods by abstracting tabular test definitions to the model level has not been considered yet but is promising. An integrated model–driven tabular test definition approach combines the benefits of both approaches. Additionally, the approach is suitable for system testing of service–centric systems because it provides an abstract technology and implementation independent view on tests, is flexible in considering changes, supports high–quality test design in a very early phase of system development, is suitable for security testing, and is domain–oriented supporting test design by customers and system analysts.

1.2 Problem Statement

Motivated by the previous section, the problem considered in this thesis is the combination of tabular test definitions and model–driven testing to a novel tool–based methodology for model–driven system testing of service–centric systems. Although this integration results in a methodology combining the benefits of tabular test definition and model–driven testing listed in the previous section, it has not been considered so far.

Besides the main research task to develop and evaluate the model–driven system testing methodology itself, our approach has to address the main issues for system testing of service centric–systems mentioned above.

Based on that, the main research challenges tackled in this thesis are as follows:

RC1 – *Development and application of a tool–based system testing methodology for service–centric systems based on model–driven testing and tabular test design.* This contains the definition of system and testing artifacts, their domain–specific languages, their relationship, and their integration into a testing process, a tool implementation for it, and its evaluation by case studies.

RC2 – *Validation of test models.* This contains the definition and evaluation of validation rules, coverage criteria and arbitrations for test models and its application to case studies.

RC3 – *Security requirements testing for service–centric systems.* This contains the application of model–driven system testing as proposed in RC1 in the context of security requirements testing and its evaluation by a case study.

RC4 – *Evolution of service–centric systems and its propagation to tests.* This contains the general state–based management of system and test changes, its integration into a testing process, the generation of regression test suites based on the system and test states plus its evaluation by a case study.

1.3 Problem Solution and Contribution

In this thesis we contribute to the open problem of combining tabular test definitions and model–driven testing to gain a novel system testing methodology applicable for service–centric systems. We solve this problem by representing tests in separate models and integrate tabular test data definition into this representation. The requirements and the system design are also provided as models used to validate them against the test model and to represent the test results on the model level.

In this respect our approach continues the line of tabular test definition languages and abstracts it to the model level. In Figure 1.1 the development of system testing approaches is shown.

In the past, acceptance or system testing has been implemented ad–hoc based on textual requirements and test definitions. Today, approaches such as FIT provide framework support for the tabular definition and execution of test cases. In our approach, called *Telling TestStories*, tests are represented as separate test models which integrate behavioral models and test data tables. Our test models are designed in such a way that they can be validated against the requirements model and the system model. Our approach can be applied in a test–driven way, i.e., the test models are defined before the behavioral artifacts of the system, and the test results are visualized in the test model due to traceability between all artifacts.

Our methodology is based on a testing process, which is test–driven, validates the models, directly transforms test models to executable test code, and integrates the test result into the test models. Tests in our respect are typically modeled manually by domain experts. But we support this process by model validation and the generation of tests by model transformations.

The methodology is tool–based. We have applied the Unified Modeling Language (UML) [OMG07b, OMG07a] and its profiling mechanism as modeling language because there are several UML tools available that can be customized for our

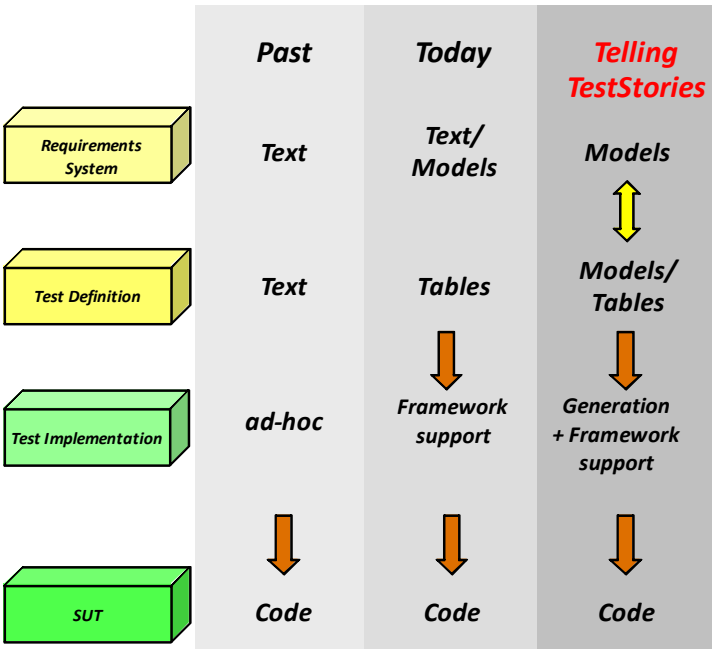


Figure 1.1: System Testing Approaches

needs. We have implemented our models with UML2Tools [uml], a modeling tool for UML based on the Eclipse platform [ecl]. Our methodology is not limited to UML as representation language but we have used it to optimize the tool support and to keep the system model scalable. We consider test stories as partial specification of the system workflow, therefore we represent test stories as activity diagrams. Additionally, sequence diagrams for representing abstract tests are also supported. For model validation we have used the Object Constraint Language (OCL) [OMG06b] as implemented in the SQUAM tool [COGIOT06].

In this environment, we have defined a domain–specific language that fulfills our specific requirements and allows the integration of tabular test data. The tests are linked to requirements and can be directly transformed to executable tests by adapters. The test results are annotated in the test models due to the traceability between all artifacts.

Our approach is also applicable for testing security requirements of service–centric systems. We attach security requirements to functional requirements and test them in an integrated way. This provides methodological support for testing security requirements in a systematic way.

On the model level we have separated but connected requirements, system, and test models. We handle the evolution of requirements, system artifacts and tests by a state–based change management process. The process triggers and propagates state changes on the model level, supports consistency management in the evolution process and the generation of regression test suites.

The research method applied in this thesis is based on the design science paradigm [HMPR04, PTRC08]. This paradigm essentially advocates the construction of a demonstration implementation of a proposed new approach followed by the evaluation of its usability, effectiveness, and efficiency. We have evaluated our approach in two industrial case studies from the telecommunication and home networking domain.

Contribution. The results presented in this thesis extend and improve previous achievements in the field of model–driven system testing of service–centric systems.

As shown in Figure 1.2, system test design methods can be classified according to two dimensions, i.e., Formality and Tabularity.

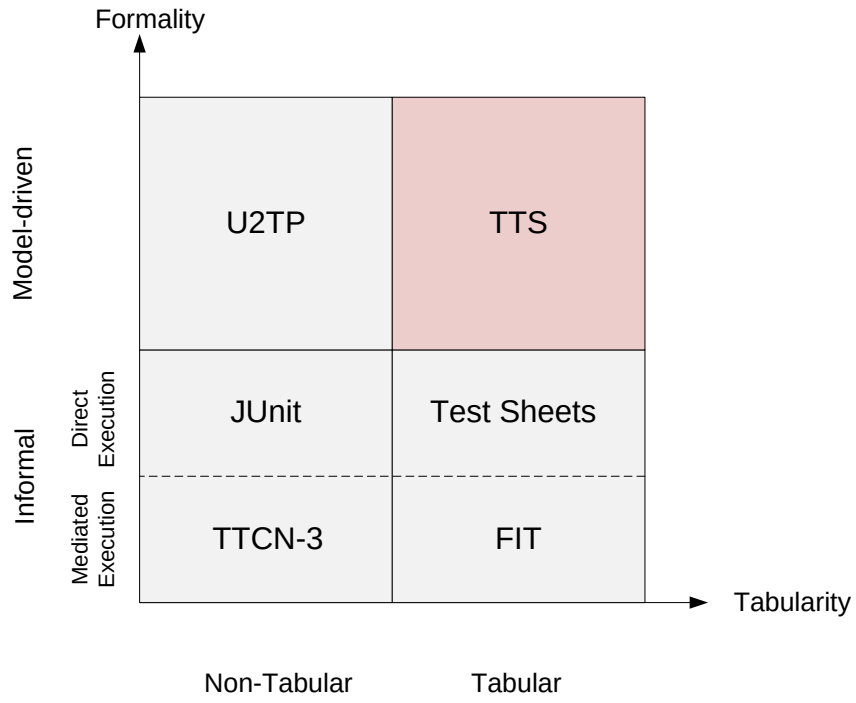


Figure 1.2: Classification of System Test Design Methods

Formality addresses whether the tests are designed formally, i.e., based on a metamodel (*Model–driven*), or as test scripts but without a metamodel (*Informal*). Tests defined with informal methods can be executed directly (*Direct Execution*) or executed after a transformation or adaptation (*Mediated Execution*). Today, practical model–driven approaches are always executed after a transformation and therefore not distinguished any further. *Tabularity* addresses whether test information and results are represented in tables (*Tabular*) or in other structures (*Non–Tabular*). In contrast to non–tabular approaches, tabular approaches allow the pure test logic, test data, and test results to be presented without additional noise or programmatic detail. A tester writing a test table focuses exclusively on defin-

ing input data and expected results, and a framework presents test results simply highlighting places where the subjects behavior deviates from that expected. The non–tabular approaches typically focus on the testing of concrete artifacts whereas tabular approaches focus on testing relationships between input and output.

In Figure 1.2 we also provide an example for each category of our classification. In *FIT* [Mug05] tests are represented as informal tables executable via fixtures. *TestSheets* [ABB10] provide a directly executable tabular test design and test reporting notation. *TTCN–3* [WDT+05] is a standardized textual test specification language that can be transformed to executable Java code. In *JUnit* [jun] tests are written programmatically as informal source code that can be executed directly. The UML 2.0 Testing Profile (*U2TP*) [OMG05] is a modeling languages for tests which supports model–driven testing. So far a tabular and model–driven approach has not been considered. The thesis at hand closes this gap by defining a tabular and model–driven system testing methodology called Telling TestStories (*TTS*).

Below we list the main contributions (**C**) of this thesis and assign them to the research challenges (**RC**) defined in the previous section.

C1 – Definition, implementation and evaluation of a tool–based model–driven system testing methodology for service–centric systems called Telling TestStories (TTS). The approach is model–driven and tabular. TTS is novel according to the classification in Figure 1.2 (contributes to **RC1**).

C2 – Classification of TTS and interrelationship to other model–driven and tabular system testing approaches (contributes to **RC1**).

C3 – Automatic generation of executable test code from a test model. The transformation directly generates test code from test models without an intermediate step (contributes to **RC1**, **RC2**, **RC4**).

C4 – Systematic validation of consistency, completeness and coverage criteria in and between tabular test models, requirements models and system models (contributes to **RC1**, **RC2**, **RC4**).

C5 – A–posteriori validation of test runs by arbitrations. The formalism is capable for functional and performance constraints based on a test run and its verdicts (contributes to **RC1**).

C6 – Application of model–driven and tabular testing to security requirements testing (contributes to **RC3**).

C7 – State–based management of evolving systems, requirements and tests, plus generation of regression test suites (contributes to **RC4**).

With our tool–based model–driven system testing methodology we address key problems stated in the fundamental research category of the testing road map

in [Har00]. We contribute to the key problem of *testing component–based systems* because service–centric systems can be considered as generalization of component–based systems, to *testing based on precode artifacts* because we define test models based on and validate them against the system design and the requirements, to *testing evolving software* because we consider the state–based management of system and test models plus regression testing, and to *using testing artifacts* because we implement full traceability between all artifacts integrated in our testing process.

Our contribution resulted in several publications listed in the next paragraph.

Publications. In the course of this PhD thesis, the following conference and journal papers have been published (in ascending order of their publication date):

1. J. Chimiak-Opoka, M. Felderer, C. Lenz, and C. Lange: Querying UML Models using OCL and Prolog: A Performance Study. In: Model Driven Engineering, Verification, and Validation. (2008)

2. M. Felderer, J. Chimiak-Opoka, and R. Breu: Telling TestStories - Modellbasiertes Akzeptanz-Testen Serviceorientierter Systeme. In: Softwaretechnik Trends 28(3). (2008)

3. M. Felderer, R. Breu, J. Chimiak-Opoka, and F. Schupp: TestStories - Ausfuehrbare Requirements fuer Serviceorientierte Architekturen. In: Informatik 2008 Beherrschbare Systeme dank Informatik Band 2. (2008)

4. M. Felderer, R. Breu, J. Chimiak-Opoka, M. Breu, and F. Schupp: Concepts for model-based Requirements Testing of service–centric Systems. In: Proceedings of the IASTED Software Engineering 2009. (2009)

5. J. Chimiak-Opoka, S. Löw, M. Felderer, R. Breu, F. Fiedler, F. Schupp, and M. Breu: Generic Arbitrations for Test Reporting. In: Proceedings of the IASTED Software Engineering 2009. (2009)

6. M. Felderer, P. Zech, F. Fiedler, J. Chimiak-Opoka, and R. Breu: Model-driven System Testing of service–centric Systems. In: Proceedings of the 9th International Conference on Quality Software. (2009)

7. M. Felderer, F. Fiedler, P. Zech, and R. Breu: Model–driven System Testing of a Telephony Connector with Telling Test Stories. In: Proceedings of the CONQUEST 2009. (2009)

8. M. Felderer, F. Fiedler, P. Zech, and R. Breu: Modellgetriebene Systemtests mit Telling TestStories. In: SQ Magazin (12). (2009)

9. M. Felderer, J. Chimiak-Opoka, and R. Breu: Model-driven System Testing of service–centric Systems. In: Proceedings of the 11th International Conference on Enterprise Information Systems. (2010)

10. M. Felderer, B. Agreiter, and R. Breu: Security Testing by Telling TestStories. In: Proceedings of the Modellierung 2010. (2010)

11. M. Felderer, P. Zech, F. Fiedler, and R. Breu: A Tool-based methodology for System Testing of Service-oriented systems. In: Proceedings of the VALID 2010. (2010) (received a **best paper award**)

12. M. Felderer and P. Zech: Telling TestStories – A Tool for Tabular and Model-driven System Testing. In: Testing Experience (12). (2010)

13. M. Felderer, B. Agreiter, and R. Breu: Managing Evolution of Service–centric systems by test models. In: IASTED International Conference on Software Engineering. (2011) (accepted for the IASTED–SE 2011)

14. M. Felderer, B. Agreiter, and R. Breu: Evolution of Security Requirements Tests for Service-centric Systems. In: International Symposium on Engineering Secure Software and Systems. (2011) (accepted for the ESSOS'11)

15. M. Felderer, J. Chimiak-Opoka, P. Zech, C. Haisjackl, F. Fiedler, and R. Breu: Model Validation in a Tool–based Methodology for System Testing of Service–oriented Systems. In: International Journal On Advances in Software. (2011) (to appear in May 2011)

1.4 Structure of the Thesis

The thesis at hand provides a comprehensive report on various aspects of the Telling TestStories framework. In this thesis, we cover foundations, the basic methodology, formalizations, security testing, and regression testing with Telling TestStories. We provide case studies based on industrial applications for all presented ideas. Each chapter starts with an overview and a motivation and closes with related work and a summary. The main content of each chapter is as follows.

Chapter 1. In Chapter 1 we motivate the problem discussed in this thesis (Section 1.1) and formulate a problem statement (Section 1.2). Based on the problem statement we sketch the solution to it and formulate our contribution (Section 1.3). Finally, we present the structure of this thesis (Section 1.4).

Chapter 2. In Chapter 2 we define the underlying concepts of the proposed methodology. We present the background of model–driven testing, i.e., software testing basics, model–based testing and model–driven engineering (Section 2.1). Then we determine our view on service–centric systems (Section 2.2).

Chapter 3. In Chapter 3 we present our basic testing methodology. We first introduce its system and testing artifacts (Section 3.1). Based on these artifacts, we define the testing process (Section 3.2), and domain specific languages for the requirements, system, and test model (Section 3.3). We then employ the framework on a callmanager case study (Section 3.4). Afterwards we present the tool implementation of our methodology (Section 3.5) and related approaches (Section 3.6). We close with a classification of our approach (Section 3.7).

Chapter 4. In Chapter 4 we provide formal foundations for the basic testing methodology. We first give a set–based formalization of TTS concepts needed for the precise definition of arbitrations and test evolution (Section 4.1). We then define an arbitration mechanism also considering performance testing (Section 4.2), and the validation of models by consistency, completeness and coverage checks (Section 4.3). Finally, we discuss the test model transformation (Section 4.4).

Chapter 5. In Chapter 5 we employ Telling TestStories on security requirements testing. We first determine security and security testing (Section 5.1), and then apply our security requirements testing approach to a case study from the home networking domain (Section 5.2).

Chapter 6. In Chapter 6 we consider the evolution of test models. We first give an extensive motivation for the problem (Section 6.1) and define a tailored metamodel (Section 6.2). Based on that metamodel, we define the evolution process (Section 6.3) and employ it on a case study (Section 6.4).

Chapter 7. In Chapter 7 we summarize the thesis (Section 7.1), evaluate the research results (Section 7.2), and identify future work based on the results of this thesis (Section 7.3).

The Chapters 1 to 7 sketched above depend on each other. In Figure 1.3 the relationship between the chapters is shown.

Chapter 1 motivates and defines the problem discussed in the thesis and is the basis for all other chapters. Chapter 2 presents the background of the methodology. Chapter 3 then explains the various basic aspects of the Telling TestStories methodology. Based on that, Telling TestStories can be extended in several directions reflected by two separate chapters. Chapter 4 defines a formalization of the artifacts, the arbitration concept, model validation techniques, and test model transformations. Chapter 5 introduces security requirements testing with TTS. Based on the formalization of the artifacts presented in Chapter 4 and the security requirements testing case study of Chapter 5, in Chapter 6 test model evolution management and regression testing is discussed. The thesis closes with conclusions in Chapter 7 referring to all preceding chapters.

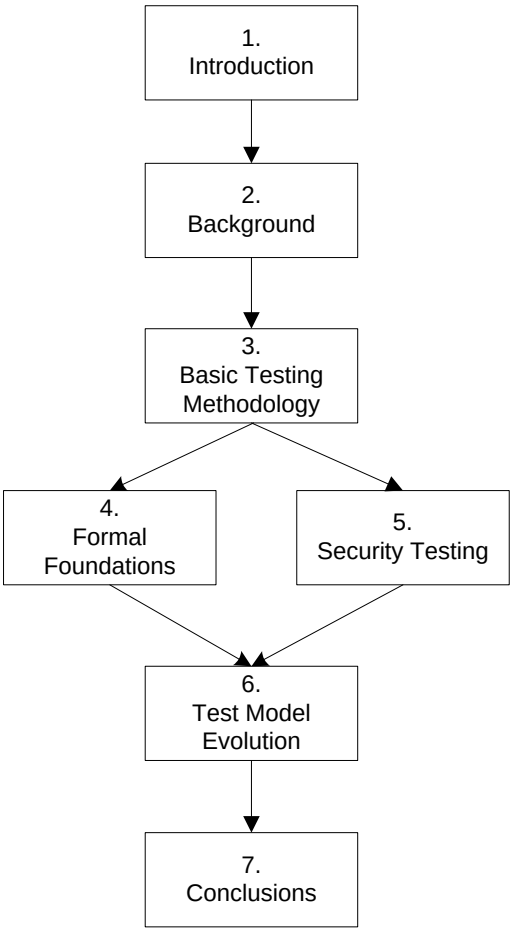


Figure 1.3: Relationship between the Chapters of the Thesis

Conventions. To make this thesis readable and clear we follow some conventions. For frequently used terms we define acronyms that are collected in Section A. TTS metamodel elements are written in a sans-serif font, arbitrary model elements, operations, types or output lines are written in a `typwriter` font, system states and paragraph titles are written in **bold** font, and key terms plus important text passages are written in *italic* font.

Chapter 2

Background

He who wants to build high towers must dwell with the fundament for a long time.

Anton Bruckner

In this chapter, basics about model–driven testing, service–centric systems, and domain specific languages are presented. We first define model–driven testing and present software testing basic, model–based testing, and model–driven engineering (Section 2.1). Afterwards we define service–centric systems and its underlying technologies (Section 2.2). Finally, we sum up (Section 2.3).

2.1 Model–driven Testing

Model–driven testing (MDT) refers to a particular style of model–based testing, inspired by model–driven engineering (MDE). Therefore MDT comprises the *modeling of tests* and the *automatic transformation to executable tests*. In this section, we present the basic principles and the terminology of model–driven testing including basics of software testing, model–driven engineering, and model–based testing as far as needed in the remainder of this document.

Models which are the basis for all methodologies and technologies presented in this thesis can be characterized as follows [Sta73]:

- Models are *mappings* from a concrete (the "original") into a more abstract (the "model") world;
- Models serve a specific *purpose*;
- Models are *simplifications*, in that they do not reflect all attributes of the concrete world.

Models are typically denoted in a modeling language defined by a syntax and a semantics (see Section 2.1.3 for more details). We sometimes use the term "model"

also for a projection on a specific aspect of a model ("view") or its representation, e.g., a diagram in case of UML.

2.1.1 Software Testing Basics

A *fault* is a defect in the software. A *failure* is an external, incorrect behavior with respect to the requirements or some other description of the expected behavior. A fault is the cause of a failure. Failures are typically observed during the execution of the system being tested. The term *error* is used synonymously to fault in this thesis.

Testing is the evaluation of software by observing its execution [AO08]. The executed system is called *system under test* (SUT). The term "testing" is used as synonymous for "dynamic testing". Software Testing consists of the *dynamic* verification of the behavior of a program on a *finite* set of test cases, suitably *selected* from the usually infinite executions domain, against the *expected* behavior [IEE04b]. In a test case, the actual and intended behavior of a SUT are compared and result in a *verdict*. Generally, verdicts can be either of *pass* (behaviors conform), *fail* (behaviors do not conform), and *inconclusive* (not known whether behaviors conform). A *test oracle* is a mechanism for determining the verdict.

Validation is the process of evaluating software during or at the end of the development process to determine whether it satisfies specified requirements. *Verification* is the process of evaluating software to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase [IEE90b]. Testing is a means for the verification and validation of software.

[Tre04] defines the type of testing along the three dimensions *level of detail*, *characteristics*, and *accessibility* (see Figure 2.1).

According to the level of detail, we can distinguish the following types of testing:

- *Unit testing* is applied to the smallest unit of program code
- *Module testing* tests the conformance of distinct components of a system
- *System testing* is applied to a complete system. System testing is usually supported by related techniques such as *Integration testing* and *Interface testing*.

System testing in our respect is based on the system requirements. We therefore do not consider *acceptance testing* as separate testing category because acceptance testing is requirements testing from the customers perspective and an extension of system testing focusing on usability requirements [BM05].

According to the characteristics, we can distinguish the following types of testing:

- *Performance testing* tries to determine whether the software is correct with respect to timing requirements.
- *Stress testing* evaluates the system behavior under heavy load.

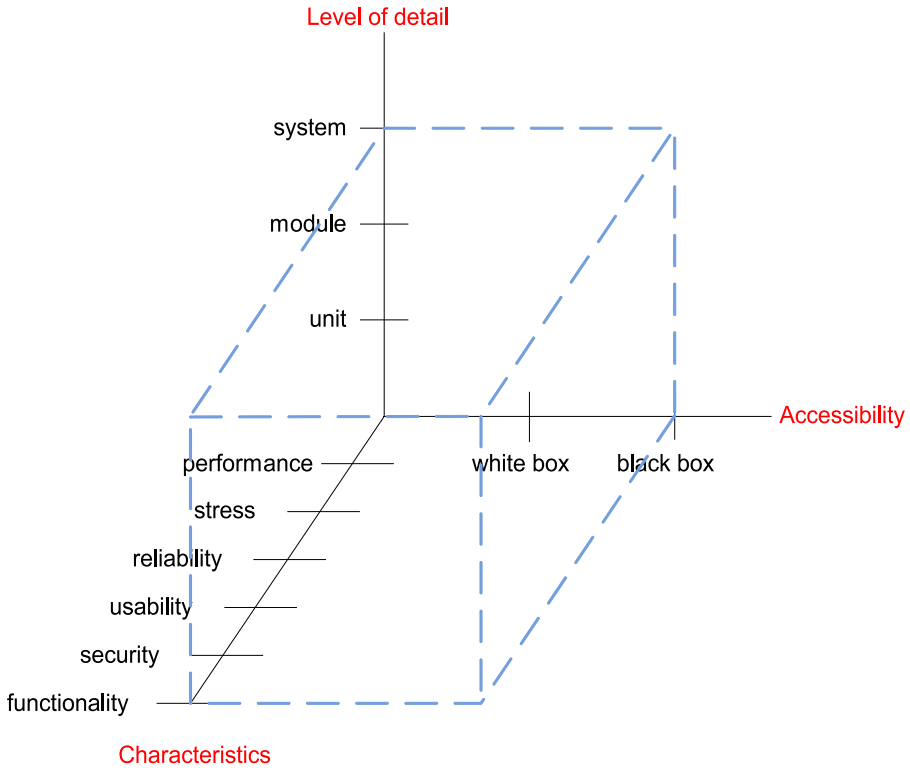


Figure 2.1: Types of Testing [Tre04]

- *Reliability testing* determines whether a system behaves correctly over a longer period of time.
- *Usability testing* evaluates the user interface which makes the software difficult to use or may cause users to misinterpret the output.
- *Security testing* determines whether a system protects data and maintains functionality as intended.
- *Functional testing* examines whether a system is a correct refinement of its design or specification.

It is possible to consider many more characteristics, e.g., following the classification in the ISO/IEC 9126 standard [ISO01].

According to accessibility we can distinguish the following types of testing:

- *Black–box testing* is the process of deriving tests from external descriptions of the software, including specifications, requirements, and design. The system under test is considered as a black box, where only inputs and outputs are known, the internal details are unknown. This type of testing is also known as *behavioral testing*.
- *White–box testing* is the process of deriving tests from the source code internals of the software, specifically including branches, individual conditions, and

statements. White–box testing is also known as *structural testing*.

White–box testing allows for creating such test cases that execute certain parts of the code. In contrast to black–box testing, white–box testing techniques allow for testing all the program code. Black–box testing, however, has the potential to detect missing functionality. Often, a combination of such approaches is used, for instance, when creating functional tests under consideration of a system's module structure. This type of testing is called *grey–box testing*.

The list of possible categorizations could be continued further. There are many different testing techniques taxonomies available in the literature, for example in Beizer's classical book on software testing [Bei90].

Regression testing is the selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements [IEE90a].

A *testing methodology* defines a testing process, its artifacts such as the models or the generated test code, and its integration with the system development process. Note that a testing method, i.e., a testing technique differs from a testing methodology and may be defined as part of a testing methodology.

In the remainder of this thesis, the term *test* is used for singular test cases or test suites, and the term *testing* for the test process.

2.1.2 Model–based Testing

In principle, any form of software testing can be seen as model–based. The tester always forms a mental model of the system under test before engaging in activities such as test case design [Bin99]. The term model–based testing (MBT) is applicable when these mental models are documented and subsequently used to generate tests, to execute tests or to evaluate their results [HIM00]. There are many definitions of MBT [RBGW10] but each contain at least one of the following two aspects:

- modeling of tests
- generation of tests from models

Models must be formal enough to allow, in principle, a machine to derive tests from these models, which is not the case for UML use case diagrams for instance [UPL06]. In the reminder of this document we only consider models for the derivation of tests and not of the test system or the test architecture.

In [UPL06] a taxonomy for model–based testing is introduced that identifies seven different dimensions and instantiations for each dimension. The seven dimensions *Subject*, *Redundancy*, *Characteristics*, *Paradigm*, *Test Selection Criteria*, *Technology*, and *On/Offline* are depicted in Figure 2.2.

Based on the artifacts generated from models, [UL07] distinguishes four different approaches to model–based testing:

1. Generation of test input data from a domain model

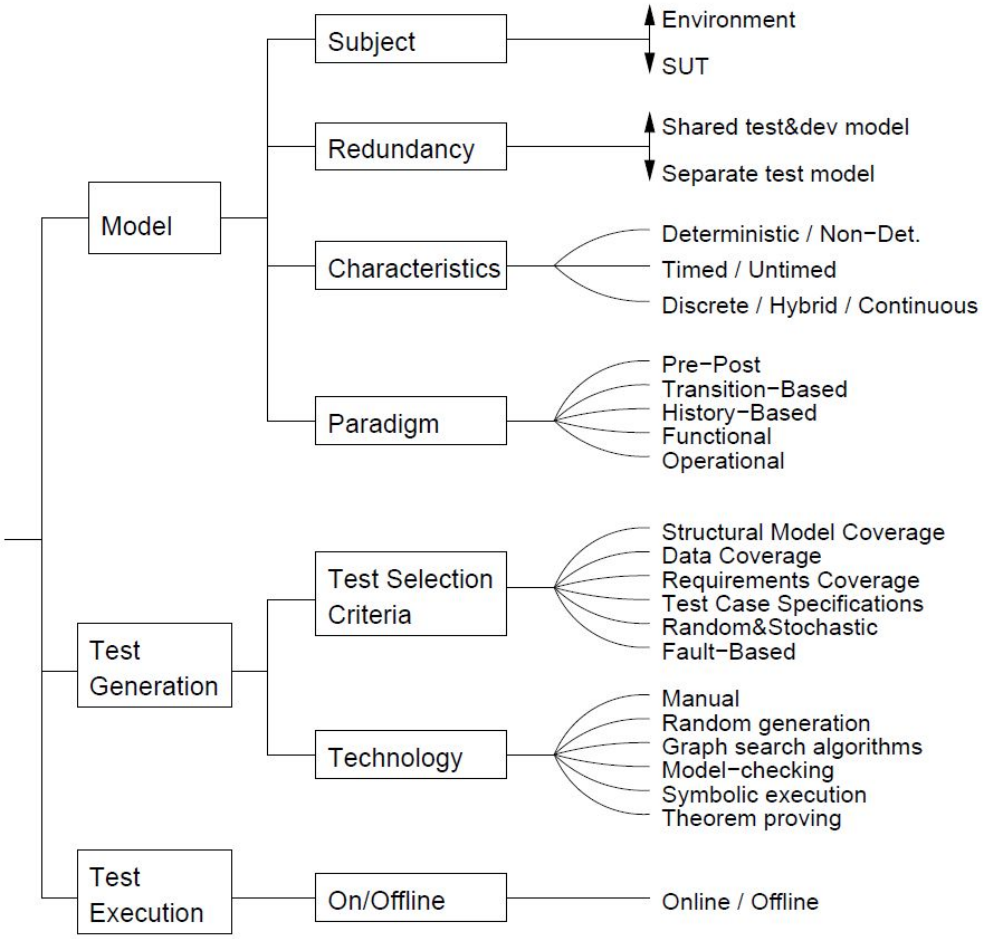


Figure 2.2: Model–based Testing Taxonomy [UPL06]

2. Generation of test cases from an environment model
3. Generation of test cases with oracles from a behavior model
4. Generation of test scripts from abstract tests

Generally, models are not only used for the test but also for the system development. Based on the relationship between system and test models plus their application in the system and test development process, [Sch07] distinguishes six variants depicted in Figure 2.3.

System model–driven approaches (variant (a)) only use a system model for the system and test generation. The system and its tests are not independent, and faults in the system model cannot be recognized by the tests.

Test model–driven approaches use separate test models for the test generation (variant (b)) and also for the system generation (variant (c)). If a test–first and model–based test methodology is implemented, this is a minimalistic approach to it.

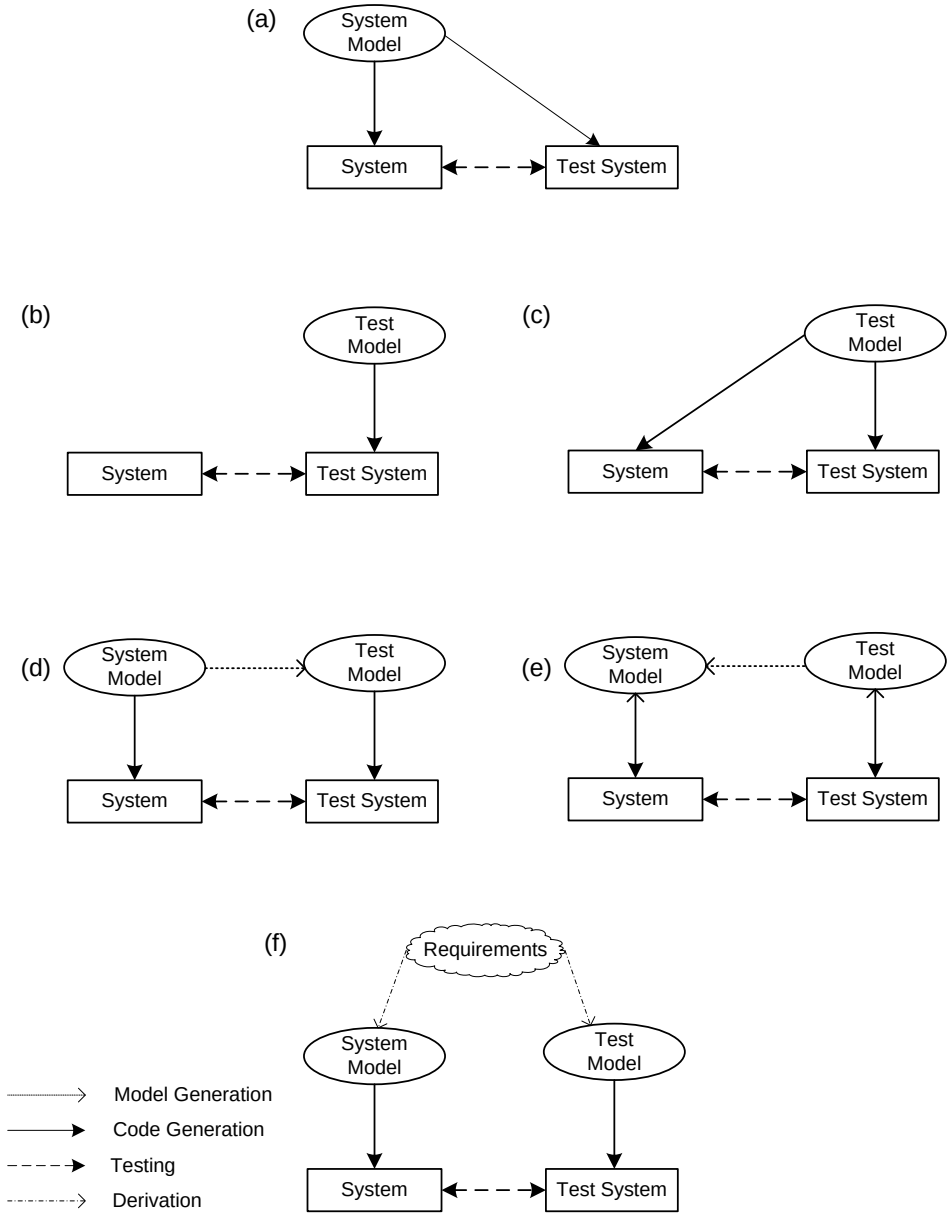


Figure 2.3: Variants of Model–based Testing [Sch07]

System and test model–driven approaches is the most consequent approach because models are used for the system and for the tests. This approach is based on two models which can be used to check the consistency or coverage even before the implementation exists. The overhead of defining two models is compensated by higher efficiency resulting in a higher system quality. The process can be system model–driven (variant (d)) or test model–driven (variant (e)) or separated (variant (f)). Variant (f) using independent system and test model, i.e., not just models generated from each other, is an optimal MBT approach according to [PP04].

In [PP04] four scenarios are discussed that concern the interplay of models

used for test case generation and code generation. The first scenario concerns the process of having one model for both code and test case generation. The second and third scenarios concern the process of building a model after the system it is supposed to represent either by model extraction or by manual modeling. The last and optimal approach considers two distinct models.

Various types of paradigms and notations have been used to describe models for MBT. According to the taxonomy in [UPL06] these notations can be grouped into the following paradigms:

- *State–based (or Pre/Post) notations* such as Z, B, VDM, OCL, or JML
- *Transition–based notations* such as state machines or labeled transition systems
- *History–based notations* such as temporal logics
- *Functional notations* such as algebraic specifications
- *Operational notations* such as petri nets or communicating sequential processes
- *Stochastic notations* such as Markov chains
- *Data–flow notations* such as Lustre or block diagrams

UML [OMG07b, OMG07a] contains a large set of diagrams and notations that are defined by a metamodel and have some freedom allowed for different interpretations of the semantics of the diagrams. Therefore there are many possible ways for model–based testing on top of UML models. The most common application scenarios are as follows:

- OCL preconditions and postconditions are used to define the behavior of methods. OCL constraints are applied for model–based testing e.g., in [ASA05] or in combination with other UML diagram types e.g., in [BL02, UL07, LLQC07].
- UML activity diagrams are well suited for modeling processes and workflows. They are a good basis for test generation and applied for model–based testing e.g., in [KKBK07, LJX+04, CXX06, CMK08, BL02], model–based testing has successfully been applied on activity diagrams.
- UML state machines model the internal behavior systems and applied for model–based testing e.g., in [UL07, OA99].
- UML sequence diagrams are used for the representation of abstract test cases or for specifying the behavior of distributed processes and applied for model–based testing e.g., in [LS06, LLQC07].

When using the terms “activity diagrams”, “state machines” or “sequence diagrams” in the remainder of this document, we mean the UML variants and depending on the context even the diagram’s underlying model.

2.1.3 Model–driven Engineering

Model–driven Engineering (MDE) involves the systematic use of models as essential artifacts throughout the software development process [Sch06]. MDT can therefore be considered as application of the general MDE principle in the testing step of the development process. At the core of MDE are the concepts and technologies of *domain specific languages* and *model transformations* which are briefly discussed in Section 2.1.3 and in Section 2.1.3. A standardized approach to MDE is Model–driven Architecture (MDA) which is discussed in the context of MDT in Section 2.1.3.

Domain Specific Languages

The task of a modeling language, called language in the following, is to provide a set of concepts and an associated notation that allows the description of subjects of interest. In the case of software engineering, models are frequently used as construction plans, i.e., as descriptive models. The main purpose of a domain specific language (DSL) is to provide the "right" or the "best" abstractions for the problem in hand [AK07], the so called "domain".

Each language is defined by an *abstract syntax*, *well-formedness rules*, a *concrete syntax*, and its *semantics*.

The abstract syntax defines the basic set of concepts that can be used to make statements in the language together with rules for using them correctly. One particular useful way to specify the abstract syntax of a modeling language is to create a model of the abstract syntax using the notation of the modeling language to be defined. For instance, in Figure 3.10 the abstract syntax of our test design language is defined by a *metamodel*, i.e., a model that specifies an intended set of models.

Well–formedness rules complement the restrictions introduced by the abstract syntax with further constraints which cannot be expressed using the metamodeling notation. For instance, the correctness checks defined in Section C.1 can be considered as well-formedness rules.

The concrete syntax defines the notation to be used to present structures which are denoted using the abstract syntax. For instance, we use annotated UML activity diagrams to define tests. The concrete syntax of a language can be textual or visual.

The semantics defines the interpretation given to language constructs, i.e., expressions written in the abstract syntax. Well–known ways of defining this interpretation include [AK07]:

- *informal semantics*: in terms of natural descriptions;
- *operational semantics*: in terms of relating abstract syntax concepts to execution machinery;
- *denotational semantics*: in terms of mathematical mappings to known semantic domains;

- *translational semantics*: in terms of mapping the language to a target language.

Concerning the degree of formality, an *informal semantics*, i.e., described in a natural description, is distinguished from a *semi–formal semantics*, i.e., based on a formal notation such as a graph that cannot be verified systematically, and a *formal semantics*, i.e., based on a mathematical notation that can be verified.

In UML domain specific modeling is supported by UML profiles [OMG07a]. Profiles are collections of stereotypes, tagged values and constraints [Har06] usually defined as Object Constraint Language (OCL) expressions [OMG06b]. [Sel07] presents a systematic method for defining UML profiles. In this approach, the definition of a UML profile involves the construction of a domain model which is then mapped to a profile. The mapping contains the selection of a base UML metaclass whose semantics are closest to the semantics of the domain concept considering well–formedness rules, checks to determine if any of the attributes of the selected base metaclass need to be refined, checks to determine if the selected base metaclass has no conflicting associations to other metaclasses. UML does not provide a formal semantics, therefore the semantics of the DSL has to be specified.

DSLs differ in their degree of domain specifity from more general–purpose to more domain–specific, and in their level of abstraction from problem–oriented to solution–oriented.

Due to the context–sensitive nature of testing, domain–specific approaches offer many attractive advantages [Har06]. The basic idea of using a domain–specific solution is to introduce a language solely for the purpose of test modeling in the particular domain at hand. This way, it is possible to tailor the modeling language for the needs of the testers in a specific domain, i.e., system tests of service–centric systems.

Model Transformations

A model transformation is a *specification* or a *process* which takes as input a source model which takes as input a source model conforming to a given metamodel and produces as output a target model conforming to a metamodel. The transformation is itself a model conforming to a metamodel. The generic model transformation process is depicted in Figure 2.4. Note that the transformation model and the transformation process are called model transformation and that the meaning depends on the context.

Model transformations can be classified along several dimensions [CH03]. Concerning the language of source and target models, we can distinguish *endogenous transformations* between models in the same language, e.g., refactoring is such a type of transformation, and *exogenous transformations* between models in different languages, e.g., code generation. Concerning the level of abstraction *horizontal transformations* where the source and target models reside on the same abstraction level, e.g., language mitigation is such a type of transformation, and *vertical transfor-*

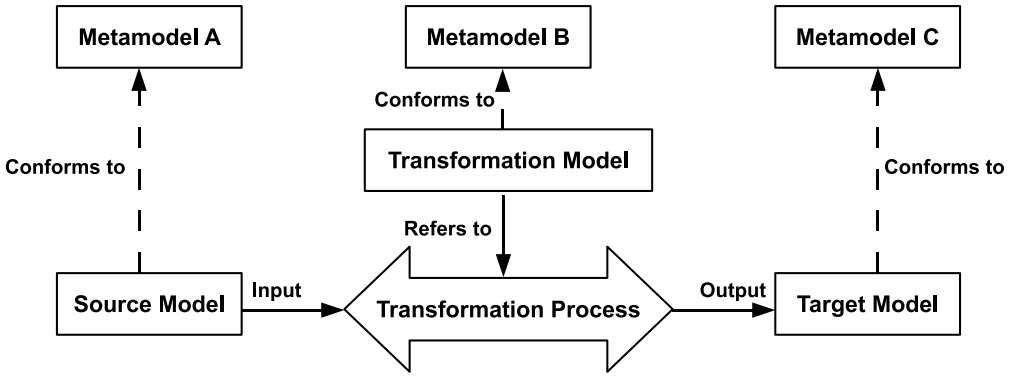


Figure 2.4: Generic Model Transformation Process

mations, e.g., a refinement, can be distinguished. Concerning the type of the target model, we can distinguish between *model–to–model transformations* where the target model conforms to a metamodel and *model–to–text transformations* where the source model is serialized to a textual representation. Concerning the transformation processing and the definition of the transformation, two types of model–to–text transformations are distinguished, namely the *visitor–based* approach, where the entire internal representation of the source model is processed, and the mostly applied *template–based* approach, where specific fragments of the source model defined by templates are processed.

There are various model transformation tools available. For the implementation of TTS, the openArchitectureWare framework [oaw] which supports model–to–model transformations and template–based model–to–text transformations has been applied.

Model–driven Architecture and Model–driven Testing

Model–driven Architecture [OMG03] is a MDE approach standardized by the OMG. It raises the abstraction level of software development by using models which are used for automatic code generation. MDA is based on the Meta–Object Facility (MOF) [OMG06a] standard for metamodeling.

The underlying principle behind MDA is the separation between platform specific and platform independent aspects of the software, and the use of automated transformation to pass between different levels of abstraction. Applied to model–driven testing, this means that the testing model is independent of the testing platform and that a transformation is used to pass from platform independent to platform specific ones [HIM00]. According to [ZDSD05] *model–driven testing* can be defined as testing–based MDA [OMG03], i.e., the derivation of executable test code from test models.

In MDA a computation independent model (CIM), platform independent model (PIM), platform specific model (PSM) and system code are defined. The corresponding artifacts in the model–driven testing (MDT) domain are a computation independent test model (CIT), a platform independent test model (PIT), a platform

specific test model (PST), and test code. MDT separates the platform independent and platform specific aspects of test models, and uses automatic transformations to pass between different levels of abstraction. Furthermore MDA and MDT can be related because test models might be generated from system models or the system models are used to check consistency and coverage properties of the test model. According to MDA, a PIT can be transformed either directly to test code or to a PST. Normally, the transformations from PIM to PIT and from PSM to PST cannot be done completely automatically. The overall process and the relationship between the artifacts is depicted in Figure 2.5.

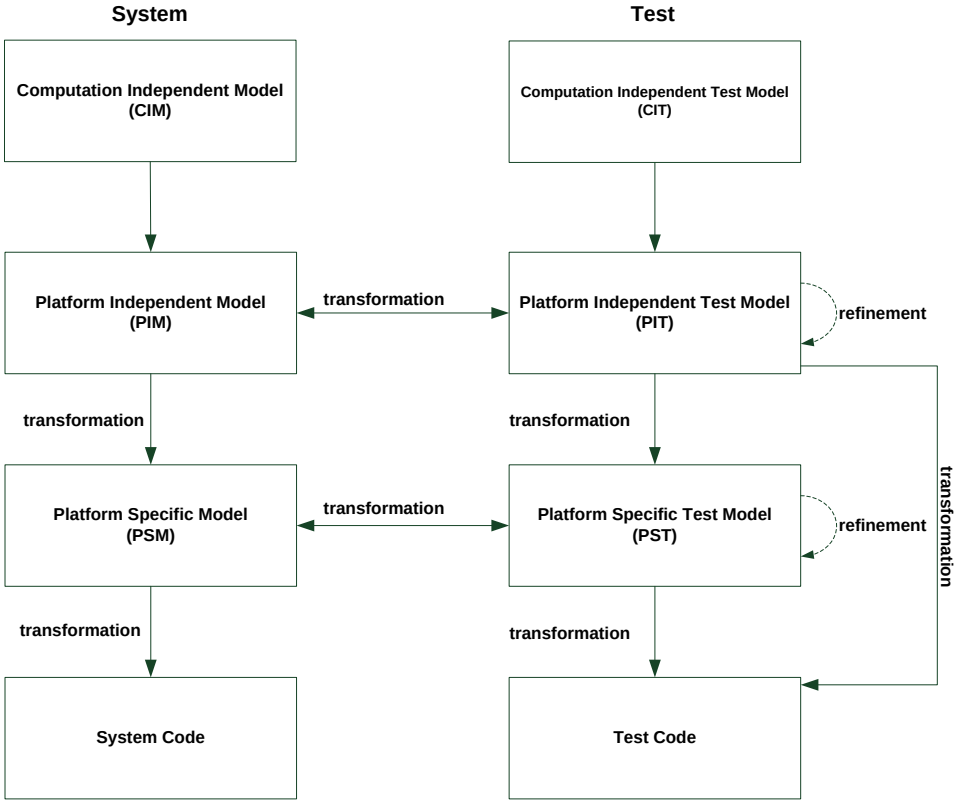


Figure 2.5: Model–Driven Testing Process [Sch07]

Model–driven testing in this sense is the most consequent approach to model–based testing because it uses models on the system and test side [Sch07]. If these models are independent, i.e., not just generated from each other, then this approach to model–based testing is optimal according to [PP04]. It has the advantage that models can already be checked for consistency and coverage before they are actually implemented. The overhead of defining two models is compensated by higher efficiency resulting in a higher system quality.

The transformations in the model stack of Figure 2.5 are as follows. We have *exogenous and vertical transformations* between different levels of abstraction in the system and test domain, e.g., from PIT to PST or from PST to test code, *exogenous horizontal transformations* from PIM to PIT and from PSM to PST, and

endogenous transformations by refining the PIT and PST. All transformations are defined on but also consistency and coverage checks are defined on MOF–compliant metamodels [OMG06a].

The *UML 2.0 Testing Profile* (U2TP) [OMG05] is a standardized UML 2.0 profile for the definition of CIT, PIT, and PST models (see Section 3.6.2 for a more detailed description of U2TP). In [ZDSD05] PIT models in U2TP are transformed to test code in *TTCN–3* [WDT+05], the Testing and Test Control Notation version 3, which is a standardized test specification and implementation language for black box testing mainly of distributed systems. The TTCN–3 test code is then transformed to executable test code in Java.

2.2 Service–centric Systems

Basically, a *service–centric system* consists of a set of independent peers offering services that provide and require operations via interfaces [EHH+08, CDP06]. Services are *loosely coupled* separating the service description from its execution environment, *dynamically bound* where the exact implementation is determined at runtime, and *composable* making services part of some other services. Orchestration and choreography technologies allow the flexible composition of services to workflows. A orchestration describes a workflow between services controlled by a single peer, whereas a choreography describes a collaborative workflow between services where no single peer controls the communication("peer–to–peer communication"). Orchestrations are themselves executable as services of the controlling peer whereas choreographies are typically non–executable. Service–centric systems narrow the conceptual gap between its implementation and its actual business function. This supports a business-purpose oriented combination of several services to a complex service composition.

Service–centric systems are applied for implementing flexible inter–organizational IT–based business processes and their number and complexity is steadily increasing.

Arising application scenarios have demonstrated the power of service–centric systems. These range from the exchange of health related data among stakeholders in health care, over new business models like SAAS (Software as a Service) to the cross-linking of traffic participants. Service–centric systems have specific system properties that limit their testability including the integration of various component and communication technologies, the dynamic adaptation and integration of services, the lack of service control, the lack of observability of service code and structure, the cost of testing, and the importance of service level agreements (SLA) [CD08]. Non–functional properties have a special meaning for service–centric systems because SLAs may include Quality of Service (QoS) constraints or security requirements are defined because services are often exposed over the Internet and can thus be subject to security attacks.

Service–centric systems follow the architectural style of a Service Oriented Architecture (SOA) which realizes a decentralized system architecture that delivers application functionality by the integration of services across domain boundaries [Haf08]. Typical technologies for realizing SOA are web services which use the XML based languages Web Services Description Language WSDL [CMRW03] for the interface description, the Universal Description and Discovery Interface (UDDI) [CHvRR04] for service registration, and SOAP [ML07] for the message exchange. Orchestrations can be defined in the Business Process Execution Language for Web Services (WS–BPEL) [OAS07], and choreographies in the Web Services Choreography Description Language (WS–CDL) [KBR+05].

For system modeling of service–centric systems specifications like the Service Oriented Architecture Modeling Language (SoaML) [OMG09b] or SECTET [Haf08] can be applied.

SECTET comprises two model views, namely the workflow and the interface view, necessary to cover all aspects needed for the design of SOA.

The *Workflow View* is further divided into the *Global Workflow*, which captures the interactions between cooperating partners, and the *Local Workflow* that describes an executable process, which is local to each partner and implements application logic.

The *Interface View* links the global and local workflow models. It describes the partner nodes as components offering a set of services with given properties and permissions. Its three sub–models – the *Interface Model*, the *Document Model*, and the *Role Model* – correspond to the public part of the local application logic in terms of the local workflow. The names of the model elements conform to the uniform technical and syntactical specifications the partners agreed upon when designing the global workflow (e.g., parameter format, interaction protocol, role names, service interfaces and parameters, operation semantics etc.).

SoaML extends UML 2.0 in four main areas: Participants, ServiceInterfaces, ServiceContracts, and ServiceData.

Participants allow for defining the service providers and consumers in a system. *ServiceInterfaces* enable the explicit modeling of the provided and required operations. Each operation has a precondition, a postcondition, input and output data and describes a *ServiceCapability*. *ServiceContracts* are used to describe interaction patterns between participants. *ServiceData* represent service messages and attachments. Finally, the metamodel provides elements to model service messages explicitly.

The concepts of SECTET can be mapped to SoaML as shown in Table 2.1. Basically, global and local workflows are represented by behaviors attached to Service Architecture and Participant which additionally provide a structural view on the service oriented system. Contracts of SoaML are just partially reflected by policies in SECTET.

SECTET	**SoaML**
`Global Workflow`	`Behavior` attached to a `Service Architecture`
`LocalWorkflow`	`Behavior` attached to a `Participant`
set of `Interface` elements	`ServiceInterface`
`Role`	`Participant`, `Agent`
`Message`	`MessageType`
policies on documents not covering complete policy functionality	`ServiceContract`
	additional concepts (`Agent`, `Milestone`, ...)

Table 2.1: Mapping of SECTET to SoaML

The SoaML standard defines additional concepts that are not directly reflected in SECTET, such as `ServicesArchitecture`, `Agent`, `Milestone`, `CollaborationUse`, `ServiceChannel`, and `RealWorldEffect`. SoaML conforms to OASIS standards because in [OMG09b] a mapping between SoaML and OASIS standards is defined. Therefore we do not consider this additional notation here.

Especially on the system side, the metamodel of SECTET is not standardized and the metamodel of SoaML is still changing and not concise. We therefore build our approach on a concise metamodel that can be mapped to SECTET and the core concepts of SoaML. Additionally, this metamodel is traceable with requirements and can be integrated with test stories, i.e., test models, which can be used to implement a test–driven approach (see Section 3.4.6 on traceability). The system metamodel is very abstract and can be mapped to arbitrary distributed technologies such as web services [FFZB09] or classical RMI [FZF+09].

2.3 Summary

In this chapter we have defined the basic concepts of model–driven system testing of service–centric systems.

In Section 2.1 we have defined the basics of model–driven testing. First, we have defined software testing as the evaluation of software by observing its execution and categorized it along the dimensions level of detail, characteristics and accessibility. We have then defined model–based testing as test generation, execution and evaluation based on explicit models. We provided a taxonomy for MBT, and categorized MBT approaches whether they are system–model and/or test model–driven. We have also categorized model–based testing based on UML models. Finally, we defined the relevant basics of model–driven engineering, i.e., domain specific languages, model transformations and MDA which allowed us to define model–driven testing as test–based MDA and special type of model–based testing.

In Section 2.2 we have defined service–centric systems as systems consisting of a set of independent peers offering services that provide and require operations via interfaces. Service centric systems follow the architectural style of a Service Oriented Architecture and require special model–based testing techniques. Service–centric systems can be modeled with SoaML or SECTET.

Chapter 3

Telling TestStories Testing Methodology

There is nothing more
practical than a good theory.

Kurt Lewin

In this chapter, we define the basic testing methodology of Telling TestStories (TTS). TTS is a model–driven testing approach for service–centric systems. In the following, when we use the terms "methodology", "framework" or "approach" without a defined context, we refer to TTS.

We first present the involved system and testing artifacts and their relationship (Section 3.1). Based on the artifacts, we describe our testing process (Section 3.2). Afterwards we define a domain specific language for requirements, system and test modeling by its abstract syntax, its concrete syntax and its semantics (Section 3.3). We then apply TTS to the callmanager case study from the telecommunication domain (Section 3.4), and provide an overview of the tool implementation and its underlying technologies (Section 3.5). After the presentation of related approaches (Section 3.6), we classify and evaluate our approach (Section 3.7). Finally, we present related work (Section 3.8) and sum up (Section 3.9).

3.1 TTS System and Testing Artifacts

In this section we define the artifacts, i.e., textual descriptions, models, code and running systems of TTS and their interrelationship in an informal way. The TTS methodology is an adaptation of the variant (f) of the MBT approaches in Figure 2.3. This variant has the advantage that consistency or coverage can be checked even before the actual implementation exists.

Figure 3.1 shows the artifacts of the TTS framework and their relationship. In the TTS framework we can distinguish three formalization levels with informal artifacts (at the top), model artifacts (in the middle), and implementation artifacts

(at the bottom). Informal artifacts are depicted by clouds, (semi–)formal models by graphs, code by transparent blocks and running systems by filled blocks. Formalized relationships between artifacts are depicted by solid lines, whereas informal dependencies are depicted by dashed lines.

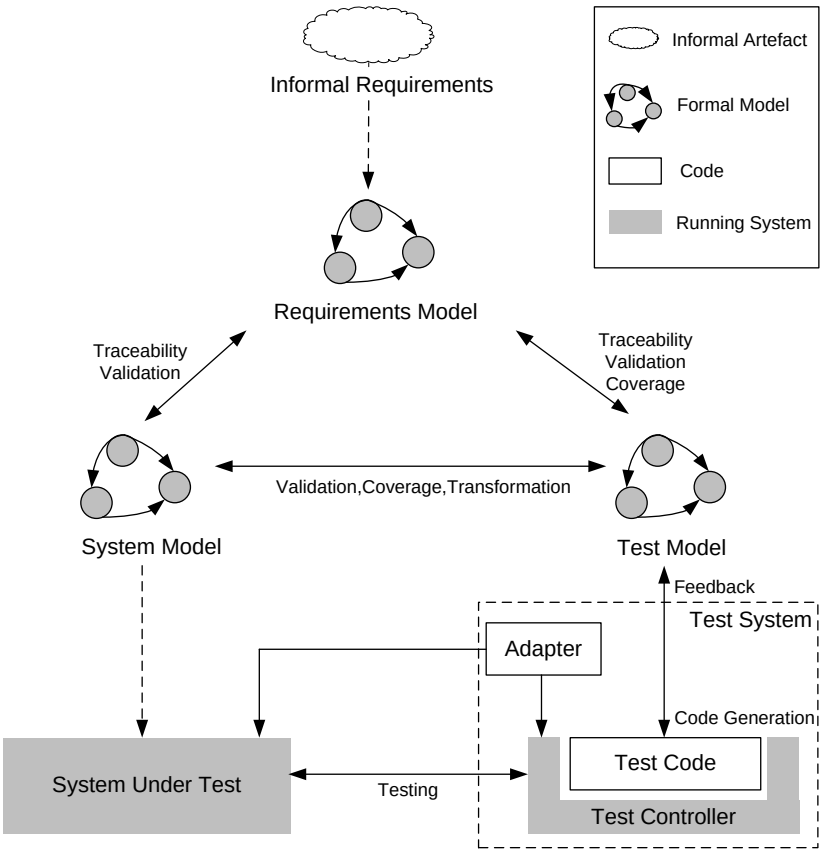


Figure 3.1: Overview of the TTS Artifacts and their Relationship

Based on the *Informal Requirements*, a **Requirements Model** is defined. Its model elements are traceable to model elements of the *System Model* and the *Test Model*. The system model is needed for validation, coverage checks and transformation to tests. The test model is used for validating the system model. From the test model *Test Code* is generated. The test code is executed by the *Test Controller* which uses *Adapter* classes to invoke operations on the *System Under Test*. We do not consider the transformation from the system model to the system under test which is symbolized by the dashed line in Figure 3.1. The underlying testing process is presented in the next section.

We have modified the variant (f) of the MBT approach in Figure 2.3 in the following points:

- We have integrated a requirements model which allows us to integrate traceability to high–level requirements into the model level. Additionally, we can consider tests as part of an executable requirements model.

- Although our system and test model are principally independent, our approach can be applied in a system–driven way, e.g., by generating tests from the system and in a test–driven way, e.g., by using tests for defining the system behavior.
- Adapters which make services executable have been integrated

In the following paragraphs we explain the artifacts of Figure 3.1 in more detail.

On the *informal level* there is only one type of artifact defined, namely **Informal Requirements**, i.e., written or non–written capabilities and properties. Informal requirements are not discussed in detail because they are not in the main focus of our testing methodology and cannot be validated automatically.

On the *model level* there are three models defined: requirements, system and test model. The models are formally defined by a metamodel. Based on the metamodel inter–model and intra–model relationships are defined, namely traceability, validation, coverage, and transformation. For readability reasons in Figure 3.1 only the inter–model relationships are depicted, but in each of the models it is possible to define validation, coverage and transformation rules based on its metamodel.

Requirements Model. The requirements model contains the functional and non–functional requirements for system development and testing. Its structured part consists of a requirements hierarchy. The requirements are based on informal requirements depicted as a cloud. The requirements model provides a way to integrate textual descriptions of requirements which are needed for communication with non–technicians into a modeling tool. The requirements metamodel is defined in Section 3.3.1.

System Model. The system model describes the system structure and system behavior in a platform independent way. Its *static structure* is based on the notion of services. Each (business) service or even each operation is assigned to a requirement. We assume that each service in the system model corresponds to an executable service in the running system to guarantee traceability. Additionally, configuration services for configuring the system or modifying types and information services providing information about the business objects may be defined. The latter two types of services are not corresponding to requirements. But requirements, (business) services, and executable services are traceable. The *dynamic structure* is based on local processes, which are controlled by a specific service and correspond to an orchestration, and global processes which integrate several services and correspond to a choreography. The system metamodel is defined in Section 3.3.2.

Test Model. The test model defines the test requirements, the test data and the test scenarios as so called test stories or in a general context so called tests. We use the terms 'test story' and 'test' interchangeably in this thesis depending on

the context. If we address the more abstract view, we use the term 'test'. If we address the more application–oriented and process–oriented view, we use the term 'test story'. *Test stories* are controlled sequences of service operation invocations exemplifying the interaction of services and assertions for computing verdicts. Tests may be generic in the sense that they do not contain concrete objects but variables which refer to test objects provided in tables. Tests can also invoke configuration services for the system setup and tear down. The concept of a test is principally independent of its representation. We employ UML sequence diagrams [FABA10] and UML activity diagrams [FCOB10] but are not restricted to these notations. Tests include references to a table of test data including values for all free parameters of the test. Each line in this table defines test data for one test case. The test data elements are stored in a *data pool*. *Test requirements* can contain global arbitrations that define when a test is successful by restrictions on the verdicts, or coverage constraints that define what parts of a model have to be inspected by a test. The syntax and semantics of tests is defined in Section 3.3.3.

Each test is linked to a requirement and can therefore be considered as executable requirement by analogy to FIT tests [Mug05]. Tests in our sense can also be used as testing facets [CD08] defining built–in test cases for a service that allows potential actors to evaluate the service. Together with appropriate requirements the tests may serve as executable SLAs.

Traceability. For model maintenance, transformations and validations traceability between different model elements is required. In the TTS framework traceability between elements on the model level is guaranteed by links between model elements, and between the model level and the implementation level by adapters for each service. The adapters link service calls in the model to executable services. Therefore every service invocation is traceable to a requirement.

Transformation. In the TTS methodology, we consider model–to–model transformations within the test model and between the system and the test model. In this thesis we introduce a model–driven methodology for the manual definition of tests and support for it, but we do not consider automatic test generation techniques from the system model as in classical model–based testing. We therefore only sketch existing model–to–model transformations from the system model to the test model and how they could be integrated into TTS as related work in Section 4.5). In Section 4.4 we define a transformation from abstract tests feasible for domain experts to executable test stories.

Validation. Models designed manually require tool supported validation. Our approach is suitable for test–driven modeling because the test model is used to validate the system. In the context of TTS, we consider the properties consistency, completeness and coverage. In Section 4.3 validation is discussed in more detail.

Consistency checks assure that there is no conflicting information in models. Consistency of a model enables error–free transformation from the model to another model or to the source code. For manually designed (parts of) models consistency within and between them should be automatically checked. Within the TTS framework we specified and implemented consistency criteria for all three models and between pairs of them.

Completeness checks assure that one artifact is complete, i.e., contains all essential information. Similarly like for consistency, we can consider completeness within one model (for elements and their properties) and between models. Completeness of the system model is crucial for the TTS framework and determines whether transformations from the system model to the test model can be applied. If the system model is complete, then behavioral parts of the test model can be generated by model transformations.

Coverage can be considered as a variant of inter–model completeness where one model is the test model and the other one is the requirements model or the system model. Coverage is very important in the context of testing[1] and is used to check to what extend the test model covers the requirements and system model and implicitly the system. We adopted a series of coverage criteria from testing [AO08] and model–driven testing [UL07] to fit into the TTS framework.

On the *implementation level* the test code generated from the test model is executed by the test controller against the system under test. The executable services of the system under test are invoked by adapters.

Feedback. The feedback is responsible for linking the test results back to the model level and annotate them to the model elements. Annotations are represented by coloring and annotating actions and tables. The test code generation is discussed in Section 3.5.6.

Code Generation. The test code is generated automatically by a model–to–text transformation from the test model as presented in [FFZB09]. For each test in the test model, a test code file is generated. The test code generation is discussed in Section 3.5.4.

Test Code. The test code language is Java. Adapters which bind abstract service calls in the test code to running services of the system under test make the test code executable. The structure of the test code is discussed in Section 3.5.4.

[1]Due to the importance of coverage for testing purposes, it is explicitly denoted in Figure 3.1 although in our approach coverage is a type of validation.

Adapters. The adapters are needed to access service operations provided and required by components of the system under test. For a service implemented as web service, an adapter can be generated from its WSDL description. Adapters for each service are the link for traceability between the executable system, the test model and the requirements. Adapters make it possible to derive executable tests even before the system implementation has been finished which supports test–driven development. The adapters are discussed in Section 3.5.3.

Test Controller. The test controller executes the generated test code and accesses the system services via adapters. Our implementation of the test controller executes test code in Java but other JVM–based programming or scripting languages are also executable without much implementation effort. The test controller is discussed in Section 3.5.5.

Test System. The test controller, the adapter and the test code form the test system. The model of the test system is defined in Section 3.3.5.

Testing. The evaluation of the service–centric system by observing its execution [AO08] is called testing. Services are invoked in the test code executed by the test controller via adapters.

System Under Test. The system under test (SUT) is a service–centric system as defined in Section 2.2 that may contain additional configuration services. The model of the SUT is defined in Section 3.3.4.

The informal requirements can be considered as external input to the TTS framework, and the system under test is the target of the application of TTS. This is shown by two dashed arrows in Figure 3.1. The first dashed arrow goes from the informal requirements to the TTS requirements model, and the second dashed arrow goes from the test controller and adapters to the system under test. Both, the informal requirements and the system under test, are out of the scope of this thesis.

3.2 TTS Testing Process

Based on the artifacts presented in the previous section, we define the stages of the TTS testing process and the affected artifacts. The process consists of a *design*, *validation*, *execution*, and *evaluation* phase and is processed in an iterative way. Initially, the process is triggered by requirements for which services and tests have to be defined. The process is depicted in Figure 3.2.

The first step is the definition of requirements. Based on the requirements, the system model containing services and the test model containing tests are designed.

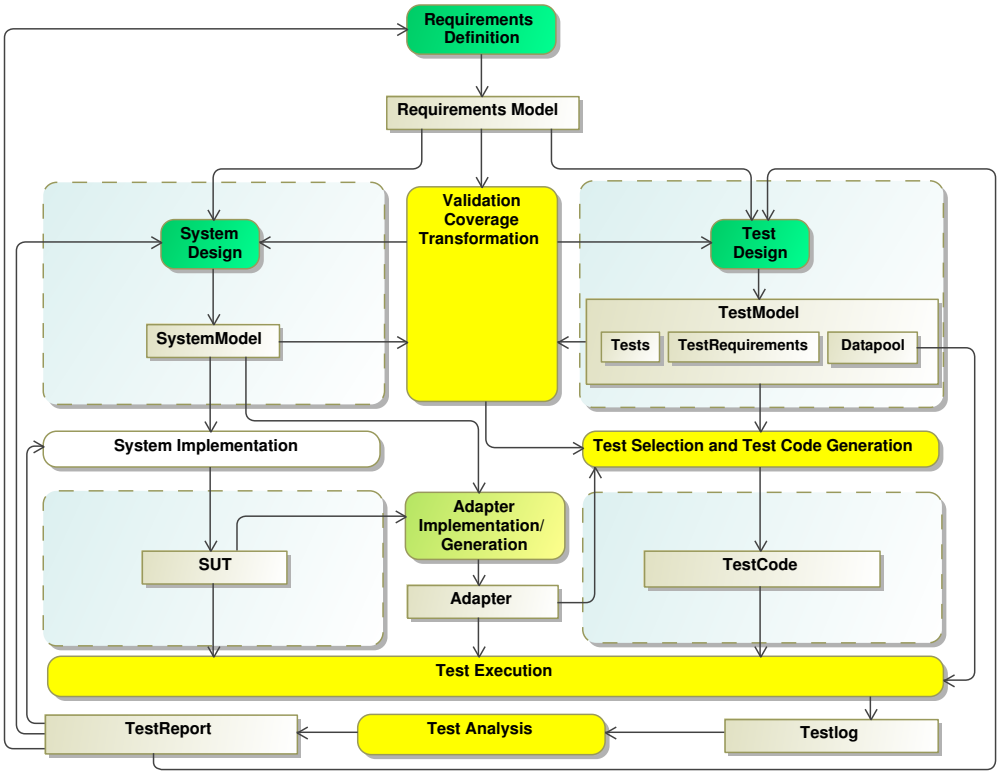


Figure 3.2: Model–driven Testing Process

The test design includes the data pool definition and the definition of test requirements. The system model and the test model, including the tests, the data pool and the test requirements, can be validated for consistency and completeness and checked for coverage. In a system–driven approach tests can be generated from the system model by model–to–model transformations, and in a test–driven approach tests can be integrated in the system model. This validity checks allow for an iterative improvement of the system and test quality. Our development process can be considered as test model–driven because test models are applied for the generation of test code and support the design of the system model. Our methodology does not consider the system development itself but is based on traceable services offered by the system under test. As soon as adapters which may be – depending on the technology[2] – generated (semi–)automatically or implemented manually are available for the system services, the process of test selection and test code generation, i.e., model–to–text transformation can take place. The generated test code is then automatically compiled and executed by a test controller which logs all occurring events into a test log. The test evaluation is done offline by a test analysis tool which generates test reports and annotations to those elements of the system and test model influencing the test result. Test reports and test logs are implementation artifacts that are not important for the overall process but for the practical evalua-

[2]In the tool implementation, adapters for web services can be generated automatically based on a WSDL description, adapters for RMI access can only be generated semi–automatically.

tion. Therefore test reports and test logs have not been considered in the previous section.

In [FBCO+09] we have introduced the term *test story* for our way of defining tests as analogy to the agile term user story defining a manageable requirement together with acceptance tests.

The separation of the test behavior and the test data has been influenced by the column fixture of the Framework for Integrated Test (FIT) [Mug05] which allows the test designer to focus directly on the domain because tests are expressed in a very easy–to–write and easy–to–understand tabular form also suited for data–driven testing.

The manual activities in the TTS testing process are performed by specific roles. In Figure 3.3 these roles and their activities are shown.

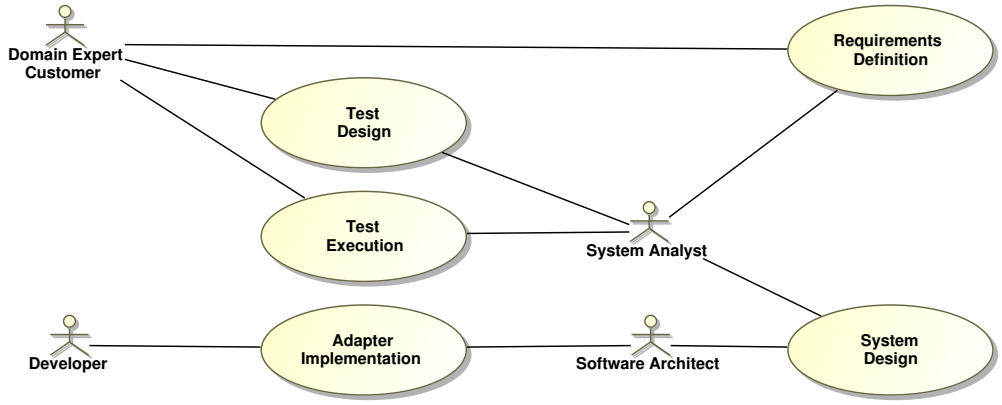


Figure 3.3: Roles in the TTS Process

A *Domain Expert* or *Customer*, which are represented by one role in Figure 3.3, are responsible for the test design and the requirements definition. Additionally, domain experts or customers may initiate the test execution and all related automatic activities (validation, test code generation, test analysis). Domain experts and customers are responsible for the same activities but have different views on testing, i.e., domain experts represent the internal view performing system tests and customers represent the external view performing acceptance tests.

A *System Analyst* is partially responsible for the test design, the requirements definition, and the system design. Additionally, the system analyst may also initiate the test execution and all related activities. The system analyst is especially responsible for the definition of non–functional requirements, e.g., for security or performance and corresponding tests.

A *Software Architect* is partially responsible for the system design and the adapter implementation. The software architect defines interfaces and component technologies in coordination with the system analyst and the developer.

A *Developer* implements adapters if they have to be developed manually.

3.3 TTS Metamodel and Domain Specific Language

In this section we define metamodels for the requirements model (Section 3.3.1), the system model (Section 3.3.2), and the test model (Section 3.3.3), and models for the system under test (Section 3.3.4) and the test system (Section 3.3.5) of Section 3.1. The metamodels are considered as abstract syntax of a domain specific language and extended by a concrete syntax and a semantics to a domain specific language for modeling requirements, systems and tests. Well-formedness rules are defined in natural language as part of the abstract syntax. As concrete syntax, we use UML diagrams annotated with stereotypes and tagged values. Well–formedness rules are defined in OCL. Examples for the concrete syntax of all elements are described in Section 3.4. Our approach is not restricted to the UML notation and tools for it – in principle, also a textual representation in a tool like openArchitectureWare can be used. But UML is a widely used standard and is especially convenient for designing the system model which would be much more complex and error–prone in other notations.

Some parts of the metamodel are refined in Chapter 4 for special purposes such as the evolution of tests.

Our metamodel adheres to the following design principles.

- testers disregard details of the target platform and other implementation specifics during modeling [Har06]
- testers focus on user experience and system boundary – as opposed to the need to model precise internal system behavior by developers

3.3.1 Domain–specific Language for Requirements

In this Section we define the abstract syntax, the concrete syntax and the semantics for the domain–specific language of requirements.

Abstract Syntax

In Figure 3.4 the requirements model of our approach is depicted.

The package Requirements Model defines the element Requirement which is supertype of the types FunctionalRequirement and NonFunctionalRequirement. The type NonFunctionalRequirement may have further subtypes for security (SecurityRequirement) or performance (PerformanceRequirement). In Figure 3.4 the element Requirement is highlighted because it has relations to other packages shown in Figure 3.15. The requirements themselves can be of an arbitrary level of granularity ranging from abstract goals to concrete performance requirements. Requirements are modeled explicitly on the metamodel level to define traceability between requirements and other model elements such as tests.

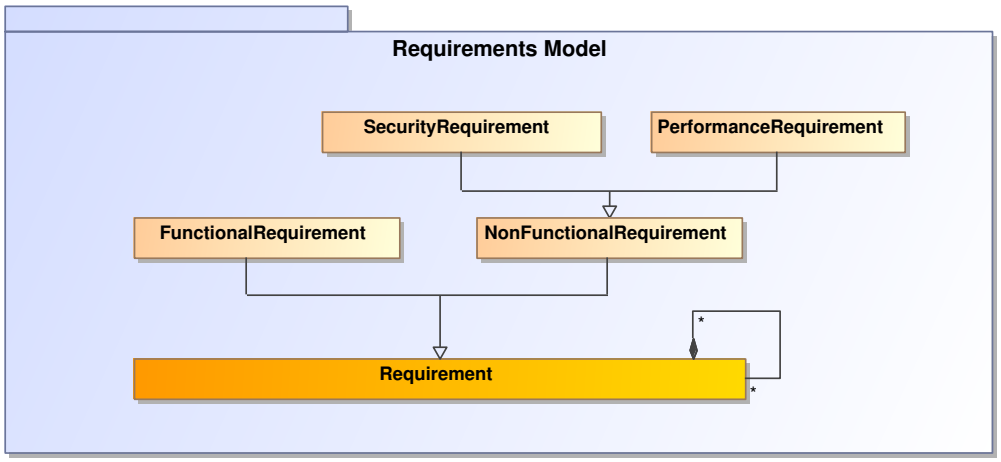


Figure 3.4: Requirements Metamodel of TTS

Concrete Syntax

We use annotated UML class diagrams to represent requirements hierarchies. A class may have the stereotype Requirement, FunctionalRequirement, or an arbitrary type of non–functional requirement such as SecurityRequirement or PerformanceRequirement. The requirements are associated and form a requirements hierarchy. Each requirement has a textual description and is linked to a Service of the system model or to a TestElement of the test model. All these attributes and links are denoted by tagged values in our implementation. Due to the links between requirements and tests, this representation allows us to visualize test results on the requirements level. Our concrete syntax follows the requirements diagrams of SysML [OMG10] but we do not consider different types of relationships between requirements and other model elements such as *satisfies*, *refines*, or *verifies*. It may be the case, that the requirements hierarchy is incomplete by not denoting the root element. In this case the root element is added implicitly and is the super element of all elements without an explicit root element.

It is also possible to represent requirements as hierarchical text or in tables (see [OMG10] for a tabular representation) but we do not consider this representation in the remainder of this thesis.

Our UML profile for requirements models is depicted in Figure 3.5.

The profile defines stereotypes for the elements Requirement, FunctionalRequirement, NonFunctionalRequirement, SecurityRequirement, and PerformanceRequirement extending the UML metaclass Class. A Requirement can be linked to a service and a test by the tagged values service and test, respectively. A NonFunctionalRequirement may additionally be linked to an assertion by a tagged value assertion. The stereotype Requirements is assigned to packages to define a container for requirements.

In Figure 3.6, an example for the concrete syntax of a functional requirement

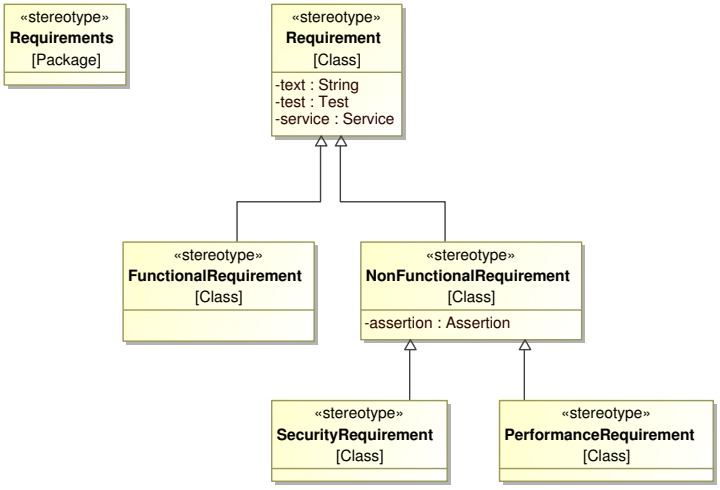


Figure 3.5: UML Profile for Modeling Requirements

and a performance requirement is given. The syntax of requirements follows the representation of requirements in SysML [OMG10].

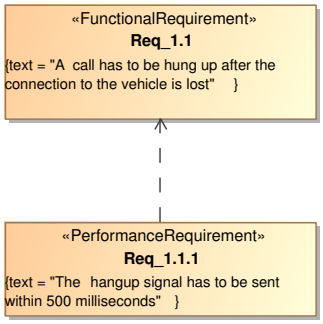


Figure 3.6: Example for the Concrete Syntax of a Functional and a Performance Requirement

Both, the functional requirement `Req_1.1` and the performance requirement `Req_1.1.1` have a textual description. Additionally, `Req_1.1.1` is a subrequirement of `Req_1.1`. Further examples for the concrete syntax of requirements are given in Sections 3.4.1 and 5.2.1.

Semantics

Requirements are textual specifications of a capability or a condition that must (or should) be satisfied by a system. Therefore requirements have a natural informal semantics defined by its hierarchy and its textual description. Additionally, requirements have some kind of translational semantics defined by the behavior of the assigned services and tests.

3.3.2 Domain–specific Language for Systems

In this Section we define the abstract syntax, the concrete syntax and the semantics for the domain–specific language of systems.

Abstract Syntax

Our concise system view of a service–centric system has been inspired by SECTET [Haf08], a framework for security–critical, inter–organizational workflows based on services. In Figure 3.7 the system model of our approach is depicted.

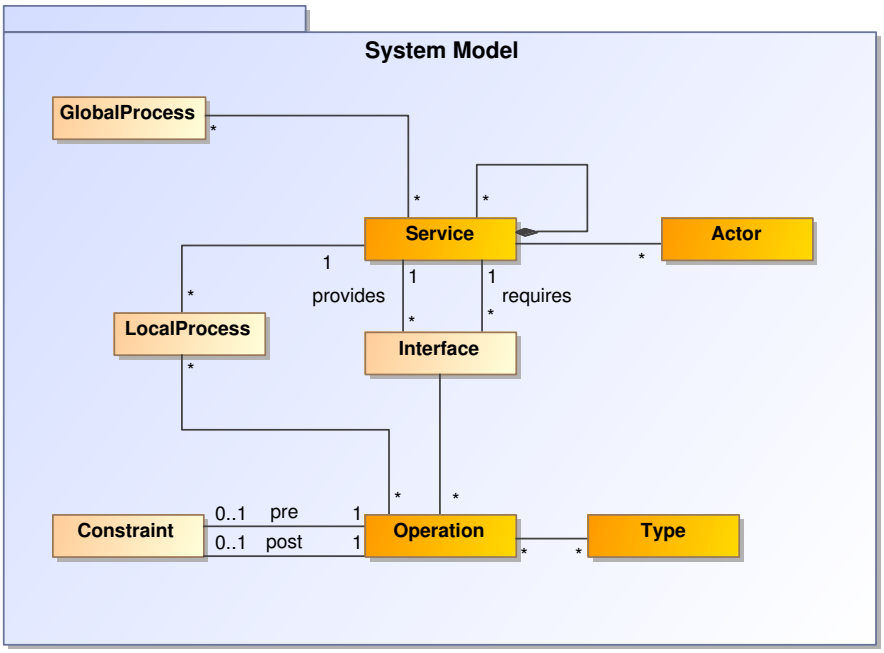


Figure 3.7: System Metamodel of TTS

The package System Model defines Service elements which provide and require Interface elements and are composed of basic services. Each interface consists of Operation elements which refer to Type elements for input and output parameters. Types may be primitive types, enumeration types or reference types. Operations may also have a precondition (pre constraint) and a postcondition (post constraint). Each service has a reference to Actor elements. It is also possible to identify services with a service operation (if there is only one) and to identify them with service calls. Services can therefore be considered as executable use cases. Services may have LocalProcess elements that have a central control implemented by a workflow management system defining its internal behavior. Different services may be integrated into a GlobalProcess without central control. Orchestrations can be modeled as local processes, and choreographies as global processes. In Figure 3.7 the elements Actor, Operation, Service, and Type are highlighted because they have relations to other packages shown in Figure 3.15.

Concrete Syntax

Services as the central concept are either denoted by classes in a UML class diagram or by components in a UML implementation diagram. The types are defined in a UML class diagram. Constraints are written in OCL. Local processes and global processes are modeled by behavior diagrams. Local processes, i.e., workflows with a central control, are typically visualized by state machines or activity diagrams, and global processes, i.e., workflows without a central control, are typically visualized by sequence diagrams. Types are modeled by classes, primitive types or enumerations.

The UML profile for system models is depicted in Figure 3.8.

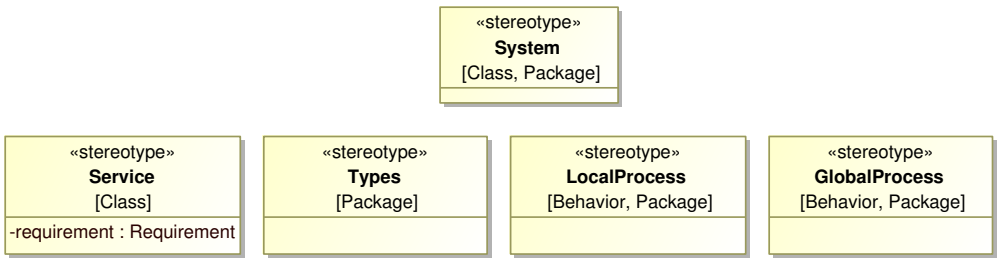


Figure 3.8: UML Profile for Modeling Systems

The stereotypes System, Types, LocalProcess, and GlobalProcess are applied on packages defining the respective elements. The metamodel elements Interface, Operation, Actor, and Constraint are identical with the UML metaclasses with the same names and therefore not considered in Figure 3.8.

The stereotype Service is applied on classes defining services. In Figure 3.9, the concrete syntax of a service is shown.

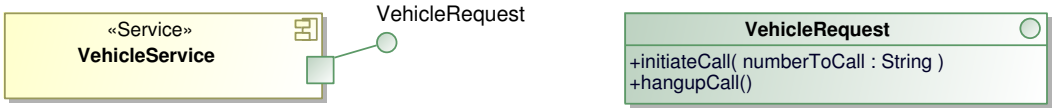


Figure 3.9: Example for the Concrete Syntax of a Service

A service is depicted as a component with provided and required interfaces. The service `VehicleService` in Figure 3.9 provides one interface `VehicleRequest`.

The syntax of the remaining metamodel elements is equal to the syntax of standard UML. In Sections 3.4.2 and 5.2.2 we give further examples for the concrete syntax of all system models elements.

Semantics

The semantics of the system model is defined by an operational and a translational semantics. Services are mapped to running services and their execution defines the meaning of services.

Additionally, we can define a translational semantics by mapping our system model to SECTET and SoaML. The mapping is defined in Table 3.1.

System Model	SECTET	SoaML
Actor	`Role`	`Participant`,`Agent`
Constraint	constraint on operation	constraint on operation
Global Workflow	`Global Workflow`	`Behavior` attached to a `Service Architecture`
Local Workflow	`Local Workflow`	`Behavior` attached to a `Participant`
Operation	operation in `Interface`	operation in `ServiceInterface`
Type	`Message`	`MessageType`
provides interface	`implement` association end	`Service`
requires interface	`use` association end	`Request`
set of provided and required interfaces of a Service	set of `Interface` elements	`ServiceInterface`

Table 3.1: Mapping of the TTS System Metamodel Elements to SECTET and SoaML

Table 3.1 sketches a mapping of all model element types of the system model, i.e., Actor, Constraint, Global Workflow, Local Workflow, Operation, Type, and interfaces to SECTET and SoaML.

3.3.3 Domain–specific Language for Tests

In this section we define the abstract syntax, the concrete syntax and the semantics for the domain–specific language of tests.

Abstract Syntax

The test model of our approach is depicted in Figure 3.10.

The package Test Model defines all elements needed for system testing of service–centric systems. A Test has a Status such as new, retestable or obsolete, some TestRequirement that the test must satisfy or cover, and it consists of TestElement artifacts which are one of the following elements.

- Assertion elements for defining expressions for computing verdicts,
- Call elements, i.e., Servicecall elements for invoking operations on services, or Trigger elements for operations that are called by a service,

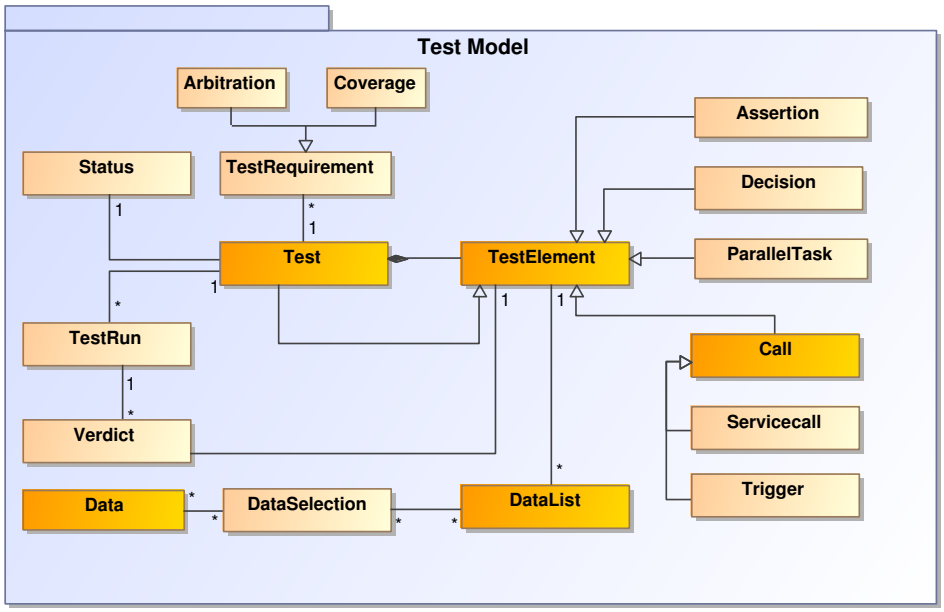


Figure 3.10: Test Metamodel of TTS

- ParallelTask elements for the parallel execution of behavior,
- Decision elements for defining alternatives, and
- Test elements itself.

A test element also has a DataList containing Data elements that may be generated by a DataSelection function. A test has some TestRun elements assigning Verdict values to assertions. The verdict has one of the values pass, fail, inconclusive, or error. *Pass* indicates that the SUT behaves correctly for the specific test case. *Fail* indicates that the test case has been violated. *Inconclusive* is used where the test neither passes nor fails. An *error* verdict indicates exceptions within the test system itself. In the model itself only a pass or a fail can be specified. Inconclusive or error are assigned automatically. This definition of verdicts originates from the OSI Conformance Testing Methodology and Framework [ISO94].

Assertions are boolean expressions that define criteria for computing pass, fail or inconclusive verdicts. Assertions can access all variables in the actual evaluation context. The grammar of assertions is printed in Listing F.1. An editor for writing and testing assertions has been defined in [Aic10].

In Figure 3.10 the elements Call, Data, DataList, Test, and TestElement are highlighted because they have relations to other packages shown in Figure 3.15.

The system model and the test model are created manually or are partially generated from each other. If the system model and the test model are created manually, we ensure that they are consistent with each other and that the test model fulfills some coverage criteria with respect to the system model by validation rules

(see Section 4.3 for consistency, completeness and coverage checks). Alternatively, if the system model is complete then test scenarios, test data and oracles can be generated, or otherwise if the test model is complete, behavioral fragments of the system model can be generated.

Test requirements can contain *global arbitrations* expressed by Arbitration elements that define when a test is successful by restrictions on the verdicts (see Section 4.2), or *coverage constraints* expressed by Coverage elements that define what parts of a model have to be inspected by a test (see Section 4.3).

In the case of a test, a data list defines a test table, i.e., a list of lists. Data selection functions for example randomly generate integers within a specific range. This function is then denoted by `genInt(a,b)` and generates a random integer between the integers `a` and `b`.

The test elements form a tree. The top–level element of type Test can be considered as test suite and each line in the test table assigned to an element of type Test in that hierarchy can be considered as test case. Therefore our generic notation is compatible with the notation in frameworks like JUnit [jun]. Tests are completely recursive and, in principle, there is no limit to the number of levels to which tests can be nested. However, in practice nesting depths greater than three are not applied and it is even not clear what use nesting depths greater than two or three would be.

Concrete Syntax

We employ UML sequence diagrams in Section 5.2 and UML activity diagrams in Section 3.4 for the representation of tests. Test elements are denoted as actions or activities. Both, test requirements and assertions are denoted in an OCL–like notation. Arbitrations and coverage criteria are attached as tagged values to tests, and the pass and fail constraints are tagged values of assertion elements.

The UML profile for test models is depicted in Figure 3.11.

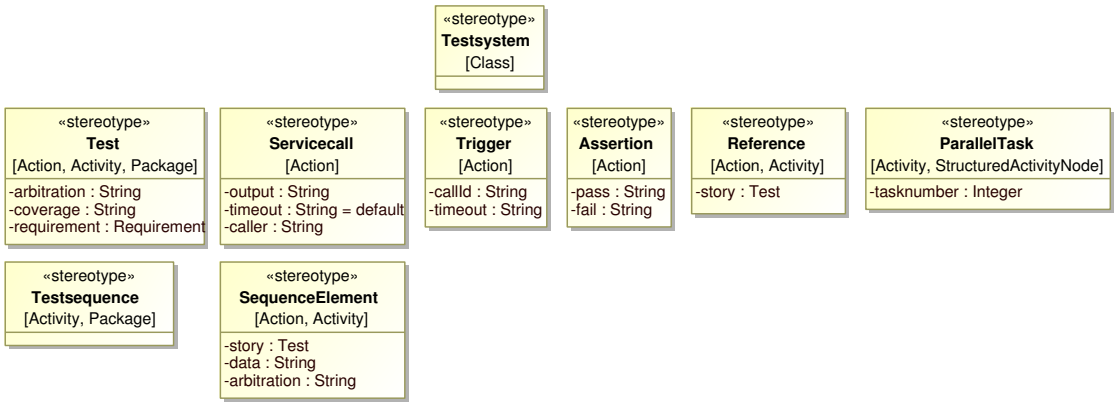


Figure 3.11: UML Profile for Modeling Tests

The UML profile for test models defines the stereotypes Testsystem, Test, Servicecall, Trigger, Assertion, Reference, ParallelTask, Testsequence, and SequenceElement to model tests. The stereotype Testsystem is used to group the test model. The stereotypes Test, Servicecall, Trigger, Assertion, Reference, and ParallelTask are used to define tests. The stereotypes Testsequence and SequenceElement are used to define top–level tests, i.e., test suites and avoid test recursion in the implementation together with the stereotype Reference for invoking tests from other tests. In this context, Test elements describing concrete test scenarios are also called TestStory. All metamodel elements of the implemented profile are shown in Figure B.1.

Figure 3.12 shows examples for the concrete syntax of a Servicecall element, a Trigger element, and an Assertion element.

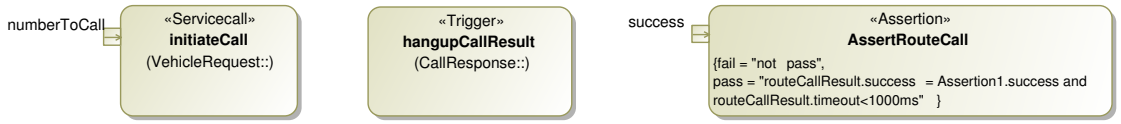


Figure 3.12: Example for the Concrete Syntax of a Service Call, a Trigger, and an Assertion

The service call `initiateCall` is assigned to the operation `inititiateCall` of the interface `VehicleRequest` and the trigger `hangupCallResult` is assigned to the operation `hangupCallResult` of the interface `CallResponse`. The assigned operations are defined in the properties of the model elements. The assertion `AssertRouteCall` includes a `fail` and a `pass` constraint.

The service call `initiateCall` of Figure 3.12 has one input parameter `numberToCall`, and the assertion `AssertRouteCall` has one input parameters `success`. Pins are used to make a test easier to understand. In the test `RouteCall`, pins visualize the free parameters that have to be filled with concrete test data from the test tables. Pins are only shown for readability reasons, because the list of free parameters is extracted automatically from service calls (input parameters) and assertions (free variables of an assertion) when generating the structure of a test table. In Sections 3.4.3 and 5.2.3 we give examples for the concrete syntax of all model elements in the test model.

Test data is denoted in tables. Each column represents an input parameter for the test. Input parameters are all free variables of all actions in a test and the names of input parameters are unique in a single test. We do not use object flows but the correlation between variable names in activities and tables to model data dependencies (see the test `RouteCall` in Figure 3.22 and the corresponding test data table `RouteCallTestData` in Table 3.2). Test runs are listed in tables and the corresponding verdicts determine the color of table rows as in JUnit. In Tables 3.2 and 5.1 examples for test data tables are given.

Semantics

There are various ways how to define the semantics for test models. The semantics can be defined by the mapping of tests in the test model to executable test code as presented in Section 3.5.4. The mapping itself already defines a translational semantics. Furthermore, it is possible to define the semantics of a test operationally by the execution of the generated test code. By analogy to system models the semantics of test models can also be defined by a transformation of test models to a well-defined modeling notation such as the UML 2.0 Testing Profile [OMG05] or test sheets [ABFJ08]. In Section 3.6.2 we sketch a mapping of our test model to the UML 2.0 Testing Profile and in Section 3.6.3 we define a mapping of our test model to test sheets.

3.3.4 SUT Model

The model of the SUT is depicted in Figure 3.13.

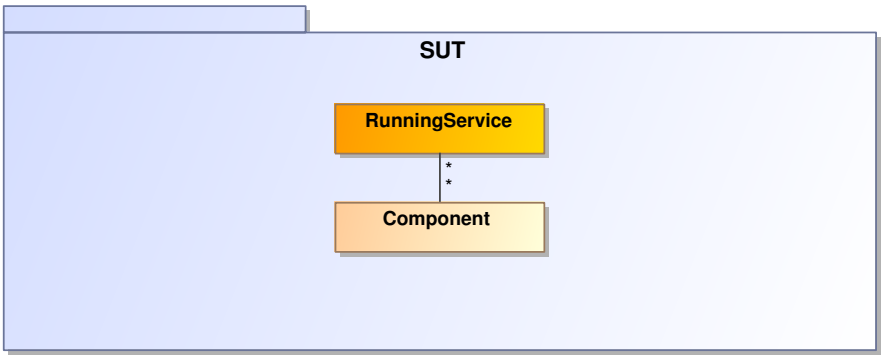


Figure 3.13: SUT Model

A SUT consists of a set of RunningService elements which have been deployed on Component elements. On every component an arbitrary set of services can be deployed and every service can be deployed on various components. In Figure 3.13 the element RunningService is highlighted because it has relations to other packages shown in Figure 3.15.

3.3.5 Test System Model

The model of the test system is depicted in Figure 3.14.

The test system contains ExecutableTest elements with an execute method that can be invoked by the test controller. Each executable test refers to Adapter elements – one for each service invoked by at least one service call in the corresponding test – with an invoke method callable by the executable test. Each executable test has a ConcreteDataList assigned which consists of ConcreteData elements. In Figure 3.14 the elements Adapter, ConcreteData, ConcreteDataList, and ExecutableTest

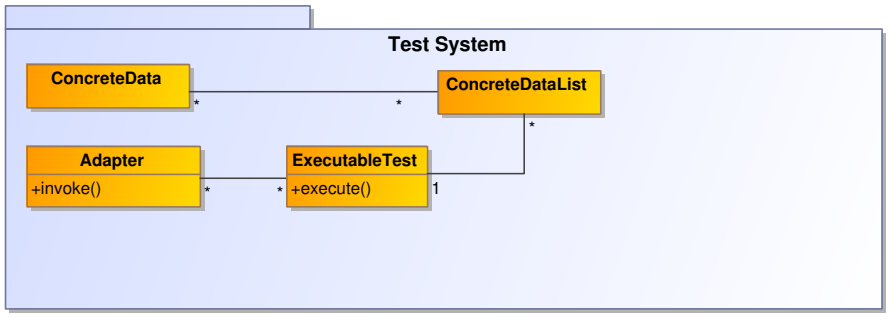


Figure 3.14: Test System Model

are highlighted because they have relations to other packages shown in Figure 3.15. The mapping of the model elements from the package Test System Model to model elements of the package Test Model is shown in Section 3.3.6.

3.3.6 Integration

Figure 3.15 integrates the packages Requirements Model, System Model, Test Model, SUT, and Test System. Model elements that enable traceability have a stronger color and traceability links are bold.

A Requirement can be linked to Service and Operation elements from the system model, and to arbitrary TestElement elements from the test model. Each Call refers to an Operation. A Service may be assigned to a RunningService of the SUT. A Test is associated to an ExecutableTest, Data is linked to ConcreteData, and a DataList is linked to a ConcreteDataList. Every Adapter refers to a RunningService.

Our approach can also be used for testing classical use cases. In that case a use case corresponds to a Service which is informally described by a Requirement. A signature and a corresponding precondition and postcondition can be assigned to the service by identifying it with the signature, precondition and postcondition of a singular operation assigned in a one–to–one correspondence to the service.

3.4 Callmanager Case Study

In this section we present our industrial callmanager application that has been tested with our framework[3]. The *callmanager* is an application in the area of Computer Telephony Integration (CTI) and parts of it have already been tested with unit tests before. But the whole application can currently only be tested by manual and unsystematic tests. Therefore the combination of our methodology with the callmanager provides a good case study but also provides significant savings of effort

[3]The case study was generously provided by Softmethod (`http://www.softmethod.de` [accessed: November 25, 2010]) within the Telling TestStories research project (`http://www.teststories.info` [accessed: November 25, 2010]).

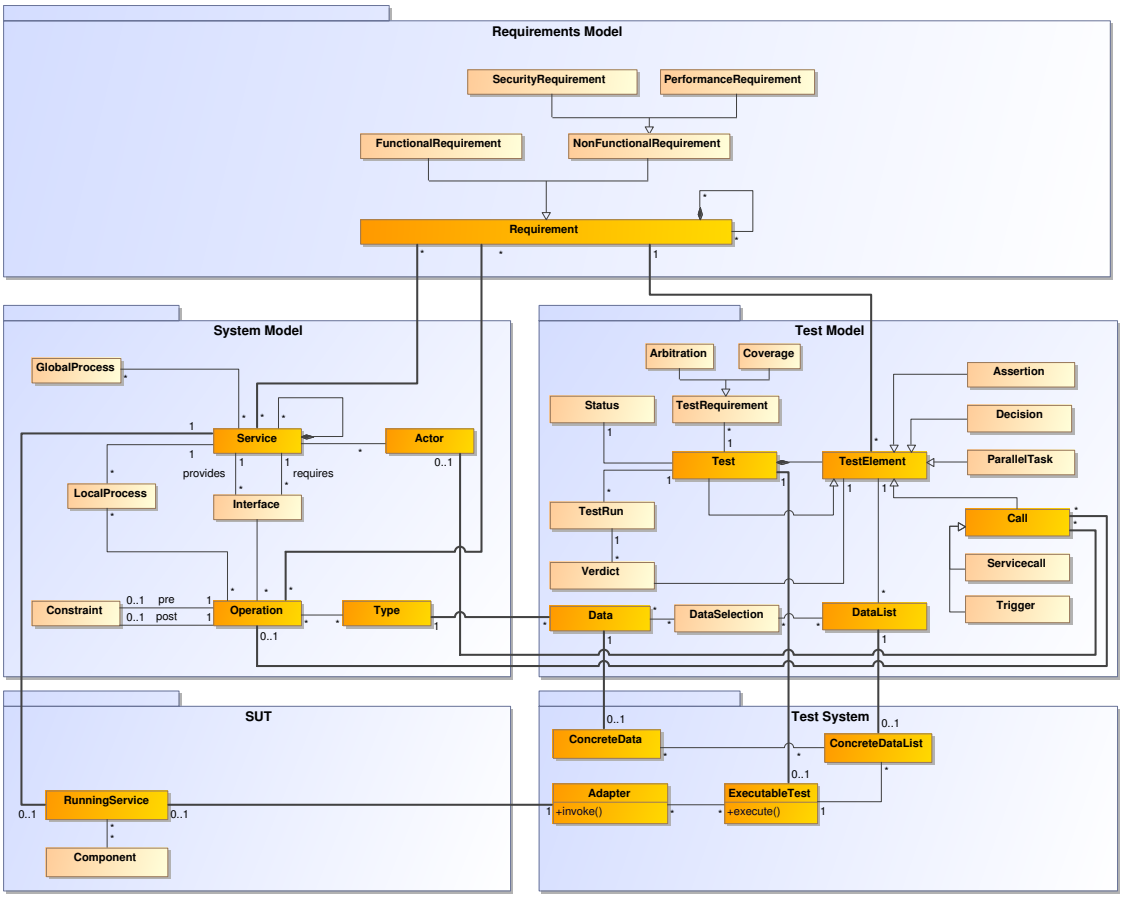


Figure 3.15: Metamodel for Requirements, System and Test

because TTS enables more efficient testing than manual testing.

The callmanager application is a telematics system for the automotive industry developed for a German car vendor. The application has been developed according to the European Unions eCall proposal enforcing any newly homologated car to be equipped with a technical device automatically initiating emergency calls when an accident happened. In this context car specific information has to be transmitted to the call center answering the phone call including the GPS data of the current car position or the number of fired airbags. After the data transmission the call center agent has the ability to speak with the people inside the crashed car. The callmanager and its environment are depicted in Figure 3.16.

The *callmanager* is a standalone server application bridging the actual telephone system on the car vendor's side based on private branch exchange (*telephony system*) with the remaining infrastructure of the car vendor (*backend*) like database systems holding car specific data. To this end, the callmanager provides several operations as a service to the backend. This includes routing the call from the car to another destination, like the locally responsible police station (*routeCall*) or the termination of a call from a car (*hangupCall*), e.g., if the car accidentally calls the call center and

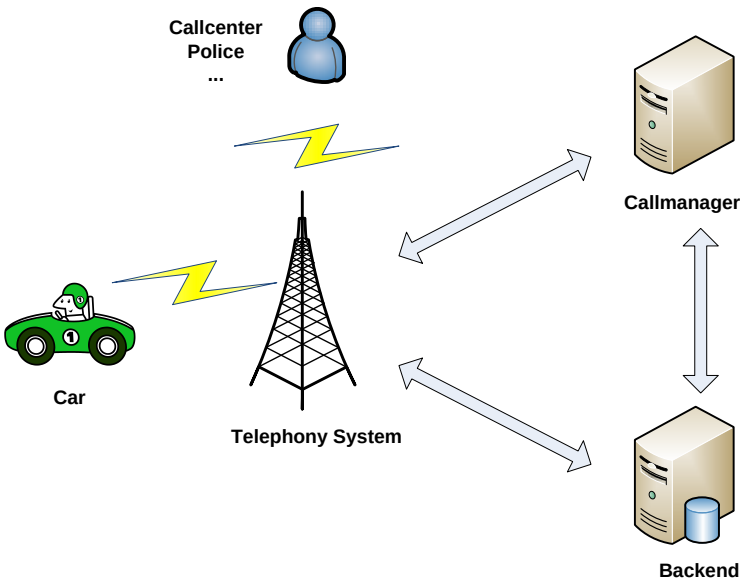


Figure 3.16: Overview of the Actors and Components of the Callmanager Application

there is no emergency or if all necessary actions were initiated. The callmanager application bridges the telephony system with the remaining backend of the car vendor, including the actual call center in order to fully control the handling of the call.

In almost all scenarios the car executes the initial call and therefore the callmanager is equipped with an asynchronous notification mechanism to inform the backend of incoming calls or the termination of the call maybe due to signal loss of the mobile device. Not only the initial notification but all services are executed on asynchronous service calls due to the nature of telephone systems. As already mentioned the callmanager has a wide variety of possible use cases. In this case study we focus on the following concrete scenario: We assume a car has an accident, collects all necessary data and initiates a call to the call center. This lets the callmanager come into action and start its internal procedure of collecting the transmitted data, which results in the notification of the backend of an incoming call. After the initial call was received the scenario will route the call to another destination, which can be the local police station. Afterwards the backend decides to terminate the call by hanging it up. As earlier mentioned the whole communication is done in an asynchronous way, such that each service call is acknowledged by asynchronous events. As an additional challenge the callmanager is a standalone server application running on a dedicated server separated from the telephony system. Therefore the TTS framework has no possibility to startup the callmanager as a child process. The adapter concept of TTS was used to connect the callmanager SUT to the Test Controller. In Section 3.5 architecture of the TTS tool and its application to the callmanager case study are presented.

In the subsequent sections we define the requirements model (Section 3.4.1), the

system model (Section 3.4.2), the test model (Section 3.4.3), the system under test (Section 3.4.4), the test system (Section 3.4.5) and traceability of the callmanager application (Section 3.4.6).

3.4.1 Requirements Model

The requirements model provides a hierarchical representation of the functional and non–functional requirements to the system. In Figure 3.17, the requirements for routing a call are depicted.

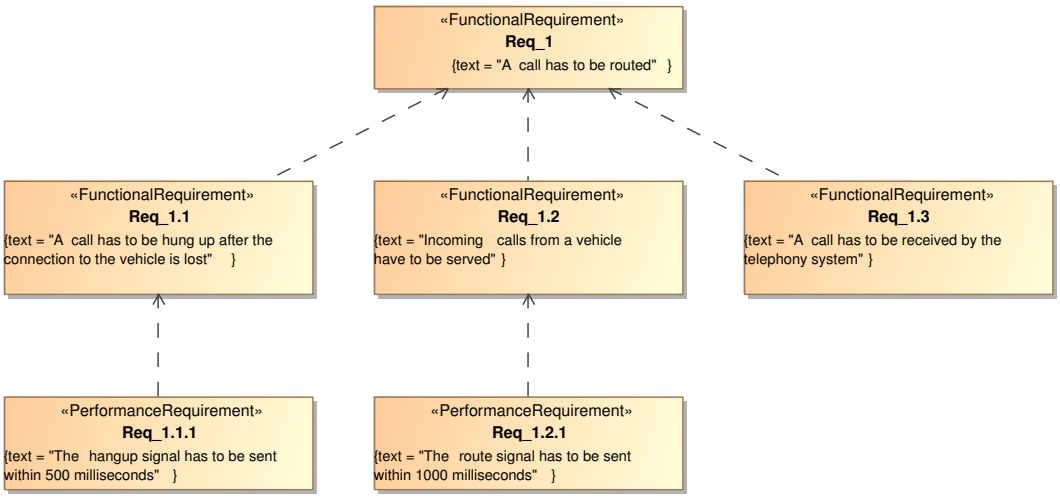


Figure 3.17: Requirements of the Callmanager Application

The functional requirements (`Req_1`, `Req_1.1`, `Req_1.2`, `Req_1.3`) have the stereotype FunctionalRequirement and the non–functional performance requirements (`Req_1.1.1`, `Req_1.2.1`) have the stereotype PerformanceRequirement which is a subtype of the metamodel element NonFunctionalRequirement.

3.4.2 System Model

In this section we define the system model of the callmanager. We define the types of the system, the services, and its processes.

Types

Services have interfaces with operations. These operations use Java standard types (primitive types, String), which are not modeled explicitly and user defined types such as the complex type `ByteArray` and the enumeration type `ErrorTypes` of Figure 3.18.

The complex type `ByteArray` contains binary data trasmitted by the operation `sendData` of the interface `CallCommand`. The enumeration `ErrorTypes` lists possible error types, e.g., `NOLINE` if no telephone line is available, for call command and

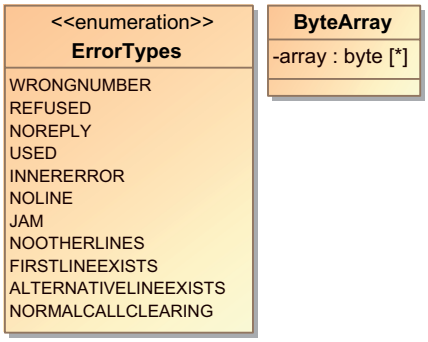


Figure 3.18: Types of the Callmanager Application

response operations. The callmanager contains additional enumeration types which are not relevant in this case study.

Services and Interfaces

The system consists of four services providing and requiring interfaces. The services are depicted in Figure 3.19 and the interfaces in Figure 3.20.

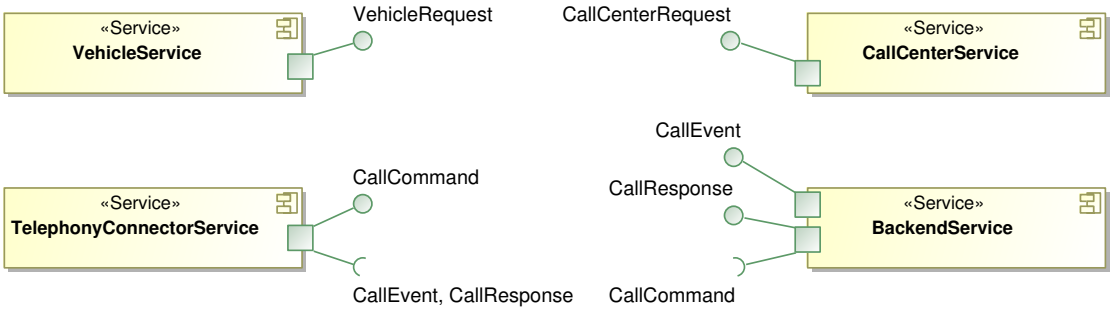


Figure 3.19: Services of the Callmanager Application with Provided and Required Interfaces

The system provides the following services:

- The `VehicleService` represents the car that has an integrated telephone to initialize a call or to hang it up.
- The `CallCenterService` represents a call center that receives routed calls from the vehicle via a telephony system.
- The `TelephonyConnectorService` is the service under test and models the callmanager. Technically, the callmanager is a standalone server application bridging the actual telephone system on the car vendor's side based on private branch exchange.

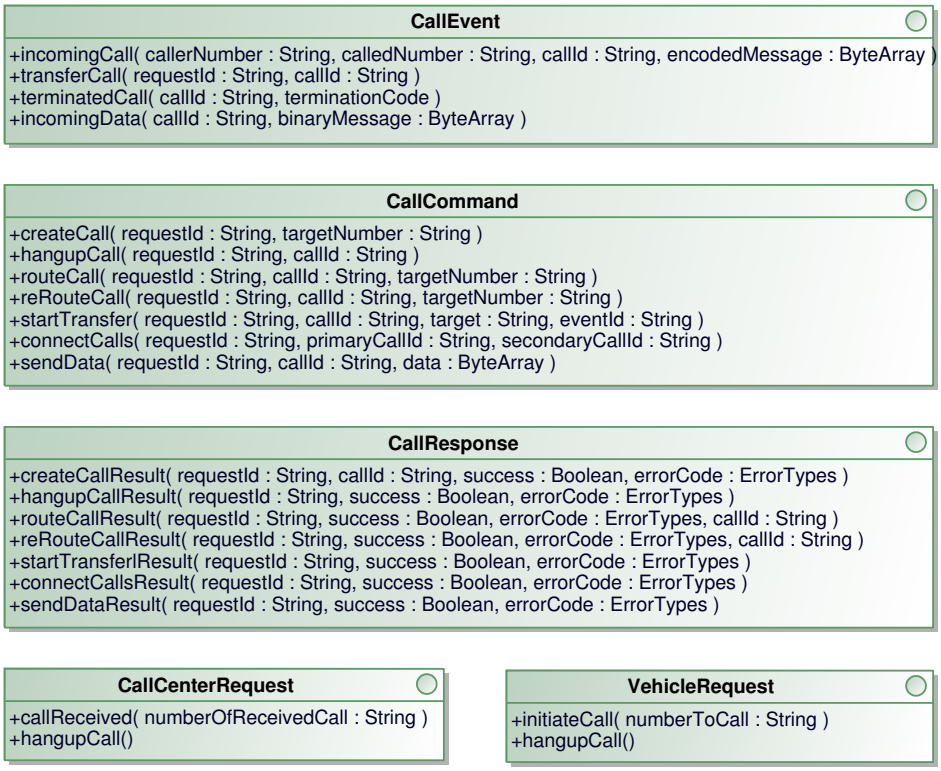


Figure 3.20: Interfaces of the Callmanager Application

- The `BackendService` models the remaining infrastructure of the car vendor (backend) like database systems holding car specific data. To this end the callmanager provides several operations to the backend. This includes routing the call from the car to another destination, like the locally responsible police station (`routeCall`) or the termination of a call from a car (`hangupCall`), e.g., if the car accidentally calls the call center and there is no emergency or if all necessary actions were initiated. The TelephonyConnector application bridges the telephony system with the remaining backend of the car vendor, including the actual call center in order to fully control the handling of the call. The backend is the only service without an underlying telephony infrastructure.

Each operation may have a precondition and a postcondition denoted in OCL. The operation `initiateCall` for instance has the precondition of Listing 3.1.

```
callerNumber.size>=0 and calledNumber.size>=0 and
  callId.size>=0 and callerNumber <> calledNumber
```

Listing 3.1: Precondition of the Operation `initiateCall`

The precondition of Listing 3.1 specifies that all input parameters of type String have a length greater than 0 and that the number of the caller differs from the called number.

In this case study the operations are asynchronous calls without return values. Therefore there are no postconditions defining a relationship between input and output parameters. In general failed preconditions are an indicator for an irregularity in the environment and failed postconditions indicate irregularities in the operation under test [MFC+09]. We do not need the full capabilities of pre- and postconditions in our example but we just automate the application of oracles by evaluating assertions. Therefore in our implementation, if a precondition of a service call evaluates to false we assign a verdict inconclusive, and if a postcondition evaluates to false we assign a verdict fail. Additional pre- and postcondition based testing techniques for test generation are not considered in this thesis but can principally be integrated into our framework. We sketch such approaches in Section 4.5.

Processes

A local process for the `VehicleService` is shown as state machine in Figure 3.21.

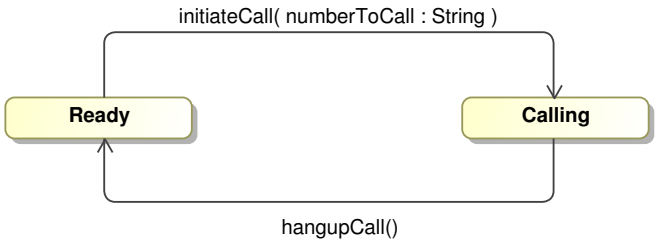


Figure 3.21: Vehicle States

The local process defines two internal states (**Ready** and **Calling**) for the vehicle. If the operation `hangupCall` is executed in state **Calling** then the state machine goes to state **Ready**. If the operation `initiateCall` is executed in state **Ready** then the state machine goes to state **Calling**.

A global process for routing a call can be denoted by a UML activity diagram. We do not consider a global workflow here but in other scenarios global workflows can be the basis for the generation of tests or for checking the consistency of tests.

3.4.3 Test Model

In this section we define a test model for the callmanager case study referring to the requirements and system model defined above. The tests are denoted as UML activity diagrams and the corresponding data in tables.

Figure 3.22 shows one parametrized test `RouteCall` that is fed with data from Table 3.2 and invoked in the test `TestSuite` shown in Figure 3.23.

In the test `RouteCall`, a telephone call is initiated (`initiateCall`), and a trigger in the `BackendService` receives the call (`incomingCall`). Then the call is

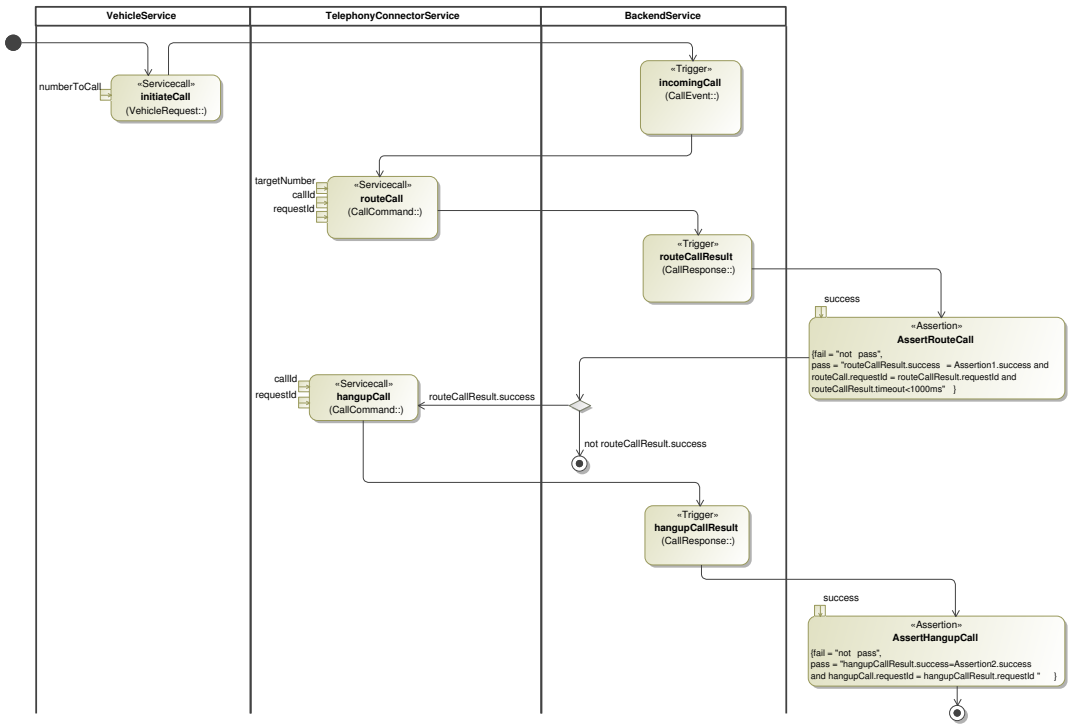


Figure 3.22: Test `RouteCall` for Routing a Call

routed via the service call `routeCall` and the trigger `routeCallResult`, and finally the telephone call is terminated via the service call `hangupCall` and the trigger `hangupCallResult`. Assertions check whether the routing and hangup return the expected values and react within time. Assertions are always evaluated by the test controller, regardless of whether they are assigned to a lifeline or not. The pins visualize the free variables for which concrete input data is provided by test data tables and which are defined by input parameters of service calls and free variables of assertions. As mentioned before, in an activity diagram pins are only shown for readability reasons because the information they represent is automatically extracted from the calls and assertions to generate a test data table.

Parameter	**Test1**	**Test2**
`initiateCall.numberToCall`	`SIP:1234`	`SIP:1234`
`routeCall.requestId`	`genInt(1,100)`	`genInt(1,100)`
`routeCall.callId`	`incomingCall.callId`	`incomingCall.callId`
`routeCall.targetNumber`	`SIP:transfer`	`SIP:invalid`
`AssertRouteCall.success`	`true`	`false`
`hangupCall.requestId`	`routeCallResult.requestId`	
`hangupCall.callId`	`routeCallResult.callId`	
`AssertHangupCall.success`	`true`	

Table 3.2: Test Data `RouteCallTestData` for Test `RouteCall`

The test data `RouteCallTestData` in Table 3.2 defines two test cases for the test

`RouteCall`. For determining the parameter `routeCall.requestId` a data selection function `genInt` is used to generate an integer between `1` and `100` which serves as request identifier. After a test execution a verdict is assigned to each assertion as so called test run. If there is only one verdict assigned to one line in a test table, a test run can be visualized by coloring the line of the table as depicted in Figure 3.30. The verdict pass is visualized by a green line, the verdict fail by a red line, and the verdict inconclusive by a yellow line.

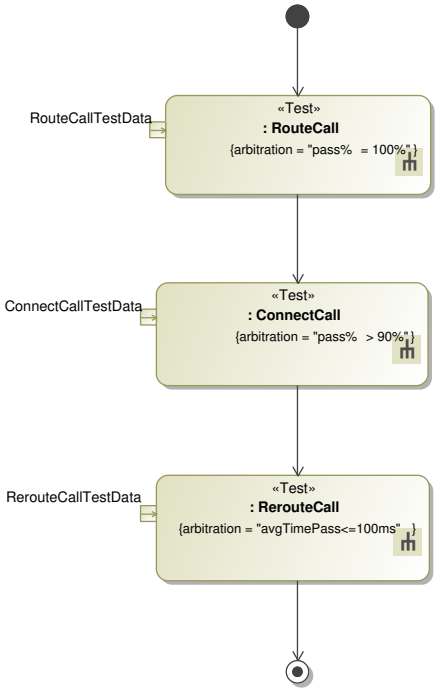


Figure 3.23: Overall Test for Routing a Call, Connecting a Call, and Rerouting a Call

The test `TestSuite` in Figure 3.23 calls the parametrized tests `RouteCall` and `ConnectCall` with the data tables `RouteCallTestData` and `ConnectCallTestData`. The assertions check the percentage of pass which has to be 100% for `RouteCall`, more than 90% for `ConnectCall`, and an average time for passed test cases below 100 milliseconds. Basically the tests define activity flows of the system. The attached stereotypes define the operational semantics of each node: Servicecall elements are invoked by the test execution engine, Trigger elements are invoked on the test execution engine and Assertion elements define checks for computing verdicts. The test `CallmanagerTestSuite` is the top–level test containing all other tests in a hierarchy. This test may have an attached coverage constraint to define that all operations of the system have to be invoked in at least one service call.

3.4.4 SUT

In the system under test each running service corresponding to a modeled service is deployed on one specific component. There is a vehicle simulator component

for the VehicleService, a voice manager component for the BackendService, a callmanager component for the TelephonyConnectorService. The call center is simulated by a simple Java class. All interfaces are provided as Java interfaces.

3.4.5 Test System

The test system is presented in Section 3.5 on the tool implementation. The adapter implementation in the callmanager case study is based on the Java Remote Method Invocation (RMI) technology [rmi]. The visualization of the test results in the diagrams and tables is also explained by the tool implementation of the case study in Section 3.5.

3.4.6 Traceability

Traceability between requirements and other artifacts generated during the system development and testing process is essential for systems evolution and testing because in both cases all affected artifacts have to be checked or even changed. Our approach comprises traceability between the requirements model, the system model, the test model and the running services. Each test element is assigned to a requirement, and implicitly to its super–artifacts. Tests can therefore be considered as executable requirements because they assign an executable model to requirements. If the execution of a test story fails, then the requirements themselves and all its super–artifacts fail. In Table 3.3 we define the mapping of the test elements of the test RouteCall to the requirements, i.e., the traces between the packages Test and Requirements in Figure 3.15.

TestElement	Type	Related Requirements
RouteCall	Test	Req_1
initiateCall	Servicecall	Req_1.3
incomingCall	Trigger	Req_1.3
routeCall	Servicecall	Req_1.2
routeCallResult	Trigger	Req_1.2, Req_1.2.1
Assertion1	Assertion	Req_1.2, Req_1.2.1
hangupCall	Servicecall	Req_1.1
hangupCallResult	Trigger	Req_1.1, Req_1.1.1
Assertion2	Assertion	Req_1.1, Req_1.1.1

Table 3.3: Traceability between Test Elements and Requirements

Due to this mapping, if the execution of a service call via its adapter fails, the failure can be propagated via the test element to specific requirements.

Technically we have implemented traceability between the requirements and the system model or the test model via tagged values storing links to the associated test elements or requirements.

3.5 Tool Implementation

In this section, we describe the TTS tool implementation[4] that has been applied on the case study presented in Section 3.5. The explanations are based on the tool description in [FZFB10].

For the tool implementation we have introduced the concept of a test sequence (see UML profile of the implementation in Figure B.1) which represents the top–level test. A test sequence consists of test sequence elements each referring to one test story including the corresponding test data table and the arbitration. We use a UML profile for modeling the requirements, the system and the tests which allows us to reuse existing UML concepts and tools. The notation is based on the concrete syntax presented in Section 3.3. By introducing test sequences the concept of a test is not completely recursive any more. But this is not a practical restriction because nesting of more than three levels do not occur in practice.

The tool is designed to fulfill the following general and specific requirements to testing tools:

- General requirements to testing tools defined in [Har00]:
  - The tool is easy to learn and use.
  - The output of the tool is presented in a clear and understandable way.
  - The tool is automated as much as possible.
- Specific requirements according to the TTS methodology:
  - The tool integrates the complete testing process of Figure 3.2.
  - The tool integrates all artifacts and processes for system testing of service–centric systems.
  - The tool presents test results on various abstraction levels, namely log files, test tables, models and reports.
  - The tool is test model–driven.
  - The tool supports iterative test development.
  - The tool is model–driven and tabular, i.e., it supports the integrated modeling of tests with models and tables.
  - The tool supports test model validation by consistency, completeness, and coverage checks.
  - The tool supports various adaptation technologies by a generic adapter concept.
  - The tool is based on existing modeling technologies and exchange formats.

[4] The TTS tool implementation is available at `http://teststories.info` [November 25, 2010].

Additionally, our tool has been implemented for the integrated model–based design, generation, execution and evaluation of tests within Eclipse [ecl]. Developed as a set of Eclipse plugins, the tool consists of various components setting up the whole environment, to keep a high level of modularity. The main components are depicted in Figure 3.24 and correspond to the activities of our testing methodology depicted in Figure 3.2.

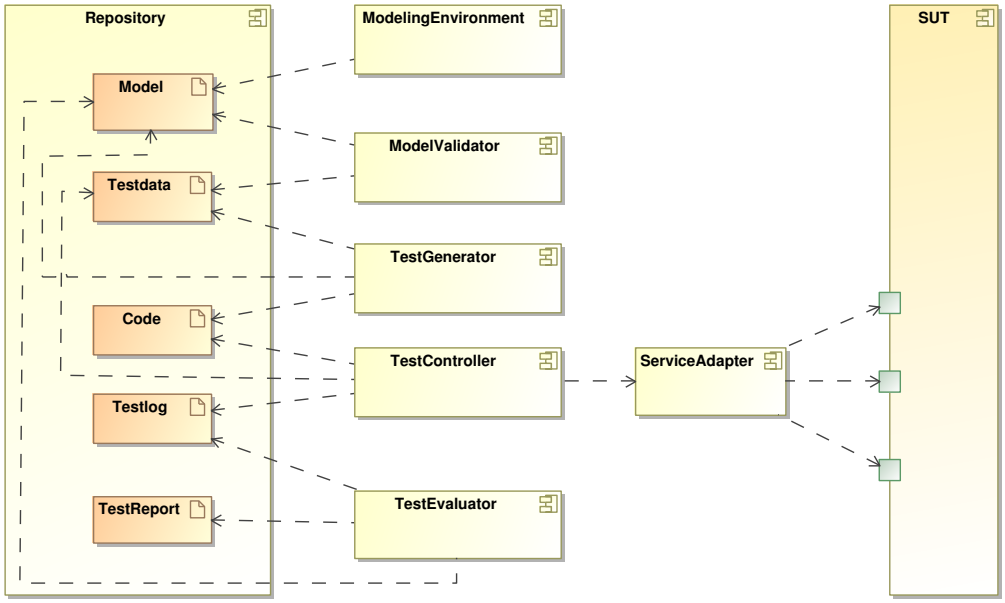


Figure 3.24: System Architecture of the TTS Tool Implementation

The **Modeling Environment** is used for designing the requirements model, the system model and the test model. It processes the workflow activities *Requirements Definition*, *System Model Design*, and *Test Model Design*.

The **Model Validator** is based on the SQUAM framework [COGIOT06] and uses OCL as constraint language. It processes the workflow activity of *Validation*, *Generation*, and *Integration*.

The **Test Code Generator** generates executable Java code from the test model. It processes the workflow activity *Test Selection and Test Code Generation*.

The **Service Adapters** are used by the test controller to invoke services on the system under test (SUT). They can be created manually or generated automatically depending on the service technology. Adapters correspond to the workflow activity *Adapter Implementation/Generation*.

The **Test Controller** executes the tests against the SUT. It processes the

workflow activity *Test Execution*.

The **Test Evaluator** generates test reports and visualizes test results within the models. It corresponds to the workflow activity *Test Analysis*.

In its modularity, the tool is only bound to the Eclipse framework. The various components can be exchanged by more custom triggered extensions as the tool follows established practices (test data modeling in XML, test case modeling in XMI). In the subsequent subsections, the components will be outlined in more detail. Additional technical details concerning the implementation are presented in [Zec09].

Figure 3.25 shows the Eclipse testing project for the callmanager application.

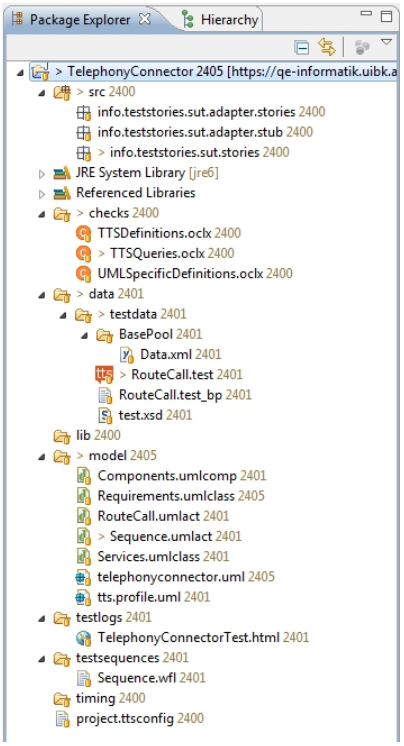


Figure 3.25: TTS Project in Eclipse

The Eclipse project contains the repository of a testing project. It keeps all artifacts except the *TestReport* elements which are generated automatically based on templates and the test log data. *Code* is stored in the directory *src*, the *Model* in the file *telephonyconnector.uml* of the directory *model*, *Testdata* in the directory *data*, *Testlog* data in the directory *testlogs*. The remaining directories and files contain metadata, generated data and libraries.

3.5.1 Modeling Environment

The requirements, system, and test models are denoted as UML models based on the TTS UML profile and stored in the XMI format. Our modeling language presented

in Section 3.3 is implemented as UML profile within the UML2Tools [uml]. The UML2Tools implementation of the profile is depicted in Figure B.1. Stereotypes are used to label model elements and tagged values are used to define additional attributes. In the following paragraphs we present the model environment by the model of the callmanager case study.

The requirements are visualized in a class diagram and form a hierarchy. Low level functional and non–functional requirements are aggregated to high level requirements. This representation is analogous to requirements diagrams of SysML [OMG10] and guarantees that requirements are integrated into the model which simplifies the implementation of traceability.

The requirements for routing a call are depicted in Figure 3.26.

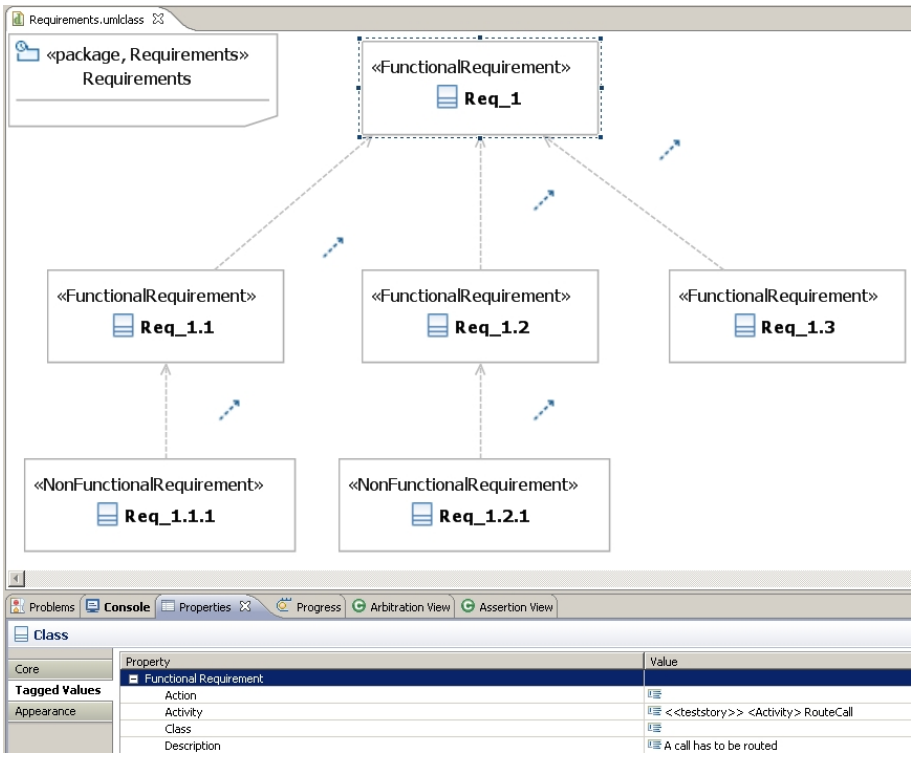


Figure 3.26: Requirements to the Callmanager Application

We have modeled a requirement for routing a call (`Req_1`) and its parts, including non–functional performance requirements to hangup a call within 1000ms (`Req_1.1.1`) and to send the route signal within 1000ms (`Req_1.2.1`). Compared to the representation of requirements in Figure 3.17, the textual description of a requirement is not directly annotated to the requirement but written in the properties editor in the lower part of Figure 3.26.

In the system model types are visualized as class diagrams, processes as state machine, and services as classes with their provided and required interfaces. To provide an overview, in Figure 3.27 an extract of the services of the callmanager

application is depicted.

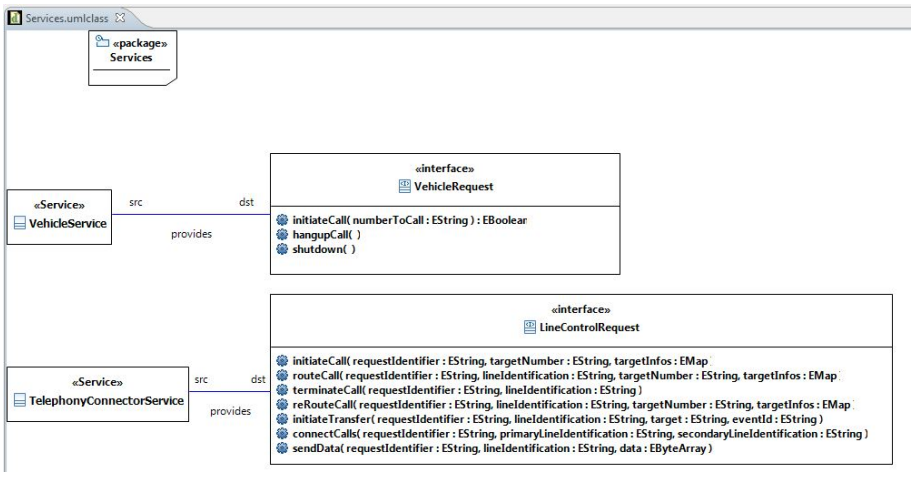


Figure 3.27: Extract of the Services of the Callmanager Application

The service *VehicleService* provides the interface *VehicelRequest*, and the service *TelephonyConnectorService* provides the interface *LineControlRequest*. In the implemented diagram shown in Figure 3.27, the separated visualization of services in Figure 3.19 and of interfaces in Figure 3.20 is integrated in one diagram.

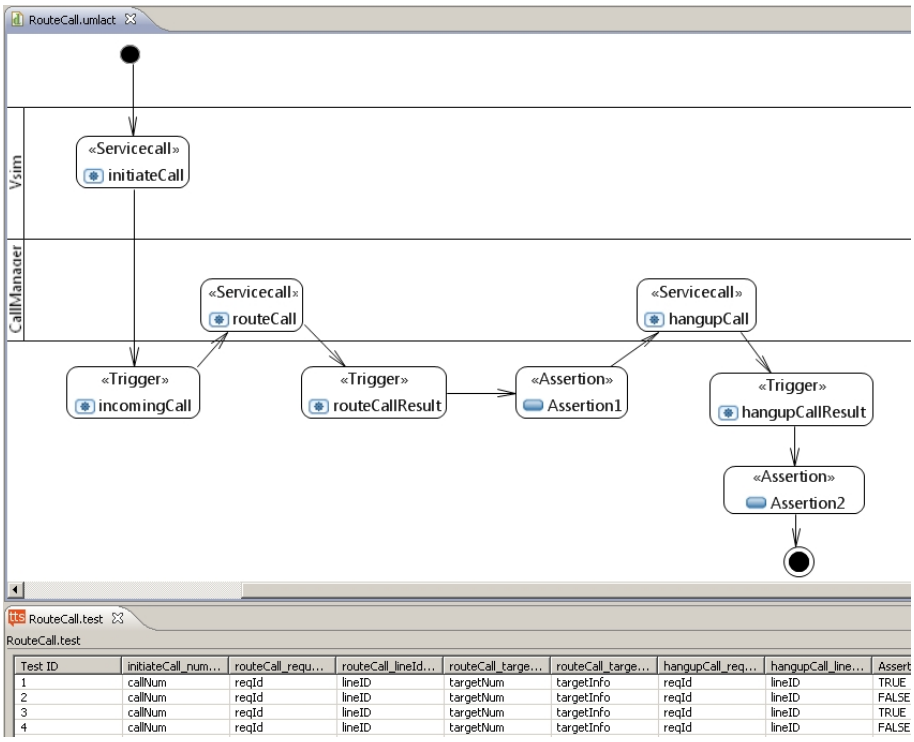


Figure 3.28: Test `RouteCall` for the Callmanager Application

Tests are modeled as activity diagrams or sequence diagrams. Test sequences or high level test suites are modeled as activity diagrams. In Figure 3.28, a test in the TTS tool for the callmanager application is shown. In the upper part of the

test in Figure 3.28, the test `RouteCall` for routing a call is depicted. After the car service (`Vsim`) initiated a call, the callmanager routes the call and terminates the call. The results of these calls are triggered on the test controller. Intermediate assertions check whether the result provided by the trigger equals to the expected one. Each test may have test data which is defined for the test story `RouteCall` in the table `RouteCall.test` depicted in the lower part of Figure 3.28.

The test `RouteCall` of Figure 3.28 is the implementation of the test `RouteCall` of Figure 3.22. Both representations have the same meaning but some syntactical differences. The tool implementation does not support the visualization of pins for service call and assertion elements. Instead, the free parameters are shown in the properties of service call and assertion elements or in the corresponding data table. Due to practical requirements, we have used alternative names for the lifelines of the implemented test `RouteCall` in Figure 3.28 each representing the same service as in Figure 3.22. `VehicleService` is represented by `Vsim` in the tool, `TelephonyConnectorService` is represented by `CallManager`, and the trigger elements of the `BackendService` are executed by the test controller and therefore not depicted in a separate lifeline. Finally, `AsserRouteCall` has the identifier `Assertion1` in the tool implementation, and `AssertHangupCall` has the identifier `Assertion2`.

Test stories and their corresponding data files are executed as test sequence elements as shown in Figure 3.30 which defines an additional arbitration to define a global verdict specifying when all tests of a story have passed.

Test data is modeled in tables. This makes the modeling of tests more complex, because tests are then not directly integrated within the UML model representation. Nevertheless we use tables for test data representation and not model elements because this is cumbersome and cannot be handled practically for data–driven tests. We support the automatic generation of table templates for each test. We also integrate checks for test data within our consistency and coverage checks. In Figure 3.29 a test data table for the route call test of Figure 3.22 is shown.

Test tables follow the approach of the FIT framework [Mug05], which allows for assigning concrete test data (in our case concrete object identifiers) to every input parameter and free variables of assertions occurring in a test. The various test data objects referred in the test tables are provided from a data context, holding instances of the concrete data objects needed for system testing. By making use of the *Inversion of Control* (IoC) container of the Spring Framework [spr], TTS allows for modeling complex data objects and not only primitive types.

Hence, these test data tables allow for describing and modeling the execution flow of test cases in a very fine grained and deterministic way, by manually manipulating the object contents. Additionally, it is possible to trace down the erroneous execution of test cases to the test data level by making use of the IoC's user-defined object identifiers.

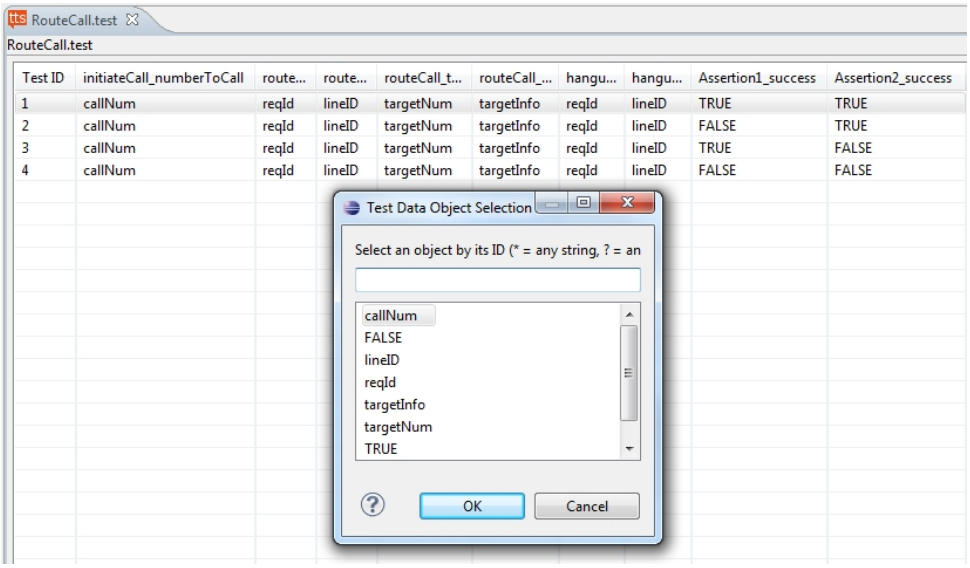


Figure 3.29: Test Data Table Specifying Test Cases

3.5.2 Model Validator

The model validator provides support for consistency and completeness in OCL. This is the basis for the definition and generation of high–quality system and test models. Consistency checks guarantee that the model is non–contradictory and completeness checks guarantee that all necessary information is defined in the model. Consistency and completeness are the prerequisite for successful test code generation and execution. Therefore the test code generator uses the model validator to check test model validity prior to test code generation. In Listing 3.2 a sample top–level query for assuring test model validity is denoted.

```
context Model

  query checkIsValidTestModel:
    let result:Boolean =
      if Package.allInstances()->
       any(o|o.profileIsTypeOf('Test')).oclIsUndefined()
      then false else
        Package.allInstances()->
        any(o|o.profileIsTypeOf('Test')).isValidTestModel()
      endif
    trigger true
    message result endmessage
```

Listing 3.2: OCL Query in SQUAM for Validity of a Test Model

The OCL query of Listing 3.2 checks for all packages of type test whether the test model is valid.

The model validator is based on the SQUAM framework (Systematic Quality Assessment of Models) [COGIOT06] and the query in Listing 3.2 is aligned with its syntax. In Section 4.3 these validity checks are motivated and explained in more detail and in the Appendix C the source code of some consistency and completeness queries is listed. The model validator can also be used for the definition of coverage criteria. The extension mechanism of SQUAM which allows for defining external Java calls, enables the integration of test data into OCL based coverage checks.

The community edition of SQUAM[5], which is integrated into TTS, provides basic features for defining queries and an interface for executing them.

SQUAM contains features for editing OCL expressions like syntax highlighting, code completion, or code formatting and considers UML profiles for defining and executing queries. Especially the support of UML profiles in SQUAM is needed for the TTS integration because TTS is based on a UML profile. The query editing support of SQUAM is very convenient for the definition of domain– and project–specific test requirements by test managers which are not OCL experts. SQUAM has itself been implemented as a set of Eclipse plugins and therefore it can easily be integrated into TTS.

3.5.3 Service Adapters

The communication of the test controller with the SUT is encapsulated inside generated or manually–implemented adapters, tailored to the concrete SUT. By encapsulating service invocations and the data mapping, the modeling of test stories and the generation of test code is eased (see Sections 3.5.1 and 3.5.4) because it can be abstracted from technical details. The TTS tool supports the automatic generation of adapters for services providing a proper WSDL description [CMRW03] of their operation interfaces. The WSDL description itself can be generated from a proper system model. Additionally, there is one reference implementation for Java RMI–based adapters [rmi] in the callmanager case study.

All generated or manually–implemented adapters have to implement the interface `IAdapter` which is printed in Listing 3.3.

```
public interface IAdapter {
 public Object
   invoke(String servicename, Object... arguments);
}
```

Listing 3.3: Service Adapter Interface

[5]The professional edition of SQUAM provides additional modeling features like templates or reporting not relevant for the validation purpose of TTS.

The interface contains one method `invoke` with a service name and a list of arguments as input parameters. A tailored adapter has to implement only one single method `invoke` to be ready–for–use in the tool environment during the testing process. TTS supports the manual or automatic development of an adapter for different technologies such as web services, RMI or CORBA as much as possible.

Adapters are key artifacts for traceability because they are the link between test models and the running system. Each service in the system model has an assigned adapter which invokes the corresponding running service of the SUT. The test generator generates an invoke–method call for each service call in a test. For instance, in Listing 3.4 a generated invoke–method call for the service call `routeCall` is shown.

```
CallManager.invoke("routeCall",
                   context.retrieveValue("routeCall_requestId"),
                   context.retrieveValue("routeCall_callId"),
                   context.retrieveValue("routeCall_targetNumber")
                   );
```

Listing 3.4: Service Adapter Invocation

On a `CallManager` adapter object, the method `routeCall` is invoked. The input parameters `requestId`, `callId`, and `targetNumber` are retrieved from the test context which holds the test data during the test execution. Adapters can also contain mocks if specific services (or service operations) are not available but the invoking test should be executed. Mocking is especially useful if a specific service is temporarily unavailable or if its invocation causes cost.

3.5.4 Test Generator

The test generation constitutes one of the core components of the tool environment. The code generator is implemented based on oAW [oaw] which is a framework for domain modeling and model–driven development, allowing to realize model–to–text transformations as needed in our case. The architecture and the functionality of the test generator has been published in [FFZB09] and is sketched in this section.

The test code generator enables to generate executable Java code out of the modeled test stories. In its implementation the generator traverses each model element of the test model and produces test code in Java. Our template–based model–to–text transformation generates test code which is composed of predefined code templates, called and evaluated during the visitation of the various model elements. The templates for test code generation are depicted in Listing D.1. For pre– and postconditions aspects in the notation of AspectJ [asp] are generated to be evaluated as aspects on service calls during the test execution.

The above–mentioned evaluation of the code templates focuses on the pro-

cessing of the applied stereotypes (see Section 3.5.1) defined as a part of the tool environment. As already mentioned earlier, those stereotypes define element specific tagged values, containing the required information for proper test case generation. To assure that the test model meets the requirements posed by the test code generator, prior to test code generation, the test model is validated as presented in Section 3.5.2. The validation rules assure that on the one hand side, the test model only invokes service calls defined by the SUT and uses data types processable by the SUT. On the other side, those OCL rules are used to check, that the test model is valid, in a sense, that the tagged values of the test specific stereotypes are set.

In our approach the tests are executed offline because the tests are strictly generated before they are executed.

3.5.5 Test Controller

The test controller processes the test code which is executed with concrete test data and logged afterwards. Additionally, the engine also provides a communication interface to the SUT to realize asynchronous service communication, i.e., the execution of a workflow in the SUT whose completion is indicated by a callback method onto the invoking client.

The test controller itself consists out of various components explained in the subsequent paragraphs.

Data Management. This component is responsible for data management and provides the test data objects referenced in the various test data tables (see Section 3.5.1). Again, the IoC container of Spring is used to provide the test data objects to the test engine. Inside this container the objects are retrieved by their unique object identifier used in the test data tables.

Event Handling. This component is used by the whole tool environment to process events thrown during test execution, i.e., a *VerdictEvent* to indicate the evaluation of an assertion during test execution. Additionally, this component generates tables as in the FIT framework [Mug05], illustrating the successful or erroneous execution of a set of test data.

Assertion Evaluation. This component evaluates assertions and computes verdicts. In contradiction to JUnit and the like, Telling TestStories allows complex test data to be used, and hence, also this component allows for iterating through complex object structures for test evaluation.

Timing. This component ensures that timeouts are met for service responses which is needed for handling asynchronous service calls. This is encapsulated inside this component by ensuring that responses to specific service invocations receive the test controller within a pre–specified duration. Responses are assigned

to corresponding service invocations by their method signatures.

In the execution phase, the test controller passes through three states. In an initial state, the test workflow is parsed and according to its contents, *test tasks* are generated encapsulating the modeled test cases. After completion of creating the tasks the engine enters its second, main state in which the tests are executed against the SUT. In a third and final state, the engine generates the above–mentioned result table containing the test outcomes, after all events have been processed (due to concurrent execution of the test controller events have to be queued for proper handling).

3.5.6 Test Evaluator

The test evaluator is responsible for evaluating the results of a test run. The evaluation of a test run is performed by assigning verdicts based on the evaluation of assertions, the evaluation of preconditions and postconditions of service calls, and errors originated in the test environment. As in the FIT framework, our test evaluator colors test case lines in test tables green if a test case passes or red if a test case fails. Additionally, test case lines are colored yellow if a test case neither passes nor fails. Figure 3.30 shows a colored test table for our case study.

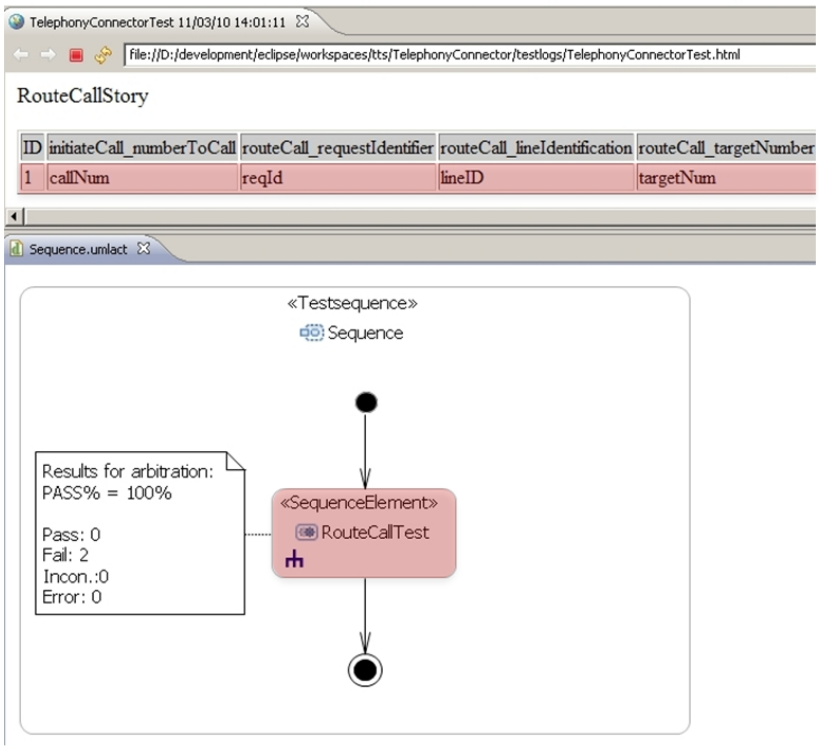


Figure 3.30: Test Result of a Test Run

If a failure can be assigned to a specific model element, our tool is able to color it in the activity diagram. The sequence elements are colored green or red

depending whether its arbitration is fulfilled or not. The sequence element *RouteCall* of Figure 3.30 is colored red because its two test cases fail but its arbitration requires passing of all its test cases.

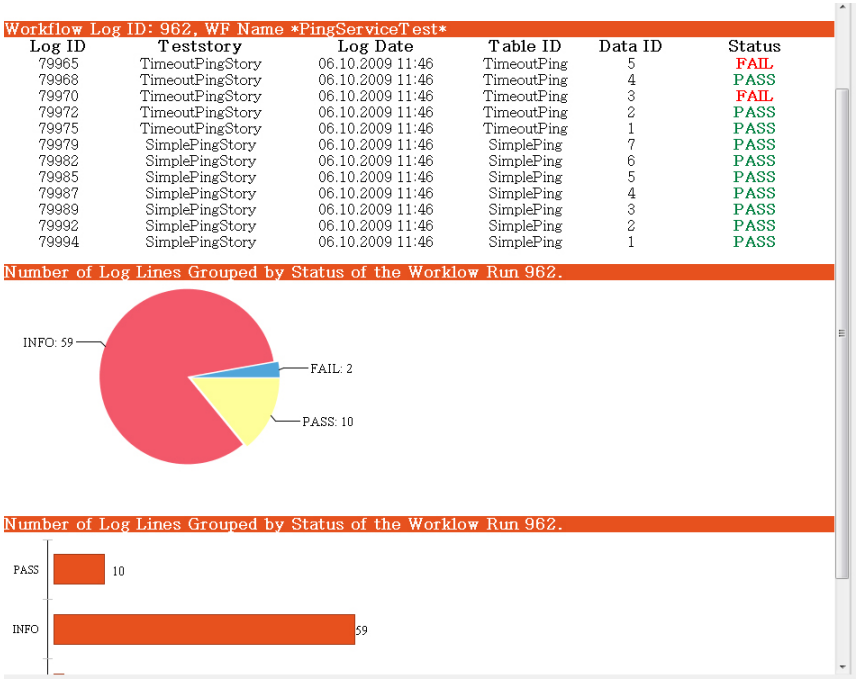


Figure 3.31: BIRT Report for Test Results

We have also integrated high–level test reporting based on BIRT [bir] which generates different types of graphical test summary reports as depicted in Figure 3.31. The report lists all test cases and their test result and depicts it in a pie chart and in a bar chart.

3.6 Related Approaches

TTS is a tool–based methodology that covers the model–based test design, generation, execution, evaluation, and management of tests. According to the classification presented in Figure 1.2, TTS is a model–driven and tabular test design method. In this section we cover the most prominent system test design approaches, which are closely related to TTS, i.e., which differ from it – according to the classification in Figure 1.2 – only in formality or tabularity. Namely we consider the Framework for Integrated Test (Section 3.6.1), the UML 2.0 Testing Profile (Section 3.6.2), and test sheets (Section 3.6.3). Each of these approaches has a relationship to TTS and can be – at least partially – mapped to TTS. We provide an overview of each approach and relate it to TTS. It is intended to not consider programmatic approaches like JUnit [jun] as related approaches because they reside on a lower level of abstraction, i.e., the code level.

3.6.1 Framework for Integrated Test

In this section we provide an overview of the Framework for Integrated Test and relate it to TTS.

The Framework for Integrated Test (FIT) [Mug05] uses a tabular representation of tests, mainly intended for acceptance testing. FIT allows the test designer to focus directly on the domain requirements and business rules because the test logic is expressed in a very easy–to–write and easy–to–understand tabular form. The desired relationship between the inputs and outputs determine the form of the table and how test logic and test data are specified. In order to transform the tables into invocations to the SUT, FIT introduces *fixtures* based on the Adapter pattern [GHJV95]. Three different kinds of fixtures are available to define how the cells of the table are interpreted, i.e., mapped to the functionality of the SUT:

- A *ColumnFixture* maps columns in the test data to fields or operations of its subclasses
- A *RowFixture* invokes functionality in the SUT. An algorithm matches the specified values in the rows of the table to the returned objects
- A *ActionFixture* is used to test GUIs. The first column contains one of a small number of commands and the subsequent columns contain values interpreted by the particular command. The rows are thus interpreted as a sequence of commands to be performed in the specified order. The default commands are *start* to open a particular GUI screen, *enter* for entering a certain value to a field in the GUI screen, *press* to invoke the operation of a button, and *check* to read a value from a GUI and to check it against a given value.

Test results are shown in the table by labeling cells red or green depending on whether the corresponding relationship has been satisfied by the SUT. This allows all relevant test information to be displayed on one page and makes the test result easy to see.

A simple example of a FIT test using a ColumnFixture is shown in Figure 3.32

Division		
numerator	denominator	quotient
1000	10	100.00
-1000	10	-100.00
1	1	0.00
0	1	0.00

Figure 3.32: FIT Test for Division

The example defines a test for a division service. The service accepts two parameters and calculates their quotient. The rows “numerator” and “denominator”

represent the parameters that are passed to the SUT. The expected results are listed in the "quotient" row. In the result FIT table in Figure 3.33 the labeled cells are shown.

Division		
numerator	denominator	quotient
1000	10	100.00
-1000	10	-100.00
1	1	0.00 *expected*
		1.00 *actual*
0	1	0.00

Figure 3.33: FIT Test Result for Division

The first two and the last test pass, and the third test fails. For the failing test the expected and the actual result are printed.

The underlying motivation for this approach is to encourage customers, business people, and even customers to write tests, since tests are high–level. This simplicity is gained at the price of reduced expressive power, e.g., there is no control flow, but only sequences of operation invocations and not all features of complex parameter types can be handled. A FIT test needs to be manually transformed into executable code, known as fixture, which is itself an error prone process that also requires programming knowledge.

TTS is based on the tabular test representation of FIT and its motivation to encourage domain experts to write tests. But in TTS tests are directly executable via automatically generated adapters and TTS tests are more expressive. Additionally, TTS has the advantages of a model–driven approach, its graphical representation of tests and early test definition and validation.

3.6.2 UML 2.0 Testing Profile

In this section we provide an overview of the UML 2.0 Testing Profile and relate it to TTS by a mapping between the metamodel concepts.

For platform independent test modeling the UML 2.0 Testing Profile (U2TP) [OMG05] has been standardized. It defines four concept groups for test architecture, test behavior, test data and time. Table 3.4 gives an overview of the most important concepts of the U2TP.

As UML models primarily focus on the definition of structure and behavior they provide only limited means for describing tests. The supplementary concepts of the testing profile were identified on the basis of the existing test specification languages TTCN–3 and JUnit. The concept groups are presented in the following paragraphs.

Test Architecture Concepts	Test Behavior Concepts	Test Data Concepts	Time Concepts
SUT	Test objective	Wildcards	Timer
Test context	Test case	Data pool	Time zone
Test component	Defaults	Data partition	
Test configuration	Verdicts	Data selection	
Test control	Validation action	Coding rules	
Arbiter	Test log		
Scheduler			

Table 3.4: U2TP Testing Concepts

Test Time. This concept is related to time constraints and observations within a test specification. These concepts in addition to the existing UML 2.0 time concepts are needed to provide a complete test specification. A timer controls the test execution and reacts to start and stop requests as well as timeout events. The graphical syntax for timer actions is adopted from the Message Sequence Charts used by TTCN–3.

Test Data. The data a test is based on is supplied via so–called data pools. These either have the form of data partitions (equivalence classes) or as explicit values. The test data is used in stimuli and observations of a test. A Stimulus represents the test data sent to the SUT in order to assess its reaction.

Test Behavior. A test specifies the interaction of the SUT with test components in order to realize the test objective. The test is specified in terms of sequences, alternatives, loops, stimuli, and observations from the SUT. During execution a test verdict is returned to the arbiter. The arbiter assesses the correctness of the SUT and finally sets the verdict of the whole test.

Test Architecture. The concepts of the test architecture are related to the structure and the configuration of tests, each consisting of test components and a test context. Test components interact with each other and the SUT to realize the test behavior. The test context encapsulates the SUT and a set of tests as well as an arbiter interface that evaluates and generates the verdict, and a scheduler interface that controls the execution of the test cases. The composite structure of the test context is referred to as the test configuration.

More details on the informal semantics of the concepts can be found in [OMG05]. U2TP does not offer the possibility to directly execute the described tests. In order to execute the tests a transformation or mapping to another language needs to be employed. Such language mappings exist for JUnit and TTCN–3.

In [BRDG+07] an operational semantics is defined via a mapping from the U2TP to JUnit [jun] and TTCN–3 [WDT+05] — two common test execution languages that have been used for testing web services [SDA05]. Even though the TTCN–3 description language itself is not executable, it provides tools for this task. Based on a given description, a compiler creates an executable program in either Java, C, or C++. However, additional checks and modifications in the target language have to be performed before the tests can be executed. Not all concepts of U2TP can be transformed to JUnit which is specialized for component tests. A direct transformation from U2TP to Java – as from TTS to Java – is also possible but not common.

Table 3.5 defines a mapping from our test metamodel to the UML 2.0 Testing Profile.

Test	**U2TP**
Assertion	`ValidationAction`, `Arbiter`
Data	`Data`
DataList	`Class`
DataSelection	`DataSelection`
Decision	decision node
ParallelTask	`Coordination`
Service	`TestComponent`
Servicecall	`Stimulus`
Status	attribute of `TestContext`
System	`SUT`
Test	`TestContext`, `TestControl`, `TestCase`
TestRequirement	`Scheduler`, `TestObjective`, `Arbiter`
TestRun	`TestLog`, `LogAction`
Trigger	`Observation`
Verdict	`Verdict`

Table 3.5: Mapping of Model Elements from the Package Test to U2TP

Most of the metamodel element mappings in Table 3.5 are one–to–one mappings. However, there are some metamodel elements like Assertion or Test that cannot be transformed one by one. A Test is mapped to a `TestContext` and `TestControl` in the case of a high–level test which corresponds to a TestSequence in the UML profile of our implementation. A low–level test is mapped to a `TestCase` which is linked to a behavior diagram corresponding to the behavior of TTS tests.

For instance, the test sequence of the callmanager case study depicted in Figure 3.23, which contains a sequence of three tests *RouteCall*, *ConnnectCall* and *RerouteCall*, is mapped to the following U2TP test context and test control shown in Figure 3.34 and Figure 3.35.

The test context in Figure 3.34 contains three test cases, each corresponding to one test story in our TTS test. Additionally, the test context contains three data pools corresponding to the data tables of each test story.

Figure 3.35 shows the test control resulting from the mapping of the test se-

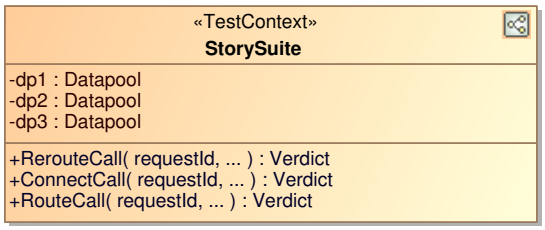


Figure 3.34: U2TP Test Context Mapping Target

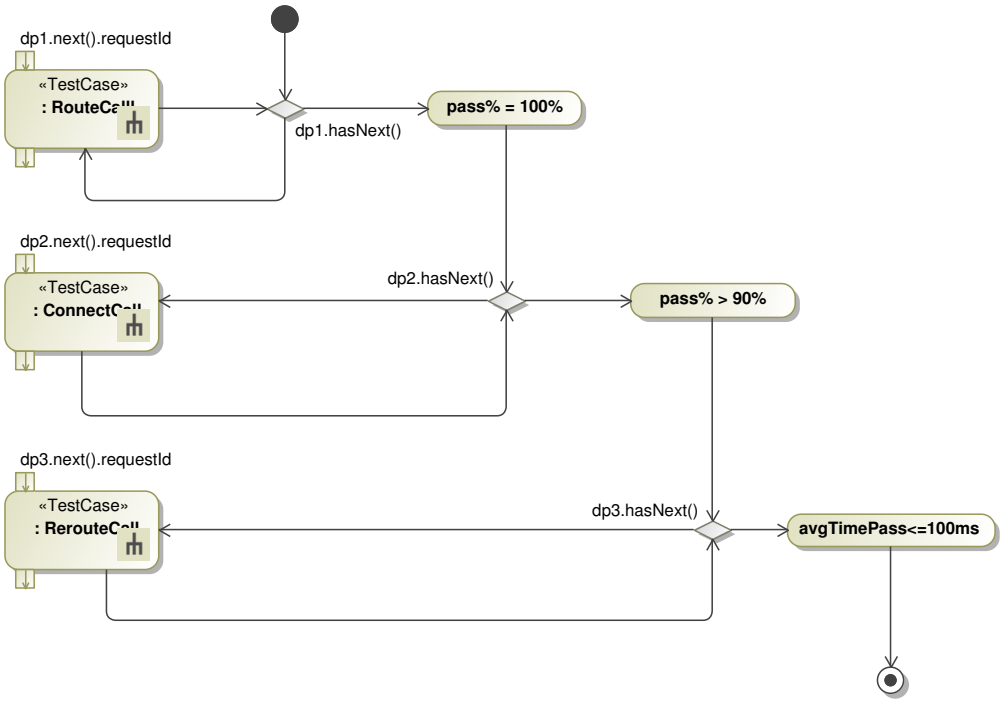


Figure 3.35: U2TP Test Control Mapping Target

quence in Figure 3.23. It contains the three test cases corresponding to the test stories and, their arbitrations and the input data from data pools.

Each data pool, implements the iterator pattern providing the operations *next()* which return the next element of the data pool, and *hasNext()* which is true if there is another element in the data pool. The test story behavior itself is quite similar in TTS and the U2TP. U2TP uses plain UML actions and validation actions for the assertions. As mentioned in Section 3.3, the mapping can be used to define a translational semantics for the test model of TTS.

Both, the U2TP and TTS provide a graphical notation that can be transformed to executable test code in Java. U2TP has a strong connection to TTCN–3 because a mapping to TTCN–3 has been defined and the definition of U2TP has strongly been influenced by TTCN–3. U2TP is therefore best suited for application areas

where TTCN–3 has also been successfully applied, mainly in protocol testing and for distributed systems. U2TP tests are typically not executable, i.e., directly transformable to Java test code but translated via TTCN–3 test code to Java [ZDSD05]. TTS tests are directly translated to Java test code which guarantees traceability between all artifacts.

U2TP introduces much overhead and test specific information in the test models, which makes them hard to understand. U2TP has not been defined for direct test result reporting whereas the tabular description of tests in TTS follows the FIT approach of direct tabular test result representation. TTS is reduced to a small number of concepts for system testing of service–centric systems usable by domain–experts.

In [OMG05] a model–driven approach to testing SOA with the U2TP is defined. The approach focuses on web services technology and uses the complete set of U2TP concepts which makes it quite hard to understand for non–test–experts. Our approach can be mapped to that approach but additionally, TTS is designed for arbitrary service technologies, provides a service–centric view on tests, supports the tabular definition of tests and guarantees traceability between all involved artifacts.

3.6.3 Test Sheets

In this section we provide an overview of test sheets and relate them to TTS. We present test sheets based on the explanations in [ABB10] and use the callmanager case study (see Section 3.4) as running example to sketch the mapping from test sheets to TTS.

Test sheets [ABFJ08] come in two basic forms: *input test sheets* and *result test sheets*. Input test sheets define the input values and the expected results of the SUT. As such they represent test cases. Result test sheets, on the other hand, describe the results of a test execution. They contain all the information of the input test sheet, plus additional information about the reaction of the SUT to the stimulus.

The basic input test sheet for the test of *RouteCall* (see Figure 3.22) is depicted in Figure 3.36.

Each row of the test sheet represents a call to the system's functionality. The input parameters and output values are always separated into two distinct areas separated by the so called invocation line depicted as vertical double line between the columns `E` and `F` in Figure 3.36.

In the lines `1` to `3` objects for the `Vehicle`, the `TelephonyConnector` and the `Backend` are created. They correspond to the partitions in TTS. In line `4` the object created by `Vehicle.create()` is referenced by `F1` and the operation `initiateCall` with the input literal `''SIP:1234''` is invoked on it. The calls of `incomingCall` and `routeCall` are similar – `rand()[1...100]` generates an randomized integer number between `1` and `100`. The call `getRouteCallResult` in line `7` has a timeout with value `1000` as input parameter. In lines `8`, `9`

	A	B	C	D	E	F
1	'Vehicle	create				
2	'TelephonyConnector	create				
3	'BackendMock	create				
4	F1	initiateCall	"SIP:1234"			
5	F3	incomingCall				
6	F2	routeCall	"SIP:transfer"	F4	rand()[1...100]	
7	F3	getRouteCallResult	1000			
8	F7	getCallId				D6
9	F7	getSuccess				true
10	F7	getRequestId				E6
11	F2	hangUpCall	D4	rand()[1...100]		
12	F3	getHangupCallResult				
13	F12	getSuccess				true
14	F12	getRequestId				D11

Figure 3.36: Input Test Sheet `RouteCallTest` for Testing `RouteCall`

and 10 the values for the variables `callId`, `success`, and `requestId` are retrieved and compared to the expected return values in column F. These checks correspond to three assertions `routeCall.CallId=routeCallResult.CallId`, `routeCall.requestId=routeCallResult.requestId`, and `routeCallResult.success=Assertion1.success`. After the invocation of `hangUpCall`, the values for `success` and `requestId` are retrieved and checked by analogy to the lines 8 to 10. The `Backend` provides the operations `getRouteCallResult` and `getHangupCallResult` to retrieve objects that provide access to the variables `success`, `requestId`, and `callId` provided by the corresponding asynchronous service calls.

When the tests are applied to the SUT, the returned values are displayed in a new test sheet, known as result test sheet. The result test sheet is similar to the input test sheet but gives an indication of whether each expected result was actually satisfied. This is done using the familiar green/red coloring scheme by analogy to TTS. Thus, for example, a result test sheet for the execution of the test sheet in Figure 3.36 might have the form shown in Figure 3.37.

Each return value in column F is colored green if the expected value of the variables `callId`, `success`, or `requestId` equals to it (see lines 8, 9, 10, 14). Otherwise the expected value and the red colored return value are printed (see line 13).

The test sheet in Figure 3.36 is linear because it describes a single execution scenario based on the linear execution of their rows. To support conditional branches or loops, it is possible to opt out of the linear execution semantics of single–scenario test sheets and instead describe a more complex control flow by enriching the test sheets with additional behavioral information. This additional behavioral informa-

	A	B	C	D	E	F
1	'Vehicle	create				
2	'TelephonyConnector	create				
3	'BackendMock	create				
4	F1	initiateCall	"SIP:1234"			
5	F3	incomingCall				
6	F2	routeCall	"SIP:transfer"	F4	rand()[1...100]	
7	F3	getRouteCallResult	1000			
8	F7	getCallId				D6
9	F7	getSuccess				true
10	F7	getRequestId				E6
11	F2	hangUpCall	D4	rand()[1...100]		
12	F3	getHangupCallResult				
13	F12	getSuccess				true false
14	F12	getRequestId				D11

Figure 3.37: Result Test Sheet `ResultRouteCallTest` for Testing `RouteCall`

tion, which appears at the bottom of the test sheet below the invocation rows, specifies the order in which the rows are executed. Visually it is separated from the invocation columns by a double line, similar to the invocation line described previously. Test sheets with additional control logic are called *non–linear test sheets* or *multi scenario test sheets*. Each alternative is represented in the following general form:

```
SelectionCriteria -> NextBehavioralRow / ExecutionSequence
```

The selection criteria can either be a guard condition or a probability, depending on whether the behavior is deterministic or stochastic. In the former case the selection criterion is a boolean guard and in the second case it is a probability value. For deterministic behavior several guard conditions are mutually exclusive, in the case of stochastic behavior the sum of the probabilities of the guards must be `1`.

The second part of a transition expression, after the arrow symbol, identifies the next behavioral row that the behavior enters when the transition is made. By convention, the last row represents the end state and, thus, has no transition expression defining how to proceed. In other words, the last row is empty.

The final part of the expression, after the slash symbol, defines the sequence of operations that are performed as part of the transition. This takes the form of a list of row numbers separated by a space character.

Figure 3.38 shows a multi scenario test sheet considering the condition in the test `RouteCall`.

In line `13` a deterministic guard condition is specified. After the execution of the lines `1` to `10`, the value of the variable *success* reflected by the return value of the operation *getSuccess* in cell `F9` is checked. If the return value of `F9` is true, then line `11` is executed, otherwise the execution terminates in line `14`.

	A	B	C	D	E	F
1	'Vehicle	create				
2	'TelephonyConnector	create				
3	'BackendMock	create				
4	F1	initiateCall	"SIP:1234"			
5	F3	incomingCall				
6	F2	routeCall	"SIP:transfer"	F4	rand()[1...100]	
7	F3	getRouteCallResult	1000			
8	F7	getCallId				D6
9	F7	getSuccess				true
10	F7	getRequestId				E6
11	F2	hangUpCall	D4	rand()[1...100]		
12	F3	getHangupCallResult				
13	F12	getSuccess				true
14	F12	getRequestId				D11
13	[F9 == true] -> 11 / 1 2 3 4 5 6 7 8 9 10	[F9 != true] -> 14				
14						

Figure 3.38: Multi Scenario Test Sheet `MultiScenarioRouteCallTest` for Testing `RouteCall`

By introducing *higher–order test sheets*, it is possible to decompose tests and test scenarios into multiple, separately understandable elements. The examples above are then termed *lower–order test sheets* since they define single tests on the lowest level of the hierarchy. However, by parameterizing (lower–order) test sheets, in which the actual values used in a test are provided from outside as parameters, it is possible to execute the same test logic multiple times with different values. This is the role of higher–order test sheets.

The usage of transition expressions in test sheets, supports the definition of any kinds of behavioral protocol or algorithm, or any probabilistic operational profile. For writing practical tests, test sheets are as expressive as programmatic approaches.

Input test sheets can be parametrized by replacing the normal expressions or literals by appearing in cells with special parameters. A higher–order parameter is simply a question mark followed by a character. The first parameter must have the character `A`, the second `B`, and so on, thus, replicating the input rows of the higher–order test sheet.

Higher–order test sheets are not different from lower–order test sheets except that they are executed on lower–order test sheets. Higher–order test sheets are useful for invoking the same test logic several times with different test data. From the point of a higher–order test sheet a parametrized lower–order test sheet is like a service with a single operation, `test()`, which has an input argument for every parameter of the test sheet and return a boolean value indicating whether the test represented by the test sheet was successful (i.e., if any of the invocation rows failed).

Figure 3.39 shows a higher–order test sheet for routing a call.

The higher order test sheet contains two calls of the lower order test sheet `LowerOrderRouteCallTest`, calling its `test()` operation with three input parameters, i.e., the free parameters of the lower–order test sheet, and the expected return

	A	B	C	D	E	F
1	'LowerOrderRouteCallTest	test	"SIP:1234"	"SIP:transfer"	true	true
2	'LowerOrderRouteCallTest	test	"SIP:1234"	"SIP:invalid"	false	true

Figure 3.39: Higher–order Test Sheet `HigherOrderRouteCallTest` for Testing `RouteCall`

values `true` in column `F`.

Higher–order test sheets are normal test sheets in every way – they just happen to have a test sheet as SUT rather than a service interface – they can themselves be parametrized and used as the SUT of higher–order test sheets and so on. The approach is therefore completely *recursive*.

Figure 3.40 shows the corresponding lower–order test sheet `LowerOrderRouteCallTest` for the higher–order test sheet of Figure 3.39.

	A	B	C	D	E	F
1	'Vehicle	create				
2	'TelephonyConnector	create				
3	'BackendMock	create				
4	F1	initiateCall	?A			
5	F3	incomingCall				
6	F2	routeCall	?B	F4	rand()[1...100]	
7	F3	getRouteCallResult	1000			
8	F7	getCallId				D6
9	F7	getSuccess				?C
10	F7	getRequestId				E6
11	F2	hangUpCall	D4	rand()[1...100]		
12	F3	getHangupCallResult				
13	F12	getSuccess				true
14	F12	getRequestId				D11

Figure 3.40: Lower–order Test Sheet `LowerOrderRouteCallTest` for Testing `RouteCall`

The test sheet `LowerOrderRouteCallTest` has the same behavior as the test sheet `RouteCallTest` in Figure 3.36 but is parametrized. It contains three parameters, indicated by `?A`, `?B`, and `?C`. The concrete values are the input values retrieved from the invoking higher–order test sheet.

Test sheets have been used for specifying high quality services in [ABFJ08] and for testing web services in [ABB10].

The development of test sheets is significantly simplified by the use of a model–driven approach like TTS in which tests are designed graphically, validated and traceable. Likewise, the use of TTS is likely to be improved by the availability of a concise, understandable, and semantically self–contained tabular definition of executable tests than is offered by programming languages (Java in the case of the TTS implementation).

We therefore consider the integration of TTS and test sheets as future work. As TTS is a tabular and model–driven testing approach, we can enhance the TTS framework by test sheets in two ways. On the one hand test models in TTS can be transformed to test sheets as internal test representation, and on the other hand test sheets may replace the test data tables of TTS. The integration will result in a clean, comprehensive test methodology that supports all phases of the test development process, from analysis and design through to implementation, execution and evaluation, in a systematic and traceable way.

3.7 Classification of TTS

Classifications are very helpful for the evaluation of a methodology and for the comparison with similar approaches. In this section we classify our approach according to established taxonomies. Beside the motivating classification of TTS as model-driven testing and tabular system testing approach defined in Section 1.3, we classify TTS according to established testing and model–based testing taxonomies. Additionally, we classify the TTS artifacts due to the model–drivenness of the TTS methodology according to the two dimensions language–level and platform-dependency, and relate TTS to the Living Models paradigm.

Testing Classification. According to the classification in [Sch07] our approach can be characterized as *dynamic*, *systematic*, *active*, and *model–based* testing process. We focus on testing the system behavior indicating a dynamic approach by sending stimuli to it and observing its response which is active (see Section 3.5.5 for details on the test execution). Our approach is based on a formal metamodel (see Section 3.3 on the TTS metamodel) and therefore TTS is model–based. We specify a basic process model (see Section 3.2 on the TTS testing process) and therefore TTS is systematic.

Model–based Testing Classification. According to the model–based testing taxonomy of Figure 2.2 our approach can be classified as follows. Model subject is the *SUT* but the *environment* can be integrated by considering actors which are also reflected by services. Our approach separates test and development models but integrates them by reusing model elements and by validation and coverage checks. We think that this is one of the great advantages of our approach. Our approach can be adapted to handle deterministic, timed, and discrete systems. Our test models are *state–based* or pre/post where the system is a collection of variables representing a snapshot of the internal state of the system. We consider *structural model coverage*, *data coverage*, *requirements–coverage*, and *ad–hoc test case specification* (see Section 4.3 on coverage criteria). Our approach is capable for *manual generation*, *graph search*, and *symbolic execution* as test generation technologies (see Section 3.5.4 on test generation). Tests are executed *offline*.

Model–driven Testing and TTS. Our methodology is a model–driven testing approach. It applies models for testing purposes, separates between platform specific and platform independent aspects of the software, and applies automated transformations according to the definition in Section 2.1.3. The artifacts of the TTS process defined in Section 3.2 can therefore be classified according to the two dimensions platform dependency (*Platform Independent*, *Platform Dependent*) and language level (*Metamodel*, *Model*, *Implementation*, *Execution*). In Figure 3.41 the TTS artifacts are classified according to their platform dependency and the language level. Additionally, also the connecting activities of the testing process are integrated.

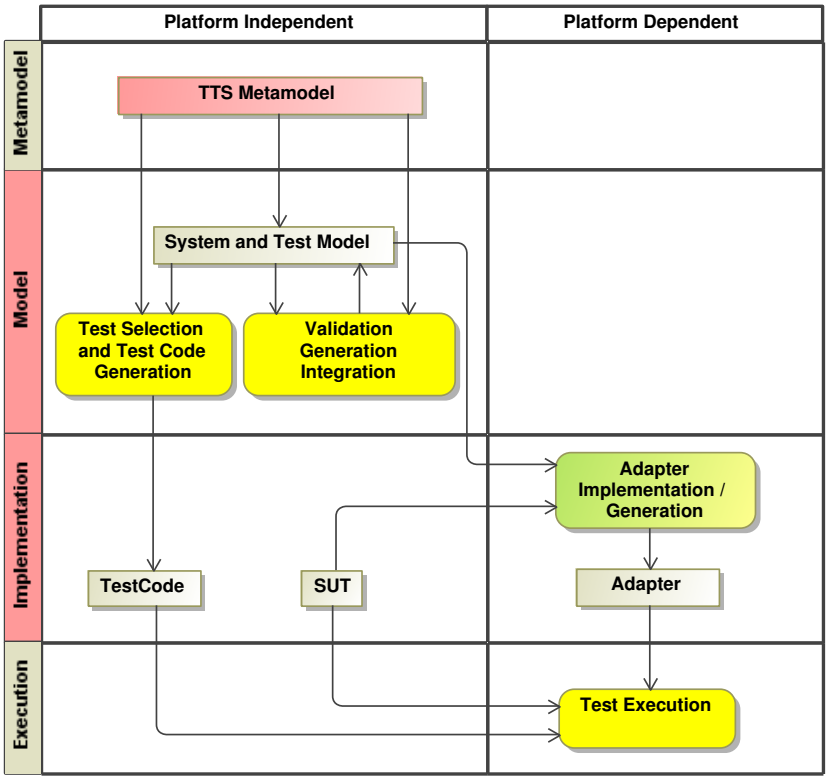


Figure 3.41: Model–driven Testing Classification of TTS Artifacts

The metamodel provides the abstract syntax for the system and test model. Additionally, the rules for validation and the rules for test code generation are defined based on the metamodel. The system and the test model are created iteratively with integrated model validation. Then the test code is generated. Additionally, the adapters for accessing the SUT services are implemented or generated by extending the abstract adapter depending on the SUT implementation. Finally, the test code can be executed. Compared to Figure 3.2 a rectangle for the artifact *TTS Metamodel* has been added. The model–to–text transformation from the system and test model to the test code is obviously a exogenous and vertical transformation. We have not considered the computation independent view, i.e., the requirements because they

do not affect the transformations itself. In our approach platform independent test models are directly transformed to test code which is a big difference to many other approaches, e.g., based on platform–independent test models specified in U2TP.

Living Models and TTS. Living Models focuses on model–based management, design and operation of dynamically evolving systems. The innovation of Living Models is threefold [Bre10]. First, Living Models is based on the tight coupling of models and code. This comprises a bidirectional information flow from models to code and from code and the runtime system back to the models. Second, Living Models supports the management of quality requirements fostering the cooperation between stakeholders in IT management, software engineering and systems operation. Third, Living Models is attached with a novel process model focusing on change and change propagation. The idea of Living Models is manifested by 10 principles (*P1* to *P10* listed in the following paragraph).

TTS has been designed and implemented based on these principles. Due to the traceability between all types of artifacts, TTS provides a close coupling of models and running systems (*P2 – Close Coupling of Models and Running Systems*) and a bidirectional information flow between models and code (*P3 – Bidirectional Information Flow between Models and Code*). For different stakeholders specific environments are provided (*P1 – Stakeholder Centric Modeling Environments*), e.g., a test report view for IT managers or a test model view for quality managers. TTS provides an integrated metamodel for the system, the tests, and the requirements under consideration of non–functional properties (*P4 – Common System View*) and checks for model consistency and coverage (*P6 – Information Consistency and Retrieval*). The remaining principles (*P5 – Persistence*, *P7 – Domains and Responsibilities*, *P8 – Model States*, *P9 – Change Events*, *P10 – Change–driven Process*) are considered by the state–based evolution management of TTS test models by attaching state machines to all model elements (see Chapter 6 on test model evolution and regression testing). The test evolution framework then guarantees the consistent evolution of the system and its tests and uses the states of model elements to generate regression test suites.

3.8 Related Work

TTS is a model–driven testing methodology for service–centric systems with tool support. In this section we provide related work concerning model–driven testing methodologies, the testing of service–centric systems, and testing tools.

Model–driven testing methodologies. Many approaches to the generation of executable tests from annotated non–UML models [MS04] and UML models [HVFR05] have been developed. But only a few approaches define separate test models based on a UML profile as metamodel such as [HN04, mdt] or the

most prominent and standardized approach based on the UML 2.0 Testing Profile [OMG05]. System level testing is based on the formalization of use cases using interactions and their mapping to UML 2.0 Testing Profile test specifications. The aim of our approach is a business oriented view on the test models. Test models can be created together with domain experts in a test–driven way even before the system model has been completed. Therefore the system and test model share concepts, are business oriented – not considering the test architecture and the underlying technology – and support the tabular description of data. Our approach is compatible to U2TP because our concepts can be mapped to U2TP concepts. Nevertheless we do not use U2TP directly because we emphasize a more business oriented view on system test modeling. The generation of executable test code from U2TP specifications has already been addressed in the specification itself [OMG05] providing mappings to JUnit [jun] and TTCN-3 [WDT+05]. In [ZDSD05] transformation rules are defined mapping U2TP to TTCN–3 which is afterwards compiled to executable test code in Java. In our approach, the test models are directly transformed to test code in Java which is also the language for adapter implementation.

FIT/Fitnesse [Mug05] is the most prominent framework which supports system test–driven development of applications allowing the tabular specification, observation and execution of test cases by system analysts. Our framework is due to the tabular specification of test data based on the ideas of FIT/Fitnesse but integrates it with model–based testing techniques.

System testing based on SoaML [OMG09b] specifications has not been considered so far. We address this problem indirectly by defining a generic system model that can be mapped to SoaML and relating the system model to a test model. This guarantees that our approach can be integrated with other modeling approaches for service–centric systems such as Quasar Enterprise [EHH+08] or the OASIS SOA Reference Model [OAS06] which are compatible to SoaML.

[APT+09] defines a requirements model and a system model similar to our approach based on domains models, used and required interfaces, and state machines. But in that approach only the mapping to Qtronic[6], a tool for automated test design, and not traceability or the relationship between the system and an independent test model are considered. [BGL+07] defines a similar model–based testing approach based on a subset of UML including state machines, instance diagrams, class diagrams and OCL. Each operation in the SUT can be labeled by requirements identifier to support the generation of a traceability matrix. The approach is implemented in the Smartesting tool[7]. Again this approach does not consider the relationship between independent system, and test models, test data is not defined in tables, and the approach is not optimized for service–centric systems.

A model–driven approach to testing SOA with the UML 2.0 Testing Profile [OMG05] is defined in [BRDG+07]. This approach focuses on web services

[6]The Qtronic tool is available at`http://www.conformiq.com/` [November 25, 2010].

[7]The Smartesting tool is available at `http://www.smartesting.com` [accessed: November 25, 2010].

technology and uses the whole set of U2TP concepts. TTS can be mapped to that approach but additionally our methodology is designed for arbitrary service technologies, provides a service–centric view on tests, supports the tabular definition of tests and guarantees traceability between all involved artifacts.

In [ZDSD05] model–driven testing is defined and implemented by mapping test models in U2TP to test code in TTCN–3. The focus of that work is on the transformation of test models to executable code whereas we focus on the relationship between requirements, test models and system models.

In [MS04] a system testing method whose test model is based on test graphs similar to our test stories is presented. Therein consistency and correctness is validated via model checking but the relationship to system models and traceability such as in our approach are not considered.

In [ABFJ08] test sheets, an extension of FIT, are used for specifying high quality services. The approach is compatible to our approach but it is based on a pure tabular representation and it does not consider abstract requirements, system, and test models as we do.

Testing service–centric systems. In [CD08] unit testing, integration testing, regression testing and non–functional testing of service–centric systems are discussed but in contrast to our approach system testing is not considered explicitly. As already mentioned above, in [BRDG+07] a model–driven approach for testing service–oriented architectures with U2TP is defined.

In [SDA05] TTCN-3 has been applied for functional and load testing of web services. Our approach can be mapped to the UML Testing Profile and therefore also to TTCN-3. But our metamodel focuses on the core concepts of testing service–centric systems This allows the investigation of model properties such as consistency and coverage especially for service–centric systems in the next chapter.

[BHH10] provides an overview of approaches for testing service–centric systems considering our approach as model–driven testing approach for service–centric systems.

Testing Tools. Model–based testing approaches always have a methodological and a tool aspect [UL07]. There are already some industrial tools available [GNRS09]. TDE/UML [HVFR05], a tool suite for test generation from UML behavioral models, is related to our approach but focuses on GUI–based systems whereas TTS focuses on service–centric systems.

FIT/Fitnesse [Mug05] is the most prominent framework which supports system test–driven development of applications allowing the tabular specification, observation and execution of test cases by system analysts. Our framework is due to the tabular specification of test data based on the ideas of FIT/Fitnesse but integrates it with model–based testing techniques.

Although test sheets [ABFJ08] define a fully tabular approach, support for model–driven testing as in TTS is missing there.

In [MS04] a model–driven system testing approach and a tool implementation that enables test engineers to graphically design complex test cases based on METAFrame Technologies' Application Building Center [SM99] has been defined. The approach is similar to TTS but not based on UML and its generic profiling mechanism.

The PLASTIC framework [BDAFP09] provides a collection of tools for online and offline testing both functional and non–functional properties of service–oriented applications. Some of the tools are model–based but the tools are not integrated and do not follow a model–driven testing approach as in the TTS tool implementation.

3.9 Summary

In this chapter we have defined the basic testing methodology of Telling TestStories.

Figure 3.42 graphically summarizes the TTS artifacts and the concrete syntax defined in Section 3.1 and in Section 3.3.

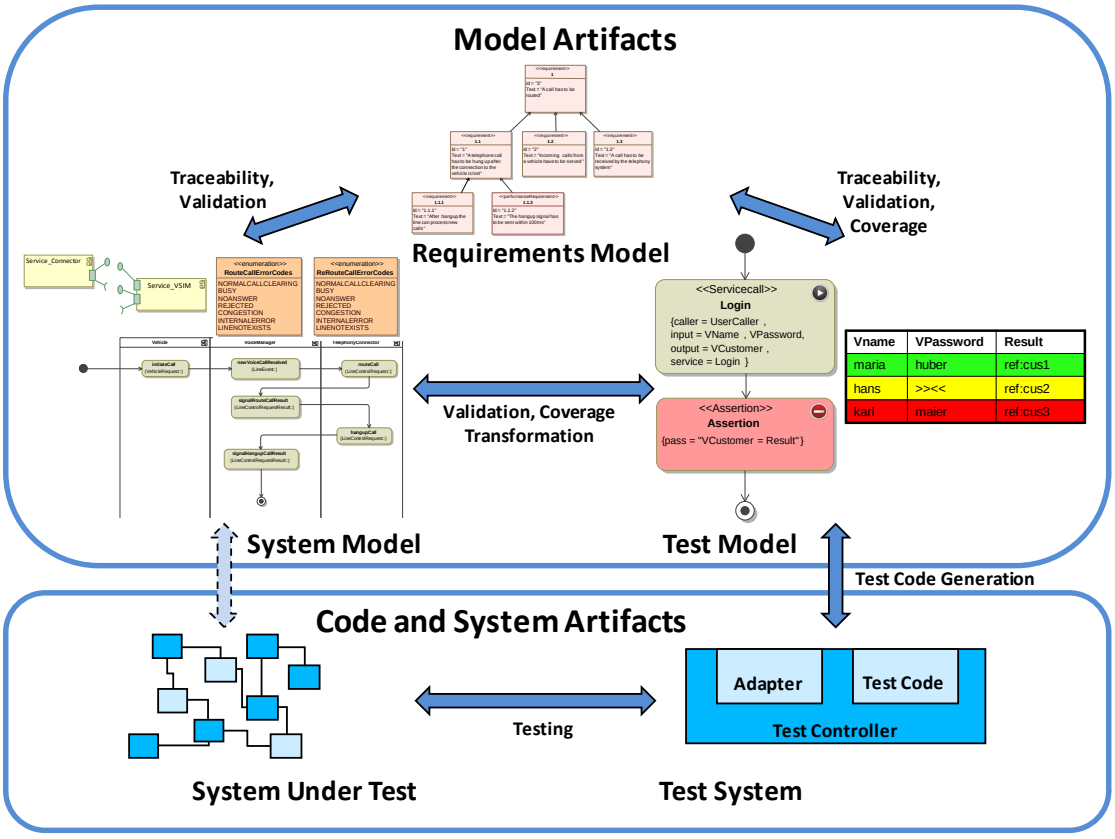


Figure 3.42: Telling TestStories Comprehension

In Section 3.1 we have defined the system and testing artifacts of TTS and their relationship. The main artifacts are the requirements model, the system model, the test model, test code, adapters, the test controller, the test system, and the system under test.

In Section 3.2 the main stages of the TTS testing process and the affected artifacts have been presented. The process consists of the requirements definition, the system model design, the test model design, the validation, generation and integration of the models, the test selection and test code generation, the adapter implementation and generation, the test execution and the test analysis. The process can be applied in a system–driven and in a test–driven way.

In Section 3.3 the metamodel and domain–specific language for TTS has been defined. We have presented a domain–specific language for requirements, systems and tests by its abstract syntax, its concrete syntax and its semantics. For the system under test and the test system we have provided a model.

In Section 3.4 a case study from the telecommunication domain has been presented by its requirements model, system model, test model, SUT, test system and the traceability between the elements in the different packages.

In Section 3.5 we have presented to tool implementation of the TTS methodology. We have explained the main components of the tool, namely the modeling environment, the model validator, the test code generator, service adapters, the test controller, and the test evaluator.

In Section 3.6 we have sketched prominent model–driven or tabluar system test design approaches, namely the Framework for Integrated Test (FIT), the UML 2.0 Testing Profile (U2TP), plus test sheets, and have related them to TTS.

In Section 3.7 we have classified our methodology in various categories of model–based testing and model–driven testing. Additionally, we classify the artifacts according to the two dimensions language–level and platform–dependency, and relate TTS to the Living Models paradigm.

In Section 3.8 we have presented related work to our tool–based model–driven system testing methodology for service–centric systems. We consider model–driven testing methodologies, the testing of service–centric systems, and test tool implementations.

Chapter 4

Formal Foundations

Everything should be made as simple as possible, but not simpler.

Albert Einstein

In this section, we provide formal foundations for the testing methodology introduced in the previous chapter. We first formalize the basic concepts of TTS (Section 4.1). Based on that we discuss mechanisms for arbitration (Section 4.2), the validation of system and test models (Section 4.3), and test transformation (Section 4.4). Finally, we present related work (Section 4.5) and sum up (Section 4.6).

4.1 Set–based Formalization of TTS Concepts

In this section we provide a basic formalization of the TTS requirements, system and test metamodel. This formalization is needed for a precise and concise presentation of our arbitration concept in Section 4.2 and of the test model evolution in Chapter 6. Our formalization is based on set theory and employs the reduced metamodel presented in Section 4.1.1.

4.1.1 Reduced Metamodel

The formalization in this section is based on the reduced metamodel in Figure 4.1. This metamodel contains all concepts relevant for the formal definition of arbitrations and of the evolution process. A motivation for this metamodel reduction (compared to the general TTS metamodel presented in Section 3.3.6) is given in Section 6.2. In this section we only define the concepts.

The reduced metamodel just contains the packages Requirements Model, System Model and Test Model. The Test Model is simplified compared to the corresponding one in the complete metamodel of Figure 3.15. Missing multiplicities indicate an

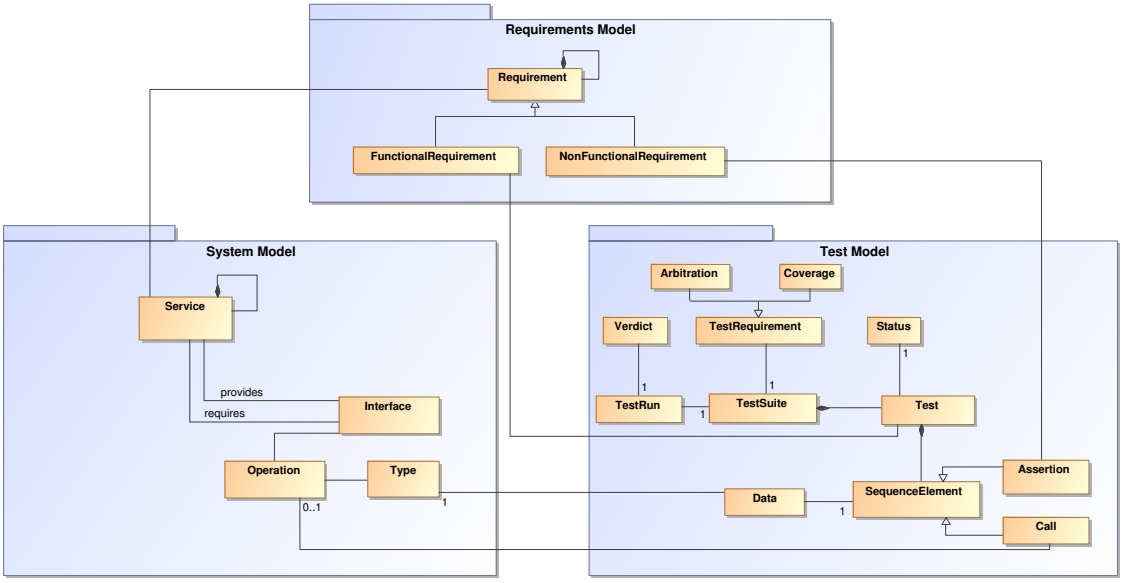


Figure 4.1: Reduced Metamodel for the Formalization

arbitrary number of elements, i.e., `0..*`. It does not contain a generic model element TestElement which allows arbitrary nesting of elements and does not consider parallel tasks and alternatives. The reduced metamodel contains a TestSuite element which is an aggregation of tests. Each test is a sequence of Assertion and Call elements. This reduction is valid because alternatives, parallel tasks and nesting can be serialized and expressed in the reduced metamodel. Note that data is assigned to the elements itself. Therefore every test is a single test case and not parametrized.

4.1.2 Terminology

We formalize the metamodel of Figure 4.1 and the test execution as far as needed in the subsequent sections to provide a precise and unambiguous presentation of arbitrations, coverage and test evolution. Our formalization is based on an extensional description with sets and functions.

Consider a fixed service–centric system. The following sets define all elements of a specific type in that service–centric system.

In the requirements model we define $\mathcal{R} = \mathcal{FR} \cup \mathcal{NFR}$ to be the set of all requirements ($\mathcal{R}$) defined as union of the set of functional requirements ($\mathcal{FR}$) and non–functional requirements ($\mathcal{NFR}$).

In the system model we define $\mathcal{S}$ to be the set of all services, $\mathcal{I}$ the set of all interfaces, $\mathcal{OP}$ the set of all service operations, and $\mathcal{D}$ the set of all data types.

In the test model we define $\mathcal{T}$ to be the set of all tests, $\mathcal{C}$ the set of all calls, $\mathcal{A}$ the set of all assertions, $\mathcal{TS}$ the set of all test suites, $\mathcal{TR} = \mathcal{AR} \cup \mathcal{COV}$ the set of all test requirements defined as union of the set of arbitrations $\mathcal{AR}$ and coverage criteria $\mathcal{COV}$, and $\mathcal{TE}$ the set of all test runs. The set of verdicts $\mathcal{V}$ and the set of

status $\mathcal{ST}$ are enumerations. Data elements in the set $\mathcal{DATA}$ are instantiations of types.

For an arbitrary set $\mathcal{M}$, the set of all sequences of arbitrary length $\mathcal{M}^*$ is defined as $\mathcal{M}^* = \bigcup_{i \geq 0} M_i$ with $\mathcal{M}_0 = \emptyset\,, \mathcal{M}_i = \{1, \ldots, i\} \rightarrow \mathcal{M}$ for all natural numbers $i > 0$.

In the following definitions, we determine elements of the sets defined above. If there is no definition for a specific element, then it is not of specific interest or there are no specific restrictions on instances of that set. In that sense a service is just a tuple of requirements and interfaces.

Definition 1 (Status)**.** The set $\mathcal{ST}$ of test status is defined as $\mathcal{ST} = \{\texttt{evolution}, \texttt{regression}, \texttt{stagnation}, \texttt{obsolete}\}$.

The status is mainly used for generating regression test suites and based on the classification in [LW89] where the status of a test can be can be *evolution* for testing novelties of the system, *regression* for testing non–modified parts ensuring that evolution did not impact parts supposed not to be modified, *stagnation* for ensuring that evolution did actually take place and changed the behavior of the system, and *obsolete* for tests which are not relevant any more.

Definition 2 (Verdict)**.** The set $\mathcal{V}$ of verdicts is defined as $\mathcal{V} = \{\texttt{pass}, \texttt{fail}, \texttt{inconc}, \texttt{error}\}$.

This definition of verdicts originates from the OSI Conformance Testing Methodology and Framework [ISO94]. *Pass* indicates that the SUT behaves correctly for the specific test case. *Fail* indicates that the test case has violated. *Inconclusive* is used where the test neither passes nor fails. An *error* verdict indicates exceptions within the test system itself. In the model itself only a pass or a fail can be specified. Inconclusive or error are assigned automatically.

Definition 3 (Test)**.** A test $t \in \mathcal{T}$ is defined as pair with a type $st \in \mathcal{ST}$ as first element, an arbitrary sequence of service calls and assertions $seq \in (\mathcal{SC} \cup \mathcal{AS})^*$ as second argument, i.e., $t = \langle st, seq, fr \rangle$, and a functional requirement $r \in \mathcal{FR}$.

The elements of a list are accessed by point operators, i.e., the status of a test t is invoked by $t.status$.

Example 1. Consider an operation `add(a,b):c` of a service `AdditionService`. A test which checks whether the return value equals a specific value, may then be defined as follows: $t_1 = \langle$`add(1,2):c`, `(c=3)`$\rangle$, where `add(1,2):c` defines a service call and `(c=3)` an assertion.

Definition 4 (Test Requirement)**.** A test requirement is a specific constraint that a test must satisfy or cover. A test requirement is an arbitration or a coverage criteria.

Arbitrations and coverage criteria can be expressed as predicates. A precise definition and examples of arbitrations are presented in Section 4.2 and the precise definition and examples for coverage criteria are defined in Section 4.3.

Definition 5 (Test Suite)**.** A test suite can be defined extensionally or intensionally. Extensionally, a test suite $ts \in \mathcal{TS}$ contains an arbitrary sequence of tests $tests \in \mathcal{T}^*$, an arbitration $ar \in \mathcal{AR}$, and a coverage criteria $cov \in \mathcal{COV}$, i.e., $ts = \langle tests, ar, cov \rangle$. Intensionally, a test suite can be defined by all tests which fulfill a specific selection criteria P, i.e., $ts = \langle t | P(t) \rangle$ which is formulated as coverage criteria, i.e., $P \in \mathcal{COV}$.

Example 2. Consider two tests $t_1 = \langle$`add(1,2):c`, `(c=3)`$\rangle$ and $t_2 = \langle$`add(2,2):c`, `(c=3)` in the situation of Example 1. A test suite can then extensionally be defined as follows: $ts_1 = \langle \langle t_1, t_2 \rangle$, `(pass%=100%)`, `(true)`$\rangle$. ts_1 is a sequence of the two test cases t_1 and t_2, contains an arbitration that requires passing of all tests, and the coverage criteria true selecting all tests and not restricting the set of test cases at all. Intensionally, ts_1 can be defined as the sequence of all test cases which call the operation `add` which can be defined as $ts_1 =$ { `t | t.servicecalls.exists(s | s.name = 'add' )` } in an OCL–like predicate definition language.

The restriction that every test suite contains exactly one arbitration and one coverage criteria is not reflected in the metamodel representation of Figure 4.1. But this restriction is possible because a set of arbitrations or coverage criteria can be expressed by one element via conjunction.

Definition 6 (Test Run)**.** A test run $te \in \mathcal{TE}$ represents the execution of a test suite. A test run is defined by a test suite ts and a verdict assignment function $va : \mathcal{A} \rightarrow \mathcal{V}$ which assigns to all assertions in all tests of the test suite ts a verdict. The test run te is then defined as $te = \langle ts, va \rangle$.

If every test is supposed to have exactly one assertion then the verdict assignment can be defined as function $va : \mathcal{T} \rightarrow \mathcal{V}$ and its elements are called a *test execution*.

Example 3. A test run `te1` for test suite `ts1` of Example 2 can be defined as follows: $te_1 = \langle ts_1, \{t_1($`c=3`$) \mapsto$ `pass`$, t_2($`c=3`$) \mapsto$ `fail`$\}\rangle$, or if we suppose va due to uniqueness of assertions as mapping from tests to verdicts, $te_1 = \langle ts_1, \{t_1 \mapsto$ `pass`$, t_2 \mapsto$ `fail`$\}\rangle$.

Definition 7 (Test Result)**.** A test result is an element of a verdict assignment function, i.e., it assigns a verdict $v \in \mathcal{V}$ depending on the context to an assertion $a \in \mathcal{A}$ $tr = \langle a, v \rangle$ or test $t \in \mathcal{T}$ $tr = \langle tc, v \rangle$.

Example 4. In the verdict assignment function of the test run in Example 3, two test results $\langle t_1,$ `pass`$\rangle$ and $\langle t_2,$ `fail`$\rangle$ are defined.

Definition 8 (Hierarchy, Subhierarchy, Superset)**.** A *hierarchy* of model elements is a directed n–ary tree of model elements. For an arbitrary model element m in a hierarchy, the *subhierarchy* $Sub(m)$ is defined as the set of all model elements of the subtree of m, and the *superset* $Sup(m)$ is defined as the set of all model elements on the path from m to the root model element of the hierarchy.

Example 5. Figure 3.17 defines a hierarchy of requirements. The hierarchy contains the elements `Req_1`, `Req_1.1`, `Req_1.2`, `Req_1.3`, `Req_1.1.1`, and `Req_1.2.1`. In this situation $Sub(\texttt{Req\_1.1}) = \{\texttt{Req\_1.1.1}\}$ and $Sup(\texttt{Req\_1.2.1}) = \{\texttt{Req\_1.2}, \texttt{Req\_1}\}$.

Services and requirements are organized in hierarchies, so called *service hierarchies* or *requirements hierarchies* which are needed to propagate changes.

4.2 Arbitrations

In this section we introduce arbitrations which allow for a decision on the highest abstraction level whether a sequence of test results satisfies a given arbitration criteria. We first motivate arbitrations, introduce behavioral and performance verdict functions and show how we have integrated arbitration mechanisms in the TTS methodology. The arbitration mechanism presented in this section covers the generic arbitrations approach for TTS published in [COLF+09].

4.2.1 Definitions and Motivation

We first provide definitions for verdicts functions, behavioral verdict functions, performance verdict functions and arbitrations. Afterwards, we motivate the usage of arbitrations in the context of various domains and service–centric systems.

A *verdict function* returns an encoded value which is based on a single result or on a sequence of results. We consider two types of verdict functions: *behavioral verdict function* and *performance verdict function*.

A *behavioral verdict function* returns a result based on the verdicts of test results. Behavioral verdict functions are discussed in Section 4.2.2.

A *performance verdict function* returns a result based on the test execution time of test results, i.e., the time needed to execute the test to obtain the test result. Performance verdict functions are discussed in Section 4.2.3.

An *arbitration* is a decision on the highest abstraction level whether a sequence of test results satisfies a given arbitration criteria. It uses logical and arithmetical operators to combine results of verdict functions.

Arbitrations can be considered for reporting purposes but also as dynamic coverage criteria. Arbitrations are higher–order assertions because they provide restrictions on the results, i.e., verdicts of the arbitrations introduced above.

The increasing size of software systems triggers off new challenges. Because of the large size of models it becomes hardly feasible to get a in–depth understanding

of a whole system at a low abstraction level. Methods for simplification become necessary to retrieve information from a system.

Models and metrics have been proposed to alleviate the complexity problem. In this context arbitrations, consolidating test evaluation results, and reporting can help to gain a *better understanding of the testing process* shifting it to a more general level.

Another important issue for arbitrations and test reporting is the *monitoring of implementation progress*. Defining well designed system tests, may result in a statement of the overall work progress and can early help pointing out the system being behind a schedule.

Moreover, in some areas arbitrations are the only means for test evaluation. Many service–centric systems have to fulfill service level agreements. These can be very strict or fuzzy agreements. In order to prove the system fulfilling the requirements several tests must be run to state an overall system reliability. Complex arbitrations calculated over verdict functions can be seen as metrics over test runs. Arbitrations allow for a convenient integration of tests in our sense as *built–in tests* to specified services in a service centric system.

Contractual specifications of large service–centric systems usually contain (informal) functional tests, reliability and performance requirements for these systems. State of practice is that these requirements are typically managed manually, i.e., functional tests are executed manually according to a commonly agreed test script. Reliability and performance tests are mainly part of the final acceptance procedures. The effort for manual testing is extremely high and may cover up to 50% of the development costs. Even partial automation of test execution and arbitration can cause significant *reduction of the development costs*. For service–centric systems, arbitrations can be seen as a type of SLAs.

The trend in industrial software development is going towards agile development processes that provide continuously growing executable system code. Therefore there is a high practical demand, to formally describe those tests, to support for (semi-) automatic test runs as part of continuous integration and to document and aggregate the test results in form of arbitration reports.

Arbitrations have originally been implemented as requirement for testing the callmanager application (see Section 3.4) because of general specifics of the telecommunication domain.

In the area of *telecommunication* the software system has to fulfill strict requirements. Imagine a software system reacting to incoming calls. Dependent on a caller's number, the desired service has to calculate some parameters and decide to which destination number the incoming call has to be routed. There are two kinds of requirements to the services related to behavior and performance. Firstly, a correct implementation must be assured, i.e., the correct number must be selected. Secondly, the number must be selected as fast as possible, since otherwise the person calling might hang–up the call even before having been routed. In this example the behavioral requirements are essential but without assuring high performance they

become useless. In a case when the correct number is provided too late, the service has to exit and route the call to a predefined default number, in order not to loose the calling customer.

Additionally, in discussions with practitioners we found out that the definition of arbitrations is useful for other techniques service–centric systems are based on.

In *real–time database systems* a lot of information is stored. As an example we will consider a database with many personal data records. At the service agreement level we can have, apart from behavioral requirements, the following performance requirement: queries to the database have to provide the result within one second in 99% of all cases, and within less than three seconds in the remaining cases. It is not possible to decide if the system really fulfills the service agreement based on results of single test runs. In such situation we need arbitrations over sequences of test runs to obtain a reliable decision on the system performance.

In case of capacity of *concurrent systems* less restrictive arbitrations are required. If a maximum of concurrent users a system can handle is equal to 80, we would need a verdict of a test scenario, that out of 100 parallel login requests, at least 80 requests must succeed, whereas the others may fail explicitly or stay inconclusive.

The aforementioned issues show the need for verdicts combining behavioral and performance aspects and evaluated over sequences of test results. Additionally, reporting is requested to summarize the overall performance and correctness of the system. Reports give an overview for a development team, as they show progress of the development process and enable effort prediction. Reports are also an important source of information for the ordering customer who wants to be kept informed about the project's progress. In the next section we show work related to arbitrations and reporting.

4.2.2 Behavioral Verdict Functions

As mentioned in the previous section, behavioral verdict functions are related to the expected behavior of a SUT. In this section we describe how behavioral functions are calculated for verdict assignments. We demonstrate how an arbitration can be built upon behavioral verdict functions and demonstrate in which way arbitrations defined in the TTCN–3 standard can be expressed using our definitions.

Behavioral Verdict Functions for a Test Result

We consider a *test result* tr, which represents an executed test tc and the corresponding test verdict v obtained for that test tc. The verdict equals to one of the values defined in Definition 2, $\mathrm{v} \in \{\texttt{pass}, \texttt{inconc}, \texttt{inconc}, \texttt{error}\}$.

Example 6. Consider a test that queries the SUT to find a user with a particular family name. If we take a concrete family name and define an expected result, e.g., a user id, we get a test(tc_0). If we run tc_0 we get a test result tr_0 with a

verdict (e.g., $v_0 = \texttt{pass}$, if the expected result and the return value are equal): $tr_0 = \langle tc_0, v_0 \rangle = \langle tc_0, \texttt{pass} \rangle$.

Now we define a **behavioral verdict function for a test result** $\mathcal{B}(tr)$, which returns an encoding of the obtained verdict:

$$\mathcal{B}(tr) = \begin{cases} \langle 1,0,0,0 \rangle & \text{if v} = \texttt{pass} \\ \langle 0,1,0,0 \rangle & \text{if v} = \texttt{inconc} \\ \langle 0,0,1,0 \rangle & \text{if v} = \texttt{fail} \\ \langle 0,0,0,1 \rangle & \text{if v} = \texttt{error} \end{cases} \tag{4.1}$$

Verdicts are encoded as unit vectors in a vector space whose length corresponds to the number of different verdicts. We introduce this encoding to enable an easier manipulation (e.g., projection) and implementation.

Example 7. For the test run tr_0 from the previous example, we get $\mathcal{B}(tr_0) = \langle 1,0,0,0 \rangle$.

The positions of the verdict types in the unit base correspond to the definition of a total order on the set of all verdict types, $\texttt{pass} > \texttt{inconc} > \texttt{fail} > \texttt{error}$. This is a shorthand notation for the following structure: whenever we obtain a verdict v, we ignore all verdicts v' for which we have $v' > v$. In other words if we have `inconc`, we ignore `pass`, if we have `fail`, we ignore `pass` and `inconc`, if we have `error`, we ignore `pass`, `fail`, and `inconc`. The order was introduced to enable an easier implementation and it is compatible with orders introduced in [WDT+05, BRDG+07]. In TTCN–3 [WDT+05] the following verdict types are defined: `none>pass>inconclusive>fail>error`. In U2TP [BRDG+07] the verdict types and their order are the same, only `none` is not considered.

To access a **value in the unit base** we introduce the projection operator π, e.g., to access the value for `pass` verdict we use $\pi_{\texttt{pass}}(\mathcal{B}(tr))$. Additionally, we define $\pi_{\texttt{all}}(\mathcal{B}(tr))$ to obtain the sum for all types:

$$\pi_{\texttt{all}}(\mathcal{B}(tr)) = \sum_{type \in \{\texttt{pass},\texttt{inconc},\texttt{inconc},\texttt{error}\}} \pi_{type}(\mathcal{B}(tr)) . \tag{4.2}$$

Example 8. For tr_0 defined in Example 6 we obtain:

$$\begin{aligned} \pi_{\texttt{pass}}(\mathcal{B}(tr_0)) &= 1, \\ \pi_{\texttt{inconc}}(\mathcal{B}(tr_0)) &= 0, \\ \pi_{\texttt{fail}}(\mathcal{B}(tr_0)) &= 0, \\ \pi_{\texttt{error}}(\mathcal{B}(tr_0)) &= 0, \\ \pi_{\texttt{all}}(\mathcal{B}(tr_0)) &= 1. \end{aligned}$$

Behavioral Verdict Functions for a Sequence of Test Results

A **sequence of test results** TR is a collection of single test results and the extensional description of a verdict assignment function. As the collection represents aggregated test results, it enables the interpretation of their execution results on a more abstract level.

Example 9. We define TR_0 as a sequence of test results that search different users. Assuming we executed three test runs, we can obtain $TR_0 = \langle\langle tc_0, \texttt{pass}\rangle, \langle tc_1, \texttt{pass}\rangle, \langle tc_2, \texttt{inconc}\rangle\rangle$ where tc_0 is defined as in Example 6 and tc_1, tc_2 are defined as tc_0, but with different test data.

For defining **behavioral verdict functions for a sequence of test results** $\mathcal{B}(TR)$, we have to extend the definition given in (4.1). This is the case because $\mathcal{B}(TR)$ is a function that represents the sequence of added occurrences of particular verdict types aggregated over all test results in the test result sequence TR:

$$\begin{aligned} \mathcal{B}(TR) \;=\; & \langle \pi_{\texttt{pass}}(\mathcal{B}(TR)), \\ & \pi_{\texttt{inconc}}(\mathcal{B}(TR)), \\ & \pi_{\texttt{fail}}(\mathcal{B}(TR))\rangle, \end{aligned} \tag{4.3}$$

where

$$\pi_{\texttt{pass}}(\mathcal{B}(TR)) = \sum_{tr \in TR} \pi_{\texttt{pass}}(\mathcal{B}(tr))$$

and by analogy we define projections for `inconc` and `fail`.

Example 10. For TR_0 defined in the previous example, we may obtain $\mathcal{B}(TR_0) = \langle 2, 1, 0\rangle$.

Arbitrations over Behavioral Verdict Functions

To define the ratio of test runs with particular verdict types we use projections:

$$\pi^{\%}_{\texttt{pass}}(\mathcal{B}(TR)) = \frac{\pi_{\texttt{pass}}(\mathcal{B}(TR))}{\pi_{\texttt{all}}(\mathcal{B}(TR))} \tag{4.4}$$

and by analogy we define projections for `inconc` and `fail`. Using percentages we can define **arbitrations** over verdicts. For example, we can express upper or lower bounds for test verdicts.

Example 11. We can define that at least three quarters of all test runs must obtain a pass verdict, which can be expressed as $\pi^{\%}_{\texttt{pass}}(\mathcal{B}(TR)) \geq 75\%$. Considering the criterion TR_0 from Example 10 is not satisfying the arbitration, as $\pi^{\%}_{\texttt{pass}}(\mathcal{B}(TR_0)) = 66\%$.

Moreover we can obtain more complex arbitrations by combining constraints with the **logical operators** *and* ($\wedge$), *or* ($\vee$) and *not* ($\neg$).

Example 12. With logical operators we can define that at least three quarters of all test results must obtain a pass verdict and no more than 5% may obtain an inconclusive verdict, which can be expressed as $\pi^{\%}_{\texttt{pass}}(\mathcal{B}(TR)) \geq 75\% \wedge \pi^{\%}_{\texttt{inconc}}(\mathcal{B}(TR)) \leq 5\%$. Considering the criterion TR_0 from Example 10 is not satisfying the arbitration, as $\pi^{\%}_{\texttt{pass}}(\mathcal{B}(TR_0)) = 66\%$ and $\pi^{\%}_{\texttt{inconc}}(\mathcal{B}(TR_0)) = 33\%$.

Expressing TTCN–3 Arbitrations

Our definitions can be used to express concepts provided by TTCN–3, where every component computes a local verdict for a test case which is then combined by the arbiter to a global verdict. The default arbitration from TTCN–3 can be expressed as follows (for completeness `none` and `error` can easily be added):

$$TTCN(TR) = \begin{cases} \texttt{fail} & \text{if } \pi^{\%}_{\texttt{fail}}(\mathcal{B}(TR)) > 0 \\ \texttt{inconc} & \text{if } \pi^{\%}_{\texttt{fail}}(\mathcal{B}(TR)) = 0 \\ & \quad \wedge\, \pi^{\%}_{\texttt{inconc}}(\mathcal{B}(TR)) > 0 \\ \texttt{pass} & \text{otherwise} \end{cases}$$

Example 13. Considering TR_0 from Example 9 we obtain $TTCN(TR_0) = \texttt{inconc}$.

The TTCN–3 standard mentions that users can define their own arbitration scheme, e.g., for performance tests, but no further details are given. In the next section we show how the performance aspect can be added to our formalism.

4.2.3 Performance Verdict Functions

As mentioned in Section 4.1, performance verdict functions are related to the test execution time, which is an important aspect for industrial applications. In this section we describe how performance verdict functions are calculated for a single test run and for a sequence of test runs. We give some examples and show how arbitrations can be defined based on both types of verdict functions.

Performance Verdict Functions for a Single Test Run

To handle performance constraints we extend the definition of a **test run**, given in (6), to the following triple:

$$tr = \langle tc, \mathrm{v}, t \rangle, \tag{4.5}$$

where t is time needed to execute a test case tc obtaining the verdict v. The time is defined over natural numbers, representing time in a given unit (e.g., milliseconds), extended with a symbol for undefined time ($\perp$), in case of a time–out occurrence and the # symbol for not relevant results, in case when a projection returns no value. Hence we get $t \in \mathbb{N} \cup \{\perp, \#\}$. We introduce the following semantics of additional symbols: $\perp$ is treated as positive infinite in comparisons, and # is omitted in arithmetical operations.

Example 14. Example 6 including test execution time can be represented as $tr_0 = \langle tc_0, \texttt{pass}, 5 \rangle$ if it takes 5 time units to execute tr_0.

To access the time information we define the **performance verdict function for a single test run** $tr = \langle tc, \mathrm{v}, t \rangle$ as follows:

$$\mathcal{T}(tr) = t. \tag{4.6}$$

Example 15. For tr_0 from the previous example, we obtain $\mathcal{T}(tr_0) = 5$.

To combine **performance and behavior** information we use the projection operator π to obtain time from a given test run $tr = \langle tc, \mathrm{v}, t \rangle$:

$$\pi_{\texttt{pass}}\left(\mathcal{T}(tr)\right) = \begin{cases} t & \text{if } \mathrm{v} = \texttt{pass} \\ \# & \text{otherwise} \end{cases} \tag{4.7}$$

By analogy we define projections for `inconc` and `fail`.

Example 16. For tr_0 defined in Example 14 we obtain
$$\begin{aligned} \pi_{\texttt{pass}}\left(\mathcal{T}(tr_0)\right) &= 5, \\ \pi_{\texttt{inconc}}\left(\mathcal{T}(tr_0)\right) &= \#, \\ \pi_{\texttt{fail}}\left(\mathcal{T}(tr_0)\right) &= \#. \end{aligned}$$

Performance Verdict Functions for a Sequence of Test Runs

Performance verdict functions for a sequence of test runs can use arithmetical functions, such as *sum*, *average*, *max*, *min*, and *count* to express restrictions over test execution time:

$$\bigotimes \mathcal{T}(TR) = \bigotimes_{tr \in TR} \mathcal{T}(tr), \tag{4.8}$$

where $\bigotimes$ is a place holder for any arithmetical operation.

Example 17. Adding performance information to TR_0 from Example 9 we get $TR_0 = \langle tr_0, tr_1, tr_2 \rangle = \langle \langle tc_0, \texttt{pass}, 5 \rangle, \langle tc_1, \texttt{pass}, 10 \rangle, \langle tc_2, \texttt{inconc}, \perp \rangle \rangle$. If we assume that the maximum time for a single test run must not exceed value of 10, i.e., $\min \mathcal{T}(TR) \leq 10$, then TR_0 does not satisfy this condition, as tr_2 was executed in undefined time.

In Example 17, we can see that such restrictions over test execution times are not expressive enough to cover all practical situations. To have a more useful mechanism we need again to combine information on **behavior and performance** using the projection again:

$$\bigotimes \pi_{\texttt{pass}}\left(\mathcal{T}(TR)\right) = \bigotimes_{tr \in TR} \pi_{\texttt{pass}}\left(\mathcal{T}(tr)\right). \tag{4.9}$$

Example 18. With the aforementioned extension we are able to express the following arbitration for TR_0 from Example 17, $\max \pi_{\texttt{pass}}\left(\mathcal{T}(TR)\right) \leq 5$, which means, we restrict maximal time for all test runs in TR with pass verdicts to 5 time units. For given TR_0 this is too restrictive, as tr_1 needs 10 time units to pass.

Arbitrations over Performance Verdict Functions

Based on performance verdict functions it is possible to express tests related to performance and accessibility of SUT.

Example 19. Combining performance verdict functions with logical operations we can express arbitrations with conjunction, such as one expressed by the following formula:

$\max \pi_{\texttt{pass}}(\mathcal{T}(TR)) \leq 5 \wedge \max \pi_{\texttt{inconc}}(\mathcal{T}(TR)) < \bot$.

The arbitration ensures that the maximal execution time of test runs that obtained a pass verdict is lower or equal to 5 time units and no time–out occurred for test runs that obtained an inconclusive verdict.

4.2.4 Generic Arbitrations and Test Reports

Based on definitions introduced in Section 4.2.2 and Section 4.2.3 we can define generic arbitrations. To achieve more flexibility in arbitration strategies we introduce additional labeling and for better overview we integrate arbitrations into test reports.

Arbitrations over Behavioral and Performance Verdict Functions

With all functions introduced in the previous Sections 4.2.2 and 4.2.3 it is possible to cover a broad range of expressions that are important for practical applications. By combining behavioral and performance verdict functions, one can formulate arbitration strategies which cover both aspects.

Example 20. We can combine behavioral and performance aspects by logical conjunction of previously defined arbitrations from Example 12 and Example 19.

Arbitration Labeling

Additionally, we introduce arbitration labeling to make the arbitration mechanism more generic and flexible. To each arbitration we can assign a set of labels. We consider labels related to development stages of SUT, different parts of the SUT and labels indicating verdict types. With labels fine tuning of restrictions over test runs depending on a project progress are possible. Analysis of border cases enables to specify the focus of future development of the system.

Example 21. We can consider a weak arbitration associated with a *first system release* label. The arbitration can state that the system may take up to 10 seconds for 2% of all searches. In the ideal situation the system should always return the complete list of all matching records within one second, but in a real situation due to the huge amount of data, this is not always possible. In the development phase it might be interesting to investigate cases with a part of the resulting list obtained within the predefined time or cases with the complete result list where the searching process took a little longer than one second.

Test Reports

To utilize all information gathered during test execution by the help of the verdict functions described in Sections 4.2.2 and 4.2.3 test reporting is of great significance.

A test report manager essentially takes care of two jobs (Figure 4.2): firstly, it has to evaluate the information gathered by former test runs, and secondly, it has to present the evaluated data by generating a test report. In order to evaluate the test run information, the report manager queries the database and receives the logging data. On this basis, it evaluates the verdict functions and generates a configurable test report.

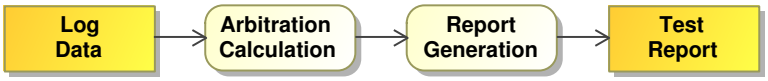


Figure 4.2: Steps from Log Data to a Test Report

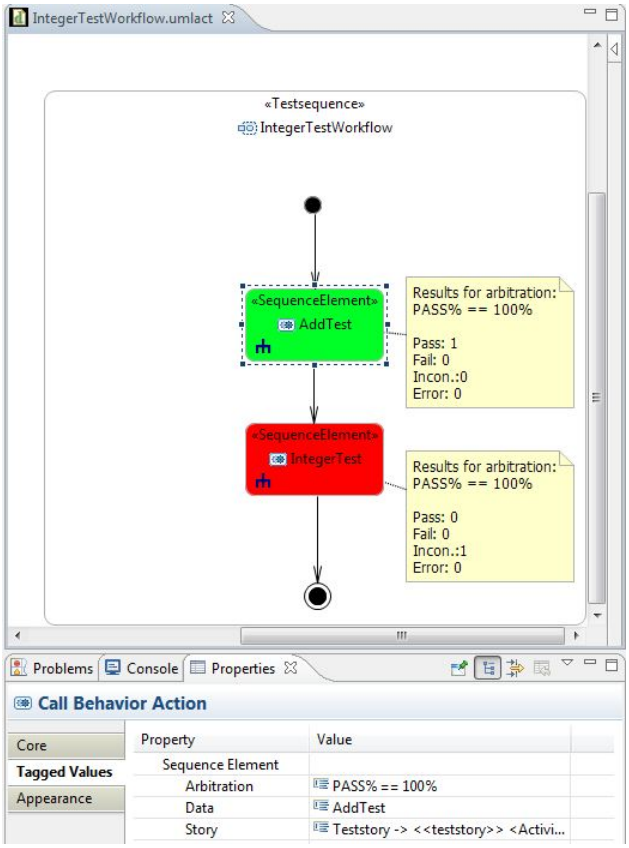


Figure 4.3: Implementation of Arbitration Calculation and Reporting

Based on the experience from software system development in the telecommunication and real–time databases domains we gathered the following **requirements for the report manager**:

- *flexibility of report definitions* — users should have the possibility to select the required level of detailedness (how the test runs should be selected and

grouped), types of used arbitrations (selection of expressions and labels), and the type of visualization (as for example in BIRT),
- *traceability to artifacts* involved in the testing process — the test report should enable navigation to the artifacts, such as, tested parts of SUT in appropriate versions, test specifications and test data used at the point of testing, and detailed log information for single test runs,
- *progress report* support — reports over regression tests showing evolution of SUT and progress statistics, e.g., as defined in [Fre08].

We have implemented the evaluation of arbitrations in our tool as presented in Section 3.5 on the tool implementation and depicted in the screenshot of Figure 4.3. In our implementation the verdict function is interpreted by a parser generated with ANTLR [ant]. The grammar of arbitrations is printed in Listing F.2 and for writing arbitrations we have implemented an editor in [Aic10]. The arbitration calculation of the log data is done in Java. The report generation is done by a modification of the UML2Tools representation of the test sequence elements by adding comments and coloring green if the arbitration holds and red if it does not hold (see Figure 4.3).

4.3 Model Validation

Models designed manually require tool supported validation since manual checks are error–prone and time–consuming. Validation is an activity where the model is analyzed against a set of validation rules. The outcome of the validation is a report specifying which model elements, if any, violate which rule. Validation is an evaluation activity that does not alter the model. In the context of TTS three different types of validation rules, i.e., *consistency*, *completeness* and *coverage* rules are considered and presented below. We do not consider correctness properties, i.e., the conformance with the modeling language as a separate category as in [Abb09]. This is justified from a formal point of view, where correctness is usually meant to be the combination of consistency and completeness. From a pragmatic point of view, conformance with the modeling language can either be checked by consistency or completeness checks [ZG03].

Validation of the model has the advantage that the quality of the design can be checked very early in the system development process. A valid model is the prerequisite for test code generation. If the model is invalid, then the expensive process of the test code generation can be avoided as long as the test model has not been improved. Validation can also be applied to validate the test model against the system model supporting test–driven modeling.

From the point of view of automatic validation, different approaches can be distinguished:

- *Model Checking* examines checks in temporal logics on system models specified as special graph structures [Fra07]. Due to the state explosion problem it

has drawbacks for big practical models. Additionally, it is to complex for practitioners to modify temporal logics formulas. Model checking is applied in model–based testing especially when the system model is specified as state machine.

- *Constraint Solving* requires specifying the checks and the system model in formal logic [GBR98]. Most tools work semi–automatically and the handling of basic data types is problematic [BW05]. Additionally, formal logic is very complex for practitioners.

- *Static Analysis* is directly checked on the UML model and applied in our approach. It does not support the dynamic simulation of a model but due to our experience this is replaced by early test execution in iterative software testing and therefore not a severe restriction.

In TTS, the models are validated automatically. Due to the dynamic evaluation of test models provided by early test execution, static analysis, i.e., validation of rules defined on metamodel elements without program execution, provides the most efficient and effective validation result: dynamic validation is provided by early test execution, and the modeling effort and computational complexity for static analysis is not as high as for model checking and constraint solving. The focus on static analysis is also supported by empirical research because in [LC04] it is shown that incompleteness and inconsistencies in UML models are already detected with OCL–based static analysis and that formal methods such as model checking or constraint solving do not improve the detection rate significantly. In TTS, SQUAM is applied as static validation framework.

In the remainder of this section we first discuss our validation framework, and then we consider consistency, completeness and coverage rules in more detail.

4.3.1 Validation Framework

In this section, we describe the SQUAM tool implementation that has been used for the validation of the TTS model. SQUAM [COGIOT06] (Systematic QUality Assessment of Models) supports OCL–based quality analysis which is optimal for our UML–based model representation. We have also experimented with validation rules in Prolog [COFLL08] but due to our experiments query execution in our respect is more performant with the Object Constraint Language (OCL) [OMG06b] than with Prolog. Additionally, queries for UML–based models are shorter in OCL and due to our opinion also easier to understand than in Prolog.

As discussed in Section 3.5.2, the SQUAM framework supports the definition and evaluation of queries. In Figure 4.4 the SQUAM perspective within Eclipse is shown.

The OCL library project `info.teststories.queries` opened in the OCL perspective contains all consistency, completeness and coverage rules defined for

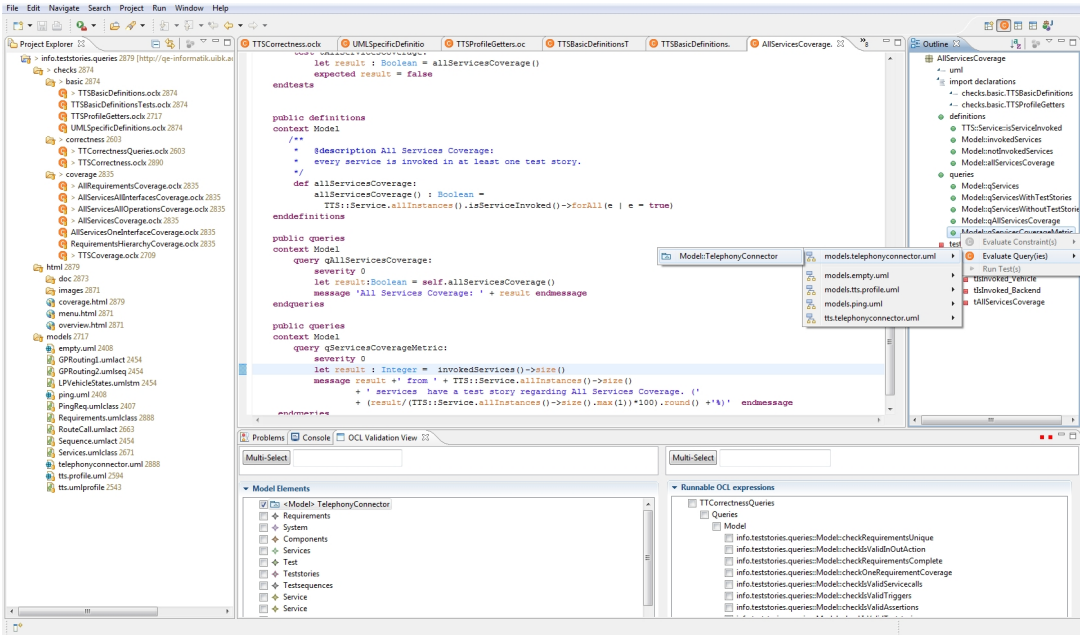


Figure 4.4: SQUAM Perspective with TTS Validation Rules

TTS. As shown in Figure 4.4 queries can be executed manually in the OCL Validation View or in the Outline view. The TTS queries for consistency and completeness are executed automatically before tests are generated, coverage criteria can be executed on demand. SQUAM defines an OCL library [CO09] for validating TTS models which contains OCL definitions, OCL queries and OCL tests.

A *definition* is a method within a specific context. The defined methods can be used in other methods or in queries. In Listing 4.1 example definitions are shown.

```
public definitions

  context Model
    def getRequirementByName:
      getRequirementByName (aName : String) : TTS::Requirement =
        TTS::Requirement.allInstances()->any(r | r.name = aName)

    def uniqueNamesOfRequirements:
      uniqueNamesOfRequirements() : Boolean =
        TTS::Requirement.allInstances().name->isUnique(n|n)

enddefinitions
```

Listing 4.1: Examples for OCL Definitions

Listing 4.1 contains two definitions. The first definition `getRequirementByName` has one input parameter `aName` and returns a requirement element with the same name. The second definition `uniqueNamesOfRequirements` checks whether the names of all requirements in a model are unique.

A *query* adds information specific for model querying to a definition, i.e., a severity level and a message. Queries are the interface for manual and automatic validation. In Listing 4.2 example queries are shown.

```
public queries

  context Model
    query qUniqueNamesOfRequirements:
      severity 1
      let result:Boolean=uniqueNamesOfRequirements()
      message result endmessage

  context Model
    query qAllRequirementCoverage:
      severity 2
      let result:Boolean = self.allRequirementsCoverage()
      message 'All requirement coverage: ' + result endmessage

endqueries
```

Listing 4.2: Examples for OCL Queries

Listing 4.1 contains two queries. The first query `qUniqueNamesOfRequirements` checks whether all requirements have unique names by invoking the definition `uniqueNamesOfRequirements`. The second query `qAllRequirementCoverage` checks for all requirements whether they are tested and prints whether all requirements are covered or not as a message.

In TTS, three different severity levels are considered in queries. The severity level *error* (severity 0) indicates that the test code generation is not possible if a specific query evaluates to false. Most consistency checks have the severity level error. The severity level *warning* (severity 1) indicates that the test code generation is possible but that the test code execution might be incomplete or even fail if that check evaluates to false. Incompleteness checks often have the severity level warning. The severity level *information* (severity 2) shows an information message but the check does not influence the test code generation and test execution. Coverage criteria often just provide informative messages. In Listing 4.2 the query `qUniqueNamesOfRequirements` has the severity level warning, and the query `qAllRequirementCoverage` has the severity level information.

Although queries are the interface for validation in SQUAM, we only denote whole queries in the remainder of this document if their severity level or message is important. Otherwise we only denote the definition that is invoked by a query.

A *test* for an OCL definition specifies the method, the input values and the expected result. The handling of the underlying test framework for OCL expressions is similar to JUnit for Java and therefore called OCLUnit [CO09]. In Listing 4.3 OCLUnit tests for the definition `getRequirementByName` are shown.

```
private tests

  modelinstance models.telephonyconnector.uml

    test tRequirementsHierarchyCoverage_Req_1:
      let result:Boolean =
        getRequirementByName('Req_1').
          requirementsHierarchyCoverage()
        expected result = true

    test tRequirementsHierarchyCoverage_Req_1_1:
      let result:Boolean =
        getRequirementByName('Req_1.1').
          requirementsHierarchyCoverage()
        expected result = true

endtests
```

Listing 4.3: Examples for OCLUnit Tests

The two OCLUnit tests `tRequirementsHierarchyCoverage_Req_1` and `tRequirementsHierarchyCoverage_Req_1_1` of Listing 4.3 are defined for the method `getRequirementByName` for the model named `telephonyconnector.uml`, i.e., the model of the callmanager, which is the model instance on which the tests are executed.

To specify queries we followed the model analysis and OCL library development process (Figure 4.5). The upper swimlane corresponds to the manual model analysis, the lower swimlane to the library development process.

First, common requirements for model analysis and library development are specified. A quality aspect is selected, e.g., a completeness criterion defining that each requirement should have a unique name. For this aspect OCL definitions and queries are specified in the development step.

The next step is quality assessment, where the results of the manual and au-

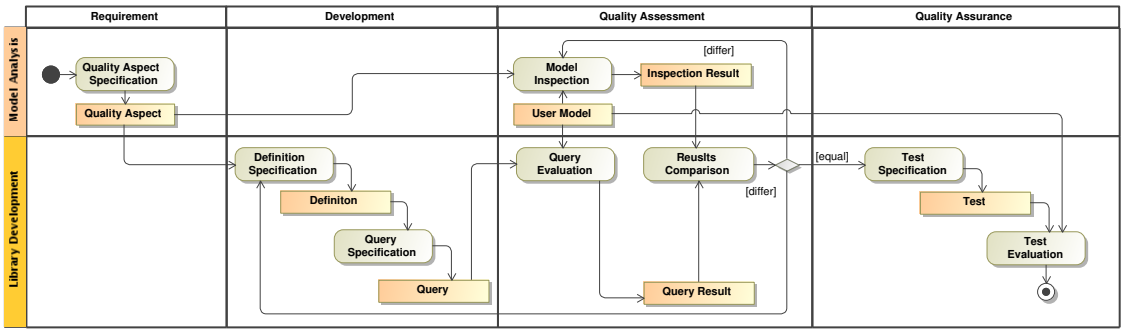


Figure 4.5: Model Analysis and Library Development Process (from [COAB10]).

tomatic analysis are cross–checked. For the selected aspect, manual inspection is used to determine the result of this aspect for the model. Simultaneously, appropriate queries are evaluated on the model. If the results of the model inspection and the query evaluation differ, the reason has to be determined and either the OCL definition specification or manual inspection needs to be repeated. The manual inspection of the model is performed as long as correctness of a query achieves a defined convenience level. Afterwards the query can be used for automatic model analysis.

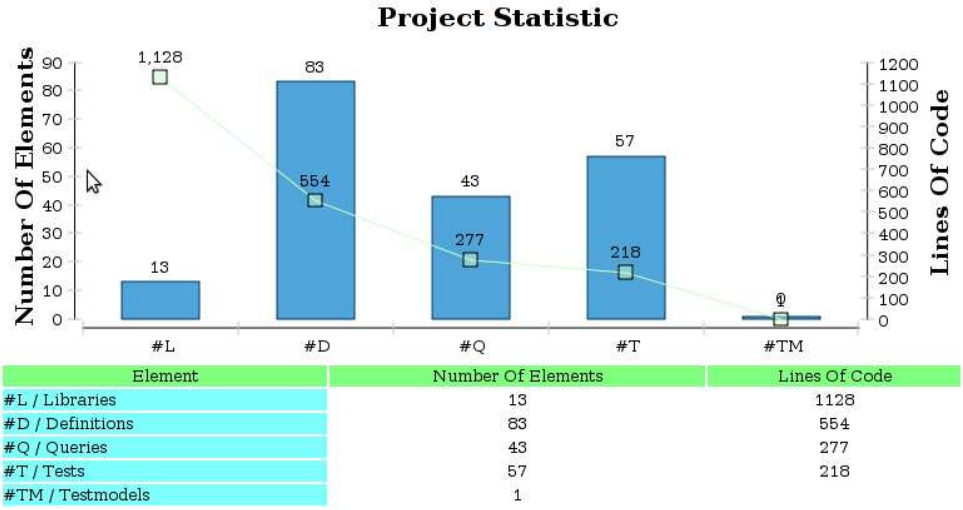
If the results are equal, the last step, i.e., quality assurance, can be executed. The aim of this step is to assure semantic correctness of OCL expressions in the future development of the library. For this purpose OCL unit tests [CO09] are specified and evaluated regularly. In the test evaluation step, OCL unit tests with corresponding OCL test models are required. The OCL test model is an instance of the requirements, system and test metamodel. Note that the OCL test model is not the same as a test model in TTS but a model instance of the considered metamodel for the OCL expression under test. This instance is used as test data for OCL unit tests to assess the desired semantics of definitions. OCL unit tests are similar to JUnit [jun] tests applied to assess the semantic correctness of source code.

In Figure 4.6 we show the size of the OCL project developed for the TTS approach.

We have specified 83 definitions used in 57 queries and split into 13 libraries. The number of OCL expressions is higher than the total number of the criteria and metrics as it was necessary to define helper methods. The helper expressions can be used in the specification of further criteria/metrics and make their development less time consuming. To assure correctness of OCL expressions we wrote 57 OCL unit tests [CO09] evaluated on one test model.

4.3.2 Validation Rules

Validation rules which are applied for the static model analysis in our approach can validate just one model (*intra–model* validation) or the relationship between models (*inter–model* validation). For instance, a check whether the name of every



Element	Number Of Elements	Lines Of Code
#L / Libraries	13	1128
#D / Definitions	83	554
#Q / Queries	43	277
#T / Tests	57	218
#TM / Testmodels	1	

Figure 4.6: OCL Project Statistics

requirement is unique is an intra–model validity check, and the check whether every service call in a test of the test model refers to a service of the system model is an inter–model check.

The inter–model relationship between the requirements, system and test model is implemented by tagged values or associations. For instance, the relationship between a requirement (in the requirements model) and a test (in the test model) is implemented by a tagged value of the requirement and the test model element referencing each other. In Figure 3.26 the requirement `Req_1` is associated to the test `RouteCall`.

As mentioned above in TTS we investigate three types of validation rules: consistency, completeness, and coverage rules.

Consistency checks assure that there is no conflicting information in models. Consistency of a model enables error–free transformation from the model to another model or to the source code.

Completeness checks assure that one artifact is complete, i.e., contains all essential information. Completeness of the system model is crucial for generation and execution of tests in the TTS framework.

Coverage can be considered as a variant of inter–model completeness where one model is the test model. This aspect is very important in context of testing and is used to check to what extent the test model covers the system model and implicitly the system. We adopted a series of coverage criteria from testing [AO08] and model–driven testing [UL07] to fit into the TTS framework.

Additionally, we distinguish between criteria and metrics. *Criteria* provide only a boolean result: true if a model fulfills a given criterion, false otherwise. They provide a warning mechanism implemented, i.e., the severity levels information, warning, and error, to inform modelers about an incorrectness in a model. *Metrics* have a numeric result. Typically they provide information to what extent a corresponding criterion is fulfilled, thus they are fractions ranging from `0` to `100%`. They can be used for the evaluation of models in a similar way as for the evaluation of source code in [LMD05]. Metrics are well suited for summarizing particular aspects of models and for detecting outliers in large models. They scale up and aggregate many details of models.

Table 4.1 provides an overview of all types of validation rules that have been defined for TTS.

		consistency		completeness/coverage		
Type	Models	Metrics	Criteria	Metrics	Criteria	
Intra	Requirements		✓	✓	✓	Complet.
	System		✓	✓	✓	
	Test		✓	✓	✓	
Inter	Requirements–System		✓	✓	✓	Cov.
	Requirements–Test		✓	✓	✓	
	System–Test		✓	✓	✓	

Table 4.1: Types of Validation Rules

For consistency we have only defined criteria because in our approach a model is either consistent or not but there is no number to define the degree of consistency. For completeness and coverage we additionally have defined metrics because there is a degree of completeness and coverage. For instance, it is possible to state that 80% of all requirements are covered by tests. We have defined intra–model validation rules for the requirements, system and test model and inter–model validation rules between all combinations of them, i.e., requirements–system, requirements–test, and system–test. Coverage rules are inter–model validation rules where one model is the test model. In the following, we discuss consistency, completeness and coverage aspects and give examples.

Consistency

Consistency of a model expresses that a model is non–contradictory. There are various formal and informal definitions for consistency in software engineering [Küs04]. Referring to [LC04], consistency in UML can broadly be divided into the complete approaches, which define a fully formal semantics for the complete notation, and the partial approaches, which define a well-defined meaning for a subset. The partial approaches can be divided into two principle types: formal approaches that map a partial model onto a formalism and design–oriented approaches that focus

on modeling UML using a metamodel and OCL in order to analyze properties. Our approach can be classified as design–oriented approach. In [LC04] it is shown that this approach supports the detection of incompleteness and inconsistencies as well as the quantification of completeness, while it does not require the more complex machinery of fully formal methods.

According to the pragmatic consistency definition of TTS, a model is consistent if there is *no conflicting information in the model*, if it is *free from undesired non–determinism* and if *test generation and execution is possible*. Our consistency definition focuses on the test model. Consistency of the requirements and system model is only considered if it is a prerequisite to guarantee the consistency of the test model. For instance, a system model is inconsistent if there are services with the same name because in that case the invocation of service calls in the test model is not clearly defined.

The early definition of a test model can be used to improve the quality of the system model and vice versa because the additional model information allows for defining additional consistency checks that cannot be defined within one of the models.

The concrete validation rules of this section and all subsequent sections are formulated based on the implemented metamodel depicted in Figure B.1 which is a specialization of the generic metamodel (see Figure 3.15).

We have defined several consistency criteria to assure that there is no conflicting information in the models. The criterion `isServiceUnique` in Listing 4.4 guarantees the uniqueness of service definitions in a system model.

```
context TTS::Service
  def isServiceUnique:
    isServiceUnique() : Boolean =
      TTS::Service.allInstances()
       ->select(r|r.base_Class.name=base_Class.name)
       ->size() = 1

context Model
  def allServicesUnique:
    allServicesUnique() : Boolean =
      TTS::Service.allInstances().
       isServiceUnique()->forAll(s | s = true)
```

Listing 4.4: OCL Definition for Service Uniqueness

Listing 4.4 contains two definitions. The first definition `isServiceUnique` checks for a specific service whether its name identifier is unique. The second definition `allServicesUnique` checks whether all services in the system model are unique.

The criterion `isAssertionConsistent` in Listing 4.5 guarantees the consistency of an assertion. The definition `isAssertionConsistent` guarantees that the pass constraint of an assertion differs from its fail constraint.

```
context TTS::Assertion
  def isAssertionConsistent:
    isAssertionConsistent() : Boolean =
    not (pass = fail)

context Model
  def allAssertionsConsistent:
    allAssertionsConsistent() : Boolean =
      TTS::Assertion.allInstances()
        .isAssertionConsistent()->forAll(a | a = true)
```

Listing 4.5: OCL Definition for Consistency of Assertions

The first definition of Listing 4.5 checks that the pass constraint and the fail constraint are not equal. If pass does not differ from fail, then the assertion is inconsistent because it is not possible to compute a meaningful verdict. The second definition `allAssertionsConsistent` checks whether all assertions in a specific model are consistent.

In Section C.1 the source code of basic consistency checks is listed. In TTS we have implemented consistency checks for the following tasks:

- uniqueness of requirements, i.e., all requirements have different identifiers
- uniqueness of services, i.e., all services have different identifiers
- uniqueness of tests, i.e., all tests have different identifiers
- consistency of assertions, i.e., the pass and fail constraint of assertions differ
- consistency of tests, i.e., each test has exactly one initial and one final node

Our consistency checks guarantee the uniqueness of identifiers and the proper structure of tests for test code generation and execution.

Completeness

Completeness guarantees that a *model contains all essential information*. Completeness of models has high relevance for testing because it influences the quality of the derived tests. Incomplete test models can typically be transformed to test code but the test execution is not possible or not useful. For instance, if there is no service in a system model, then test execution is not possible, if there is no assertion in a test

then the execution of that test is not useful. As for consistency, our completeness definition focuses on test models[1]. Completeness of the requirements and the system model is only considered as far as relevant for test generation and execution.

We consider coverage criteria as special completeness checks between the test model and the system or requirements model. Due to the high importance of coverage in the respect of testing not just for validation but also for test selection, we discuss coverage criteria in the next section.

We have defined several completeness criteria which guarantee that artifacts contain all essential information. In our respect the completeness of test models is very important because otherwise no meaningful test code can be generated. In Listing 4.6 a criterion to check the completeness of a test is defined.

```
context TTS::Test
  def hasAssertion:
    hasAssertion() : Boolean =
      base_Activity.allOwnedElements()->select(o |
      o.profileIsTypeOf('Assertion'))->size() > 0

context TTS::Test
  def hasServicecall:
    hasServicecall() : Boolean =
      base_Activity.allOwnedElements()
      ->select(o|o.profileIsTypeOf('Servicecall'))
      ->size() > 0

context TTS::Test
  def isTestComplete:
    isTestComplete() : Boolean =
          hasAssertion() and hasServicecall()
```

Listing 4.6: OCL Definitions for Completeness of Tests

A test is complete if it has at least one assertion to compute a verdict and a service call to interact with the system. The definition `hasAssertion` of Listing 4.6 checks whether a test has at least one assertion and the definition `hasServicecall` checks whether a test has at least one service call. Finally, the definition `isTestComplete` checks whether a test has at least one assertion and one service call by invoking `hasAssertion` and `hasServicecall`.

In Section C.1 the source code of basic completeness checks is listed. In TTS we have implemented completeness checks for the following tasks:

- completeness of requirements, i.e., each requirement has an identifier and its

[1] Other definitions of completeness focus on the system model and consider testing as a means for validating the completeness of a system specification.

textual description is not empty

- completeness of services, i.e., each service has an identifier
- completeness of tests, i.e., each test has an identifier, at least one service call, and at least one assertion

Additionally, completeness checks between requirements and services could be considered, e.g., to guarantee that each service is assigned to a requirement and vice versa. The completeness checks implemented so far already guarantee that meaningful test code is generated and can be executed afterwards.

Coverage

Coverage criteria define whether a set of specific elements is tested by a test suite and coverage metrics define the percentage of elements that have been tested by a given test suite [Bin99]. When coverage is used without quantification, 100 percent is usually understood and the metrics can be identified with the criterion.

Coverage on the model level as defined in our approach can be considered as a *variant of inter–model completeness where one model is the test model*. The covered elements can be either elements, structures or expressions from the system or requirements model.

We apply coverage criteria on a set of tests but do not consider whether these tests have been designed manually or automatically.

We consider *structural model coverage*, *data coverage*, *requirements–coverage*, and *ad–hoc test case specification* (see Section 4.3 on coverage criteria).

Test purposes may be either functional (i.e., test a certain functionality) or structural (i.e., obtain a certain coverage like branch coverage or path coverage). Due to the informal character of test purposes, it is impossible to compute executable test sequences directly from test purposes. Thus, test purposes have to be manually turned into formal test case specifications from which test cases can be derived. A test case specification can be formulated in many ways. Simple examples are input histories, transition sequences, traces, or constraints over them.

In general coverage criteria can be interpreted as adequacy criteria and as selection criteria [GG75]. A *selection criterion* helps to select a test suite in order to fulfill a goal, whereas an *adequacy criterion* helps to check that a previously selected test suite satisfies a goal. In the case of a selection criterion tests are directly generated to fulfill a specific coverage criterion, i.e., they are exit criteria defining when test generation can be stopped. In the case of an adequacy criterion tests are generated independently and then measured against the criterion in terms of their coverage.

As we only consider the tool–supported manual definition of tests, in this thesis we only consider coverage criteria as adequacy criteria. The application of a coverage criterion as adequacy criterion is in practice much more important than the application as selection criterion because test generation is based on more formal models that require more theoretical skills and more effort to be created.

The distinction between selection and adequacy criteria has a strong theoretical basis [AO08]. A *generator* is a procedure that automatically generates tests to satisfy a criterion, and a *recognizer* is a procedure that decides whether a given set of tests satisfies a criterion. Theoretically, both problems are provably undecidable in the general case for most criteria. In practice, however, it is possible to recognize whether tests satisfy a criterion far more often, i.e., applying an adequacy criterion than is is possible to generate tests that satisfy the criterion, i.e., applying a selection criterion. If no infeasible test requirements are present then the problem becomes decidable.

A test adequacy criterion is a rule or a set of rules that impose requirements on a test set and are formulated as a test requirement in our approach. As test requirements, adequacy criteria guide the process of test definition. As coverage criteria they can be interpreted as a stopping rule which defines when enough tests have been produced to satisfy all criteria. As coverage metrics they can be interpreted as a measure for the test quality. In this sense, coverage criteria define test objectives or goals that are to be achieved while performing software testing. Especially cost considerations and available resources often determine the selection of one criterion over another.

Ammann and Offutt [AO08] distinguish criteria–based test design and human–based test design which are both needed for holistic test design. In *criteria–based* test design, test cases are defined to fulfill some coverage criteria, e.g., the criteria defined in this section. In *human–based* test design, tests are defined based on the domain knowledge of the program and human knowledge of testing. TTS considers and integrates both test design approaches because TTS defines coverage criteria and is suitable for human–based test design due to its abstractness and table–based approach.

We have developed several coverage criteria and metrics based on the coverage criteria from testing [AO08] and model–driven testing [UL07]. In the next paragraphs we demonstrate how coverage criteria and metrics are implemented in our approach.

As first example we consider the *all requirements coverage* (ARC) defined in [UL07]. In our context this represents the coverage between the requirement and the test model. In [UL07] ARC says only that *all requirements are covered*. ARC is refined in the context of the TTS metamodel to *each requirement has a test*. "Requirement" in this regard is a model element that applies the `TTS::Requirement` stereotype and "has a test" means that it has either an action, or an activity defined,

i.e., at least one tagged value referring to a test is set. This informal definition results in the OCL definitions presented in Listing 4.7.

```
context TTS::Requirement
  def hasTest:
    hasTest() : Boolean =
      (not self.action.oclIsUndefined()) or
      (not self.activity.oclIsUndefined())

context Model
  def allRequirementsCoverage:
    allRequirementsCoverage() : Boolean =
      TTS::Requirement.allInstances()
       .hasTest()->forAll(e | e = true)
```

Listing 4.7: OCL Definitions for ARC

In Listing 4.7 both definitions return a value of the type `Boolean` which provides decision support but is not very informative. To gain more information from the model we have defined informative queries that compute a metric and are based on definitions. The query `qRequirementCoverageMetric` in Listing 4.8 extracts the total number of requirements and the number of all requirements which have a test assigned. The number of all tested requirements is computed by the query `requirementsWithTests` which is based on the query `hasTest` defined in Listing 4.7. The metrics then computes the ratio between the total number of requirements and the number of tested requirements. It informs to which degree the coverage criterion is satisfied. From the metric we obtain a value between `0` and `1`. Alternatively, the ratio can be expressed as percentage.

```
context Model

  query qRequirementCoverageMetric:
    severity 2
    let result : Integer = requirementsWithTests()->size()
    message result +' from '
     + TTS::Requirement.allInstances()->size()
     + ' requirements have a test.('
     + (result/(TTS::Requirement.allInstances()
      ->size().max(1))*100).round()
     +'%)'
    endmessage
```

Listing 4.8: OCL Metrics for ARC

The metrics of Listing 4.8 determines the ratio of requirements covered by at least one test and prints the result as formatted text.

For the callmanager case study (see Figure 3.26) we obtained the following result: `6 from 6 requirements have a test.(100%)`. A coverage metrics assigns a number to a coverage criterion measuring the degree of coverage. The coverage metrics `qRequirementCoverageMetric` in Listing 4.8 measures the requirements coverage by the ratio of requirements with an assigned test to the overall number of requirements.

In general for every coverage criterion a corresponding metrics can be defined [Bin99]. We therefore identify coverage criteria and metrics in the remainder of this document if it is not essential to differ criteria from metrics.

Another important coverage criterion is the all services coverage criterion (ASC) which means that *from every service at least one operation is invoked in at least one test*. This coverage criterion is defined between the system and the test model. This informal definition results in the OCL definitions shown in Listing 4.9.

```
context TTS::Service
  def isServiceInvokedByCall:
    isServiceInvokedByCall() : Boolean =
      self.provides -> exists(i
        | i.getAllOperations()
          -> exists(o
          | TTS::Servicecall.allInstances()
            -> collect (s | s.operation)-> includes(o)
          )
       )

   def isServiceInvokedByTrigger:
        /* the same as isServiceInvokedByCall
           but with TTS::Trigger.allInstances() */

   def isServiceInvoked:
     isServiceInvoked() : Boolean =
       self.isServiceInvokedByCall() or
       self.isServiceInvokedByTrigger()

context Model
  def allServicesCoverage:
    allServicesCoverage() : Boolean =
      TTS::Service.allInstances()
      .isServiceInvoked()->forAll(e | e = true)
```

Listing 4.9: OCL Definitions for ASC

Listing 4.9 contains four definitions. The definition `isServiceInvokedByCall` checks whether at least one operation of a specific service is invoked by at least one service call of an arbitrary test. The definition `isSericeInvokedByTrigger`, which is not printed to keep the listing clear, analogously checks whether at least one operation of a specific service is invoked by at least one trigger. The definition `isServiceInvoked` checks whether a specific service is invoked by at least one service call or trigger. Finally, the criterion `allServicesCoverage` checks whether all services are invoked once in an arbitrary test.

For additional information we have defined a metrics based on the definition `allServicesCoverage` which computes the number of services covered by a set of tests. The metrics prints the number and the ratio of all covered services. For the callmanager case study (see Figure 3.27) we obtained the following result: `3 from 3 services have a test regarding All Services Coverage.(100%)`.

In [AO08] a coverage–driven approach to software testing is discussed. There are four different types of testing and corresponding coverage criteria distinguished, i.e., graph coverage, logical expression coverage, input space partitioning, and syntax–based coverage. In Table C.1 we have redefined the coverage of [AO08] for the TTS metamodel. We determine metamodel elements in TTS appropriate to define graph coverage, logical expression coverage, and input space partitioning. In the TTS profile, behaviors provide graphs, decisions provide logical expressions and types plus services provide partitions of the input space. We do not have explicit grammar definitions and therefore syntax–based coverage is not suited in our respect.

In TTS we have implemented coverage checks for the following tasks:

- all requirements coverage, i.e., each requirement is tested by at least one test
- requirements hierarchy coverage, i.e., each requirement and all its subrequirements are tested
- all services coverage, i.e., from each service at least one operation is tested
- full services coverage, i.e., from each service all operations are tested
- all services all interfaces coverage, i.e., from each service one operation of each service interface is tested

In the context of the coverage criteria listed above, testing an operation means that a specific operation is invoked by one service call in an arbitrary test. Table C.1 defines a comprehensive catalog of additional coverage criteria based on the classifications in [AO08] and [UL07] that can be implemented for specific TTS testing projects.

4.4 Test Model Transformation

In the TTS framework test models are the source and the target of model transformations. In Section 3.5.4 we have already presented the model–to–text transformation, i.e., test code generation which transforms test models as source models to executable test code in Java as target models. In this section we sketch a transformation from abstract workflows as source models to test models as target models.

The aim of TTS is to simplify and support the manual design of tests by providing a concise notation for test models. For some stakeholders, e.g., domain experts, even test stories as presented above may be too technical. Instead, abstract and computation–independent workflows may be more suitable. In such a situation the transformation from abstract computation–independent workflows to executable TTS test stories is useful. In this section we sketch a notation for high–level test stories and its transformation to executable test stories.

In this thesis we present a model–driven methodology for the manual definition of tests and support for it, but we do not consider automatic test generation techniques from the system model as in classical model–based testing. We only sketch related test generation techniques in the related work section of this chapter (see paragraph *TestGeneration* in Section 4.5) and show how these approaches could be integrated into TTS to support the automatic generation of tests. Therein we sketch which techniques can be applied in the context of TTS to integrate support for automatic test generation.

An arbitrary test story of the callmanager case study contains service calls, calls of configuration services, and triggers (see Section 3.4). All these types of calls are directly executable because they correspond to operations on running services.

Executable test stories can be directly translated to executable test code as shown in the implementation of the case study (see Section 3.5). In a technical context and for technically skilled test engineers this might be feasible. But when practitioners have to define tests or if only high–level business workflow descriptions are available, then *abstract test stories*, i.e., test stories which just contain abstract calls and assertions, may be more suitable for the definition of system tests. Abstract test stories are modeled as activity diagrams or as sequence diagrams.

An *abstract call* describes abstract business operations that cannot directly be bound to executable service calls [CD08]. An abstract call can be mapped to more than one executable service call, exactly one executable service call, or it can be part of one service call. We assume that an abstract call is part of an executable call or that it can be implemented by an adapter which corresponds to an executable service call. Abstract calls correspond to the metamodel element Call in the TTS metamodel.

Abstract test stories cannot be transformed directly to executable test code. But it is possible to transform abstract test stories to executable test stories.

The transformation from abstract test stories to executable test stories is semi–automatic because it depends on technical knowledge how to map each abstract call to an executable call. First, the information how to execute abstract calls has to be annotated manually and results in an *annotated test story*. The annotated test story can then be transformed automatically by a model–to–model transformation to an executable test story. The transformation steps are depicted in Figure 4.7.

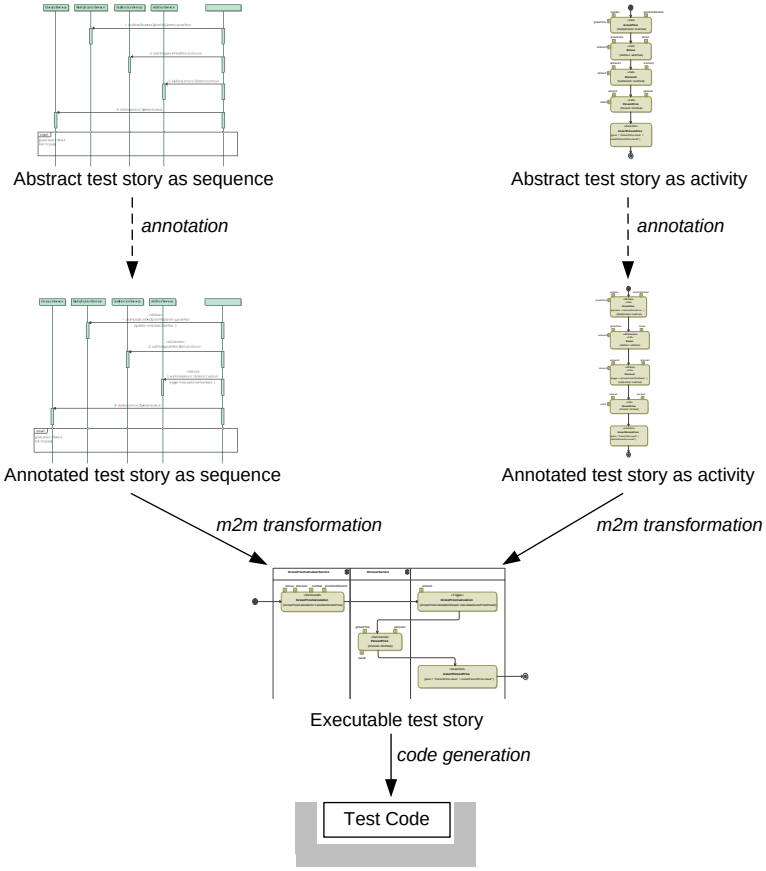


Figure 4.7: Overview of the Transformation Steps

First the abstract test story represented as sequence or activity is annotated which results in an annotated test story. The annotated test story is then automatically transformed to an executable test story by an endogenous and vertical model–to–model transformation which refines the abstract test model to an executable test model based on the same metamodel.

The transformation from abstract test stories to concrete test stories can be automated if annotations are applied. If there are no annotations, it is assumed that the abstract call can be mapped to an executable service with the same name.

According to our assumption, a group of abstract calls can be executed by one service call. The abstract calls of a group can be annotated by the following elements:

- WFStart indicates the first element of an executable group. It contains the operation to call and the caller instance.

- WFElement indicates a intermediate element of an executable group.
- WFEnd indicates the last element of an executable group. It contains a possible trigger operation indicating the end of the workflow.

If the caller instance is not specified, then it is a default instance identified by the service name. Test stories containing annotations WFStart, WFElement, or WFEnd are called *annotated test stories* and can be transformed to concrete test stories.

The transformation from abstract test stories to concrete test stories consists of two steps. In the first step, abstract test stories are transformed to annotated test stories and in the second step annotated test stories are transformed to internal test stories.

The first step of the transformation is done manually by annotating WFStart, WFElement, and WFEnd elements to service call elements. Specific checks have to guarantee that the annotations are assigned in a correct way. Each WFStart element needs exactly one succeeding WFEnd element with the same trigger on all of its outgoing paths to define the begin and the end of a set of abstract calls that are executed by one service call. The WFStart element needs an assigned operation that invokes the external workflow. If the trigger of the element WFEnd is defined then it specifies the callback operation and the workflow execution is asynchronous, otherwise the operation is synchronous. If a WFStart and a WFEnd element are assigned to the same abstract call, then the abstract call is mapped to the executable service call tagged to the WFStart element.

The general consistency, completeness, and coverage criteria defined in the previous section can be applied for abstract, annotated and executable test stories.

The transformation from annotated test stories to internal test stories is executed automatically. First, *workflow blocks*, i.e., WFStart elements and all associated WFEnd elements, are identified. Workflow blocks are replaced by a service call, which invokes the operation of the WFStart element, and optionally a trigger element if it is set for the WFEnd elements of the workflow block. The input parameters of the workflow block's service call are all free input parameters of all replaced service calls of the workflow block. For all service call elements, lifelines for the providing services are added and the respective service calls are added to it.

In the following, we sketch the transformation from abstract test stories to executable test stories by an example. Consider the abstract test `TestPersonPrice` depicted in Figure 4.8. It describes the scenario of calculating the price per person of tickets considering discount and bonus values. The behavior is presented in an abstract way independent from technical details. First, the gross price is computed (`GrossPrice`) by a multiplication, then the bonus is added (`Bonus`) and a discount (`Discount`) is subtracted. Afterwards the price per person is computed

(PersonPrice). Finally, it is checked whether the expected and the actual result are equal.

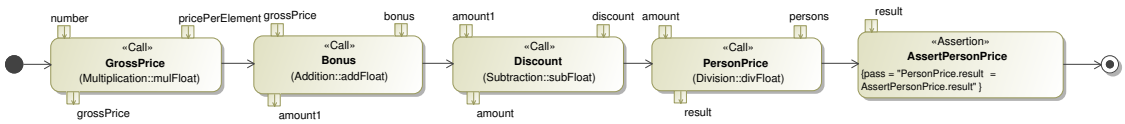


Figure 4.8: Abstract Test TestPersonPrice for Person Price Calculation

To make the tests in this section understandable and readable, we visualize all input and output parameters of calls, and the free variables of assertions by pins. The corresponding test data table TestPersonPriceTestData shown in Table 4.2 contains three test cases that are expected to pass.

id	number	pricePerElement	bonus	discount	persons	result
1	5	50	0	10	4	60
2	5	50	30	0	4	80
3	1	10	0	0	1	10

Table 4.2: Test Data TestPersonPriceTestData for Person Price Calculation

In Figure 4.9, the annotated test story corresponding to the abstract test TestPersonPrice is depicted.

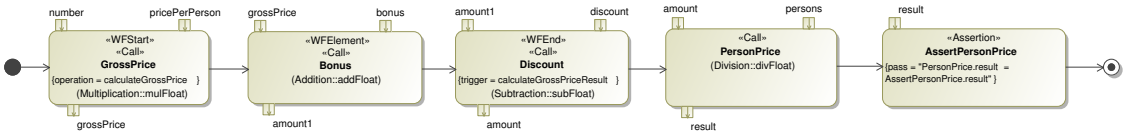


Figure 4.9: Annotated Test TestPersonPriceAnnotated for Person Price Calculation

In the annotated version of TestPersonPrice, the three service calls GrossPrice, Bonus, and Discount are annotated by WFStart, WFElement, and WFEnd, respectively because it is assumed that these three service calls are all executed by the service GrossPriceCalculator. In the next step after the annotation, it is possible to transform the test story to an executable one that can be directly transformed to executable test code. In the case of TestPersonPrice, the executable test TestPersonPriceExecutable is depicted in Figure 4.10.

The test TestPersonPriceExecutable contains a service call for the operation calculateGrossPrice of the service GrossPriceCalculationService, and a trigger for the operation calculateGrossPriceResult. The set of input parameters for the service call GrossPriceCalculation and the replaced service calls have to be equal and do not have to be transformed. The lifeline types are generated automatically due to the service types to which the invoked service calls belong, i.e., GrossPriceCalculator for the operation GrossPriceCalculation:calculateGrossPrice and DivisionService for Division:divFloat.

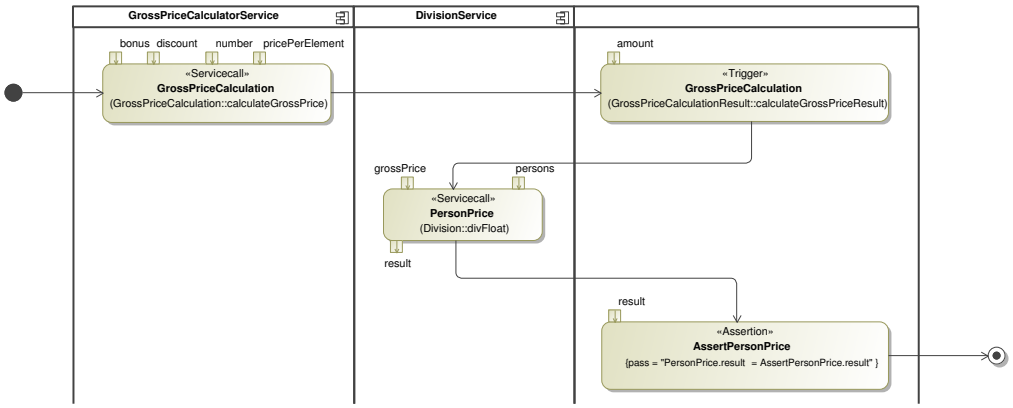


Figure 4.10: Executable Test `TestPersonPriceExecutable` for Person Price Calculation

In Figure 4.11 a possible execution semantics of the concrete test `TestPersonPrice` is depicted.

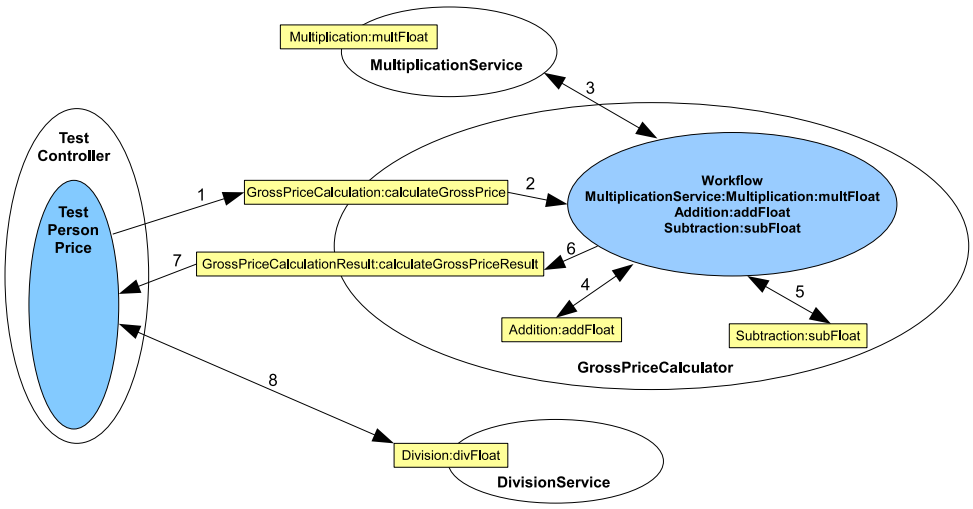


Figure 4.11: Workflow Execution Process

The test controller first executes the service operation `GrossPriceCalculation:calculateGrossPrice` on the service `GrossPriceCalculator` which triggers a workflow that sequentially executes `multFloat` which is implemented by a service `MultiplicationService`, `addFloat`, and `subFloat` which are both implemented locally in the service `GrossPriceCalculation`. Then the test controller invokes the operation `Division:divFloat` on the service `DivisionService`.

In Figure 4.12 the sequential execution of the test `TestPersonPrice` is depicted. It reflects the execution semantics of the abstract workflow without annotations. The test controller executes each service call (`multFloat`, `addFloat`, `subFloat` and `divFloat`) of the services (`MultiplicationService`, `AdditionService`, `SubtractionService`, and `DivisionService`, respectively) in a sequence.

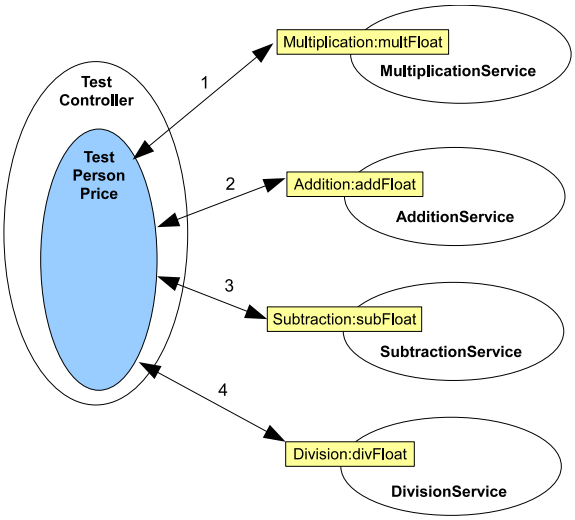


Figure 4.12: Sequential Workflow Execution Process

Depending on the context, abstract test stories can also be modeled by sequence diagrams. In Figure 4.13 the test `TestPersonPrice` is depicted in a sequence diagram. We only depict the annotated test story, because the abstract test story is similar but has no annotated stereotypes.

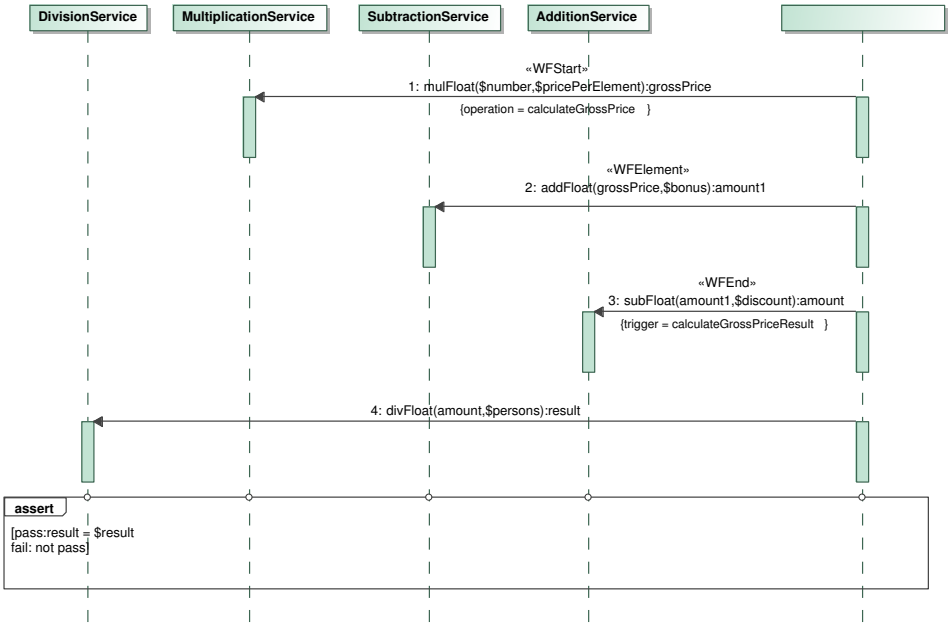


Figure 4.13: Annotated Test `TestPersonPriceAnnotated` in Sequence Diagram Representation

The abstract calls `multFloat`, `addFloat` and `subFloat` annotated by WFStart, WFElement, and WFEnd respectively are executed by the service call `calculateGrossPrice` and the trigger `calculateGrossPriceResult` of the service `GrossPriceCalculationService`. The abstract call `divFloat` is not annotated and therefore mapped one by one to an executable service call `divFloat`. The empty life-

line on the right represents the testing environment. Only the messages starting and ending in that lifeline are annotated with the workflow stereotypes and translated to executable tests.

The annotated test in Figure 4.14 – the abstract test story `TestPersonPriceConditional` is again not depicted because it is similar but has no annotated stereotypes – contains conditional elements and a workflow block with two WFEnd elements. The two WFEnd elements have the same trigger `calculateGrossPriceResult`.

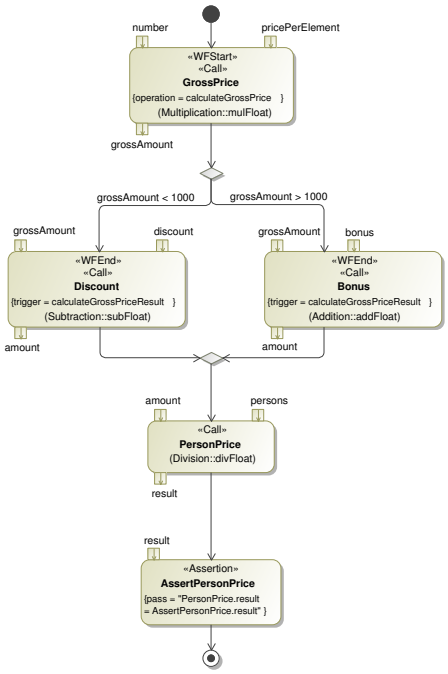


Figure 4.14: Annotated Test for Conditional Person Price Calculation

In the corresponding concrete test of Figure 4.15, the WFStart element `GrossPrice` has been replaced by the service call `calculateGrossPrice` of the service `GrossPriceCalculator` and the two WFEnd elements `Discount` and `Bonus` are replaced by the trigger `GrossPriceCalculation`.

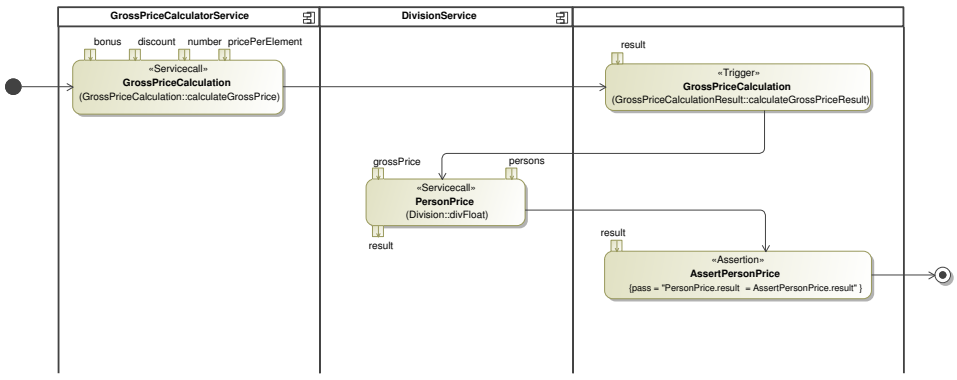


Figure 4.15: Concrete Test for Conditional Person Price Calculation

The transformation of conditional flows shows that workflow blocks can also include control flow elements. The example reflects that each WFStart element needs exactly one succeeding WFEnd element with the same trigger on all of its outgoing paths – `calculateGrossPriceResult` in this case – to define the begin and the end of a set of abstract calls that are executed by one service call.

The test execution is based on the executable test story. Therefore the test results can only be linked to the whole workflow block. A more fine grained model feedback is not possible. But on the level of abstract tests the most important information is just whether a test case passed or failed. The stakeholder defining abstract tests is not interested in more technical information. Because the tests itself represented by lines in the corresponding test table are also colored, the restricted model feedback is not relevant in practice.

4.5 Related Work

In this section we discuss related work to the formal foundations of TTS defined in this chapter. We consider related work on the formalization of TTS, on arbitrations, on model validation, on transformations, and provide test generation techniques that can be integrated into TTS.

Formalization. Generally, in software engineering the syntax and semantics of domain specific or general modeling languages like UML is typically defined in a semi–formal way based on the abstract syntax of the language. In [OMG07b] the UML and in [OMG05] the U2TP is mainly defined by its abstract syntax, its concrete syntax, a textual description of the semantics and many examples illustrating its meaning. The abstract syntax is defined in class diagrams by analogy to our abstract syntax in Section 3.3. Our formalization defines the concepts of the metamodel–based sets and functions. This way of defining concepts is similar to the syntax definition of UML objects models in [Ric01]. We provide a formalization but not a formal semantics for simulating models which is out of scope of this thesis.

Arbitrations. Regarding arbitrations some standardization and formalization proposals appeared in recent years. There are two important *standards for model–driven testing* which consider basic arbitration strategies, namely the Testing and Test Control Notation Version 3 (TTCN–3) [WDT+05] and the UML 2.0 Testing Profile (U2TP) [BRDG+07]. Both standards provide domain specific modeling languages for test specification and evaluation with a set of possible verdict types based on [ISO94] and an order over them. They define some basic arbitration strategies and offer user defined strategies, but give no further details on this topic. In our approach we rely on a subset of the verdict types defined in the standards, which can be easily extended by other types. We go further than the standards as we give

precise definitions of verdict functions.

An important work in the field of *arbitration formalization* is [Hie08], where a function returning a verdict over a set of observations is defined. This work gives formal background for behavior based verdicts, introduces formal order and properties of different verdict types. The functions defined in [Hie08] do not cover timing aspects, whereas in our approach we define verdict functions for behavior and performance in a pragmatic and uniform manner.

Complex arbitrations calculated over verdict functions can be seen as metrics over test runs. They are important from the practical point of view as they enable estimation of the current progress and predict future test efforts [Fre08]. The test progress monitoring can be achieved by *test reporting*. Among tools supporting creation of reports we can distinguish between general purpose tools, such as BIRT [bir], or tools dedicated to test reports, such as open source Testopia[2] or the commercial spiraTest[3]. General purpose tools offer flexibility, but need a lot of tailoring to be used in test reporting. Test report tools are often integrated into complex test environments and support customization of reports at the level of single test results.

Model Validation. There are several approaches for the validation of UML–based models. These approaches are either formal, i.e., mapping a UML model onto an mathematical formalism such as process algebras [EHHS02] or Z [SF97], or design–based, i.e., using a metamodel and OCL in order to analyze properties [Hna01]. In [LC04] it is shown that design–based approaches as implemented in TTS support the detection of incompleteness and inconsistencies as well as the quantification of completeness, while it does not require the more complex machinery of fully formal methods.

While validating consistency of UML models has received greater attention by researchers in recent years [EB04], completeness has also been addressed [LC04]. Even the interplay between consistency and completeness has been investigated [ZG03].

Only a small number of approaches such as [PK04] consider consistency and completeness of test models, i.e., behavioral descriptions of operations, but not consistency and completeness between requirements, system, and test models as in TTS.

Comprehensive collections of coverage criteria are defined in [UL07, AO08]. In [AO08] a coverage–driven approach to software testing is discussed. There are four different types of testing and corresponding coverage criteria distinguished, i.e., graph coverage, logical expression coverage, input space partitioning, and syntax–based coverage. In our integrated approach, we use artifacts of our metamodel as sources for test coverage. Behaviors provide graphs, decisions provide logical logical

[2]Testopia is a test case management extension for Bugzilla available at `http://www.mozilla.org/projects/testopia/` [accessed: November 25, 2010].

[3]The spiraTest tool is available at `http://www.inflectra.com/SpiraTest/` [accessed: November 25, 2010].

expressions and types or services provide partitions of the input space. We do not have explicit grammar definitions and therefore syntax–based coverage is not suited in our respect.

[MP05] collect structural UML–based coverage criteria for class diagrams, sequence diagrams, communication diagrams, state machines, activity diagrams and use case diagrams. It includes coverage criteria for state machines and activity diagrams from [OA99] where test generation from UML specifications is discussed. Our approach provides specific model–based coverage criteria for service–centric systems supporting the manual or automatic test definition.

Model Transformation. [BCD+06] describes a model transformer for test generation from system models. It automatically transforms system models denoted in a subset of UML to test models in U2TP. The approach is related to TTS because it defines a test specific model–to–model transformation from a system model to a test model in UML. But [BCD+06] only considers the transformation of the static test structure including the test components and the test context but not the transformation of the test behavior.

There are some model–to–model transformations between different types of behavior diagrams available. [PS00] defines a transformation from sequence diagrams to activity diagrams by using graph transformations. We propose a similar transformation from annotated test stories modeled as sequence diagram to executable test stories modeled as activity diagrams. Additionally, we consider specific annotations by stereotypes. In [Cib09] BPMN models [OMG09a] are transformed into UML activities, but testing aspects are not considered.

Test Generation. As mentioned in Section 4.4, we do not only consider related model–to–model transformations from system models to test models but also test generation techniques that could be integrated into our framework to provide support for automatic test generation.

Service calls in TTS refer to operations specified with *preconditions and postconditions*. Therefore testing techniques based on pre- and postconditions can be applied in TTS. There are many notations for expressing pre- and postconditions on operations [UL07] including B, Z, JML, Spec# and OCL.

Following [MFC+09] we have integrated the evaluation of preconditions and postconditions in TTS, i.e., for every precondition of a service call that evaluates to false, a verdict inconclusive is assigned, and for every postcondition of a service call that evaluates to false, a verdict fail is assigned. Additional pre- and postcondition based testing techniques for test generation have not been considered so far but can principally be integrated into our framework. We present some approaches for automatic test generation from pre- and postconditions and highlight OCL–based approaches.

An approach to *test data generation* is discussed in [BBH02] where a partition

analysis technique for test data generation from OCL specifications is presented. This approach could be integrated into TTS to define partitions and test data for specific input parameters of service calls. In general, preconditions are well–suited to generate test data, e.g., by random generation techniques [CRM07].

There are several approaches considering *test case generation* from pre- and postconditions. In [NFLTJ06] preconditions and postconditions expressed as boolean expression are assigned to use cases. Then, a use case transition system is created by a simulation algorithm that tries to apply each instantiated use case from the current state. In [UL07] a similar algorithm for test case generation from pre/post models in B is defined. Also for TTS a similar approach is possible to automatically generate tests as sequences of service calls and assertions under consideration of pre- and postconditions.

[ASA05] uses constraint solving to generate tests from mutated OCL specifications which would also fit for TTS, to generate tests as sequences of service calls and assertions.

Depending on the application context, local or global workflows can be modeled by UML activity diagrams, state machines or sequence diagrams. Therefore *test generation techniques for behavioral diagrams* which directly generate tests from behavioral artifacts of the system model can also be integrated into TTS.

UML activity diagrams are well suited for modeling processes and workflows. There are some approaches that consider activity diagrams as directed graphs and extract specific paths as test sequences, e.g., [KKBK07, LJX+04]. Such approaches can be integrated with TTS, because the test sequences extracted from activity diagrams can be considered as tests in TTS including service calls and assertions.

UML state machines model the internal behavior of systems. There are many state machine based approaches for test generation, e.g., [UL07, OA99] which simulate state machines and extract test cases. If system artifacts in TTS are modeled as state machines such approaches may be useful to generate tests.

UML sequence diagrams are used for specifying the behavior of distributed processes. In [LS06] systems are modeled as sequence diagrams including neg–operators to define invalid message sequences and assert–operators to define assertions and transformed to sequence diagrams consisting of only one lifeline modeling tests. The test model sends messages to the system under test and observes the received messages. Such tests can easily be transformed to basic test stories and therefore an integration with TTS is possible.

Note that the system model in our respect is typically incomplete and that therefore the test generation techniques discussed in this section just enhance the manual definition of test models but do not replace it.

4.6 Summary

In this chapter we have provided formal foundations for the Telling TestStories methodology.

In Section 4.1 we have formalized the basic concepts of the abstract syntax to define the arbitration mechanism in Section 4.2 and the test model evolution in Chapter 6 precisely.

In Section 4.2 we have defined arbitrations as high level decision criteria for tests. We have defined behavioral and performance arbitration criteria and shown how we have implemented the arbitration mechanism in the TTS tool.

In Section 4.3 we have discussed the validation framework of TTS which is based on SQUAM. We have presented the process of query definition plus evaluation, and have categorized the validation rules. We then have discussed the various types of validation rules, i.e., consistency, completeness and coverage by presenting concrete examples.

In Section 4.4 we have sketched a model–to–model transformation from abstract test stories to executable test stories which can be automatically transformed to test code. The transformation is semi–automatical because abstract test stories are annotated manually and then automatically transformed to executable test stories.

In Section 4.5 we have presented related work for formalization, model validation, model transformation and test generation.

Chapter 5

Security Requirements Testing

Security in IT is like locking your house or car - it does not stop the bad guys, but if it's good enough they may move on to an easier target.

Paul Herbka

In this chapter, we apply TTS for testing security requirements of service–centric systems. We show how security requirements can be specified as functional requirements according to the TTS methodology such that tests of security requirements can be treated within TTS as system tests. Functional security tests, as defined in our approach, are more powerful than tests with other security testing approaches which only consider tests of access control policies. It integrates policy testing into the overall system testing process and does not test them in an isolated way as in other approaches.

Our approach guarantees traceability between security requirements, the system and test model and the executable service–centric system. We claim that TTS is an intuitive system testing approach which allows also non–experts to define security tests which is in many other approaches only possible for experts who are able to formulate an attack model. Security tests can be modeled in the same way as functional tests. Our approach therefore provides information which security requirement is fulfilled and not just negative information claiming which requirement is not fulfilled. The security requirements testing approach presented in this chapter is mainly based on our security testing approach for TTS published in [FAB10].

We first define the basics of security and security testing (Section 5.1), then we present security requirements testing via a case study from the home networking domain (Section 5.2), afterwards we present related work (Section 5.3), and finally we sum up (Section 5.4).

5.1 Security and Security Testing

In this section we consider the basic concepts of security requirements testing. We first define security, then a classification of security requirements, and finally security testing.

5.1.1 Security

Security aims at protecting the information and the services from attacks which stretch from performing unauthorized access of information or services, tampering information, to targeting the services for shutting them down or slowing them [Mou10]. The main objective of security is to guarantee security requirements such as authentication, confidentiality, availability, authorization, integrity, and non–repudiation (see Section 5.1.2 for details on security requirements). This objective is twofold:

- Protecting the information and the services
- Providing services that run in an acceptable way and fulfill users' requests

Security is considered as *non–functional* property defining how a system is supposed to be (in contrast to what a system does). Threats to security are due to faults and flaws. Faults may lead to failures harming security requirements, and flaws are security problems which may lead to vulnerabilities. From a technical point of view, security can be classified into three main levels. Each level has its own security threats and on the shielding side, the corresponding security requirement to deal with these threats. The three security levels are as follows [Mou10]:

- *Network Security* involves tackling threats targeting the network. The main threats are (distributed) denial–of–service or network intrusion.
- *Operating System Security* is related to the operating system and includes protecting against all sorts of malware (virus, worm, spyware, etc.). The protection mechanisms involve antivirus, anti–malware, operating system level access control mechanism, firewalls, etc.
- *Application Security* is composed of the threats targeting a specific application. It includes unauthorized access, information theft, and misuse of the application. The security mechanisms include access control policies, application level encryption/decryption, etc.

All levels of security can be subject to software tests. In this thesis we focus on application security and consider the network and operating system as secure.

5.1.2 Security Requirements

Security is a quality factor of software systems and can be decomposed into a hierarchy of underlying quality subfactors [Fir04]. These quality subfactors are subject for testing to assure proper compliance. This means that if the quality requirements on the subfactors are met, a system is considered *secure* with respect to the tested factors. Nevertheless, it should be noted that the absence of problems cannot be demonstrated by testing, only their presence [Dij69].

Several classifications of security requirements can be found in the literature, e.g., [Fir03, Com09]. In this thesis we apply one of the most prominent lists of security requirements. In the following, we list the security requirements and give their definitions according to [NST06]:

- *Confidentiality* is the assurance that information is not disclosed to unauthorized individuals, processes, or devices.
- *Integrity* is provided when data is unchanged from its source and has not been accidentally or maliciously modified, altered, or destroyed.
- *Authentication* is a security measure designed to establish the validity of a transmission, message, or originator, or a means of verifying an individual's authorization to receive specific categories of information.
- *Authorization* provides access privileges granted to a user, program, or process.
- *Availability* guarantees timely, reliable access to data and information services for authorized users.
- *Non–repudiation* is the assurance that none of the partners taking part in a transaction can later deny of having participated.

It is possible to introduce additional subclassifications into this list, e.g., according to [Com09]. The proposed approach to security requirements testing is independent of the applied security requirements classification.

Security requirements can be classified into *positive requirements* and *negative requirements* [G. 97]. Positive requirements define what a system should do, e.g., "user accounts are disabled after three unsuccessful login attempts" or "network traffic must be encrypted", whereas negative requirements stress what it should not do, e.g., "unauthorized users should not be able to access data".

5.1.3 Security Testing

Security testing is software testing of security. Depending on the system under test, security requirements on each level or a combination of them can be tested. Due to the public availability of software, security is a core property and exploited security vulnerabilities can cause drastic costs, e.g., due to downtimes or the modification

of data. Therefore a structured approach to security testing is of high importance. This is especially true for service–centric systems which are often exposed over the internet [CD08].

Typically there are two types of security testing [PM04]:

- *security requirements testing* which validates the proper implementation of security requirements.
- *risk–based security testing* which simulates the attacker's approach based on a risk analysis.

Risk–based security testing typically requires more expertise than security requirements testing because the tester has to think like an attacker and the test result is not always directly observable. Risk–based security testing is often about negative requirements, so called *abuse cases*, where an attacker tries to do something he is not permitted to do. In this thesis we focus on testing security requirements, which may be formulated in a positive or negative way. Our approach to test evolution by attaching state machines to model elements as discussed in Chapter 6 can be extended to handle risk–based security testing. We already consider security tests and their evolution but it is also possible to extend our approach by risk coefficients that can be used for designing, executing and evaluating tests. We consider this as future work.

Defects or deviating behavior is normally detected by checking specific properties of a system during execution. These checks are also called *assertions*. Security requirements are an adequate source for the definition of such assertions and provide additional constraints on functional requirements. However, the same security requirement can be violated in different parts of the system. Consider the following security requirement restricting access to a home automation environment: "Unregistered users are not allowed to access the domotics system.". For testing whether this authorization requirement is not violated it has to be checked at every point where a service of the domotics system is accessed, e.g., illumination, air conditioning etc. Resulting from the nature of negative requirements, a one–to–one mapping from the security requirement to code artifacts is not easily found in most cases [MR05]. Nevertheless, to know which parts of a system could be affected by deviating (security) behavior such information is necessary. This motivates the need for traceability from assertions to security requirements and possible consequences. We discuss our security testing approach based on a case study in the next section. Functional security tests, as defined in our approach, are more powerful than tests with other model–based security testing approaches like policy testing [JMT08, PMLT08] because our methodology extends the set of testable security requirements.

5.2 Home Networking Case Study

This section presents our security testing approach with the TTS framework. This is done with the help of a case study introduced in the following. We aim for testing security functionality of a system, hence our tests can be classified as security requirements tests.

The aim of the case study[1] is to control the network access of clients in a home network depending on different client attributes. Typical attributes are for example the age of a user or a check whether the anti-malware application has been updated during the last 24 hours. An example for a resulting effect after checking the client attributes is that an underage user may only be allowed to access a restricted set of resources of the network and is only allowed to access the internet and not the private home network, e.g., if access to a music collection should be denied. The scenario is depicted in Figure 5.1.

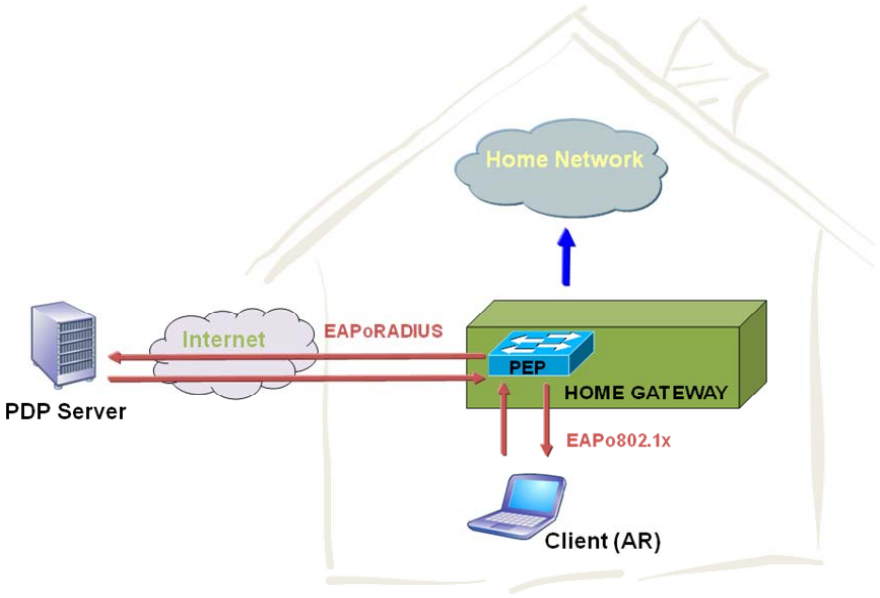


Figure 5.1: Overview of the Home Networking Case Study

The scenario consists of different peers, i.e., services distributed among the home network and the operator network. These peers are the *Access Requestor* (AR), *Home Gateway* (HG), *PolicyEnforcement Point* (PEP) and the *Policy Decision Point* (PDP). Following service–centric principles [Erl05], each service shares interfaces defining the terms of information exchange. The AR is the client application to establish and use the connection to the home network. An AR always connects to a HG. The HG is a device installed at the home of customers to control access to different networks and services (e.g., domotics, multimedia, data services).

[1]The case study was kindly provided by Telefónica (`http://www.telefonica.com` [accessed: November 25, 2010]).

The enforcement of who is allowed to access which resources on the network is made by an internal component of the HG called PEP. The PEP gets the policy it has to enforce for a specific AR by the PDP which is the only component run by the operator and not the end users themselves. Because we have four independent services only interacting via well-defined interfaces to execute a process, the example adheres to our definition of a service–centric system. Furthermore, we are focusing on testing dedicated example sequences, i.e., the test stories of the system, and verify whether certain security requirements hold under such conditions. The services are normally already tested in an isolated way before they are deployed in a service–centric setting. As mentioned earlier, the TTS framework adheres to a test-driven development approach, thus it allows the execution of test stories in early stages of system development and supports the evolution of the underlying system. Thus, we think that TTS is an ideal choice for a testing framework of the presented system.

We discuss the elicitation of some sample security requirements and their integration into the requirements model in a structured and traceable way. After that, we present the system model and, finally, we present the test model for testing the security requirements and discuss the technical realization for executing tests and asserting security requirements.

5.2.1 Requirements Model

The requirements model provides a hierarchical representation of the functional and non–functional requirements to the system. Security requirements and any other type of non–functional requirements can be integrated into this hierarchy in a natural way. The requirements hierarchy of the home networking application is depicted in Figure 5.2.

The scenario contains a functional requirement `Req_1` for controlling the network access of an agent–based network which is decomposed into the functional requirements `Req_1.1`, `Req_1.2`, `Req_1.3`, and `Req_1.4`, and a functional requirement `Req_2` for access service usage. To these functional requirements, security requirements are assigned. The Requirement `SReq_1.2.1` is an example for authentication, `SReq_1.2.2` for confidentiality, `SReq_1.3.1` for availability, `SReq_1.4.1` for authorization, `SReq_1.4.2` for integrity, and `SReq_2.1` for non–repudiation. To each security requirement as to all other requirements, model elements of the test model (test stories, assertions, test sequence elements) verifying the requirement can be assigned. In our basic requirements hierarchy security requirements are formulated as positive statements, defining how a vulnerability can be avoided. Attached test stories may then define possible vulnerabilities that make the requirement failing. In the requirements model we only consider the agent–based connection but in the system model we also integrate agent–less connection which is needed for the case study in Chapter 6.

Every functional requirement has at least one attached test and for all assigned security requirements an assertion in one of these tests exists. This guarantees

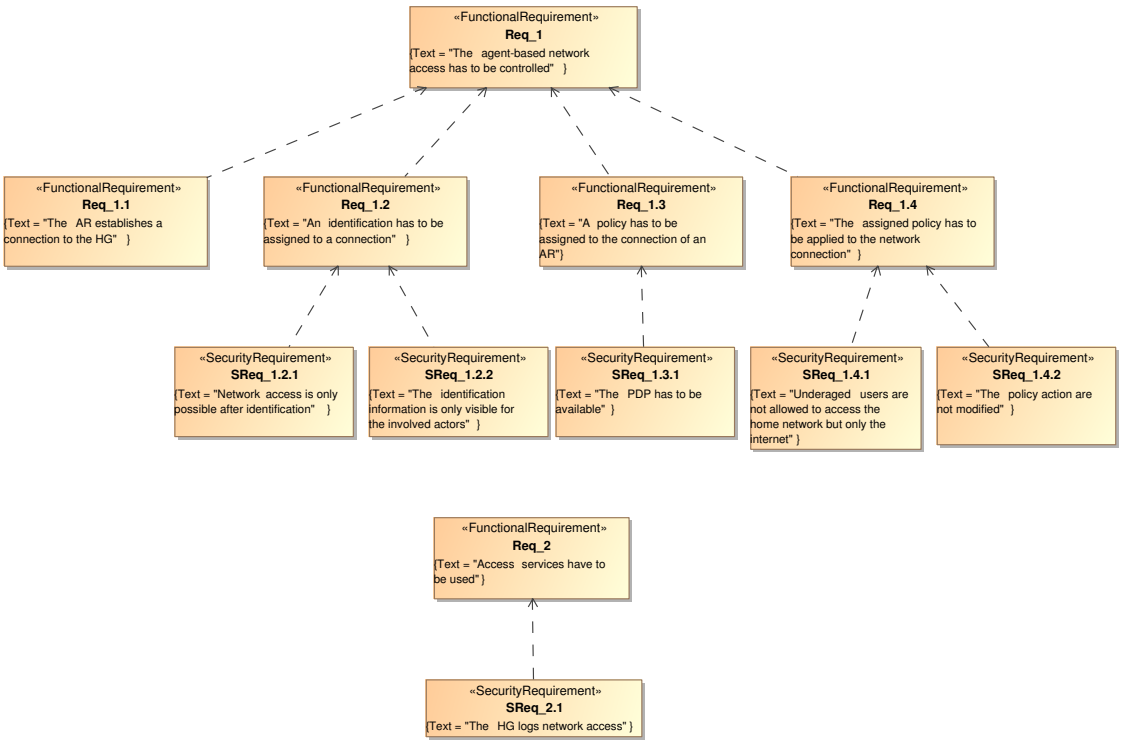


Figure 5.2: Requirements of the Home Networking Case Study

that all specified security requirements of a functional requirement are also tested. Thus, in our approach security requirements define security constraints on system functionality.

Requirements traceability refers to the ability to describe and follow the life of a requirement, in both a forwards and backwards direction [GF94]. Traceability has to be guaranteed by a system testing approach to report the status of the system requirements. Our representation of requirements allows the definition of traceability by links between model elements, i.e., by assigning test stories to requirements. Because service operation calls in test stories are linked to service operation calls in the system model which are linked to executable service operations in the system implementation, we have traceability between the requirements model, system model, test model and the executable system.

5.2.2 System Model

The system model is based on services for the AR, the HG, the PDP and the PEP. Additional services define a hierarchy on the basic services and provide additional system characteristics. We define the static system model by the definition of types, services, and interfaces.

In Figure 5.3 the types of the home networking system are depicted.

Besides some primitive class types `Integer` and `String`, the system contains the class types `Policy`, `AssessmentInfo`, `PolicyAction` with the sub-

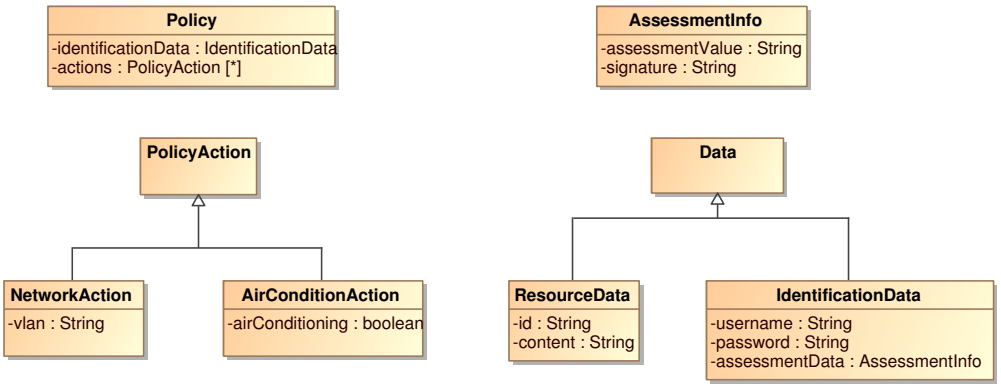


Figure 5.3: Types of the Home Networking Case Study

types `NetworkAction` and `AirConditionAction`, and `Data` with the subtypes `ResourceData` and `AirConditionData`.

In Figure 5.4 the services and in Figure 5.5 the interfaces of the home networking system are depicted.

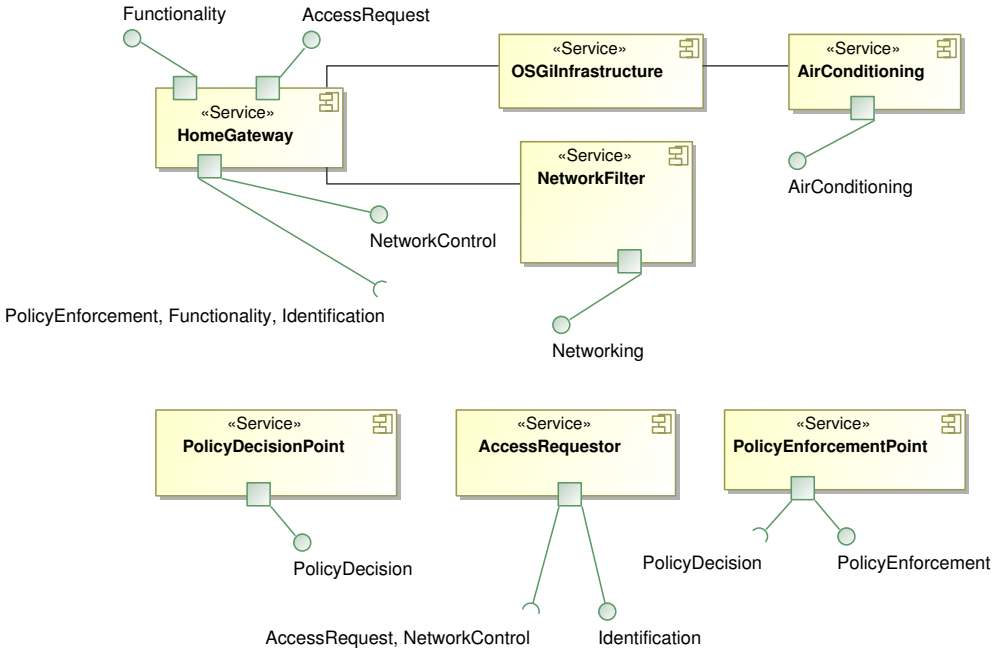


Figure 5.4: Services of the Home Networking Case Study

As already mentioned, the AR is the client application to establish and use the connection to the home network. We model this with an `AccessRequest` and `NetworkFilter` interface required by the AR, and an interface `Identification` provided by the AR. This required interface of the AR are provided by the HG because an AR always connects to a HG. The HG additionally provides the interface `Functionality` and requires the interfaces `PolicyEnforcement`, `Functionality`, and `Identification`. The HG has two

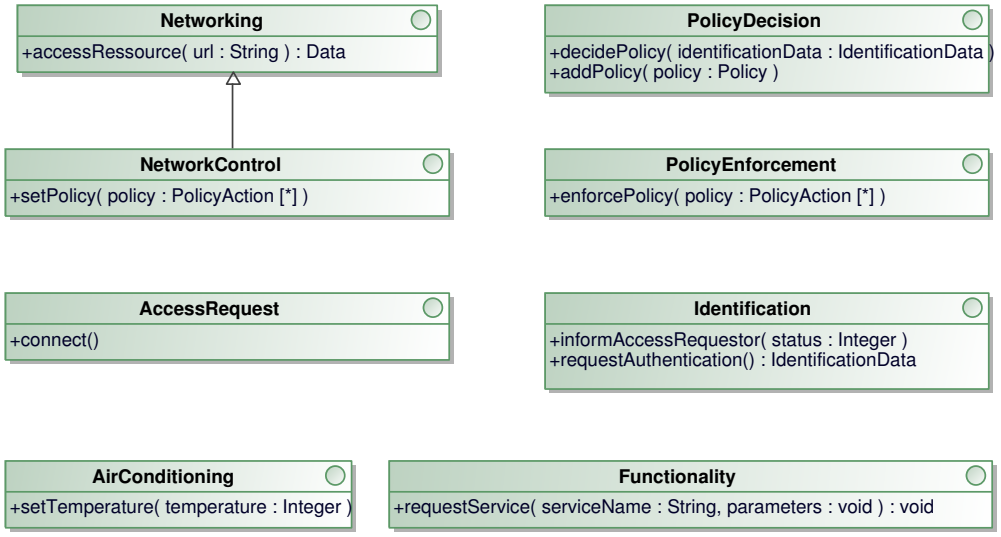


Figure 5.5: Interface Definitions of the Services of the Home Networking Case Study

subservices `OSGIInfrastructure` and `Networkfilter` which provides the interface `Networking`. The `OSGIInfrastrucutre` has a subservice `AirConditioning` providing an interface `AirConditioning`. The PDP provides the interface `PolicyDecision` and the PEP requires the interface `PolicyDecision` and provides the interface `PolicyEnforcement`.

All services in this scenario are connected with each other and the interfaces between them are well-defined (see Figure 5.4). A request by the AR will trigger the input of user credentials via the identification interface. The data returned by the AR is of type `IdentificationData` (see Figure 5.5). With this information the PDP is able to look up the appropriate policy for the request and send the corresponding list of `PolicyActions` to the PEP which enforces them. `PolicyAction` and `IdentificationData` are data types defined internally in the system model. The type `IdentificationData` describes a username, a password and optionally attestation data of an AR; the type `PolicyAction` is only used to describe to which VLAN the PEP should allow access by a specific AR. `AirConditioning` provides an additional service to the system and `Networking` and `NetworkControl` are needed for agent–less network control.

In the running system, the communication among the services is based on different protocols and standards. Authentication follows IEEE 802.1X standard [IEE04a] which defines a supplicant, an authenticator and an authentication server. In our case the supplicant is the AR, the role of the authenticator is taken over by the HG and the authentication server (a RADIUS[2] database) is represented by the PDP.

[2]RADIUS is an abbreviation for Remote Authentication Dial In User Service. RADIUS is specified in RFC 2865 [IET00].

5.2.3 Test Model

The test model contains a set of test stories whose execution order is defined in a test sequence, i.e., a top–level test. In case of the home networking case study the parametrized test stories are visualized by UML sequence diagrams which can be mapped to activities as presented in Section 4.4. The free parameters are denoted by variables with an initial '$' symbol.

Figure 5.6: Test `TestPolicy` for Testing Authorization and Integrity

The test story in Figure 5.6 defines a basic network access scenario containing two assertions. First the `AccessRequestor` connects to the `HomeGateway` (step 1 in Figure 5.6), which then requests the authentication data containing a username, a password and assessment data from the `AccessRequestor` (step 2). This information is forwarded to the `PolicyDecisionPoint` (step 3), which sends a sequence of policy actions to the `HomeGateway` (step 4) based on the identity information of the `AccessRequestor`. We then assert that there has to be a policy action that contains the expected VLAN (`$vlan`) to check the policy actions for integrity. The `HomeGateway` sends the policy actions to the `PolicyEnforcementPoint` (step 5), and then informs the `AccessRequestor` (step 6), which then accesses a specific URL (step 7). Finally, we check whether the `accessRessource()` call returns the expected data. Test cases for this test scenario are defined in Table 5.1 named `TestPolicyTestData`.

The test cases are defined in tables, each line corresponding to one test case. The columns correspond to the free variables in the test story `TestPolicy` which is integrated into the top–level test adding some policies to the `PolicyDecisionPoint` in an initial setup and executing the test `PolicyTest` (see Figure 5.7): Three policies are added to the `PolicyDecisionPoint` by the operations `addPolicy`.

Each policy assigns a sequence of policy actions, in our basic example just a sequence set of accessible VLANs, to a username/password combination. The iden-

#TC	$username	$password	$vlan	$url	$data
1	'michael'	'0815'	'HomeNetwork'	'http://74.125.43.99'	webpage_1
2	'michael'	'0815'	'HomeNetwork'	'http://192.168.1.1'	webpage_2
3	'philipp'	'0000'	'Internet'	'http://74.125.43.99'	webpage_1
4	'philipp'	'0000'	'Internet'	'http://192.168.1.1'	null
5	'guest'	'0000'	'Internet'	'http://192.168.1.1'	null

Table 5.1: Test Data for `TestPolicyTestData` for Test `TestPolicy`

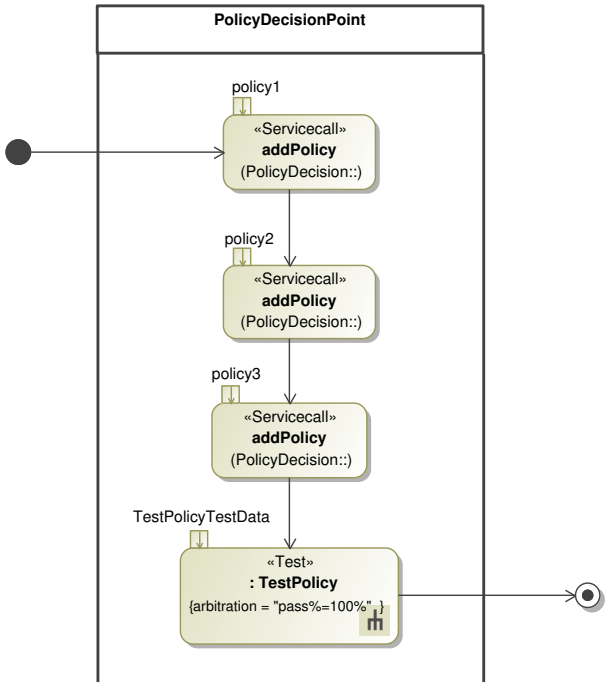


Figure 5.7: Test integrating the Test for `TestPolicy`

tification data objects are stored on our data pool and are as follows in our example:

```
policy1:('michael','0815',(['Internet','HomeNetwork'])
policy2:('philipp','0000',(['Internet'])
policy3:('*','*',(['GuestNetwork'])
```

The test `PolicyStory` executes the test `PolicyTest` for each line of the test data table `TestPolicyTestData` which is an input variable of `PolicyStory`. The test `PolicyStory` contains an arbitration `pass%=100%` that aggregates the verdicts of the stories' test cases and checks whether all test cases of a test story pass.

To make all requirements executable, we assign a test element to it which is in most cases an assertion, a service call or a test. The two assertions in the test `PolicyTest` are traceable to security requirements. In the requirements model of Figure 3.26, the first assertion can be assigned to the requirement `SReq_1.4.2` testing integrity, and the second assertion to the requirement `SReq_1.4.1` testing authorization. Additionally, the overall test story can be mapped to the functional require-

ment `Req_1.4` which is done implicitly in this case because the test story covers all sub requirements. For other types of security requirements, similar tests can be defined. Security requirements are tested if test elements are assigned to them. A requirement is tested and passes if all subrequirements pass and if all assigned test elements pass. TTS is appropriate for security testing on the system level based on security requirements. Security requirements as non–functional requirements differ from functional requirements that they just need an assigned assertion to be tested. Functional requirements need an assigned test to be tested adequately. These different restrictions, i.e., assignment of an assertion to a non–functional requirement and assignment of a test to a functional requirement, can be defined by OCL statements.

Our approach guarantees traceability between security requirements, the system and test model and the executable service oriented system. We claim that TTS is an intuitive system testing approach which allows also nonexperts to define security tests which is in many other approaches only possible for experts who are able to formulate an attack model. Security tests can be modeled in the same way as functional tests. Our approach therefore provides information which security requirement is fulfilled and not just negative information claiming which requirement is not fulfilled. Our testing approach is based on security requirements and can therefore be started very early in the software development process, even before the running system has been implemented.

In the remainder of this section we define two additional tests, `TestAgentless` in Figure 5.8 and `TestAirConditionControl` in Figure 5.9 in the context of the home networking case study that are referred to in Chapter 6 on test evolution.

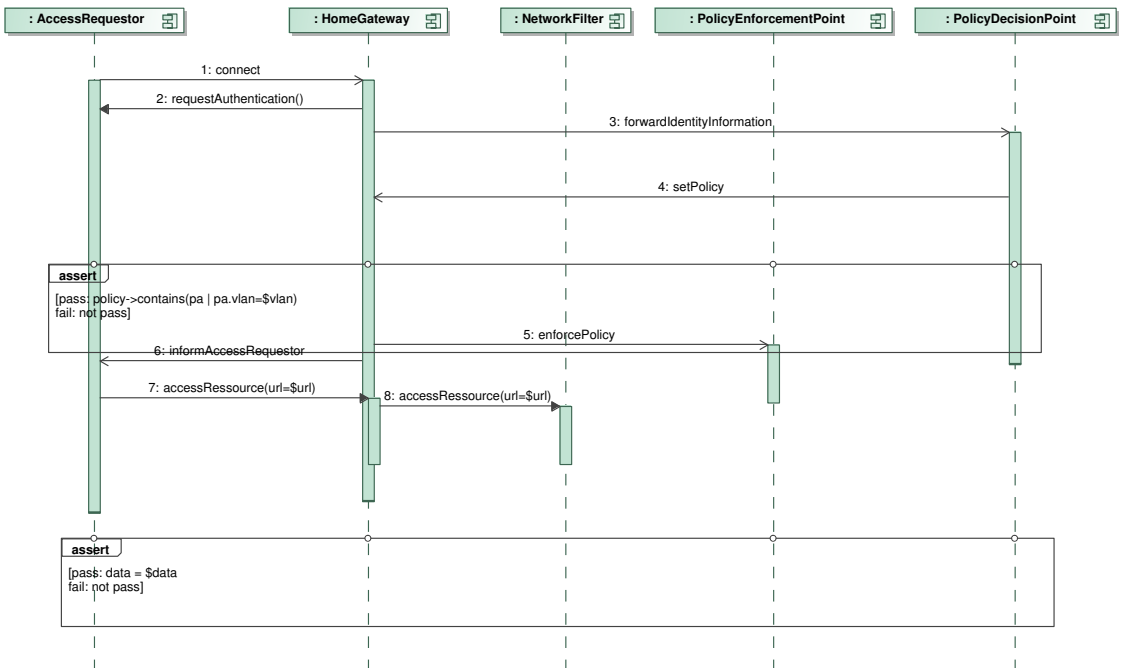


Figure 5.8: Test `TestAgentless` for Testing Agent–less Access to the Home Network

The test `TestAgentless` in Figure 5.8 validates a similar scenario to

`TestPolicy` but without an agent that checks the AR. We therefore need an additional service `NetworkFilter` on which `accessRessource` is invoked to check the request of the AR instead of the integrated agent.

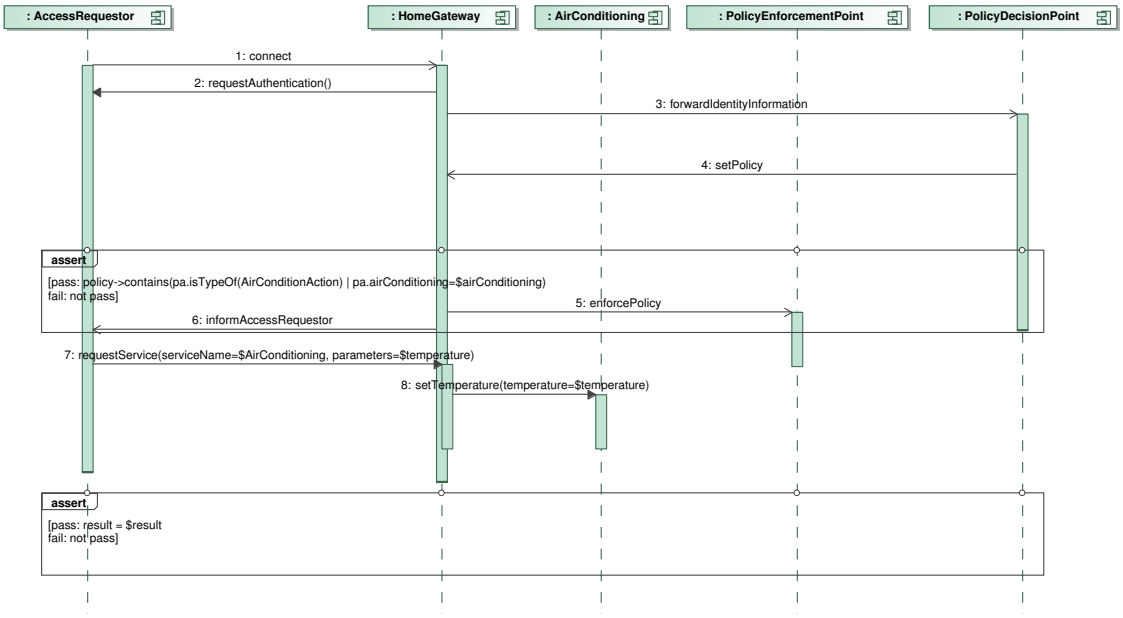


Figure 5.9: Test `TestAirConditionControl` for Testing the Air Condition Service

The test `TestAirConditionControl` in Figure 5.9 tests the functionality of an additional air conditioning service. The service provides one interface `AirConditioning` with one operation `setTemperature`. This operation is checked in an agent–based scenario by analogy the service `Networking` for accessing a resource.

5.2.4 Test Execution

The case study consists of various services communicating with each other. A crucial point of our testing strategy is that the different services are not tested individually and in an isolated way. Instead we define test stories which describe possible sequences of service invocations on the SUT. Testing each service separately is out of scope for our testing strategy because they may consist of third-party implementations, e.g., authentication server and access requester, and have normally already run through extensive test suites. What we are interested in is the behavior and value of certain parameters at specific points in a test story, i.e., the *assertions*.

Another important point of our test execution technique is that the test engine is primarily a passive participant in this process. Only messages invoked by the AR are actively called by the test controller. However, this is not a limitation of the Telling TestStories framework itself as mentioned in Chapter 3. The reason for a passive execution engine lies in the scenario itself: all services except the AR are hard-wired to each other. For instance, when the AR sends the `EAP-Response/Identity` message, i.e., the return value of the `requestAuthentication()`-call) to the HG this

will trigger a message exchange between the HG and the PDP. The AR is not aware of this communication as the AR is solely controlled by the HG. Thus, a central execution engine acting as an orchestration unit is not reasonable in this scenario because it would simply "miss" certain messages. The test execution technique for this scenario starts a test story and only interacts with the `AccessRequestor`. The parameters for this interaction are given in the data table. The remaining communication will only be observed by the execution engine. For monitoring this communication we use packet sniffers, e.g., TShark[3]) at various points in the environment. This allows us to track the full communication in a non-intrusive way, i.e., the architecture remains unchanged and no central orchestration point controlling the communication between the services has to be integrated.

Before the test story is started, the system is first set to a specific state. This is to set up the environment so that the SUT can be tested. In our case the setup consists of a number of `addPolicy()`-calls to the PDP for installing the policies in the test `TestPolicy` in Figure 5.7. After the system is initialized, the execution engine triggers the `connect()`-call by the AR. The HG then requests the user credentials from the AR in the `requestAuthentication()`-call. These are provided by the execution engine by consulting the data table, i.e., `$username` and `$password`. The next step where the AR is involved is the `informAccessRequestor()`-call where the AR is notified about the decision by the PDP. Immediately after this notification the AR can try to access a specific network resource via the `accessRessource()`-call. Again, the parameter for the requested URL is fetched from the data table, i.e., `$url`. The rest of the communication, where the AR is not involved, is only observed.

By monitoring messages, the execution engine is able to keep track of the current value of variables defined in the interfaces among services, e.g., which `PolicyActions` are returned by the PDP. This information and the content of the data table are sufficient to compose assertions and to check the behavior of the system. For example, the assertion `[pass: data = $data]` in the test story checks whether accessing a specific URL is allowed/denied as specified in the policy. This assertion can be evaluated by getting the value for `data` out of the monitored return value of `accessRessource()` and the value for `$data` out of the data table.

Technically, in the present setting there are two points where information has to be sniffed, i.e., IEEE 802.1X traffic between the AR and HG, and RADIUS traffic between the HG and PDP. For each captured message of a running test story the sniffer matches it to an interaction step of the test model and assigns the values according to the defined interface. If an assertion is encountered during this traversal then a verdict can be computed based on the current content of the variables. After the test execution the results can be evaluated as described in Chapter 3 on the TTS framework itself.

[3]The TShark tool is available at `http://www.wireshark.org` [November 25, 2010].

5.3 Related Work

Security requirements testing is based on the definition of security requirements. Several classifications of security requirements can be found in the literature, e.g., [Fir03, Com09, NST06]. We follow the most prominent list of security requirements in [NST06] which distinguishes six types of security requirements, namely confidentiality, integrity, authentication, authorization, availability, and non–repudiation. In principle our approach is independent of the applied security requirements classification but a classification provides a criterion for completeness, i.e., have all relevant security requirements been considered and can they be integrated in the classification.

Due to the openness of service centric systems, security testing for such systems has gained much interest in the last years [CDP06].

Many *risk–based testing techniques* and tools are available in industrial applications [MR09]. A very important test technique after a risk analysis has been performed and the system is implemented are penetration tests. Penetration tests [Bis07, vW08] attempt to compromise the security of a system by playing the role of a hacker trying to attack the system and exploit its vulnerabilities. Specifically, it tests missing functionality or side–effects of the system. This is often done by making the *environment* of a system the target of attack instead of the system itself, e.g., exploiting an unpatched operating system. Penetration tests are out of scope for our approach, because although we are aiming to detect false behavior of the whole SUT, we neglect its operation environment. Furthermore, it is an unsolved issue how to link penetration tests to specific system components which is a prerequisite for traceability. Penetration tests can only be applied to running systems whereas our approach can be applied already in the design phase. Vulnerability scanners are a specific class of tools applied in penetration testing, e.g., Nessus[4] constitutes a tool-based approach to perform security tests on a very low level. Furthermore, only known vulnerabilities are detectable with such tools.

Although there are many *security requirements testing* approaches available [MR09] only a few approaches are model–based such as our approach is. Model-based testing itself is well–covered in the literature as mentioned in Section 2.1.2 and many tools are already on the market supporting model-based approaches [GNRS09] but model–based security requirements testing is still an emerging field of research and industrial practice. We present a few representative approaches and put them into context with the work presented in the contribution at hand.

The aim of functional security tests is mainly the quality assessment of specified (security) requirements. In [JMT08, MJP+07] the authors describe a model-based testing approach for checking whether access control policies are properly enforced by the SUT. The functional model is written in the B language and used for the security test generation from so called *test purposes*. Test purposes are defined as

[4]The Nessus tool is available at `http://www.nessus.org` [accessed: November 25, 2010].

regular expressions and describe a general sequence of operation calls to induce a certain situation on the SUT. The approach aims at the automatic generation of test cases from the SUT. Our approach differs in several points: it supports test-driven development which also implies that test cases are not meant to be generated automatically but modeled by a domain expert, e.g., the customer; for the same reason, TTS allows for the execution of tests during system development. Our approach, furthermore, describes how the information passed among components is to be *interpreted* and how this information can be used to *check for compliance* with arbitrary security requirements and not only access control rules.

Another model-based approach for testing RBAC access control policies is presented in [PMLT08]. The creation of test targets (actual access requests) is based on three different strategies: (1) only taking into account roles and permissions, (2) considering all rules in a policy and (3) completely at random. The tests target the PDP to check whether its decisions are correct or not. Out of the actual rules in the policy mutants [DLS85] are created to check whether the tests are able to kill them, hence to assess the quality of the test generation procedure. Both approaches aim at the automatic generation of test cases out of information about the SUT. Our approach differs in several points: in our approach the test cases are not generated automatically but are modeled by a domain expert, e.g., the customer. Furthermore, TTS supports test-driven development which allows the execution of tests during system development. Such functionality is not integrated in the presented contributions. Additionally, for the scenario presented in this work an approach for generating different sequences of operation calls is not useful because of the passive viewpoint of the test execution engine. Our approach describes how the information passed among components has to be *interpreted* (interfaces) and how this information can be used to *check for compliance* with security requirements (test stories with assertions).

Other approaches, such as [WJ02], also use the system specification for generating security tests. However, our goal is not the automatic test case generation but a continuous connection among security requirements, tests and the system. Opposed to other security testing approaches, TTS supports a test-driven way of system development. This means that the system model does not have to be complete beforehand.

5.4 Summary

In this chapter we have applied TTS for testing security requirements.

In Section 5.1 we have defined the basic terminology of security and security testing. We have defined a classification of security requirements considering confidentiality, integrity, authentication, authorization, availability, and non–repudiation.

In Section 5.2 we have applied security requirements testing to a case study from the home networking domain. We have defined a requirements model, a test

model, a system model, the test execution, and traceability between the artifacts.

In Section 5.3 we have presented related work to security requirements testing and risk–based security testing.

Chapter 6

Test Model Evolution and Regression Testing

There is nothing permanent except change.

Heraclitus

In this chapter we consider the evolution of models and its influences on tests. The presented test model evolution and regression testing approach is based on our generic evolution management process for service–centric systems published in [FAB11b] and our evolution process specific for security requirements tests published in [FAB11a].

In this chapter we first motivate the problem (Section 6.1) and then wrap up the underlying metamodel (Section 6.2). Based on the metamodel we define the evolution process and its steps in detail (Section 6.3). We then employ the defined process on the home networking case study of the previous chapter (Section 6.4). Finally, we present related work (Section 6.5) and sum up (Section 6.6).

6.1 Motivation

So far we have considered systems as static. But a software system must evolve, or it becomes progressively less satisfactory [Leh80]. Although evolution has been investigated for model–driven system development [Bre10], such aspects have been neglected for model–driven *system testing* so far. Nevertheless, testing is very important during evolution [MvDZB08], and the maintenance of test models is a key factor to make model–driven testing applicable at all. The evolution of a system provides additional information that supports the selection of test models and therefore the generation of an optimal test suite by analogy to classical regression testing [RH96]. Regression testing is the selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component

still complies with its specified requirements [IEE90a]. In practice, the objective is to show that a system which previously passed a test still passes a test [PA09].

In a home networking systems as introduced in Section 5.2 only a few device types are initially integrated as actors and will successively be extended by new actor instances, improved underlying infrastructure or modified functionality. The importance for managing the evolution of service–centric systems is also reflected by a special volume on continual service improvement within the ITIL standard for service management [OGC07].

Service–centric systems are widely used nowadays and are subject to modifications which may harm their security. For instance, in a home networking system the authorization and authentication requirements have to be modified due to the successive inclusion of new actors, or the underlying infrastructure has to be changed due to new availability requirements. Based on the security requirements testing approach of Chapter 5, we consider changes of security requirements and their impact on functional requirements, services and tests.

In this chapter, we present an evolution management methodology for requirements, services and tests of dynamically evolving service–centric systems that considers security requirements and their tests. The main focus is on the overall evolution process, the impact of changes to tests, and the generation of regression test suites. Our approach is grounded on a state–model based computation of test models and allows for regression testing of service centric systems.

TTS supports the bi–directional information flow from test models to the running system via the traceability between modeled services and executable system services as presented in Section 3.3.6. The management of quality requirements is supported by tests as presented in the callmanager case study (see Section 3.4) for functional correctness and in the home networking case study (see Section 5.2) for security. TTS fosters the cooperation between IT-managers, who decide whether a product can be released based on the test report, quality managers, who are responsible for test design based on the test models, and administrators, who are responsible for the test execution based on the SUT and the test environment.

In our methodology, each changeable artifact has an attached state machine that defines the actual state of the artifact, and triggers or receives events to compute the new state. Following the widely used classification in [LW89], the *type of a test* can be *evolution* for testing novelties of the system, *regression* for testing non–modified parts ensuring that evolution did not impact parts supposed not to be modified, *stagnation* for ensuring that evolution did actually take place and changed the behavior of the system, and *obsolete* for tests which are not relevant any more. Based on the type of a test and a test requirement, a test suite is determined. This supports safe regression testing [RH96], which identifies all test cases in the original test set that can reveal one or more faults in the modified program, in a natural way.

In current practice, the tool set operating on system models, and test models is mostly unconnected, and the various stakeholders, i.e., test engineers, system architects, developers, etc., need to overlook the whole system to ensure that changes are traced to *all* affected parts. With a growing number of stakeholders, this becomes an increasingly complex issue and can result in inconsistent views of the system. Our approach provides assistance for keeping track of evolution steps by using state machines as *executable* artifacts. We assume all stakeholders to operate on the same model but different parts of it. The contribution at hand is focusing on tracking changes on any part of the system model to be able to define optimized test suites. Tool support for consolidating changes by different stakeholders and the actual *interpretation* of state machines is described in [TBL10, BBL10a].

As a model–based approach to regression test selection our approach has several advantages to test selection on the code level [BLH09]. The effort required for regression testing can be estimated earlier which is an important part of impact analysis and tools for regression testing can be largely technology independent. The management of traceability at the model level is more practical because it enables the specification of dependencies at a higher level of abstraction, and no complex static and dynamic code analysis is required. But model–based regression test selection presumes that the system, and test models are complete, internally consistent, and up-to-date which is considered in our methodology.

Our test model evolution approach (TTS with integrated test model evolution) is aligned with the core principles of the *Living Models* paradigm [Bre10] (see Section 3.7 for a detailled listing of all ten principles of Living Models). Living Models focuses on model–based management, design and operation of dynamically evolving systems. The main principles of Living Models are as follows. First, Living Models is based on a tight coupling of models and running systems. This comprises a bi–directional information flow from models to the runtime system and back to the models. Second, Living Models supports the management of quality requirements fostering the cooperation between stakeholders in IT management, software engineering and system operation. Third, Living Models is attached with a novel process model focusing on change and change propagation.

6.2 Metamodel

Our static metamodel for requirements, tests, the system and the infrastructure of service–centric systems is based on the abstract system and test metamodel defined in Section 4.1 and compatible to the general TTS metamodel in Figure 3.15. We recapture the metamodel in the context of test evolution and extend it by a package SUT which models the runtime environment.

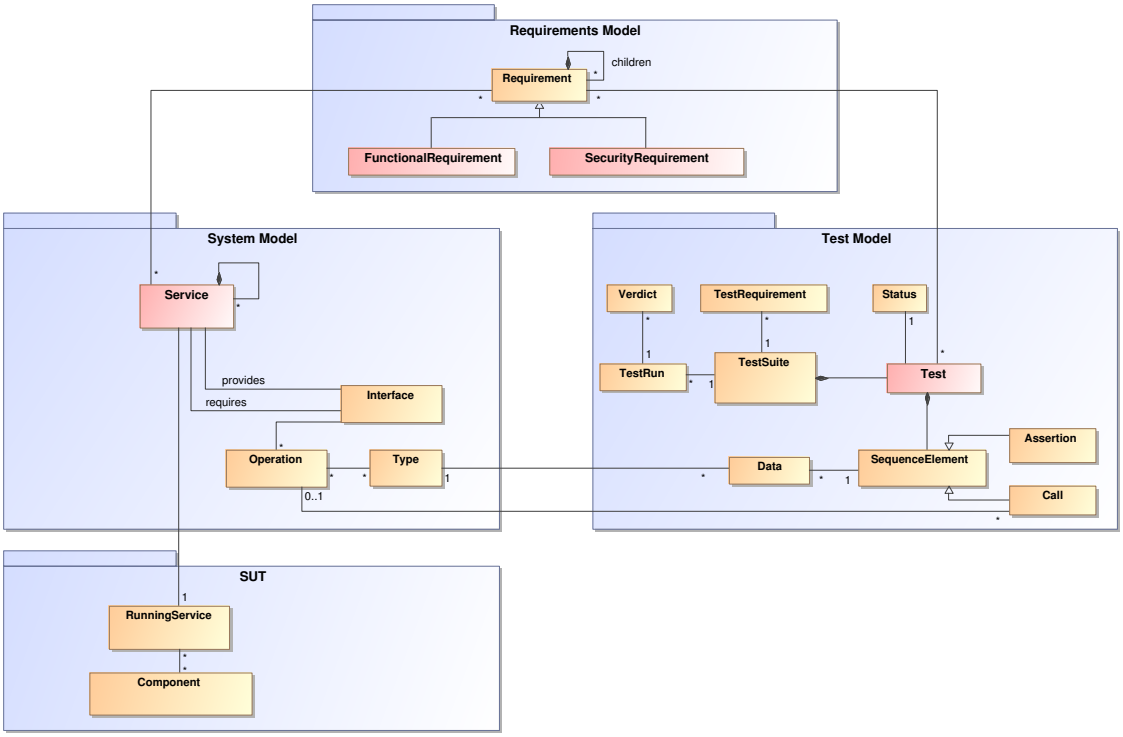


Figure 6.1: Metamodel for Requirements, System, SUT, and Test

The model–based evolution process is based on the metamodel depicted in Figure 3.15. It consists of the packages Requirements Model, System Model, SUT, and Test Model.

The Requirements Model defines a hierarchy of FunctionalRequirement and SecurityRequirement. All requirements are linked to Service elements. A FunctionalRequirement is linked to Test and SecurityRequirement elements, and a SecurityRequirement is linked to Assertion and FunctionalRequirement elements. Therefore a functional requirement has at least one attached test and for all assigned security requirements an assertion in one of these tests exists. This guarantees that all specified security requirements of a functional requirement are also tested. Thus, in our approach *security requirements define security constraints on system functionality.*

The package System Model defines Service elements which contain provided and required Interface elements. Each interface consists of Operation elements which refer to types.

The package SUT defines the deployed services and their deployment environment by the elements RunningService and Component. To keep the description of our approach clear, we do not consider choreographies and orchestrations in the metamodel because they trigger change operations by analogy to plain services. Consequently, our view represents such workflows as Service elements.

The package Test Model defines all elements needed for testing service–centric systems. A TestSuite is a collection of Test elements. A TestSuite is attached to TestRequirement elements, e.g., to select tests or define test exit criteria, and to TestRun elements assigning Verdict values to assertions. A Test has a Status, which can be either `evolution`, `stagnation`, `regression`, or `obsolete`, and consists of SequenceElement artifacts. A SequenceElement is either an Assertion defining how a verdict is computed or a Call element invoking a service operation and has some data assigned to the free variables of its assertions or calls.

The model elements FunctionalRequirement, SecurityRequirement, Service, and Test trigger further changes on other model elements and are highlighted in Figure 6.1. Each of these four types of model elements has a state machine attached. Its current state is stored in an attribute `state` not depicted in Figure 6.1. The mechanism of mutual triggers by using these state machines is discussed in detail in the following section. At this point we only want to mention that this attribute and the status of tests can be used to define test requirements in the declarative language OCL [OMG06b]. Note that the *state* refers to the internal state of an arbitrary model element and that the *status* is an additional attribute for tests.

Services and requirements are organized in hierarchies, so called *service hierarchies* or *requirements hierarchies* which are needed to propagate changes.

The evolution management in Figure 6.1 is compatible to the TTS metamodel in Figure 3.15 but reduced to focus on the proposed evolution process. In the evolution metamodel we do not consider the package Test System because we abstract from the generation and execution of tests. The package SUT are the same in both metamodels. Both packages Requirements Model are identical, but the evolution metamodel only considers SecurityRequirement elements and no other types of non–functional requirements. The System Model of the evolution metamodel does not consider processes and constraints, and considers services as central elements of system changes which is a feasible abstraction because we consider service–centric systems. The Test Model of the evolution metamodel does not consider recursive test suites. The top–level test is called TestSuite to differ it from other tests and to highlight its role as regression test suite. Because we only consider change propagation we abstract from the control flow in tests and only consider Assertion and Call elements. We also abstract from concrete data and only consider abstract data as model element Data. The reduced metamodel allows the definition of a more concise evolution process that can also be applied to the complete TTS metamodel.

6.3 Evolution Process

In this section we show how modifications of different model elements affect tests and, additionally, how tests are managed based on these modifications. The evolution process presented here defines the steps that an arbitrary modification to the model induces to obtain an executable test suite for the updated model that fulfills certain test requirements. Because our metamodel defines connections among the requirements model, the system model and the test model, we are able to propagate modifications to other relevant parts of the model via events and effects of state machines assigned to model elements. For instance, if a service is undeployed this mechanism allows for locating all service calls referring to this service and mark the according tests as "*not executable*". We first present the overall evolution process and then we present every single process step in more detail.

6.3.1 Process Overview

In Figure 6.2 the evolution process is depicted. The *evolution process* is initiated by a change of a model element, i.e., adding, modifying or removing a model element. A *change* is an event which triggers transitions in state machines. The state machines *propagate the change* to all affected model elements. The update of a model element may result in a non–executable system, e.g., a service is not deployed, or the model is inconsistent because operation signatures are not compatible to their calls. Therefore, the model elements affected by an update have to be checked for executability and consistency, and possibly changed until a consistent and executable test model is obtained. These checks, and potential modifications, are made using the modeling environment. Afterwards, based on certain test requirements, *tests are selected* and subsequently they are *executed*. This evolution process is carried out iteratively as long as changes are applied.

According to the partitions of the evolution process in Figure 6.2, its actions are implemented by different tool sets. Because the process is *change-driven* [Bre10], a *model repository* and a *modeling environment* are used for constructing an executable and consistent model. While the model repository manages changes and always provides the most current version of the model, the modeling environment consists of a set of modeling tools to manipulate and validate models. The *test system* is then responsible for selecting and executing tests conforming to a specific test requirement. The different tool sets interact via models and are interchangeable. The process can therefore be implemented by arbitrary model repositories, modeling environments and test systems as long as they operate on the same model representation.

The model element changes are processed by a *metamodel–aware* model repository, which recognizes the different model element status, and is configured by state machines. This way, every change is recorded in the repository. Because of its metamodel–awareness the model repository is able to identify the kind of modifica-

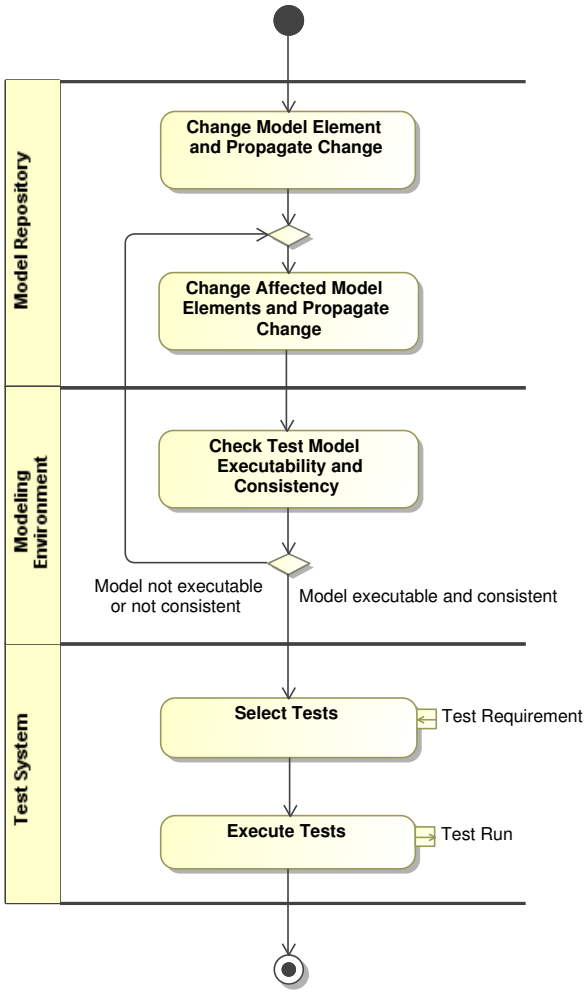


Figure 6.2: Evolution Process for Model Element Changes

tion. Consequently, the model repository is responsible for keeping various parts of the model in–sync by firing the corresponding transitions in state machines on modifications. This part of our approach realizes the integration of different models and allows the various stakeholders to collaborate effectively. Preliminary achievements of such a model repository are discussed in [TBL10] and [BBL10a].

6.3.2 Change of Model Elements and Change Propagation

All model elements can be changed. Changes can be either the creation of a model element (*add–operations*), the deletion of a model element (*remove–operations*), or the modification of a model element (*modify–operations*).

The typical test design process starts with the definition of a new requirement. Afterwards, requirements are assigned to tests. The test definition may also induce changes to the system model because the test model is connected to the system model and uses its services. Hence, an additional requirement may have impact on the rest of the model and raise the need for further changes. This describes

a test–driven development approach where the alignment of the requirement, test and system model is crucial to keep the whole model consistent. Note that any part of the model can be changed and we also cover such cases, e.g., updates of infrastructural components or bug fixes to services.

Although changes may occur on all types of model elements of the metamodel depicted in Figure 6.1, in the following we only consider changes to the elements Requirement, Service, and Test. We do so because changes on arbitrary model elements of the packages System or Infrastructure can be regarded as changes to the assigned services. In the Test package, changes on sequence elements or data elements can be considered as changes on tests. Test suites or test runs are computed in the test selection or the test execution phase but not changed by state transitions considered in this step.

For functional requirements, security requirements, services and tests, we define state machines describing how the states of these elements change. Each state transition defines a *trigger*, an optional *guard* condition and an optional *effect*, i.e., behavior to be performed when the transition fires. This means that the structure of transitions is of the following form: `trigger() [guard] / effect`. The effect part is used here to propagate changes by triggering state transition in other state machines. Simply put, the state machines show under which conditions a state transition occurs and which other elements are potentially affected.

Whenever a model element is removed, the `Status` variable of the assigned test is set to `obsolete`. In order to keep the state machines in this document more readable, we do not consider the transitions to the obsolete state when removing model elements.

Functional Requirements

Figure 6.3 – also printed in the appendix for better readability (see Figure E.3) – depicts the states of FunctionalRequirement elements.

When a new functional requirement r is created, r is in state **defined**. As soon as the requirement is assigned to a test t, its state changes to **assigned**. The requirement can only reach the state **underTest** if all of its associated subrequirements are also in the same state. The requirement is `assigned` to at least one test and all assigned tests are `executable`. This transition can be fired either by assigning a test to r or by modifying assigned tests or their services so that all tests become executable. This condition also takes into account that requirements are organized hierarchically. When r is modified in state **assigned** or **underTest** the compatibility to all assigned tests has to be checked. If the requirement and a test are not compatible anymore, the connection among the two is removed, i.e., the `unassignTest()`-operation is triggered. The trigger `modifyRequirement` also triggers transitions in the state machine of tests as depicted in Figure 6.6. The implications of this trigger are described in the section on tests. Furthermore, a requirement under test is always `assigned` to at least one test. If this is not ful-

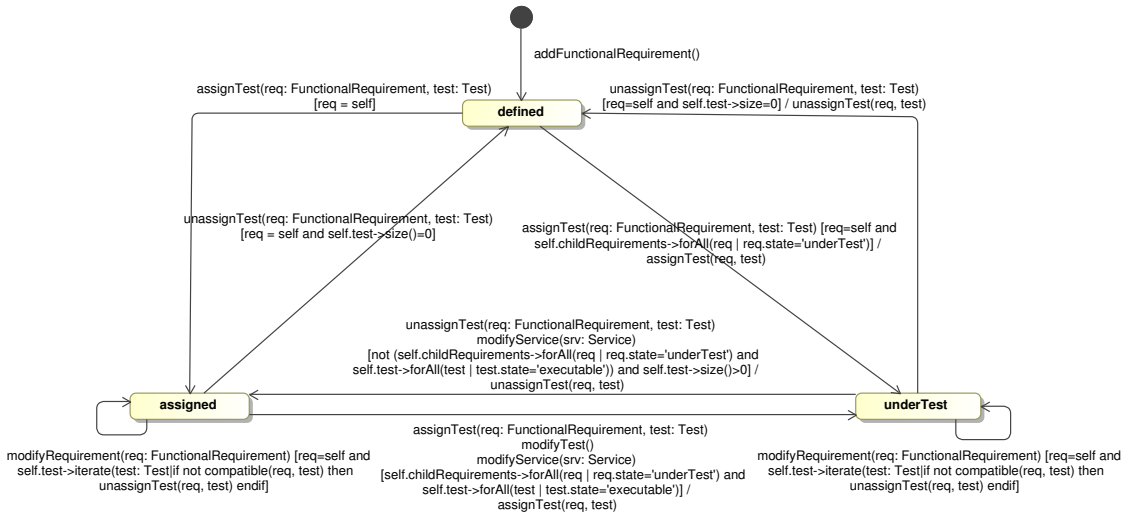


Figure 6.3: State Machine describing the Lifecycle of FunctionalRequirement Elements.

filled anymore, its state changes back to **defined**. In the subsequent semi–formal definitions, we determine the predicates `assigned` and `compatible` used as guards in the state machines.

Definition 9 (Test–Requirement Assignment)**.** The tuple set $\mathcal{TR} \subseteq \mathcal{R} \times \mathcal{T}$ denotes all assignments of tests to requirements. For every tuple $(r, t) \in \mathcal{TR}$ we say that t is `assigned` to r. When a requirement r is assigned to a test t, the tuple (r, t) is added to the set $\mathcal{TR}$. We further define the function $assigned_{TR}$ as follows: $assigned_{TR} : \mathcal{R} \times \mathcal{T} \rightarrow Bool$, where $assigned_{TR}(r, t) :=$ **true** if $(r, t) \in \mathcal{TR}$ and **false** otherwise.

Definition 10 (Test Compatibility)**.** A test $t \in \mathcal{T}$ and a requirement $r \in \mathcal{R}$ are `compatible` if the test t validates the requirement r.

The assessment whether a test validates a requirement is a manual task because the requirements are always specified in an informal way.

Security Requirements

Security requirements are different in the sense that we do not directly assign tests to them. Instead, a security requirement is assigned to a functional requirement which has to comply with it [Haf08]. This implies that the adherence to security requirements needs to be part of the tests assigned to the functional requirement.

Figure 6.4 – also printed in the appendix for better readability (see Figure E.4) – shows the lifecycle of `SecurityRequirement` elements.

When a new security requirement *sr* is created, *sr* is in state **defined**. As soon as *sr* is connected to a functional requirement, *sr* moves into state **assigned**. Upon the assignment of the functional requirement with a test, *sr* changes its state to **underTest** because it is assumed that the test also checks for the adherence to

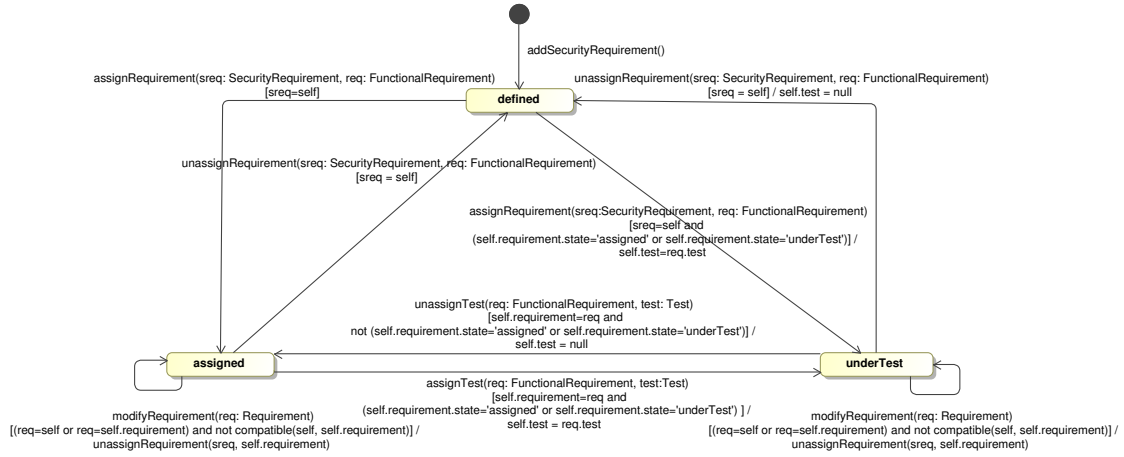


Figure 6.4: State Machine describing the Lifecycle of SecurityRequirement Elements

this security requirement. This assumption is feasible because our security tests are functional. Whenever the connection between *sr* and its functional requirement is removed, *sr* goes back to state **defined**. If a requirement is changed and *sr* is currently assigned to a functional requirement, the compatibility to this requirement is checked. If the compatibility is not given anymore *sr* is disconnected from its functional requirement. Note that this modification takes into account both, the modification of a functional requirement and the modification of a security requirement.

Definition 11 (Requirements Compatibility)**.** A functional requirement r and a security requirement sr are `compatible` if in all cases where r is satisfied, also sr is satisfied.

If a functional requirement r and a security requirement sr are compatible according to Definition 11, then a test for r can be used to additionally test sr given that there is a specific assertion for sr in the test for r.

To clarify the meaning of requirements compatibility consider the following example. A functional requirement states that registered users are allowed to post messages on a bulletin board and their user names shall appear next to their posts. Consequently, one of the assigned security requirements states that users need to be authenticated. So far, these two requirements are compatible with each other. However, if we change the functional requirement so that for guest users the string "Guest" shall appear instead of the username, this requirement is not compatible to its assigned security requirement anymore. The reason for this is that the security requirement requires authenticated users, but the functional requirement also allows unauthenticated guest users. In other words, the functional requirement would be satisfied despite a violated security requirement.

Note that also in this case, the assertion of compatibility has to be checked manually due to the informal specification of requirements.

Services

Similar to requirements, every service $s \in \mathcal{S}$ has a state. Figure 6.5 depicts the lifecycle of services and its state transitions.

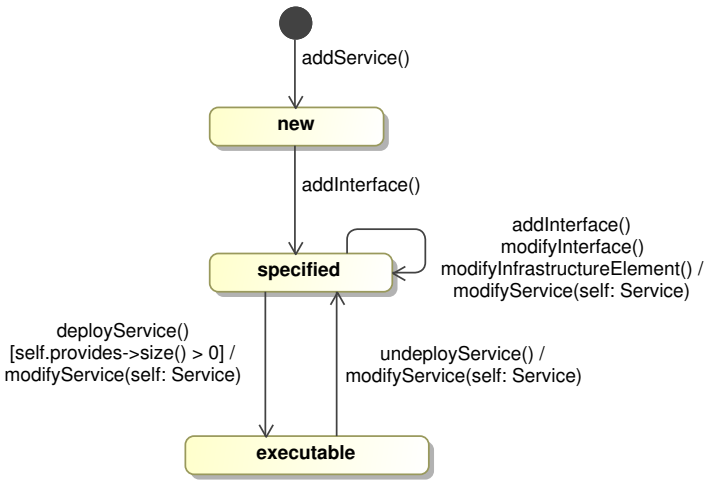


Figure 6.5: State Machine describing the Lifecycle of Service Elements

When a service is newly defined, the service is in state **new**. The state changes to **specified** as soon as an interface is added to the service by `addInterface`, because from that point on it can be referenced by tests. After a service is implemented, it can be deployed by `deployService`, which changes the state of the service to **executable**. However, deployment is only allowed if the service provides at least one interface. The deployment transition further triggers the operation `modifyService` which allows other model elements to react on the deployment of the service. Note that this operation has the current service object (`self`) as parameter. Before the interface or the infrastructural elements of a service can be modified again, the service has to be undeployed by `undeployService`. Executable services always provide at least one interface. Every modification to a service by adding an interface, modifying an interface or modifying the running services is recognized by the corresponding triggers, `addInterface`, `modifyInterface`, or `modifyInfrastructureElement`, and is propagated to `assigned` tests via the trigger `modifyService`. The implications of this trigger are described next.

Tests

Every test $t \in \mathcal{T}$ has a state. As presented above, state transitions of tests are not only caused by direct modifications to elements of type `Test` but can also be triggered by requirements and services. The possible states and transitions of tests are depicted in Figure 6.6.

Definition 12 (Test–Service Assignment)**.** The tuple set $\mathcal{TS} \subseteq \mathcal{S} \times \mathcal{T}$ denotes all assignments of tests to services. When a `Call` element, contained in a test $t \in \mathcal{T}$, is

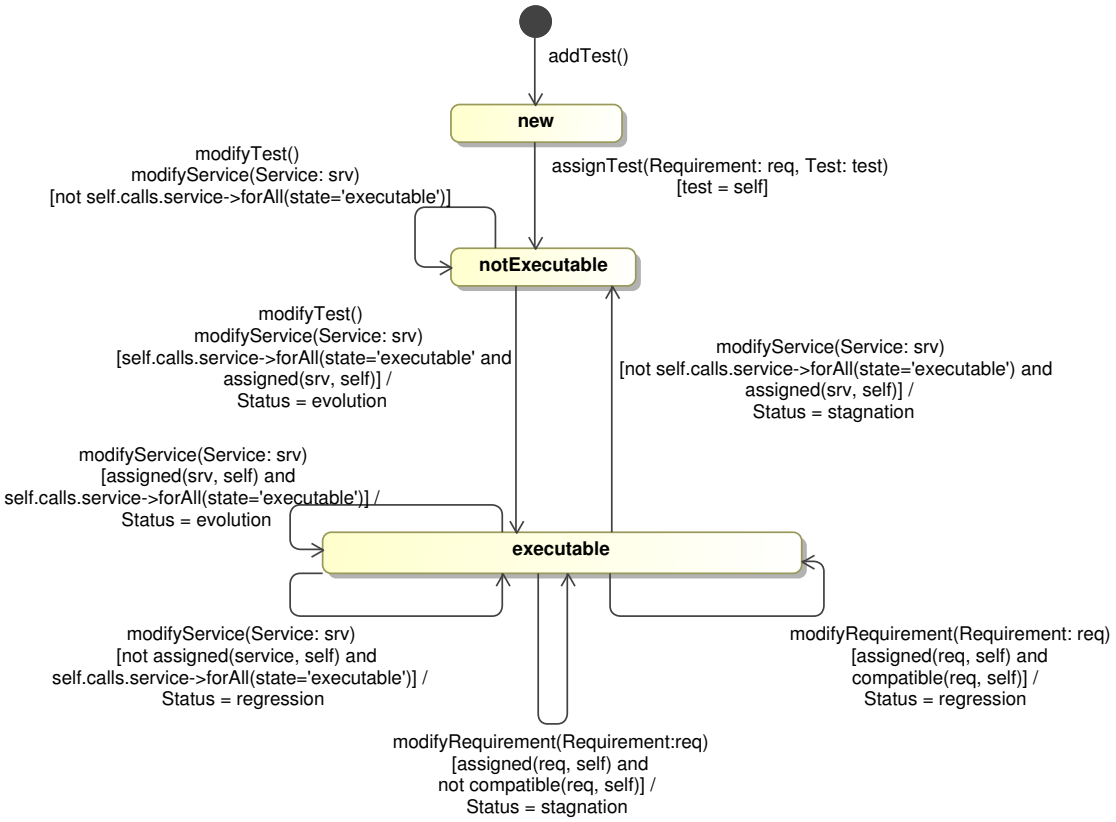


Figure 6.6: State Machine describing the Lifecycle of `Test` Elements

associated to an `Operation` element of a service $s \in \mathcal{S}$, the tuple (s, t) is added to $\mathcal{TS}$. We say that s is `assigned` to t if $(s, t) \in \mathcal{TS}$.
We further define the function $assigned_{ST}$ as follows:
$assigned_{ST} : \mathcal{S} \times \mathcal{T} \rightarrow Bool$, where $assigned_{ST}(s, t) :=$ **true** if $(s, t) \in \mathcal{TS}$ and **false** otherwise.

Note that `Call` elements can only be associated to `Operation` elements if their signature is compatible in terms of parameters and data types.

A test is in state **new** as long as it has no requirements assigned by the operation `assignTest`. Note that the same operation is also a trigger in the `Requirement` state machine, i.e., this assignment causes a state transitions for both, tests and requirements. After the assignment of a requirement, a test is in state **notExecutable** as long as all `assigned Service` elements are in state **executable**. When a test or an `assigned` service is modified such that all `assigned` services of a test are in state **executable**, the test also gets the state **executable**. A guard condition checks for adherence to this rule.

Status of Tests `Test` elements have an attribute `Status`. It describes the kind of a test regarding the last evolution step and is used for test selection. There exist four different status of tests, namely `evolution`, `regression`, `stagnation`,

and `obsolete` as stated in Definition 1. Depending on the modifications to the model, the `Status` of a test is updated. If a test transitions from **notExecutable** to **executable** its `Status` is set to `evolution` because the test is the result of an evolution step. The same is true if a test is currently in state **executable** and one of its assigned services is subject to a modification. On the other hand, if a service is modified but the current test under consideration is not assigned with that service, the `Status` of the test is changed to `regression` because the test should not be affected by this modification. Also the modification of requirements influences the `Status` attribute of executable tests. When a requirement assigned to a test t is modified such that t and the requirement are incompatible, the status is set to `stagnation` because the test *should* fail now. If, on the other hand, the requirement and the test are still compatible, the status is set to `regression`.

6.3.3 Change of Affected Model Elements

As mentioned before, the effect of a state transition propagates changes between model elements by triggering state transition in other state machines. For instance, based on a service modification, a test may change its state from **executable** to **notExecutable**. The induced state changes may be the reason for further changes of model elements. The iterative process of change and change propagation takes place until models are consistent and executable.

6.3.4 Check of Executability and Consistency

The process of changing tests, requirements and services is repeated until the model is *consistent*, i.e., it contains no internal contradictions, and *executable*, i.e., the set or a subset of all tests in a test model can be transformed to executable test code.

The consistency of models can be checked by OCL constraints. The query in Listing 6.1 checks whether the parameters in calls are compatible with the parameters of the referred operations.

```
context Model:
 Call::allInstances.parameters->forAll{ param |
  param.data.class = self.operation.class }
```

Listing 6.1: OCL Query Assuring Parameter Validity

Executability can either be checked operationally by transforming tests to executable test code, or statically, e.g., if all tests in a set defined by the predicate *test_requirement* are in state executable. This is checked by the query shown in Listing 6.2.

```
context Model:
 Tests::allInstances->select{ t |  test_requirement(t) }->
  forAll{ t | t.state='executable'}
```

Listing 6.2: OCL Query for Test Executability

Consistency is the prerequisite for executability. Therefore, executability needs only to be checked if the model is consistent. Checking consistency and executability is very important for the overall evolution process because it determines when the propagated changes have been implemented on the model.

6.3.5 Test Selection

Test selection is the process where an actual test suite is computed from the set of all tests based on test requirements. Test requirements define test selection criteria in OCL and typically consider the status of a test and the state of other model artifacts. A very general regression test could for example be based on the following test selection criteria in Listing 6.3, which selects all tests that are supposed to pass, i.e., tests of status `evolution` or `regression`.

```
context Model:
 Test::allInstances->select{ t | t.type='evolution' or
                                 t.type='regression'}
```

Listing 6.3: OCL Query Selecting Tests of Type Evolution or Regression

A more specific test selection criterion would be the OCL query in Listing 6.4, which selects all tests of type `evolution` that are associated to a specific service, called `HomeGateway` in the following example.

```
context Model:
 Test::allInstances->select{ t | t.status='evolution' and
  t.calls.services->includes{s | s.name='HomeGateway'}
```

Listing 6.4: OCL Query Selecting Tests Invoking a Specific Service

Such OCL criteria can be used to define *safe regression test suites*, selecting all tests in the original test suite that can reveal failures or *minimal regression test suites*, that select a minimal number of tests covering specific criteria.

6.3.6 Test Execution

In the test execution phase a test suite is executed. The result of the test execution is a test run which assigns verdicts to all assertions in the test suite. We abstract from the technical steps to make test models executable which includes the transformation of test to executable test code and its execution against the running services. The transformation of tests as modeled in our approach to executable test code and its execution is presented in [FFZB09]. If the test result is not as expected, e.g., if a test in the stagnation suite passes, the test model changes and the evolution process is executed iteratively.

6.3.7 TTS Tool Integration

Telling TestStories is an integrated framework and its tool implementation supports the model–based test design, the test generation, and the test execution (see Section 3.5). Our tool therefore covers the *modeling environment* and the *test system* partition of Figure 6.2. The check of the test model consistency and executability is handled by the *Model Validator*. The *Test Generator* covers the selection of tests, and the *Test Controller* executes tests. Our modeling tool operates on a standard XMI2 representation of the underlying model and a XML representation of the data. These artifacts can be stored and managed by a model repository as proposed in [TBL10, BBL10b]. Therefore TTS combined with such a model repository implements the overall evolution process.

6.4 Case Study

In this section we demonstrate our test evolution methodology by the home gateway case study presented in Section 5.2.

The home gateway is an appliance connected to an operator's network and providing certain services to customers, e.g., internet connection or video surveillance of rooms. As a service–centric system, it consists of the peers *AccessRequestor* (AR), *Home Gateway* (HG), *Policy Enforcement Point* (PEP) and the *Policy Decision Point* (PDP). The AR is the client application to establish the connection to a HG and to use its functionalities. The HG is a device installed at the home of a customer controlling access to different resources. The enforcement of who is allowed to access which resources on the network is made by an internal component of the HG called PEP. The PEP gets the policy it has to enforce for a specific AR by the PDP.

The home gateway may evolve in various directions, e.g., a new service is added on the home gateway, a new security requirement has to be applied, the underlying infrastructure is changed or a service implementation is adapted.

In this section, we discuss representative examples for the following types of evolution:

- creation of a functional requirement
- modification of a functional requirement
- addition of a security requirement
- modification of a security requirement
- modification of a component
- modification of a service

Based on the requirements for the home networking case study depicted in Figure 5.2, we consider various change scenarios. In Figure 6.7 the requirements affected by the subsequent scenarios, namely `Req_1` ("The agent–based network access has to be controlled"), `Req_1.4` ("The assigned policy has to be applied to the agent–based network connection"), `SReq_1.4.1` ("Under aged users are not allowed to access the home network but only the internet"), and `Req_3` ("The air conditioning can be controlled on the home network") are highlighted.

Figure 6.7: Evolved Requirements of the Home Networking Application

For all six evolution scenarios mentioned above, we give a concrete example and sketch the evolution process with a focus on the change and propagation events. For each scenario all relevant change events and affected model elements are shown in a sequence diagram.

6.4.1 Creation of a Functional Requirement

In the first scenario a new functional requirement with the identification `Req_3` and the text "The air conditioning can be controlled on the home network" is added. This action triggers the event `addRequirement` for this requirement, which is then in state **defined**. Then a test `Test_3` similar to the test `TestAirConditionControl` (see Figure 5.9) for testing the air condition service is added (event `addTest`) and assigned to `Req_3` (event `assignTest`). Afterwards, the requirement transitions to state **assigned** or, if the test is already executable, to state **underTest**. This scenario reflects the test–driven development process which our testing methodology is based on, because assigning a test to a requirement triggers modifications to the system model. In this case, a service `AirConditioning` for setting the temperature has to be added to make the test fully executable.

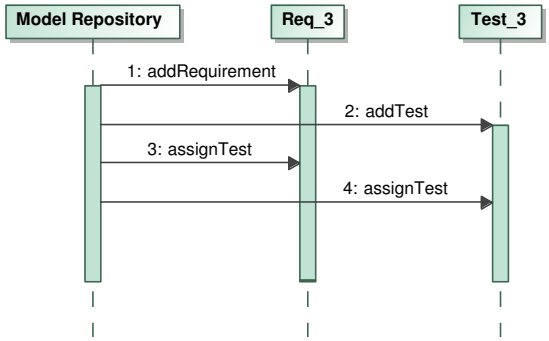


Figure 6.8: Events of the Functional Requirement Creation Scenario

The sequence diagram in Figure 6.8 gives an overview of all relevant events and affected model elements of the functional requirement creation scenario.

6.4.2 Modification of a Functional Requirement

In this scenario we assume that the functional requirement with the identification `Req_1` and the text "The agent–based network access has to be controlled" is modified. This requirement has already assigned an executable test and is modified by changing the network control from agent–based to agent–less.

The modification of the requirement triggers the `modifyRequirement` event but the requirement is then still in state **underTest**. The event `modifyRequirement` triggers a transition in the state machine of the assigned test. The modified requirement and the test for the agent–based scenario are not compatible because the agent–less scenario needs an additional network filter service. Therefore, the test has the status `stagnation` because the test is expected to fail. The assigned test is still in state **executable** because no service has been modified.

If a new test in state evolution or regression for the adapted requirement as depicted in Figure 5.8 should be defined, then a new evolution process is started. The test depicted in Figure 5.8 checks for agent–less network control, integrating

the new service `NetworkFilter`. A test `Text_1` similar to the test `TestAgentless` of Figure 5.8 is assigned to the requirement `Req_1` (event `assignTest`) in a new evolution process after the test has been added (event `addTest`). Then the test is in state **notExecutable** as long as all its service calls are executable. As soon as the new service *NetworkFilter* has been specified and is executable the event `modifyService` triggers the transition of the test to the state **executable** and its status is `evolution`. The test can then be selected by specific test requirements and executed afterwards.

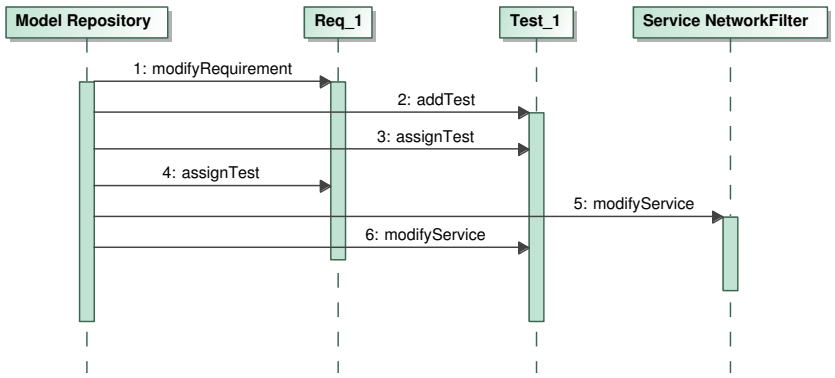


Figure 6.9: Events of the Functional Requirement Modification Scenario

The sequence diagram in Figure 6.9 gives an overview of all relevant events and affected model elements of the functional requirement modification scenario.

6.4.3 Addition of a Security Requirement

We assume that the agent–based network access is controlled (`Req_1`) and that the functional requirement `Req_1.4` "The assigned policy has to be applied to the agent–based network connection" is defined and that all requirements are in state **underTest**. Each requirement has at least one executable test assigned and all services are in state **executable**. The assigned test for `Req_1.4` is `Test_1.4` testing the policy application in an agent–based scenario similar to the test depicted in Figure 5.7. Details of this test are not relevant for this case study.

In the first step, we add the security requirement `SReq_1.4.1` "Under aged users are not allowed to access the home network but only the internet" by `addSecurityRequirement` which is then in state **defined**. Afterwards, the security requirement `SReq_1.4.1` is assigned to the requirement `Req_1.4` by the operation `assignRequirement(SReq_1.4.1,Req_1.4)` and moves into state **underTest** because the assigned requirement, `Req_1.4`, is in state **underTest**. The test `Test_1.4` already considers the restriction for under aged users and therefore no modifications are needed to even capture the meaning of `SReq_1.4.1` in the test.

In the second step, `Req_1.4` is modified by changing it from an agent–based to an agent–less network connection. This is caused by the event `modifyRequirement(Req_1)`. Assuming that the assigned test and the require-

ment are not compatible any more, the `unassignTest(Req_1.4,Test_1.4)` event is triggered and `Req_1.4` goes to state **defined** and the assigned security requirement `SReq_1.4` goes to state **assigned**. The unassigned test `Test_1.4` resumes in state `executable` and is assigned the status *stagnation* because `Test_1.4` is not compatible with the modified requirement `Req_1.4`.

In the third step, we want to define a test which is compatible with `Req_1.4` again. A test `Test_1.4(2)` similar to the one depicted in Figure 5.8 is defined via `addTest`. `Test_1.4(2)` checks for agent–less network control, integrating the new service `NetworkFilter`. Then, `Req_1.4` and `Test_1.4(2)` are assigned by `assignTest(Req_1.4, Test_1.4(2))`. Afterwards, `Req_1.4` is in state **assigned**, the security requirement `SReq_1.4.1` in state **underTest** and `Test_1.4(2)` in state **notExecutable**. To make the test executable, the test is modified by adding service calls and assertions. Also the new service *NetworkFilter* is added to the system model. After all modifications, the test is defined as in Figure 5.8 (details of the test sequence itself are not relevant). After a `modifyTest` triggered by the modifications on the test and as soon as the new service *NetworkFilter* has been specified and is executable, the event `modifyService(NetworkFilter)` triggers the transition of `Test_1.4(2)` to the state **executable** and its status becomes `evolution`. This also causes the state of `Req_1.4` to transition from state **assigned** to state **underTest**. Finally, the test model is consistent and tests can be selected and then executed.

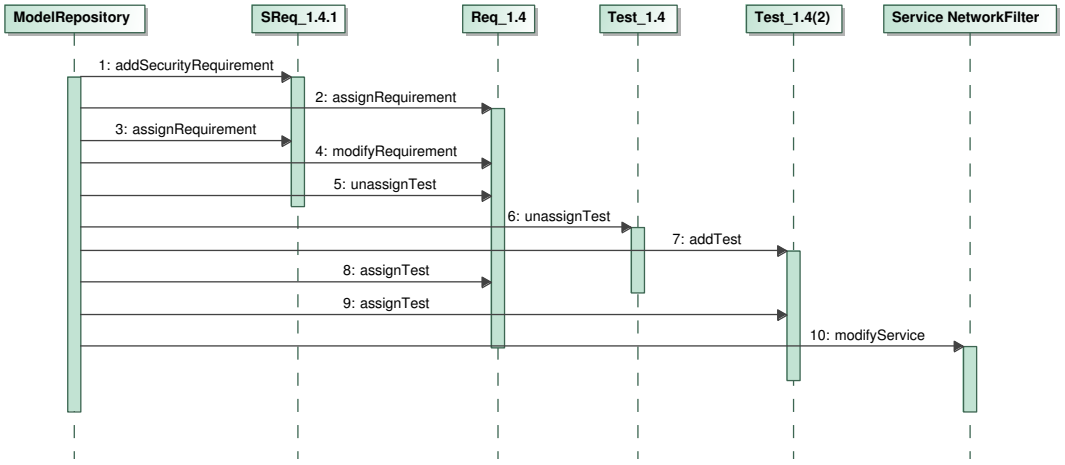


Figure 6.10: Events of the Security Requirement Addition Scenario

The sequence diagram in Figure 6.10 gives an overview of all relevant events and affected model elements of the security requirement addition scenario.

6.4.4 Modification of a Security Requirement

In this scenario we assume that the security requirement `SReq_1.4.1` is modified such that under aged users are additionally only allowed to use the internet in a restricted way, to only view web pages provided on specific web servers. Initially, `SReq_1.4.1` is in state **underTest** and assigned to the original version of requirement `Req_1.4`

which is also in state **underTest**. The modification triggers a `modifyRequirement` event. Because `compatible(SReq_1.4.1,Req_1.4)` still holds, there is no transition in the state machine of security requirements and `SReq_1.4.1` stays in state `underTest`. The `modifyRequirement` event also triggers the state machine of the assigned test `Test_1.4` which is in state **executable**. The requirement `SReq_1.4.1` and the test `Test_1.4` are not compatible and the test `Test_1.4` gets the status `stagnation`. A new test based on the stagnation test `Test_1.4` is added which is assigned to `Req_1.4` and also validates `SReq_1.4.1`. We then have a evolution to pass and a stagnation test expected to fail for the modified security requirement `SReq_1.4.1`.

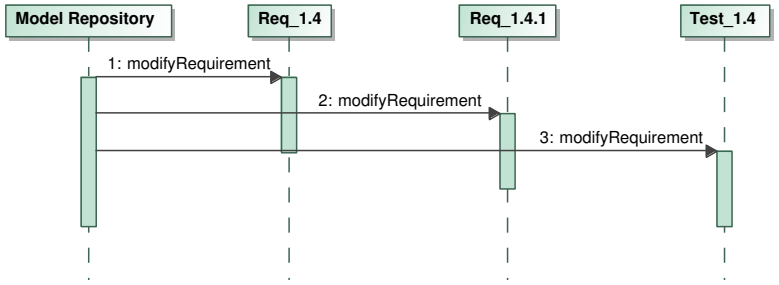


Figure 6.11: Events of the Security Requirement Modification Scenario

The sequence diagram in Figure 6.11 gives an overview of all relevant events and affected model elements of the security requirement modification scenario.

6.4.5 Modification of a Service

A typical modification of a service is the adaptation of a function signature. For instance, the parameter `temperature` of the operation `setTemperature` of the interface *AirConditioning* is changed from `integer` to `float`. The service interface has already been defined, and we assume that the corresponding service *AirConditioning* is in state **specified** and the assigned test `Test_3` of the requirement with identification `Req_3` in state **notExecutable**. The event `modifyInterface` triggers the action `modifyService` and leaves the service in state **specified**. `Test_3` is not executable and therefore a second change iteration has to be applied to make the model executable.

In the second iteration the service *AirConditioning* is deployed with the event `deployService` and the effect `modifyService` which then triggers the transition from the state **notExecutable** to the state **executable** in the test `Test_3`. In this step the status of `Test_3` is set to `evolution`.

The sequence diagram in Figure 6.12 gives an overview of all relevant events and affected model elements of the service modification scenario.

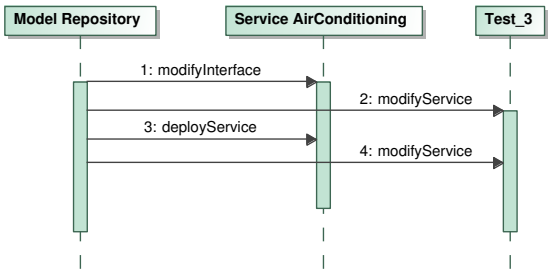


Figure 6.12: Events of the Service Modification Scenario

6.4.6 Modification of a Component

Modifications of services can also be triggered by changes on the infrastructure of the SUT as in the following scenario.

The HG service is deployed on a component implemented on an OSGi infrastructure[1], a dynamic module system and service platform for Java. Before the OSGi infrastructure is changed, all services running on it are undeployed and then in state **specified**. The change of the infrastructure is reflected by a `modifyInfrastructureElement` event firing a transition. The affected services are then deployed (`deployService`) and move to state **executable** triggering `modifyService` events with the effect that the status of all assigned tests is set to `regression`.

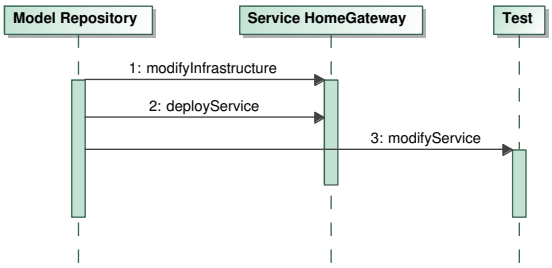


Figure 6.13: Events of the Component Modification Scenario

The sequence diagram in Figure 6.13 gives an overview of all relevant events and affected model elements of the component modification scenario.

6.5 Related Work

Evolution of software has attained much attention in the last years. Many approaches consider the evolution of the system either focusing on the source code or on the system model [MD08]. But only a few approaches highlight the role of tests in the evolution process.

[1]More information on OSGi is available at `http://www.osgi.org` [accessed: November 25, 2010].

In [MvDZB08] the interplay between software testing and evolution is investigated considering the influence of evolving tests on the system. The paper introduces *test–driven refactoring*, i.e., refactoring of production code induced by the restructuring of the tests. As in [MvDZB08], we consider the relationship of tests and system evolution, but on the model–level and not on the code–level. To the best of our knowledge, there is actually no approach that considers the evolution of security requirements tests.

Change management ensures that standardized methods and procedures are used for efficient and prompt handling of all changes to control services and their infrastructure [OGC07]. Change management on the level of UML models has been investigated in [BLO03]. Similar to our approach, [LCP08] utilizes state machines to model and manage requirements changes. In distinction from our approach, the change management approaches in [BLO03] and [LCP08] do not consider the co–evolution of tests, requirements and services on the model level.

Regression testing is one major activity to maintain evolving systems [GPCL08]. Most regression testing approaches are code–based [vMZ99]. Only a few model–based regression testing approaches have been considered. Model–based regression testing approaches determine impacts of system model changes on the baseline test suite. UML–based system models typically consider class models and a specific type of behavior models, such as state machines in [FIMN07], sequence diagrams in [BLH09] or activity diagrams in [CPS02]. Our approach abstracts from a specific type of behavior and attaches state machines to all artifacts, also tests, to compute a consistent regression test suite based on specific test selection criteria. We integrate tests of security requirements into this process which has not been considered in any other model–based regression testing approach yet.

There are only a few classical regression testing approaches focusing on service–centric systems such as [DBE+07] describing a regression testing strategy for web services considering service changes. The approach focuses on the minimization of service invocations by reusing monitoring data. In our approach not only service changes but also changes to the requirements and tests are considered.

Several approaches to model–based testing of security requirements have been developed but none of them considers evolution aspects. In [JMT08, MJP+07], the authors describe a model–based testing approach for checking whether access control policies are properly enforced by the SUT. [WJ02] uses the system specification for generating security tests. In [SKR07] a model–based framework for security vulnerabilities testing is described. In [Jür08] model–based testing with the UMLsec profile [Jür02] is presented.

For service–centric systems there are only a few approaches considering security testing [BHB+07, CFV07] and regression testing [DBE+07]. None of these approaches combines them to regression testing of security requirements for service–centric systems. Our methodology handles security requirements testing for evolving requirements by regression testing.

6.6 Summary

In this chapter we have considered the evolution, i.e., addition, modification or deletion, of arbitrary artifacts in the TTS framework and their influence on the test model and regression testing. We have especially focused on a functional and security requirements–driven evolution process.

In Section 6.1 we have motivated the evolution of systems and their requirements and the need for a co–evolution of tests. We have especially motivated the need for managing the evolution of tests for security requirements.

In Section 6.2 we have defined the underlying metamodel of the evolution process. The metamodel contains the packages requirements, system, test, and SUT. Changes are triggered by the model elements of type FunctionalRequirement, SecurityRequirement, System, and Test.

In Section 6.3 we have defined the evolution process. The process consists of the steps change of model elements and change propagation, change of affected model elements, check of executability and consistency, test selection and test execution which are executed in an iterative way. The process steps are executed by various tool sets, i.e., modeling tools, a model repository and the test system.

In Section 6.4 we have applied the evolution process on the home networking case study presented in Section 5.2. We have considered the evolution scenarios creation of a functional requirement, modification of a functional requirement, addition of a security requirement, modification of a security requirement, modification of a component, and modification of a service.

In Section 6.5 we have presented related work to system and test evolution, to model–based regression testing and regression testing of service–centric systems, and to security related testing.

Chapter 7

Conclusion and Future Work

The whole is more than the sum of the parts.

Aristotle

This chapter concludes this thesis by summing up (Section 7.1), evaluating the results (Section 7.2), and sketching future work in the categories research, implementation, and application (Section 7.3).

7.1 Summary

The goal of this thesis was to develop a novel system testing approach for service–centric systems that combines model–driven testing and tabular test definition. In Section 1.2 we have identified four research challenges to which this thesis contributes, i.e., the development and application of the methodology itself, the validation of test models, security testing for service–centric systems, and the evolution of service–centric systems plus its propagation to tests.

Based on concepts of model–driven testing and service–centric systems defined in Chapter 2, we have developed the Telling TestStories methodology in Chapter 3.

The TTS methodology defines a testing process plus its model and implementation artifacts. On the model level it contains the artifacts of a requirements model, a system model defining services, and a test model containing test stories. For each model a domain–specific language has been defined by its abstract syntax, its concrete syntax in UML, and its semantics. The models are validated by consistency, completeness and coverage checks which are defined in Section 4.3.

Test stories are controlled sequences of service calls, triggers and assertions. Test stories can be abstract and transformed by a model–to–model transformation to executable test stories (see Section 4.4). Executable test stories are the source for generating executable test code in Java. The test controller executes the generated

test code and invokes the system services via adapters. The TTS tool implementation contains the components Modeling Environment, Model Validator, Test Code Generator, Service Adapters, Test Controller, and Test Evaluator (see Section 3.5). All model and implementation artifacts are traceable supporting the visualization of the test results in the test behavior diagrams and in test tables.

The testing process consists of a design, validation, execution, and evaluation phase and is processed in an iterative way. Initially, the process is triggered by requirements for which services and tests have to be defined. Then the models are validated and test code is generated directly from test models and executed via adapters. Test results are annotated directly in the test model and summarized in reports. In Section 3.4 we employ the basic TTS testing methodology on a service–centric callmanager application from the telecommunication domain.

TTS is a model–driven and tabular testing approach. In Section 3.6 we have related TTS to approaches which differ either in the formality or tabularity from TTS, i.e., FIT, U2TP, and test sheets. Additionally, in Section 3.7 we have considered TTS as dynamic, systematic, active, and model–based testing process, have classified it according to the standard taxonomy of model–based testing, and have investigated the relationship of TTS to model–driven testing and the Living Models paradigm.

Based on a formalization of the model and implementation artifacts as sets in Section 4.1, we have introduced arbitrations which provide a mechanism for higher–order assertions defining restrictions on the verdicts of a test run in Section 4.2.

In Chapter 5 TTS has been applied for testing security requirements. Security requirements are attached to functional requirements and tested in an integrated way. Our approach guarantees traceability between security requirements, the system and test model and the executable service–centric system under test. Security requirements testing has been demonstrated by a case study from the home networking domain for which security requirements are defined and tested.

To each model element in the requirements model, system model and test model a state machine can be attached to manage the evolution of tests. Our evolution management process defined in Chapter 6 guarantees consistent evolution of a service–centric system and its tests. The additional state information is applied for the generation of regression test suites according to specific test requirements in OCL.

Finally, in this chapter the results of this thesis are evaluated, and future work in the categories research, implementation, and application is identified.

7.2 Evaluation

The research method applied in this thesis is based on the design science paradigm [HMPR04, PTRC08]. This paradigm essentially advocates the construction of a demonstration implementation of a proposed new approach followed by the evaluation of its usability, effectiveness, and efficiency. We have implemented a tool and used two industrial case studies to evaluate the usability, effectiveness and efficiency of the TTS methodology. A callmanager application from the telecommunication domain has been used to evaluate the basic TTS methodology itself, and a home networking application has been used to evaluate security requirements testing and test evolution management based on the TTS approach.

Design science is technology–oriented, it attempts to serve specific purposes, and it creates effective artifacts [MS95]. We have developed a novel system testing methodology with the purpose to improve the usability, effectiveness, and efficiency of existing approaches by combining tabular and model–driven testing. We have even implemented a practically usable tool for our methodology that is available online at `http://teststories.info` [accessed: February 28, 2011] and has been presented on an international testing conference [FZFB10].

According to the paradigm of explanatory design theory [BPH10] in design science, only two elements are essentially necessary for a complete design theory: requirements, i.e., the research challenges in our respect, and their solution, i.e., the contributing results evaluated by case studies in our respect. Based on the results presented in this thesis, we evaluate our contributions **C1** to **C7** defined in Section 1.3 according to the research challenges **RC1** to **RC4** stated in Section 1.2. We highlight to which extent the research challenges have been solved by our contributions and what their limitations are. As a consequence to these limitations we derive future work which is then sketched in the next section in more detail.

The research challenge **RC1** addresses the development and application of a tool–based testing methodology for service–centric systems based on model–driven testing and tabular test design.

We have designed and implemented the TTS framework (contribution **C1**), as tabular and model–driven system testing methodology for service–centric systems. We have implemented a tool for TTS and evaluated it by an industrial callmanager application. Due to the feedback of the TTS users, the tool has been assessed as usable in its practical application. Therefore the TTS methodology itself can be characterized as usable. Prior to the use of TTS the system test design has been done ad–hoc in an unsystematic way mainly by testers themselves. TTS allows for designing tests in an effective way because its intuitive graphical and tabular notation supports the design of tests that can be validated by consistency, completeness and coverage checks (contribution **C4**). The TTS methodology naturally integrates domain experts and customers into the process of formal and executable test design.

The integration of domain experts and customers may be helpful to reveal specific test scenarios that would not have been detected otherwise.

TTS is also efficient because the tests can be defined on an abstract visual level with tool support. After the initial effort of system model design the advantages of TTS can be applied. As future work, we consider controlled experiments to measure the effectiveness by the number of additionally observed failures, and the efficiency by the ratio of the observed failures to the effort of model design measured in person days. We can even more improve the efficiency of TTS by a textual representation of test models because test models can then be modified in a text editor which is much faster in many situations.

In contribution **C2** we have classified TTS as model–driven and tabular system testing methodology, and we have shown that there is actually no approach with the same properties due to the criteria formality and tabularity as TTS. As a tabular and model–driven system testing approach, TTS can be integrated with test sheets which can be employed as internal, compact representation of tests or as a substitute for data tables in TTS. We consider the implementation of the TTS and test sheets integration as future work.

The direct and automatic generation of test code addressed in contribution **C3** is important for the usability and efficiency of the TTS framework. The direct generation of test code by adapters guarantees traceability between the model and the implementation level. We have extensively applied the test code generator in the callmanager case study and steadily improved it.

The feedback of the test results in test reports, in test data tables and in behavior diagrams of tests by the full traceability between all artifacts has been rated as very useful for decision support and debugging. The a posteriori validation of test runs by arbitrations in contribution **C5** even improves the expressiveness of the feedback because it supports the concise representation of the verdicts. Arbitrations as defined in this thesis have explicitly been integrated based on practical requirements in the telecommunication domain and are a useful extension of the TTS methodology.

The research challenge **RC2** addresses specific validation rules for test models.

In contribution **C4** we have defined a framework for checking consistency criteria, completeness criteria/metrics and coverage criteria/metrics. Our implementation of the framework has shown that the checks provide additional support for the validation of the requirements, system, and test model, but also raises the failure detection rate due to the higher test quality. In the models of both case studies (callmanager and home networking application) we found inconsistent and incomplete model elements that have been detected with our criteria and removed afterwards. With our validation checks the effectiveness and efficiency of the approach has been improved because the quality of test models is higher and failures are detected earlier. The coverage criteria and metrics provide useful information to all stakeholders whether additional tests have to be defined manually or not. As future work, the

coverage criteria and metrics can also be applied for the automatic generation of test models. Our validation checks are defined statically on the metamodel and do not support the dynamic simulation of a model. But due to our experience this is replaced by early test execution in iterative software testing and therefore not a severe restriction compared to dynamic approaches like model–checking which need additional modeling effort and more knowledge in formal modeling. Additionally, the information provided in this process supports the system engineers who are not experts in test design when defining tests.

Test models are also validated by the direct test generation (contribution **C3**) because the generated tests are only executable if the test model is valid. We have observed in our test model that tests generated from invalid test models are practically not executable in most cases.

The research challenge **RC3** addresses security requirements testing for service–centric systems.

In contribution **C6** we have applied TTS to security requirements testing. As evaluated in a case study from the home networking domain, our security tests are functional and embed policy testing into the overall system testing process of TTS. We demonstrated that TTS provides an intuitive security testing approach which allows also non–experts to define security tests. In many other approaches this is only possible for experts who are able to formulate an attack model. The presented security testing approach only considers positive security requirements but does not address negative security requirements. As future work the security testing approach can be extended by risk–based security testing to also consider negative security requirements.

The research challenge **RC4** addresses the evolution management for service–centric systems and tests.

In contribution **C7** we define a process for evolution management of service–centric systems and their tests based on the test generation of contribution **C3** and the validation and coverage checks from contribution **C4**. The process is based on state machines attached to all artifacts of the requirements, system, and test model which trigger changes to all affected model elements in an iterative way. As soon as the test model is consistent, tests can be selected based on selection criteria, transformed to test code which is then executed. We have applied the evolution process on various changes in the home networking case study and demonstrated how system evolution can effectively be managed on the model level under the consideration of tests and their models. The effects of changes on the model level have so far been traced manually and can be automatized based on a model repository [TBL10, BBL10b] as future work.

7.3 Future Work

There are many interesting issues for future work arising from this thesis which are outlined in this section. The main contribution of this thesis is the design of a formally well–founded model–driven and tabular system testing approach for service–centric systems suitable for practical application. Besides research tasks, design–oriented research in computer science as performed in this thesis additionally takes technologic and application aspects into consideration. Therefore future work comprises research, implementation and application issues which are summarized in the following paragraphs.

Research. The Telling TestStories approach has so far been applied for model–driven system testing of functional requirements and security requirements. TTS is also capable to test other non–functional properties such as reliability, stress or performance. We consider testing such properties in future case studies. The arbitration mechanism already considers performance because it supports temporal constraints but performance testing has not been conducted systematically in a case study so far.

In Chapter 5 we have only considered tests of positive security requirements. The TTS methodology can be extended to also test negative security requirements by a risk–based testing. Risk–based testing considers risk values assigned to model elements for the design, the execution and the evaluation of tests. In the context of security testing, risk–based testing allocates testing resources to those threats that have a high risk. The integration of risk–based testing in TTS implies adaptations of the mechanisms for test execution, test requirements, test reporting, and the model feedback. Risk–based testing can also be applied to functional testing and for the definition of more selective regression test criteria.

The effectiveness and efficiency of TTS has so far only been evaluated qualitatively. We employ controlled experiments to also quantitatively evaluate TTS. We measure the effectiveness of TTS by the number of additionally observed failures, and the efficiency by the ratio of the observed failures to the effort of model design measured in person days. TTS provides a framework to conduct further empirical research on the effectiveness and efficiency of model–driven testing processes and to compare model–driven testing with code–based testing. Furthermore TTS supports test–driven development on the model–level and TTS is therefore a promising approach to integrate model–driven development and test–driven development. According to that, the relationship of TTS to various system development models is also a relevant research task.

The system model of TTS is very generic and can be extended to consider more specific behavioral elements, e.g., XOR connectors as in the Business Process Modeling Notation [OMG09a] to apply special test generation techniques or more fine–grained validation checks. From a representational and technical point of view this is not a problem because we use the extendable, general purpose language

UML for system modeling. There are also several test generation techniques based on UML and OCL, e.g., [BBH02] for test data generation or [ASA05] for test generation based on constraint solving that are relevant in the model–driven and tabular context of TTS.

So far coverage criteria in TTS only check the quality of the test model as adequacy criteria but are not applied for the generation of test cases as selection criteria. Our OCL–based coverage criteria can be applied to integrate selective test generation into the TTS framework.

Implementation. In the course of this thesis the TTS tool has been implemented to conduct case studies which show the usability, efficiency and effectiveness of the underlying methodology. The TTS tool is already quite mature and on its way to a practically usable open–source tool for model–driven system testing of service–centric systems [FZ10]. For the practical application the usability of the tool and the interoperability to integrate TTS with other testing tools has to be improved. For instance, the usability of modeling tests can be improved by an additional textual representation of test models which is synchronized with the graphical representation, to support fast text–based editing of test models.

Some ideas discussed in this thesis have not been implemented so far. TTS can be enhanced by test sheets as internal test representation or as substitute for TTS data tables. A mechanism for monitoring the communication between services allows for checking more specific assertions and especially improves security testing with TTS. The evolution management for test models could be automatized by integrating TTS with a model repository implementation.

Additionally, also most of the research tasks raised above imply implementation work. For instance, some variants for the automatic generation of test models require the integration of TTS with reasoning tools such as model checkers or constraint solvers.

Application. The more case studies are conducted for a tool–based methodology, the more mature the methodology becomes. Telling TestStories is suitable for arbitrary service–centric systems and therefore many application domains are arising. While finishing this thesis we have already started to test an industrial service–centric system from the health care domain with TTS. An application to the upcoming paradigm of cloud computing [VRMCL09] is planned. Cloud applications can be considered as a specific type of service–centric systems for which sufficient system testing is still an open research issue. Therefore the application of TTS for cloud testing seems to be promising.

Bibliography

[Abb09] J. Abbors. Increasing the Quality of UML Models Used for Automatic Test Generation. Master's thesis, Faculty of Technology, Abo Akademi University, 2009.

[ABB10] C. Atkinson, F. Barth, and D. Brenner. Software Testing Using Test Sheets. *Software Testing Verification and Validation Workshop, IEEE International Conference on*, 0:454–459, 2010.

[ABFJ08] C. Atkinson, D. Brenner, G. Falcone, and M. Juhasz. Specifying High-Assurance Services. *Computer*, 41:64–71, 2008.

[Aic10] L. Aichbauer. Test Evaluation with Telling TestStories, 2010.

[AK07] C. Atkinson and T. Kühne. A Tour of Language Customization Concepts. *Advances in Computers*, 70:105–161, 2007.

[ant] ANTLR Parser Generator. available at `http://www.antlr.org/` [accessed: November 25, 2010].

[AO08] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, Cambridge, UK, 2008.

[APT+09] F. Abbors, T. Pääjärvi, R. Teittinen, D. Truscan, and J. Lilius. Transformational Support for Model-Based Testing – from UML to QML. In *2nd Workshop on Model-based Testing in Practice*, 2009.

[ASA05] B. K. Aichernig, P. Salas, and P. Antonio. Test Case Generation by OCL Mutation and Constraint Solving. In *QSIC '05: Proceedings of the Fifth International Conference on Quality Software*, pages 64–71, Washington, DC, USA, 2005. IEEE Computer Society.

[asp] AspectJ. available at `http://www.eclipse.org/aspectj/` [accessed: October 22, 2010].

[BA04] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.

[BBH02] M. Benattou, J. Bruel, and N. Hameurlain. Generating Test Data from OCL Specification. In *ECCOP'2002 Workshop on Integration and Transformation of UML Models (WITUML02)*, 2002.

[BBL10a] M. Breu, R. Breu, and S. Löw. Living on the MoVE: Towards an Architecture for a Living Models Infrastructure. *International Journal On Advances in Software*, 3(1&2), 2010.

[BBL10b] M. Breu, R. Breu, and S. Löw. Living on the MoVE: Towards an Architecture for a Living Models Infrastructure. *International Journal On Advances in Software*, 3(1&2), 2010.

[BCD+06] M. Busch, R. Chaparadza, Z.R. Dai, A. Hoffmann, L. Lacmene, T. Ngwangwen, G. C. Ndem, H. Ogawa, D. Serbanescu, I. Schieferdecker, and J. Zander-Nowicka. Model Transformers for Test Generation from System Models. In *Software Quality Engineering – Proceedings of the CONQUEST 2006*, 2006.

[BDAFP09] A. Bertolino, G. De Angelis, L. Frantzen, and A. Polini. The PLASTIC Framework and Tools for Testing Service–Oriented Applications. In *Software Engineering, International Summer Schools, ISSSE 2006-2008. Revised Tutorial Lectures*, pages 106–139. Springer, 2009.

[Bei90] B. Beizer. *Software testing techniques (2nd ed.)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990.

[BGL+07] F. Bouquet, C. Grandpierre, B. Legeard, F. Peureux, N. Vacelet, and M. Utting. A Subset of Precise UML for Model–based Testing. In *A-MOST '07: Proceedings of the 3rd international workshop on Advances in model-based testing*, pages 95–104, New York, NY, USA, 2007. ACM.

[BHB+07] A. Barbir, C. Hobbs, E. Bertino, F. Hirsch, and L. Martino. Challenges of Testing Web Services and Security in SOA Implementations. In *Test and Analysis of Web Services*. 2007.

[BHH10] M. Bozkurt, M. Harman, and Y. Hassoun. Testing Web Services: A Survey. Technical Report TR-10-01, Department of Computer Science, King's College London, January 2010.

[Bin99] R. V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison Wesley Longman, Inc., 1999.

[bir] BIRT. available at `http://www.eclipse.org/birt/` [accessed: November 25, 2010].

[Bis07] M. Bishop. About Penetration Testing. *IEEE Security & Privacy*, 5(6), 2007.

[BL02] L. C. Briand and Y. Labiche. A UML-Based Approach to System Testing. *Software and System Modeling*, 1(1):10–42, 2002.

[BLH09] L. C. Briand, Y. Labiche, and S. He. Automating Regression Test Selection based on UML Designs. *Inf. Softw. Technol.*, 51(1), 2009.

[BLO03] L. C. Briand, Y. Labiche, and L. O'Sullivan. Impact Analysis and Change Management of UML Models. *IEEE International Conference on Software Maintenance*, 2003.

[BM05] A. Bertolino and E. Marchetti. A Brief Essay on Software Testing. Technical report, Institute of Information Science and Technologies (ISTI), 2005.

[Boe81] B. W. Boehm. *Software Engineering Economics*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[BPH10] R. Baskerville and J. Pries-Heje. Explanatory Design Theory. *Business and Information Systems Engineering*, 2:271–282, 2010.

[BRDG+07] P. Baker, P. Ru Dai, J. Grabowski, O. Haugen, I. Schieferdecker, and C. E. Williams. *Model-Driven Testing - Using the UML Testing Profile*. Springer, 2007.

[Bre10] R. Breu. Ten Principles for Living Models: A Manifesto of Change-Driven Software Engineering. In *CISIS 2010*, 2010.

[BW05] A. D. Brucker and B. Wolff. Interactive testing using HOL-TestGen. In *Formal Approaches to Testing of Software*, number 3997. Springer, 2005.

[CD08] G. Canfora and M. Di Penta. Service–Oriented Architectures Testing: A Survey. In *ISSSE*, pages 78–105. Springer, 2008.

[CDP06] G. Canfora and M. Di Penta. Testing Services and Service-Centric Systems: Challenges and Opportunities. *IT Professional*, 8:10–17, 2006.

[CFV07] M. Cova, V. Felmetsger, and G. Vigna. Vulnerability Analysis of Web–Based Applications. In *Testing and Analysis of Web Services*. Springer, 2007.

[CH03] K. Czarnecki and S. Helsen. Classification of Model Transformation Approaches. In *OOPSLA03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

[CHvRR04] L. Clement, A. Hately, C. von Riegen, and T. Rogers. *UDDI Version 3.0.2*, 2004. available at `http://www.uddi.org/pubs/uddi_v3.htm` [accessed: February 25, 2011].

[Cib09] M. A. Cibran. Translating BPMN Models into UML Activities. In *Business Process Management Workshops*, volume 17 of *Lecture Notes in Business Information Processing*, pages 236–247. Springer Berlin Heidelberg, 2009.

[CMK08] M. Chen, P. Mishra, and D. Kalita. Coverage–driven automatic Test Generation for UML Activity Diagrams. In *GLSVLSI '08: Proceedings of the 18th ACM Great Lakes symposium on VLSI*, pages 139–142, New York, NY, USA, 2008. ACM.

[CMRW03] R. Chinnici, J. Moreau, A. Ryman, and S. Weerawarana. *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language*, 2003. available at `http://www.w3.org/TR/wsdl20/` [accessed: February 25, 2011].

[CO09] J. Chimiak-Opoka. OCLLib, OCLUnit, OCLDoc: Pragmatic Extensions for the Object Constraint Language. In *Model Driven Engineering Languages and Systems, 12th International Conference, MODELS 2009*, pages 665–669. Springer, 2009.

[COAB10] J. Chimiak-Opoka, B. Agreiter, and R. Breu. Bringing Models into Practice: Design and Usage of UML Profiles and OCL Queries in a Showcase. In *Proc. of the 16th International Conference on Information and Software Technologies, IT'2010*, pages 265–273. Technologija, April 2010.

[COFLL08] J. Chimiak-Opoka, M. Felderer, C. Lenz, and C. Lange. Querying UML Models using OCL and Prolog: A Performance Study. In *Model Driven Engineering, Verification, and Validation*, 2008.

[COGIOT06] J. Chimiak-Opoka, G. Giesinger, F. Innerhofer-Oberperfler, and B. Tilg. Tool–Supported Systematic Model Assessment. In *Modellierung 2006*, 2006.

[COLF+09] J. Chimiak-Opoka, S. Löw, M. Felderer, R. Breu, F. Schupp, F. Fielder, and M. Breu. Generic Arbitrations for Test Reporting. In *IASTED International Conference on Software Engineering 2009*, 2009.

[Com09] Common Criteria Recognition Arrangement. Common Criteria for Information Technology Security Evaluation, 2009. available at `http://www.commoncriteriaportal.org/thecc.html` [accessed: November 25, 2010].

[CPS02] Y. Chen, R. L. Probert, and D. P. Sims. Specification–based Regression Test Selection with Risk Analysis. In *CASCON '02*, 2002.

[CRM07] Y. Cheon and C. E. Rubio-Medrano. Random Test Data Generation for Java Classes annotated with JML Specifications. In *International Conference on Software Engineering Research and Practice, Volume II, June 2528, 2007, Las Vegas*, pages 385–392, 2007.

[CXX06] M. Chen, Q. Xiaokang, and L. Xuandong. Automatic Test Case Generation for UML Activity Diagrams. In *AST '06: Proceedings of the 2006 international workshop on Automation of software test*, pages 2–8, New York, NY, USA, 2006. ACM.

[DBE+07] M. Di Penta, M. Bruno, G. Esposito, V. Mazza, and G. Canfora. Web Services Regression Testing. In *Test and Analysis of Web Services*. 2007.

[Dij69] E. W. Dijkstra. Notes on Structured Programming. 1969.

[DLS85] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *Tutorial software quality assurance: a practical approach*, 1985.

[EB04] M. Elaasar and L. Briand. An overview of uml consistency management. Technical Report SCE-04-18, Department of Systems and Computer Engineering, University of Ottawa, August 2004.

[ecl] Eclipse. available at `http://www.eclipse.org/` [accessed: November 25, 2010].

[EHH+08] G. Engels, A. Hess, B. Humm, O. Juwig, M. Lohmann, J. P. Richter, M. Voß, and J. Willkomm. A Method for Engineering a True Service-Oriented Architecture. In *ICEIS 2008*, 2008.

[EHHS02] G. Engels, J. H. Hausmann, R. Heckel, and S. Sauer. Testing the Consistency of Dynamic UML Diagrams. In *Proceedings of the Sixth International Conference on Integrated Design and Process Technology (IDPT 2002), Pasadena, CA (USA)*, June 2002. Pasadena, CA, USA.

[Erl05] T. Erl. *Service-oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR Upper Saddle River, NJ, USA, 2005.

[FAB10] M. Felderer, B. Agreiter, and R. Breu. Security Testing by Telling TestStories. In *Modellierung 2010*, 2010.

[FAB11a] M. Felderer, B. Agreiter, and R. Breu. Evolution of Security Requirements Tests for Service–centric Systems. In *International Symposium on Engineering Secure Software and Systems 2011*, 2011.

[FAB11b] M. Felderer, B. Agreiter, and R. Breu. Managing Evolution of Service Centric Systems. In *IASTED International Conference on Software Engineering 2011*, 2011.

[FABA10] M. Felderer, B. Agreiter, R. Breu, and A. Armenteros. Security Testing By Telling TestStories. In *Modellierung 2010*, 2010.

[FBCO+09] M. Felderer, R. Breu, J. Chimiak-Opoka, M. Breu, and F. Schupp. Concepts for Model–Based Requirements Testing of Service Oriented Systems. In *IASTED International Conference on Software Engineering 2009*, 2009.

[FCOB10] M. Felderer, J. Chimiak-Opoka, and R. Breu. Model–driven System Testing of Service–oriented Systems. In *Proceedings of the 12th International Conference on Enterprise Information Systems*, 2010.

[FFZB09] M. Felderer, F. Fiedler, P. Zech, and R. Breu. Flexible Test Code Generation for Service Oriented Systems. In *QSIC'2009*, 2009.

[FIMN07] Q. Farooq, M. Z. Iqbal, Z. I. Malik, and A. Nadeem. An approach for selective state machine based regression testing. In *A-MOST 07*, 2007.

[Fir03] D. G. Firesmith. Engineering Security Requirements. *Journal of Object Technology*, 2(1), 2003.

[Fir04] D. G. Firesmith. Specifying Reusable Security Requirements. *Journal of Object Technology*, 3(1), 2004.

[Fra07] G. Fraser. *Automated Software Testing with Model Checkers*. PhD thesis, Institute for Softwaretechnology, Graz University of Technology, 2007.

[Fre08] D. Freeman. Practical Test Reporting, 2008. available at `http://www.stickyminds.com/` [accessed: February 25, 2011].

[FZ10] M. Felderer and P. Zech. Telling TestStories – A Tool for Tabular and Model-driven System Testing. *Testing Experience*, 12, 2010.

[FZF+09] M. Felderer, P. Zech, F. Fiedler, J. Chimiak-Opoka, and R. Breu. Model-driven System Testing of a Telephony Connector with Telling Test Stories. In *Software Quality Engineering – Proceedings of the CONQUEST 2009*, 2009.

[FZFB10] M. Felderer, P. Zech, F. Fiedler, and R. Breu. A Tool–based methdology for System Testing of Service–oriented Systems. In *The Second International Conference on Advances in System Testing and Validation Lifecycle*, 2010.

[G. 97] G. Fink and M. Bishop. Property-Based Testing: A New Approach to Testing for Assurance. *ACM SIGSOFT Software Engineering Notes*, 22:74–80, 1997.

[GBR98] A. Gotlieb, B. Botella, and M. Rueher. Automatic Test Data Generation using Constraint Solving Techniques. *SIGSOFT Softw. Eng. Notes*, 23(2):53–62, 1998.

[GF94] O. C. Z. Gotel and C. W. Finkelstein. An analysis of the requirements traceability problem. In *Proceedings of the First International Conference on Requirements Engineering, 1994*, 1994.

[GG75] J. B. Goodenough and S. L. Gerhart. Toward a Theory of Test Data Selection. *IEEE Trans. Software Eng.*, 1(2):156–173, 1975.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[GNRS09] H. Götz, M. Nickolaus, T. Roßner, and K. Salomon. *iX Studie Modellbasiertes Testen*. Heise Zeitschriften Verlag, 2009. (in German).

[GPCL08] R. P. Gorthi, A. Pasala, K. Chanduka, and B. Leong. Specification-Based Approach to Select Regression Test Suite to Validate Changed Software. 2008.

[Haf08] Hafner, M. and Breu, R. *Security Engineering for Service–Oriented Architectures*. Springer-Verlag, Berlin Heidelberg, 2008.

[Har00] M. J. Harrold. Testing: A Roadmap. In *In The Future of Software Engineering*, pages 61–72. ACM Press, 2000.

[Har06] Hartman, A. and Katara, M. and Olvovsky, S. Choosing a Test Modeling Language: A Survey. In *Haifa Verification Conference*, volume 4383 of *Lecture Notes in Computer Science*, pages 204–218. Springer, 2006.

[Hie08] R. Hierons. Verdict Functions in Testing with a Fault Domain or Test Hypotheses. *ACM Trans. Softw. Eng. Methodol.*, 14(2):246–246, 2008.

[HIM00] J. Hartmann, C. Imoberdorf, and M. Meisinger. UML-Based Integration Testing. In *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT*

international symposium on Software testing and analysis, pages 60–70, New York, NY, USA, 2000. ACM.

[HMPR04] A. R. Hevner, S. T. March, J. Park, and S. Ram. Design Science in Information Systems Research. *MIS Quarterly*, 28(1):75–106, 2004.

[HN04] 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*, pages 129–132, New York, NY, USA, 2004. ACM.

[Hna01] B. Hnatkowska. Consistency Checking in UML. In *Proceedings of 4th Internation Conference on Information System Modeling*, 2001.

[HVFR05] J. Hartmann, M. V., H. Foster, and A. Ruder. A UML–based approach to system testing. *ISSE*, 1(1), 2005.

[IEE90a] IEEE. IEEE Standard Glossary of Software Engineering Terminology. Technical report, IEEE, 1990.

[IEE90b] IEEE. *Standard Glossary of Software Engineering Terminology*. IEEE, 1990.

[IEE04a] IEEE. *802.1X-REV - Revision of 802.1X-2004 - Port Based Network Access Control*, 2004. available at `http://www.ieee802.org/1/pages/802.1x-rev.html` [accessed: February 25, 2011].

[IEE04b] IEEE Computer Society. *Software Engineering Body of Knowledge (SWEBOK)*. EUA, 2004. available at `http://www.swebok.org/` [accessed: February 25, 2011].

[IET00] IETF. *Remote Authentication Dial In User Service (RADIUS)*, 2000. available at `http://www.ietf.org/rfc/rfc2865.txt` [accessed: February 25, 2011].

[ISO94] ISO/IEC. *Information technology – open systems interconnection – conformance testing methodology and framework*, 1994. International ISO/IEC multi–part standard No. 9646.

[ISO01] ISO/IEC. *ISO/IEC 9126. Software engineering – Product quality*. ISO/IEC, 2001.

[JMT08] J. Julliand, P. A. Masson, and R. Tissot. Generating Security Tests in Addition to Functional Tests. In *AST '08: Proceedings of the 3rd international workshop on Automation of software test*, New York, NY, USA, 2008. ACM.

[jun] Junit. available at `http://www.junit.org/` [accessed: November 25, 2010].

[Jür02] J. Jürjens. UMLsec: Extending UML for Secure Systems Development. In *UML '02: Proceedings of the 5th International Conference on The Unified Modeling Language*, London, UK, 2002. Springer-Verlag.

[Jür08] J. Jürjens. Model–based Security Testing Using UMLsec. *Electron. Notes Theor. Comput. Sci.*, 220(1), 2008.

[KBR+05] N. Kavantzas, D. Burdett, G. Ritzinger, T. Fletcher, and Y. Lafon. Web Services Choreography Description Language Version 1.0, November 2005. available at `http://www.w3.org/TR/ws-cdl-10/` [accessed: February 25, 2011].

[KKBK07] H. Kim, S. Kang, J. Baik, and I. Ko. Test Cases Generation from UML Activity Diagrams. *ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*, 3:556–561, 2007.

[Küs04] J. M. Küster. *Consistency Management of Object–Oriented Behavioral Models*. PhD thesis, Faculty of Computer Science, Electrical Engineering, and Mathematics, University of Paderborn, 2004.

[LC04] C. F. J. Lange and M. R. V. Chaudron. An Empirical Assessment of Completeness in UML Designs. In *In Proceedings of the 8th International Conference on Empirical Assessment in Software Engineering (EASE'04)*, 2004.

[LCP08] L. Lin, J.M. Carter, and J.H. Poore. Using state machines to model and manage requirements changes and specification changes. In *Circuits and Systems, 2008. MWSCAS 2008. 51st Midwest Symposium on*, pages 523–526. IEEE, 2008.

[Leh80] M. M. Lehman. Programs, Life Cycles, and Laws of Software Evolution. In *Proc. of IEEE, 68, 9*, 1980.

[LJX+04] W. Linzhang, Y. Jiesong, Y. Xiaofeng, H. Jun, L. Xuandong, and Z. Guoliang. Generating Test Cases from UML Activity Diagram based on Gray-Box Method. *Asia-Pacific Software Engineering Conference*, 0:284–291, 2004.

[LLQC07] B. L. Li, Z. Li, L. Qing, and Y. H. Chen. Test Case Automate Generation from UML Sequence Diagram and OCL Expression. *International Conference on Computational Intelligence and Security*, 0:1048–1052, 2007.

[LMD05] M. Lanza, R. Marinescu, and S. Ducasse. *Object–Oriented Metrics in Practice*. Springer New York, 2005.

[LS06] M. S. Lund and K. Stølen. Deriving Tests from UML 2.0 Sequence Diagrams with neg and assert. In *AST '06: Proceedings of the 2006 international workshop on Automation of software test*, pages 22–28, New York, NY, USA, 2006. ACM.

[LW89] H. K. N. Leung and L. White. Insights into Regression Testing. In *Proceedings of the Conference on Software Maintenance*, pages 60–69, 1989.

[MD08] T. Mens and S. Demeyer, editors. *Software Evolution*. Springer, 2008.

[mdt] Model Driven Test Development. available at `http://www.mdtd.de/` [accessed: May 8, 2010].

[MFC+09] B. Meyer, A. Fiva, I. Ciupa, A. Leitner, Y. Wei, and E. Stapf. Programs That Test Themselves. *Computer*, 42:46–55, 2009.

[MJP+07] P. A. Masson, J. Julliand, J. C. Plessis, E. Jaffuel, and G. Debois. Automatic generation of model based tests for a class of security properties. In *Proceedings of the 3rd international workshop on Advances in model-based testing*. ACM, 2007.

[ML07] N. Mitra and Y. Lafon. *SOAP Version 1.2 Part 0: Primer (Second Edition)*, 2007. available at `http://www.w3.org/TR/2007/REC-soap12-part0-20070427/` [accessed: February 25, 2011].

[Mou10] T. Mouelhi. *Testing and Modeling Security Mechanisms in Web Applications*. PhD thesis, RSM, University of Rennes, 2010.

[MP05] J. McQuillan and J. Power. A Survey of UML-Based Coverage Criteria for Software Testing. Technical report, National University of Ireland, Maynooth, 2005.

[MR05] C. Michael and W. Radosevich. Risk-based and functional security testing. Technical report, Technical report, US Department of Homeland Security and Cigital Inc, 2005.

[MR09] C. C. Michael and W. Radosevich. Risk–based and Functional Security Testing. Technical report, Cigital, 2009. `https://buildsecurityin.us-cert.gov/bsi/articles/best-practices/testing/255-BSI.pdf` [accessed: May 8, 2010].

[MS95] S. T. March and G. F. Smith. Design and Natural Science Research on Information Technology. *Decis. Support Syst.*, 15:251–266, December 1995.

[MS04] T. Margaria and B. Steffen. Lightweight coarse-grained coordination: a scalable system-level approach. *STTT*, 5(2-3), 2004.

[Mug05] Mugridge, R. and Cunningham, W. *Fit for Developing Software: Framework for Integrated Tests*. Prentice Hall, 2005.

[MvDZB08] L. Moonen, A. van Deursen, A. Zaidman, and M. Bruntink. On the Interplay Between Software Testing and Evolution and its Effect on Program Comprehension. In *Software Evolution*. 2008.

[NFLTJ06] C. Nebut, F. Fleurey, Y. Le Traon, and J. M. Jezequel. Automatic Test Generation: A Use Case Driven Approach. *IEEE Trans. Softw. Eng.*, 32(3):140–155, 2006.

[NIS02] NIST. *The economic Impacts of inadequate Infrastructure for Software Testing*, May 2002. available at `www.nist.gov/director/planning/upload/report02-3.pdf` [accessed: February 25, 2011].

[NST06] CNSS Instruction Formerly NSTISSI. 4009, "National Information Assurance Glossary". Technical report, Committee on National Security Systems, 2006.

[OA99] A. J. Offutt and A. Abdurazik. Generating Tests from UML Specifications. In *UML*, pages 416–429. Springer, 1999.

[OAS06] OASIS Standard. OASIS SOA Reference Model TC, 2006. available at `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=soa-rm` [accessed: February 25, 2011].

[OAS07] OASIS Standard. Web Services Business Process Execution Language Version 2.0 - OASIS Standard, April 2007. available at `http://docs.oasis-open.org/wsbpel/2.0/` [accessed: February 25, 2011].

[oaw] openArchitectureWare. available at `http://www.openarchitectureware.org/` [accessed: November 25, 2010].

[OGC07] OGC. *ITIL Lifecycle Publication Suite Books, 2nd impression*. TSO, 2007.

[OMG03] OMG. *MDA Guide Version 1.0.1*. Object Modeling Group, 2003. available at `http://www.omg.org/cgi-bin/doc?omg/03-06-01.pdf` [accessed: February 25, 2011].

[OMG05] OMG. *UML Testing Profile, Version 1.0*, 2005. available at `http://www.omg.org/spec/UTP/1.0/PDF` [accessed: February 25, 2011].

[OMG06a] OMG. *Meta Object Facility (MOF) Core Specification, Version 2.0*, 2006. available at `http://www.omg.org/spec/MOF/2.0/PDF/` [accessed: February 25, 2011].

[OMG06b] OMG. *Object Constraint Language Version 2.0*, 2006. available at `http://www.omg.org/spec/OCL/2.0/PDF` [accessed: February 25, 2011].

[OMG07a] OMG. *OMG Unified Modeling Language (OMG UML), Infrastructure, V2.1.2*, 2007. available at `http://www.omg.org/spec/UML/2.1.2/Infrastructure/PDF/` [accessed: February 25, 2011].

[OMG07b] OMG. *OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2*, 2007. available at `http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF/` [accessed: February 25, 2011].

[OMG09a] OMG. *Business Process Model and Notation (BPMN) 1.2*, 2009. available at `http://www.omg.org/spec/BPMN/1.2/PDF/` [accessed: February 25, 2011].

[OMG09b] OMG. *Service Oriented Architecture Modeling Language (SoaML) - Specificiation for the UML Profile and Metamodel for Services (UPMS)*. Object Modeling Group, 2009. available at `http://www.omg.org/spec/SoaML/1.0/Beta2/PDF` [accessed: February 25, 2011].

[OMG10] OMG. *OMG Systems Modeling Language (OMG SysML), Version 1.2*, 2010. available at `http://www.omg.org/spec/SysML/1.2/PDF` [accessed: February 25, 2011].

[PA09] S. L. Pfleeger and J. Atlee. *Software Engineering: Theory and Practice (4th ed.)*. Prentice Hall, 2009.

[PK04] A. Paradkar and T. Klinger. Automated Consistency and Completeness Checking of Testing Models for Interactive Systems. *Computer Software and Applications Conference, Annual International*, 1:342–348, 2004.

[PM04] B. Potter and G. McGraw. Software Security Testing. *IEEE Security & Privacy*, 2004.

[PMLT08] A. Pretschner, T. Mouelhi, and Y. Le Traon. Model-Based Tests for Access Control Policies. In *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*. IEEE Computer Society Washington, DC, USA, 2008.

[PP04] A. Pretschner and J. Philipps. Methodological Issues in Model–Based Testing. In *Model-Based Testing of Reactive Systems*, volume 3472 of *Lecture Notes in Computer Science*, pages 281–291. Springer, 2004.

[PS00] D. C. Petriu and Y. Sun. Consistent Behaviour Representation in Activity and Sequence Diagrams. In *UML'00: Proceedings of the*

3rd international conference on The unified modeling language, pages 369–382, Berlin, Heidelberg, 2000. Springer.

[PTRC08] K. Peffers, T. Tuunanen, M. Rothenberger, and S. Chatterjee. A Design Science Research Methodology for Information Systems Research. *J. Manage. Inf. Syst.*, 24(3):45–77, 2008.

[RBGW10] T. Roßner, C. Brandes, H. Götz, and M. Winter. *Basiswissen Modellbasierter Test*. dpunkt Verlag, 2010. (in German).

[RH96] G. Rothermel and M. J. Harrold. Analyzing Regression Test Selection Techniques. *IEEE Transactions on Software Engineering*, 22, 1996.

[Ric01] M. Richters. *A Precise Approach to Validating UML Models and OCL Constraints*. PhD thesis, Fachbereich 3, University of Bremen, 2001.

[rmi] RMI. available at `http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp` [accessed: November 25, 2010].

[RTDP+09] F. Ricca, M. Torchiano, M. Di Penta, M. Ceccato, and P. Tonella. Using Acceptance Tests as a Support for Clarifying Requirements: A Series of Experiments. *Inf. Softw. Technol.*, 51(2):270–283, 2009.

[Sch06] D. C. Schmidt. Model-Driven Engineering. *IEEE Computer*, 39(2), 2006.

[Sch07] I. Schieferdecker. Modellbasiertes Testen. *Objekt Spektrum*, (3):39–45, 2007. (in German).

[SDA05] I. Schieferdecker, G. Din, and D. Apostolidis. Distributed Functional and Load Tests for Web Services. *Int. J. Softw. Tools Technol. Transf.*, 7(4):351–360, 2005.

[Sel07] B. Selic. A Systematic Approach to Domain-Specific Language Design Using UML. In *ISORC '07: Proceedings of the 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, pages 2–9, Washington, DC, USA, 2007. IEEE Computer Society.

[SF97] M. Shroff and R. B. France. Towards A Formalization of UML Class Structures in Z. In *In Proceedings of COMPSAC'97*, pages 646–651. IEEE Computer Society, 1997.

[SKR07] P. A. Pari Salas, P. Krishnan, and Kelvin J. Ross. Model–Based Security Vulnerability Testing. *Software Engineering Conference, Australian*, 0, 2007.

[SM99] B. Steffen and T. Margaria. METAFrame in Practice: Design of Intelligent Network Services. In *Correct System Design, Recent Insight and Advances, (to Hans Langmaack on the occasion of his retirement from his professorship at the University of Kiel)*, pages 390–415, London, UK, 1999. Springer.

[spr] Spring. available at `http://www.springsource.org/` [accessed: November 25, 2010].

[Sta73] H. Stachowiak. *Allgemeine Modelltheorie*. Springer, 1973. (in German).

[TBL10] T. Trojer, M. Breu, and S. Löw. Change–driven Model Evolution for Living Models. *3rd Workshop on Model-Driven Tool and Process Integration*, 2010.

[Tre04] J. Tretmans. Model–based testing: Property checking for real. Keynote address at the International Workshop for Construction and Analysis of Safe Secure, and Interoperable Smart Devices, 2004. available at `http://www-sop.inria.fr/everest/events/cassis04`.

[UL07] M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[uml] UML2 Tools. available at `http://wiki.eclipse.org/MDT-UML2Tools` [accessed: November 25, 2010].

[UPL06] M. Utting, A. Pretschner, and B. Legeard. A Taxonomy of Model–based Testing. Technical Report 04/2006, Department of Computer Science, The Universiy of Waikato (New Zealand), 2006.

[vMZ99] A. von Mayrhauser and N. Zhang. Automated Regression Testing using DBT and Sleuth. *Journal of Software Maintenance*, 11(2), 1999.

[VRMCL09] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner. A Break in the Clouds: Towards a Cloud Definition. *SIGCOMM Comput. Commun. Rev.*, 39(1):50–55, 2009.

[vW08] K. van Wyk. Adapting Penetration Testing for Software Development Purposes, 08 2008. available at `https://buildsecurityin.us-cert.gov/daisy/bsi/articles/best-practices/penetration/655-BSI.html` [accessed: February 25, 2011].

[WDT+05] C. Willcock, T. Deiss, S. Tobies, S. Keil, F. Engler, and S. Schulz. *An Introduction to TTCN–3*. John Wiley and Sons, 2005.

[WJ02] G. Wimmel and J. Jürjens. Specification–based Test Generation for Security–critical Systems using Mutations. *Lecture notes in computer science*, pages 471–482, 2002.

[WRH+00] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering: an Introduction*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[ZDSD05] J. Zander, Z. R. Dai, I. Schieferdecker, and G. Din. From U2TP Models to Executable Tests with TTCN-3 - An Approach to Model Driven Testing. In *TestCom*, 2005.

[Zec09] P. Zech. SOA Testing by Telling TestStories, 2009.

[ZG03] D. Zowghi and V. Gervasi. On the Interplay between Consistency, Completeness, and Correctness in Requirements Evolution. *Information and Software Technology*, 45(14):993 – 1009, 2003. Eighth International Workshop on Requirements Engineering: Foundation for Software Quality.

Appendix A

Acronyms

In this thesis the following acronyms have been used:

DSL. Domain Specific Language

FIT. Framework for Integrated Test

MBT. Model Based Testing

MDA. Model Driven Architecture

MDE. Model Driven Engineering

MDT. Model Driven Testing

OCL. Object Constraint Language

QoS. Quality of Service

RMI. Remote Method Invocation

SCS. Service Centric System

SLA. Service Level Agreement

SOA. Service Oriented Architecture

SoaML. Service Oriented Architecture Modeling Language

SQUAM. Systematic Quality Assessment of Models

SysML. System Modeling Language

TTS. Telling TestStories

UML. Unified Modeling Language

U2TP. UML 2.0 Testing Profile

Appendix B

TTS Profile

In Figure B.1 the profile implementation of TTS within the UML2Tools [1] is depicted. In Table B.1 the metaclasses and their extending stereotypes are listed.

Metaclass	**List of Stereotypes**
Action	Assertion, SequenceElement, Servicecall, Trigger, WFElement, WFEnd, WFStart
Activity	ParallelTask, Testsequence, Teststory, WFElement, WFEnd, WFStart
Behavior	GlobalProcess, LocalProcess
Class	Component, FunctionalRequirement, NonFunctional-Requirement, Requirement, Service
Classifier	FunctionalRequirement, NonFunctionalRequirement, Requirement, Service
Operation	ServiceOperation
Package	Components, GlobalProcess, LocalProcess, Requirements, System, Test, Testsequence, Testsequences, Teststories, Teststory, Testsystem
StructuredActivityNode	ParallelTask

Table B.1: Metaclasses and Extending Stereotypes of the TTS Profile

[1] The UML2Tools are available at `http://www.eclipse.org/modeling/mdt/?project=uml2tools` [November 25, 2010].

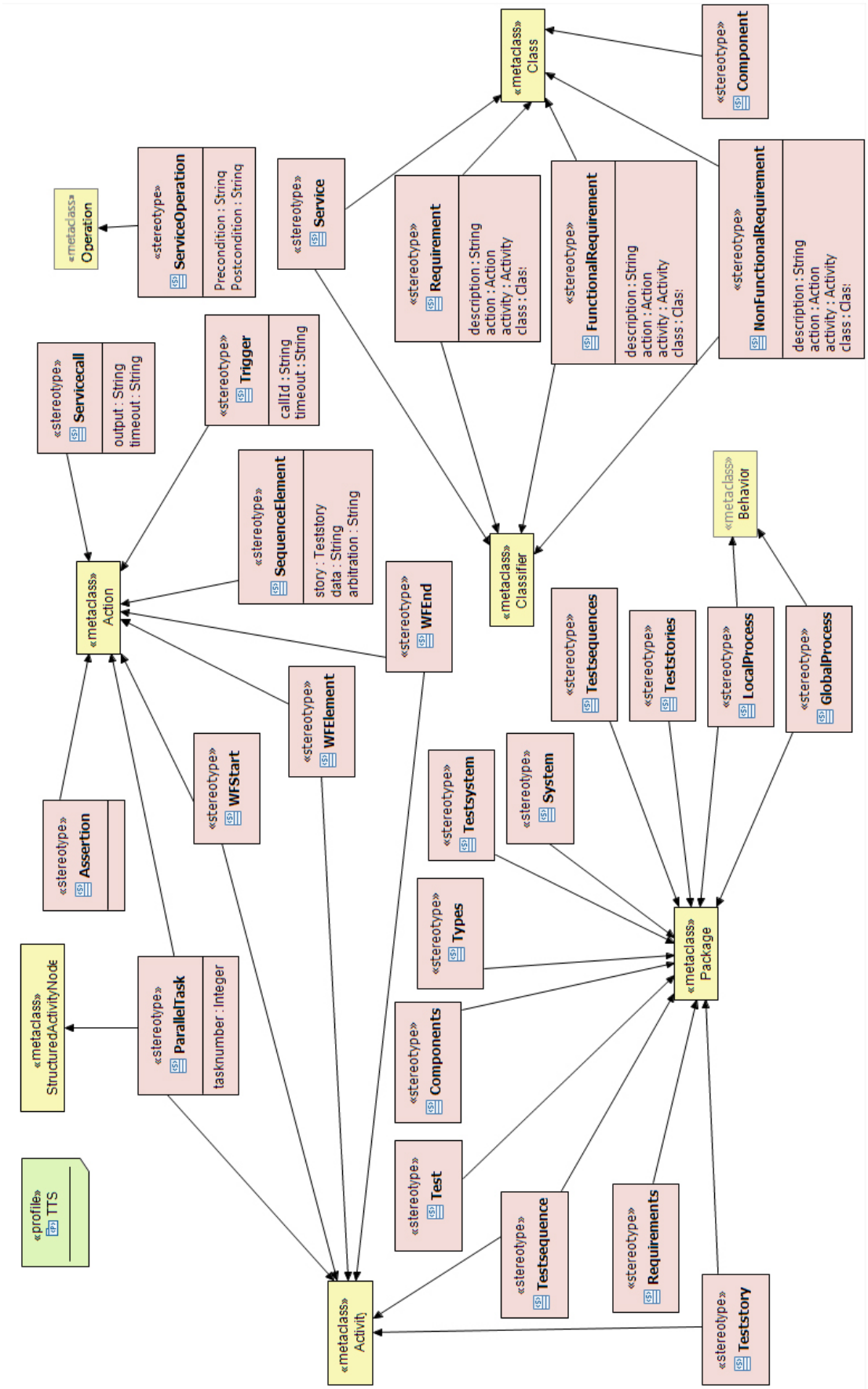


Figure B.1: UML2Tools UML Profile of TTS

Appendix C

Validity and Coverage

The validity and coverage criteria of TTS are implemented in SQUAM [COGIOT06] based on OCL [OMG06b]. Validity checks are considered in Section C.1 and a catalog of coverage criteria is considered in Section C.2.

C.1 Validity Checks

In SQUAM validity checks are implemented by definitions (see Listing C.1) and queries (see Listing C.2) that can be invoked from the runtime environment. The checks shown in the following listing have been implemented in the TTS tool and guarantee that tests can be generated and are of high quality.

```
1  library TTSCorrectness
2          metamodel uml
3          require checks.basic.UMLSpecificDefinitions
4
5          public definitions
6          context Class
7                  def hasRequirementName:
8                          hasRequirementName() : Boolean =
9                          if profileIsTypeOf('Requirement') then
10                                 name.size()>0 and
11                                 getValue(self.getAppliedStereotypes()->any(a|a.name='Requirement
                                       '),'description').oclAsType(String).size()>0
12                         else
13                                 false
14                         endif
15         enddefinitions
16
17         public definitions
18         context Class
19                 def hasServiceName:
20                         hasServiceName() : Boolean =
21                         if profileIsTypeOf('Service') then
22                                 name.size()>0
23                         else
24                                 false
25                         endif
26         enddefinitions
27
28         public definitions
29         context Activity
30                 def hasTeststoryName:
31                         hasTeststoryName() : Boolean =
32                         if profileIsTypeOf('Teststory') then
33                                 name.size()>0
34                         else
35                                 false
36                         endif
37         enddefinitions
```

```
38
39          public definitions
40          context Class
41                  def isRequirementUnique:
42                          isRequirementUnique() : Boolean =
43                           if profileIsTypeOf('Requirement') then
44                           TTS::Requirement.allInstances()->select( r | r.base_Class.name = name)
                                 ->size() = 1
45                           else
46                                   false
47                           endif
48          enddefinitions
49
50          public definitions
51          context Class
52                  def isServiceUnique:
53                          isServiceUnique() : Boolean =
54                           if profileIsTypeOf('Service') then
55                           TTS::Service.allInstances()->select( r | r.base_Class.name = name)->
                                 size() = 1
56                           else
57                                   false
58                           endif
59          enddefinitions
60
61          public definitions
62          context Activity
63                  def isTeststoryUnique:
64                          isTeststoryUnique() : Boolean =
65                           if profileIsTypeOf('Teststory') then
66                           TTS::Teststory.allInstances()->select( r | r.base_Activity.name = name)
                                 ->size() = 1
67                           else
68                                   false
69                           endif
70          enddefinitions
71
72          public definitions
73          context Activity
74                  def hasInitialNode:
75                          hasInitialNode() : Boolean =
76                           allOwnedElements()->select(o | o.oclIsTypeOf(InitialNode))->size() = 1
77          enddefinitions
78
79          public definitions
80          context Activity
81                  def hasFinalNode:
82                          hasFinalNode() : Boolean =
83                           allOwnedElements()->select(o | o.oclIsTypeOf(ActivityFinalNode))->size
                                 () > 0
84          enddefinitions
85
86          public definitions
87          context Class
88                  def isRequirementTraceable:
89                          isRequirementTraceable() : Boolean =
90                          if profileIsTypeOf('Requirement') then
91                                  getValue(self.getAppliedStereotypes()->any(a|a.name='Requirement
                                         '),'action')->notEmpty() or
92                                  getValue(self.getAppliedStereotypes()->any(a|a.name='Requirement
                                         '),'activity')->notEmpty() or
93                                  getValue(self.getAppliedStereotypes()->any(a|a.name='Requirement
                                         '),'class')->notEmpty()
94                          else
95                           false
96                          endif
97          enddefinitions
98
99          public definitions
100         context Activity
101                 def hasAssertion:
102                         hasAssertion() : Boolean =
103                          allOwnedElements()->select(o | o.profileIsTypeOf('Assertion'))->size()
                                 > 0
104         enddefinitions
105
106         public definitions
107         context Activity
108                 def hasServiceCall:
109                         hasServiceCall() : Boolean =
110                          allOwnedElements()->select(o | o.profileIsTypeOf('Servicecall'))->size
                                 () > 0
111         enddefinitions
112
113         public definitions
114         context Action
115                 def hasTestdata:
116                         hasTestdata() : Boolean =
```

```
117                     if profileIsTypeOf('SequenceElement') then
118                             getValue(self.getAppliedStereotypes()->any(a|a.name='
                                    SequenceElement'),'data')->notEmpty() and
119                             getValue(self.getAppliedStereotypes()->any(a|a.name='
                                    SequenceElement'),'story')->notEmpty()
120                     else
121                             false
122                     endif
123     enddefinitions
124
125     public definitions
126     context Activity
127             def isAlternativeDefined:
128                     isAlternativeDefined() : Boolean =
129                      allOwnedElements()->select(o | o.oclIsTypeOf(DecisionNode))->size() =
130                      allOwnedElements()->select(o | o.oclIsTypeOf(MergeNode))->size()
131     enddefinitions
132
133     public definitions
134     context ActivityPartition
135             def isValidPartition:
136                     isValidPartition() : Boolean =
137                             name.size() > 0 and represents <> null
138     enddefinitions
139
140     public definitions
141     context CallOperationAction
142             def isValidServiceCall:
143                     isValidServiceCall() : Boolean =
144                             name.size() > 0 and inPartition <> null and
145                             operation <> null
146     enddefinitions
147
148     public definitions
149     context Action
150             def isValidTrigger:
151                     isValidTrigger() : Boolean =
152                     if profileIsTypeOf('Trigger') then
153                             name.size() > 0 and
154                             getValue(self.getAppliedStereotypes()->any(a|a.name='Trigger'),'
                                    callId')->notEmpty()
155                     else
156                             true
157                     endif
158     enddefinitions
159
160     public definitions
161     context Action
162             def isValidInOutAction:
163                     isValidInOutAction() : Boolean =
164                     if not self.oclIsTypeOf(InitialNode) and not self.oclIsTypeOf(
                            ActivityFinalNode) then
165                      self.incoming->size()>0 and self.outgoing->size()>0
166                     else
167                             false
168                     endif
169     enddefinitions
170
171     public definitions
172     context Action
173             def isValidAssertion:
174                     isValidAssertion() : Boolean =
175                     if profileIsTypeOf('Assertion') then
176                             name.size() > 0 and
177                             ( getValue(self.getAppliedStereotypes()->any(a|a.name='Assertion
                                    '),'pass')->notEmpty() or
178                               getValue(self.getAppliedStereotypes()->any(a|a.name='Assertion
                                    '),'fail')->notEmpty() )
179                     else
180                             true
181                     endif
182     enddefinitions
183
184     public definitions
185     context Model
186             def hasTeststory:
187                     hasTeststory() : Boolean =
188                             TTS::Teststory.allInstances()->size() > 0
189     enddefinitions
190
191     public definitions
192     context Model
193             def hasRequirement:
194                     hasRequirement() : Boolean =
195                             TTS::Requirement.allInstances()->size() > 0
196     enddefinitions
197
198     public definitions
```

```
199         context Model
200                 def hasService:
201                         hasService() : Boolean =
202                                 TTS::Service.allInstances()->size() > 0
203         enddefinitions
204
205         public definitions
206         context Activity
207                 def isValidTeststory:
208                         isValidTeststory() : Boolean =
209                         if profileIsTypeOf('Teststory') then
210                                 hasInitialNode() and
211                                 hasFinalNode() and
212                                 hasAssertion() and
213                                 allOwnedElements()->select(o | o.profileIsTypeOf('Servicecall'))
                                        ->forAll(o | o.oclAsType(CallOperationAction).
                                        isValidServiceCall()) and
214                                 allOwnedElements()->select(o | o.profileIsTypeOf('Assertion'))->
                                        forAll(o | o.oclAsType(Action).isValidAssertion()) and
215                                 allOwnedElements()->select(o | o.profileIsTypeOf('Trigger'))->
                                        forAll(o | o.oclAsType(Action).isValidTrigger())
216                         else
217                                 false
218                         endif
219         enddefinitions
220
221         public definitions
222         context Action
223                 def isValidSequenceElement:
224                         isValidSequenceElement() : Boolean =
225                         if profileIsTypeOf('SequenceElement') then
226                                 getValue(self.getAppliedStereotypes()->any(a|a.name='
                                        SequenceElement'),'data')->notEmpty() and
227                                 getValue(self.getAppliedStereotypes()->any(a|a.name='
                                        SequenceElement'),'story')->notEmpty() and
228                                 getValue(self.getAppliedStereotypes()->any(a|a.name='
                                        SequenceElement'),'arbitration')->notEmpty()
229                         else
230                                 false
231                         endif
232         enddefinitions
233
234         public definitions
235         context Activity
236                 def isValidTestsequence:
237                         isValidTestsequence() : Boolean =
238                         if profileIsTypeOf('Testsequence') then
239                                 hasInitialNode() and
240                                 hasFinalNode() and
241                                 allOwnedElements()->select(o | o.profileIsTypeOf('
                                        SequenceElement'))->forAll(o | o.oclAsType(Action).
                                        isValidSequenceElement())
242                         else
243                                 false
244                         endif
245         enddefinitions
246
247         public definitions
248         context Package
249                         def isValidTestsequences:
250                         isValidTestsequences() : Boolean =
251                         if profileIsTypeOf('Testsequences') then
252                                 allOwnedElements()->select(o | o.profileIsTypeOf('Testsequence')
                                         and o.oclIsTypeOf(Activity) )->forAll(o | o.oclAsType(
                                        Activity).isValidTestsequence())
253                         else
254                                 false
255                         endif
256         enddefinitions
257
258         public definitions
259         context Package
260                 def isValidTeststories:
261                         isValidTeststories() : Boolean =
262                         if profileIsTypeOf('Teststories') then
263                                 allOwnedElements()->select(o | o.profileIsTypeOf('Teststory')
                                        and o.oclIsTypeOf(Activity))->forAll(o | o.oclAsType(
                                        Activity).isValidTeststory())
264                         else
265                                 false
266                         endif
267         enddefinitions
268
269         public definitions
270         context Package
271                 def isValidTestModel:
272                 isValidTestModel() : Boolean =
273                 if profileIsTypeOf('Test') then
```

```
274                         allOwnedElements()->select(o | o.profileIsTypeOf('Teststories'))->size()
                                = 1 and
275                         allOwnedElements()->select(o | o.profileIsTypeOf('Testsequences'))->size
                                () = 1 and
276                         allOwnedElements()->select(o | o.profileIsTypeOf('Testsequences'))->
                                forAll(x | x.oclAsType(Package).isValidTestsequences()) and
277                         allOwnedElements()->select(o | o.profileIsTypeOf('Teststories'))->forAll
                                (x | x.oclAsType(Package).isValidTeststories())
278                 else
279                         false
280                 endif
281         enddefinitions
282
283         /*
284         public tests
285         model models.ping.uml
286                 test testValidate:
287                         let result : Set(TTS::SequenceElement) = allSequenceElements()
288                         expected result.isValidSequenceElement() = Bag{true, true}
289
290                 test testIsValidAssertion:
291                         let result : Set(TTS::Assertion) = allAssertions()
292                         expected result.isValidAssertion() = Bag{true, true}
293
294         endtests
295         */
296
297 endlibrary
```

Listing C.1: Validity Definitions

```
1
2  library TTCorrectnessQueries
3          metamodel uml
4          require checks.correctness.TTSCorrectness
5          require checks.basic.UMLSpecificDefinitions
6
7          public queries
8          context Model
9                          query checkRequirementsUnique:
10                         severity 0
11                         let result:Set(TTS::Requirement)=TTS::Requirement.allInstances()
12                         message result.base_Class.isRequirementUnique() endmessage
13         endqueries
14
15         public queries
16         context Model
17                         query checkIsValidInOutAction:
18                         severity 0
19                         let result:Set(Action)=Action.allInstances()
20                         message result.isValidInOutAction() endmessage
21         endqueries
22
23         public queries
24         context Model
25                         query checkRequirementsComplete:
26                         severity 0
27                         let result:Set(TTS::Requirement)=TTS::Requirement.allInstances()
28                         message result.base_Class.hasRequirementName() endmessage
29         endqueries
30
31         public queries
32         context Model
33                 query checkOneRequirementCoverage:
34                 severity 0
35                 let result:Set(TTS::Requirement) = TTS::Requirement.allInstances()
36                 message result.base_Class.isRequirementTraceable() endmessage
37         endqueries
38
39
40         public queries
41         context Model
42                         query checkIsValidServicecalls:
43                         severity 0
44                         let result:Set(CallOperationAction)=CallOperationAction.allInstances()
45                         message result.isValidServiceCall() endmessage
46         endqueries
47
48         public queries
49         context Model
50                         query checkIsValidTriggers:
```

```
51                          severity 0
52                          let result:Set(TTS::Trigger)=TTS::Trigger.allInstances()
53                          message result.base_Action.isValidTrigger() endmessage
54          endqueries
55
56          public queries
57          context Model
58                          query checkIsValidAssertions:
59                          severity 0
60                          let result:Set(TTS::Assertion)=TTS::Assertion.allInstances()
61                          message result.base_Action.isValidAssertion() endmessage
62          endqueries
63
64          public queries
65          context TTS::Teststory
66                          query checkIsValidTeststory:
67                          severity 0
68                          let result:Boolean=base_Activity.isValidTeststory()
69                          message result endmessage
70          endqueries
71
72          public queries
73          context Model
74                  query checkIsValidTeststories:
75                  severity 0
76                  let result:Set(TTS::Teststories) = TTS::Teststories.allInstances()
77                  message result.base_Package.isValidTeststories() endmessage
78          endqueries
79
80          public queries
81          context Model
82                          query checkIsValidSequenceElements:
83                          severity 0
84                          let result:Set(TTS::SequenceElement)=TTS::SequenceElement.allInstances()
85                          message result.base_Action.isValidSequenceElement() endmessage
86          endqueries
87
88          public queries
89          context Model
90                  query checkIsValidTestsequences:
91                  severity 0
92                  let result:Set(TTS::Testsequences) = TTS::Testsequences.allInstances()
93                  message result.base_Package.isValidTestsequences() endmessage
94          endqueries
95
96          public queries
97          context Model
98                  query checkIsValidTestModel:
99                          severity 0
100                         let result:Boolean = if Package.allInstances()->any(o|o.profileIsTypeOf(
                                'Test')).oclIsUndefined() then false else Package.allInstances()->
                                any(o|o.profileIsTypeOf('Test')).isValidTestModel() endif
101                         message result endmessage
102         endqueries
103
104 endlibrary
```

Listing C.2: Validity Queries

C.2 Catalog of Coverage Criteria

Table C.1 lists all coverage criteria and the affected model elements of the system model. In Section 4.3.2 the implementation of some of these criteria is shown by example.

Coverage Criteria Name	**Coverage Criteria Category**	**Description**	**TTS artifacts**
All-Requirements-Coverage	Structural:Requirements	Each requirement is tested by at least one test story	FR, TE
Requirements-Hierarchy-Coverage	Structural : Requirements	A requirement and all its subrequirements are tested	FR*, TE
All-Services-All-Interfaces-Coverage	Structural : Service	From all services all interfaces are invoked at least in one test story	S, I, O, SC
All-Services-Coverage	Structural : Service	From each service an arbitrary operation is called	S,I, O, SC
Service-Hierarchy-Coverage	Structural : Service	From each service in a hierarchy an arbitrary operation is called	S*, I, O, SC
Full-Services-Coverage	Structural : Service	Each operation of each service interface is invoked at least once in a test story	I, O, SC
All-Services-All-LocalProcesses-Path-Coverage	Structural : Service : LocalProcess	Each path in each local process of each service is executed	Finding path in activity or state machine of SM and corresponding path in TM
All-Services-LocalProcess-Path-Coverage	Structural : Service : LocalProcess	Each path in one local process of each service is executed	
All-Services-All-LocalProcesses-DC-Coverage	Structural : Service : LocalProcess	Each reachable decision/branch in all local processes of all services is tested with true and false	
All-Services-All-LocalProcesses-PC-Coverage	Structural : Service : LocalProcess	Each satisfiable path in each local process of all services is tested	
All-Services-All-LocalProcesses-CC-Coverage	Structural : Service : LocalProcess	Each condition in each decision in each local process of all services is tested with true and false	
All-Services-All-LocalProcesses-MC/DC-Coverage	Structural : Service : LocalProcess	Each condition independently affecting the outcome of the decision in each local process of all services is tested to true and false	
All-Services-All-LocalProcesses-FPC-Coverage	Structural : Service : LocalProcess	Each condition directly correlated with the outcome of the decision for each local process of all services is tested to true and false	

Coverage Criteria Name	**Coverage Criteria Category**	**Description**	**TTS artifacts**
All-Services-All-LocalProcesses-Node-Coverage	Structural : Service : LocalProcess	Each node of all local processes of each service is called	
All-GlobalProcesses-Path-Coverage	Structural : Service : GlobalProcess	Each path of each global process is called	
ClassCoverage	Structural : Types	All types are referred in at least one test story	O, C, D
All-Operations-One-Value-Coverage	Data : Operation	For each input parameter of each operation one value of its domain is tested	
All-Preconditions-All-BoundaryValue-Coverage	Data : Precondition	For each precondition of an operation each boundary point that satisfies the precondition is tested	O, Pre
All-Preconditions-One-BoundaryValue-Coverage	Data : Precondition	For each precondition of an operation one boundary point that satisfies the precondition is tested	O, Pre
All-Operations-All-Values-Coverage	Data : Operation	All operations are tested with all values of their finite domain (e.g., a partition type)	
All-Operations-Pairwise-Coverage	Data : Operation : Combinatorial	Each possible combination of all values of pairs of parameters is tested for all operations	
All-Operations-N-Wise-Coverage	Data : Operation : Combinatorial	Each possible combiniation of N-triples of N parameters is tested for all operations	
All-Operations-All-Combinations-Coverage	Data : Operation : Combinatorial	Each possible combination of all parameters of all operations is tested	

Table C.1: Coverage Criteria in Telling TestStories

Appendix D

Testcode Generation

Test code is generated by a model–to–text transformation implemented in the template–based transformation tool XPand of the openArchitectureWare tool set [oaw]. In Listing D.1 the code generation templates for test stories and in Listing D.2 the code generation templates for test sequences are printed.

```
1  IMPORT org::openarchitectureware::meta::uml
2  IMPORT info::teststories::generator::profile::tts
3
4  EXTENSION info::teststories::generator::xpand::Teststory
5  EXTENSION info::teststories::generator::util::GeneratorUtils
6
7
8  DEFINE Teststory FOR uml::Model
9      resetIdCounter()
10     EXPAND Element FOREACH (List[uml::Package])ownedElement
11 ENDDEFINE
12
13
14 DEFINE Element FOR uml::Package
15         IF this.getAppliedStereotype("TTS::Test") != null
16                 FOREACH (List[uml::Package])ownedElement AS teststories
17                         IF teststories.getAppliedStereotype("TTS::Teststories") != null
18                                 EXPAND Teststories FOR teststories
19                         ENDIF
20                 ENDFOREACH
21         ENDIF
22 ENDDEFINE
23
24
25 DEFINE Teststories FOR uml::Package
26         FOREACH (List[uml::Package])ownedElement AS teststory
27                 IF teststory.getAppliedStereotype("TTS::Teststory") != null
28                         EXPAND Teststory FOR teststory
29                 ENDIF
30         ENDFOREACH
31 ENDDEFINE
32
33
34 DEFINE Teststory FOR uml::Package
35         FOREACH (List[uml::Activity])ownedElement AS teststory
36                 IF teststory.getAppliedStereotype("TTS::Teststory") != null
37                         EXPAND GenerateStory FOR teststory
38                 ENDIF
39         ENDFOREACH
40 ENDDEFINE
41
42
43 REM generates the concrete story file ENDREM
44 DEFINE GenerateStory FOR uml::Activity
45     FILE this.name + "Story.java"
46         package info.teststories.sut.stories;
47
48                 import info.teststories.adapter.rmi.controller.IAdapterRmiController;
49                 import info.teststories.adapter.controller.ISystemUnderTest;
50                 import info.teststories.evaluation.evaluator.EvaluatorFactory;
51                 import info.teststories.evaluation.evaluator.IEvaluator;
```

```
52                  import info.teststories.Verdict;
53                  import info.teststories.execution.annotation.Story;
54                  import info.teststories.execution.commctrl.adapter.AdapterRegistry;
55                  import info.teststories.execution.enactment.util.EnactmentRuntimeProperties;
56                  import info.teststories.execution.enactment.util.IProjectRuntimeProperties;
57                  import info.teststories.story.AbstractStory;
58
59                  import java.rmi.registry.LocateRegistry;
60                  import java.util.*;
61
62          @Story(name = " this.nameStory", version = "1", ID = getIdCounter().size )
63          public class  this.nameStory extends AbstractStory {
64
65              EXPAND CreateAdapters FOR this
66
67              protected IEvaluator evaluator = EvaluatorFactory.createEvaluator();
68
69              protected Verdict verdict;
70
71              public  this.nameStory()
72              {
73                  // nothing to do here
74              }
75
76              @Override
77              public void tell()
78              {
79                  context.clearMemory();
80                  context.getTestDataLine();
81                  sequence1();
82              }
83
84              public void sequence1()
85              {
86                  try {
87                      EXPAND ProcessStoryActivity FOR this
88                  } catch (Exception e)
89                  {
90                      throw new RuntimeException(e);
91                  }
92              }
93          }
94      ENDFILE
95      incrementIdCounter()
96      resetAdapters()
97  ENDDEFINE
98
99
100 DEFINE Parameter FOR String
101         context.retrieveValue(" this ")
102 ENDDEFINE
103
104
105 REM Creates all adapters for the story ENDREM
106 DEFINE CreateAdapters FOR uml::Activity
107     FOREACH this.ownedElement.typeSelect(uml::ActivityPartition) AS adapter
108         LET adapter.name AS name
109                 LET ((uml::Component)adapter.represents).name AS type
110                         putAdapterName(name, type)
111                          protected ISystemUnderTest  name  = AdapterRegistry.getSUT(" name ");
112                 ENDLET
113         ENDLET
114     ENDFOREACH
115 ENDDEFINE
116
117
118 REM Triggers processing of the activity in a visitor based manner ENDREM
119 DEFINE ProcessStoryActivity FOR uml::Activity
120     LET getInitialNode(this) AS initNode
121          IF isFinalNode(getNextNodeInFlow(initNode)) != true
122              EXPAND ProcessNode FOR getNextNodeInFlow(initNode)
123          ENDIF
124     ENDLET
125 ENDDEFINE
126
127
128 REM Processes a node for its concrete type ENDREM
129 DEFINE ProcessNode FOR uml::ActivityNode
130     REM TTS::Servicecall ENDREM
131     IF this.getAppliedStereotype("TTS::Servicecall") != null
132         EXPAND CreateCall FOR (uml::CallOperationAction)this
133          IF isFinalNode(getNextNodeInFlow(this)) != true
134              EXPAND ProcessNode FOR getNextNodeInFlow(this)
135          ENDIF
136     REM TTS::AsynchronousServiceCall ENDREM
137     ELSEIF this.getAppliedStereotype("TTS::Trigger") != null
138         EXPAND CreateTrigger FOR (uml::CallOperationAction)this
139          IF isFinalNode(getNextNodeInFlow(this)) != true
```

```
140             EXPAND ProcessNode FOR getNextNodeInFlow(this)
141         ENDIF
142     REM TTS::Assertion ENDREM
143     ELSEIF this.getAppliedStereotype("TTS::Assertion") != null
144         EXPAND CreateAssertion FOR (uml::OpaqueAction)this
145         IF isFinalNode(getNextNodeInFlow(this)) != true
146             EXPAND ProcessNode FOR getNextNodeInFlow(this)
147         ENDIF
148     REM TTS::ParallelTask ENDREM
149     ELSEIF this.getAppliedStereotype("TTS::ParallelTask") != null
150         parallel task to be generated
151         IF isFinalNode(getNextNodeInFlow(this)) != true
152             EXPAND ProcessNode FOR getNextNodeInFlow(this)
153         ENDIF
154     REM uml::DecisionNode ENDREM
155     ELSEIF this.metaType == uml::DecisionNode
156         EXPAND Decide FOR (uml::DecisionNode)this
157         IF isFinalNode(getNextNodeInFlow(this)) != true
158             EXPAND ProcessNode FOR getNextNodeInFlow(getMergeNode(((uml::DecisionNode)this)))
159         ENDIF
160     ENDIF
161 ENDDEFINE
162
163
164 REM Create a call ENDREM
165 DEFINE CreateCall FOR uml::CallOperationAction
166     LET this.inPartition.get(0).name AS adapter
167         LET this.getAppliedStereotype("TTS::Servicecall") AS st
168                 LET getValue(st, "output") AS output
169                 IF this.operation.getParameterList().size > 0
170                         IF output.toString() == null
171                                 adapter.invoke(" this.operation.name", EXPAND Parameter
                                        FOREACH standardizeParameters(getParameterList(this.
                                        operation), this) SEPARATOR "," );
172                         ELSEIF output.toString() == ""
173                                 adapter.invoke(" this.operation.name", EXPAND Parameter
                                        FOREACH standardizeParameters(getParameterList(this.
                                        operation), this) SEPARATOR "," );
174                         ELSE
175                                 context.storeValue(" output", adapter.invoke(" this.operation
                                        .name", EXPAND Parameter FOREACH standardizeParameters(
                                        getParameterList(this.operation), this) SEPARATOR "," ));
176                         ENDIF
177                 ELSE
178                         IF output.toString() == null
179                                 adapter.invoke(" this.operation.name", new Object[0]);
180                         ELSEIF output.toString() == ""
181                                 adapter.invoke(" this.operation.name", new Object[0]);
182                         ELSE
183                                 context.storeValue(" output", adapter.invoke(" this.operation
                                        .name", new Object[0]));
184                         ENDIF
185                 ENDIF
186         REM add the AdapterName to the list of SUTs to register ENDREM
187         addSUTAdapterName(adapter)
188         ENDLET
189     ENDLET
190     ENDLET
191 ENDDEFINE
192
193
194 REM Create a trigger ENDREM
195 DEFINE CreateTrigger FOR uml::CallOperationAction
196  LET getRMIHost()  AS hostname
197   LET getRMIPort() AS port
198          LET getRMIControllerName() AS controllerName
199                 LET  this.getAppliedStereotype("TTS::Trigger") AS st
200                  LET getValue(st, "callId").toString().replaceAll(" ","") AS triggers
201                         LET getValue(st, "timeout").toString().replaceAll(" ","") AS timeout
202                          LET triggers.subString(1, triggers.length-1) AS list
203                                 IF list.contains(",")
204                                  LET list.replaceAll(",","\",\"") AS indexedList
205                                         ((IAdapterRmiController) LocateRegistry.getRegistry("
                                                hostname", port).lookup(
206                                         "controllerName")).interrupt(timeout, new String[]{"
                                                indexedList"});
207                                  ENDLET
208                                 ELSE
209                                  ((IAdapterRmiController) LocateRegistry.getRegistry("hostname
                                        ", port).lookup(
210                                  "controllerName")).interrupt(timeout, new String[]{" list
                                        "});
211                                 ENDIF
212                          ENDLET
213                   ENDLET
214            ENDLET
215     ENDLET
216    ENDLET
```

```
217         ENDLET
218   ENDLET
219   ENDDEFINE
220
221
222   REM Create a conditional code block ENDREM
223   DEFINE Decide FOR uml::DecisionNode
224       REM Clear the branches counter ENDREM
225       resetBranchesCounter()
226       REM Iterate over every branch of the block ENDREM
227       FOREACH getOutgoings(this) AS branch
228           REM increment the counter ENDREM
229           incrementBranchesCounter()
230           LET ((uml::ActivityEdge)branch).getLabel() AS label
231               IF label.toString() == null && getBranchesCounter().size > 1
232                   else {
233                       EXPAND ProcessNode FOR ((uml::ActivityEdge)branch).target
234                   }
235               ELSEIF getBranchesCounter().size == 1
236                   if (formatCondition(label.toString()) ) {
237                       EXPAND ProcessNode FOR ((uml::ActivityEdge)branch).target
238                   }
239                ELSE
240                   else if (formatCondition(label.toString()) ) {
241                       EXPAND ProcessNode FOR ((uml::ActivityEdge)branch).target
242                   }
243                ENDIF
244           ENDLET
245       ENDFOREACH
246   ENDDEFINE
247
248
249   REM Create an assertion block ENDREM
250   DEFINE CreateAssertion FOR uml::OpaqueAction
251       LET this.getAppliedStereotype("TTS::Assertion") AS st
252           LET getValue(st, "pass") AS pass
253               LET getValue(st, "fail") AS fail
254                   verdict = evaluator.evaluateAssertion(nullChecker((String)pass, this) ,
                            nullChecker((String)fail, this) , context.getData());
255                   LOGGER.log(info.teststories.Verdict.fromValue(verdict.name()),
                            EnactmentRuntimeProperties.getInstance().getProperty(
                            IProjectRuntimeProperties.TTS_RUNTIME_TABLE_CURRENT_ROW), "");
256               ENDLET
257           ENDLET
258       ENDLET
259   ENDDEFINE
260
261
262   REM Create a parallel task block ENDREM
263   DEFINE CreateParallelTask FOR uml::ActivityNode
264
265   ENDDEFINE
266
267
268   DEFINE Package FOR uml::Activity
269       name
270   ENDDEFINE
271
272
273   DEFINE Package FOR uml::Element
274   ENDDEFINE
275
276
277   DEFINE Element FOR uml::Element
278   ENDDEFINE
```

Listing D.1: Test Code Generation for a Test Story

```
1   IMPORT org::openarchitectureware::meta::uml
2   IMPORT info::teststories::generator::profile::tts
3
4
5   DEFINE Workflow FOR uml::Model
6           EXPAND Element FOREACH (List[uml::Package])ownedElement
7   ENDDEFINE
8
9
10  DEFINE Element FOR uml::Package
11          IF this.getAppliedStereotype("TTS::Test") != null
12                  FOREACH (List[uml::Package])ownedElement AS testsequences
13                          IF testsequences.getAppliedStereotype("TTS::Testsequences") != null
```

```
14                                  EXPAND Testsequences FOR testsequences
15                              ENDIF
16                      ENDFOREACH
17              ENDIF
18      ENDDEFINE
19
20
21      DEFINE Testsequences FOR uml::Package
22              FOREACH (List[uml::Package])ownedElement AS testsequence
23                      IF testsequence.getAppliedStereotype("TTS::Testsequence") != null
24                              EXPAND Testsequence FOR testsequence
25                      ENDIF
26              ENDFOREACH
27      ENDDEFINE
28
29
30      DEFINE Testsequence FOR uml::Package
31              FOREACH (List[uml::Activity])ownedElement AS testsequence
32                      IF testsequence.getAppliedStereotype("TTS::Testsequence") != null
33                              EXPAND Testsequence1 FOR testsequence
34                      ENDIF
35              ENDFOREACH
36      ENDDEFINE
37
38
39      DEFINE Testsequence1 FOR uml::Activity
40              FILE this.name + ".wfl"
41                      FOREACH (List[uml::CallBehaviorAction])ownedElement AS sequenceelement
42                              IF sequenceelement.getAppliedStereotype("TTS::SequenceElement") !=
                                        null
43                                      EXPAND CreateSequenceElement FOR sequenceelement
44                              ENDIF
45                      ENDFOREACH
46              ENDFILE
47      ENDDEFINE
48
49
50      DEFINE CreateSequenceElement FOR uml::CallBehaviorAction
51              LET this.getAppliedStereotype("TTS::SequenceElement") AS sequence
52                      LET ((TTS::Teststory)getValue(sequence, "story")).base_Activity.name AS story
53                              LET getValue(sequence, "data") AS data
54                                      LET getValue(sequence, "arbitration") AS arbitration
55      story Story: data : arbitration
56                                      ENDLET
57                      ENDLET
58                      ENDLET
59              ENDLET
60      ENDDEFINE
61
62
63      DEFINE Package FOR uml::Element
64      ENDDEFINE
65
66
67      DEFINE Package FOR uml::Activity
68              name
69      ENDDEFINE
70
71
72      DEFINE Element FOR uml::Element
73      ENDDEFINE
```

Listing D.2: Test Code Generation for a Test Sequence

Appendix E

Evolution State Machines

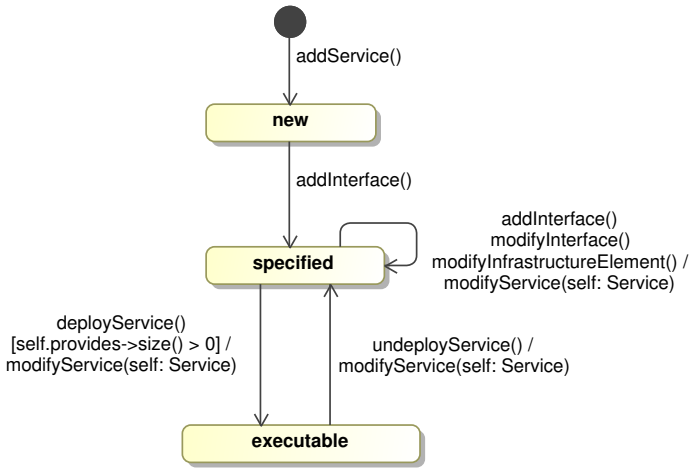


Figure E.1: State Machine describing the Lifecycle of a Service

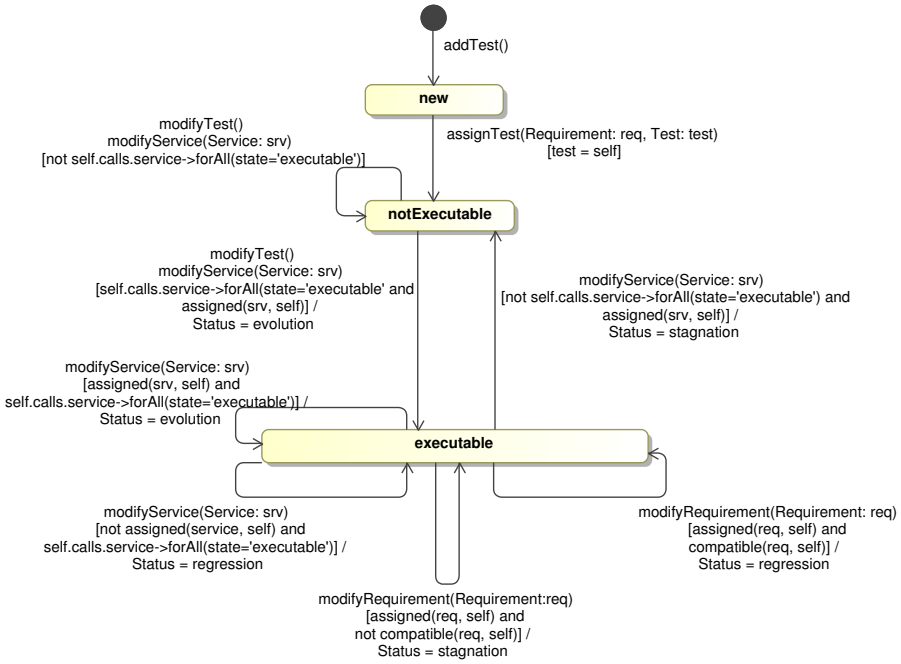


Figure E.2: State Machine describing the Lifecycle of a Test

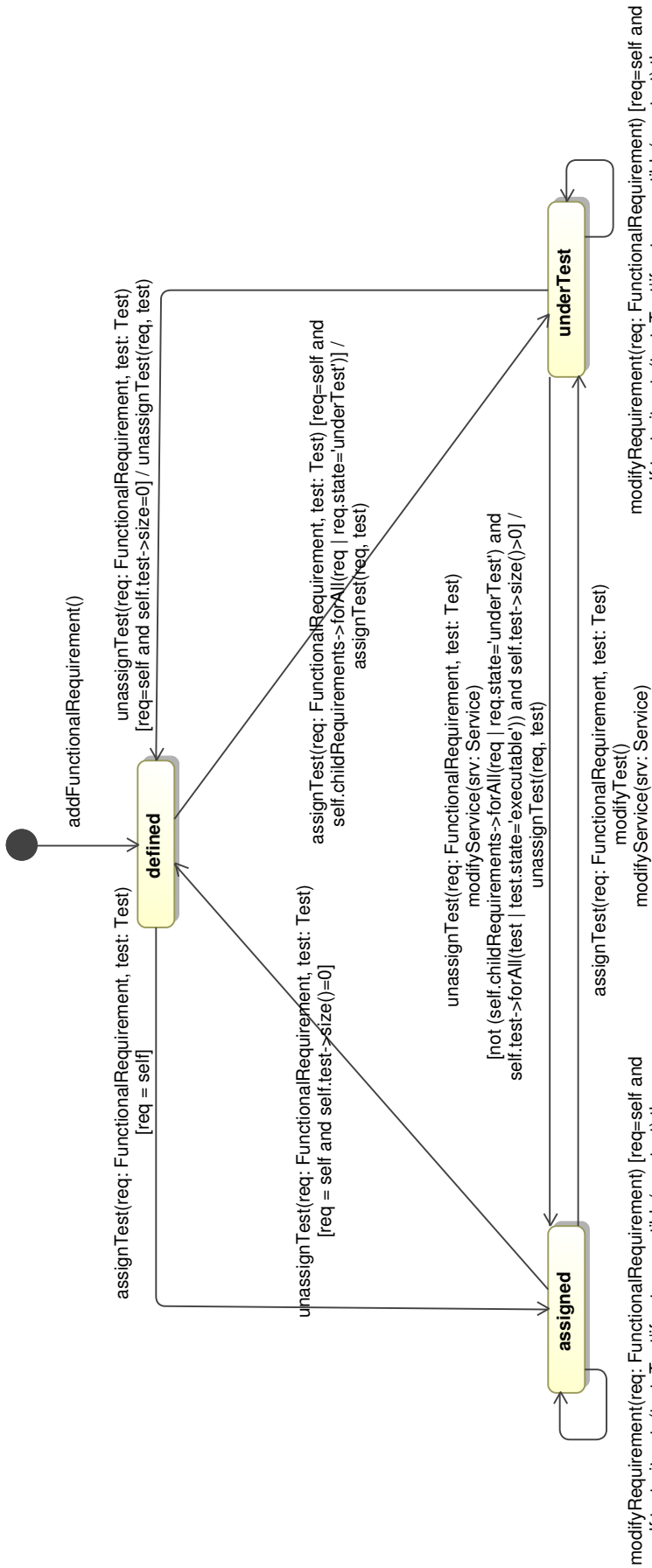


Figure E.3: State Machine describing the Lifecycle of a FunctionalRequirement

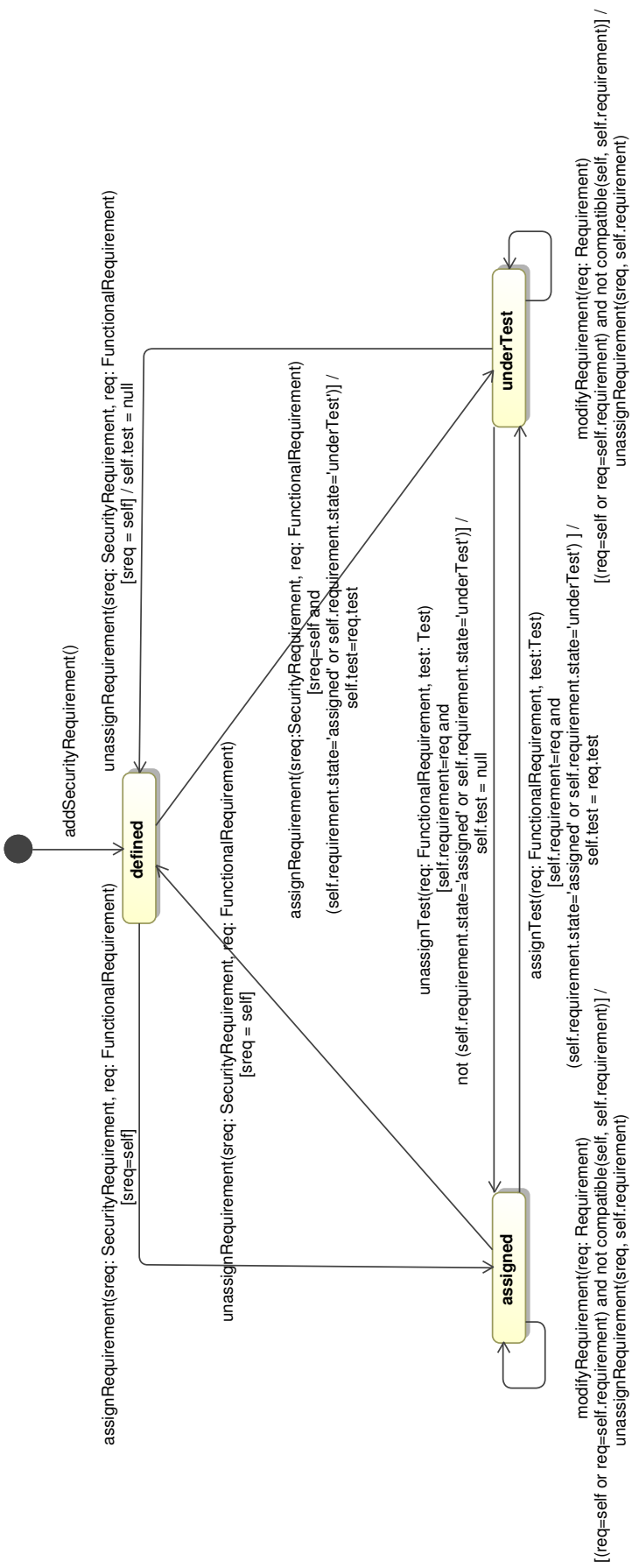


Figure E.4: State Machine describing the Lifecycle of a SecurityRequirement

Appendix F

Assertion and Arbitration Grammar

F.1 Assertion Grammar

```
1  grammar Assertion;
2
3  @header {
4  import java.lang.reflect.InvocationTargetException;
5  import java.lang.reflect.Method;
6  import java.lang.reflect.Field;
7  import java.lang.Comparable;
8  import java.lang.Number;
9  import java.text.SimpleDateFormat;
10 import java.util.Date;
11 import java.io.PrintWriter;
12 import java.io.StringWriter;
13 import org.apache.log4j.Logger;
14 }
15
16 @members {
17 private static Logger logger = Logger.getLogger(AssertionParser.class);
18
19 private IParserMgr<AssertionParser> pM;
20
21         public void setParserMgr(IParserMgr<AssertionParser> parserMgr) {
22                 this.pM = parserMgr;
23         }
24
25         @Override
26         public void reportError(RecognitionException e) {
27                 super.reportError(e);
28                 if(e instanceof NoViableAltException) {
29                         throw new RuntimeException("No alternative found at line: "+e.line+"
                                position: "+e.charPositionInLine+" token: "+e.token.getText()+"...")
                                ;
30                 }
31                 StringWriter stackTrace = new StringWriter();
32                 e.printStackTrace(new PrintWriter(stackTrace));
33                 throw new RuntimeException("Parser error...\n"+stackTrace.toString());
34         }
35
36 }
37
38 //Entry rule for assertion grammar
39 assertion returns [Object value]
40         : e=expr {$value=$e.value;} EOF;
41
42 //Lowest precedence
43 expr returns [Object value]
44         : e=expr_1 {$value=$e.value;}
45         ( AND   e=expr_1 {$value=((Boolean)$value)&&((Boolean)$e.value);}
46         | OR    e=expr_1 {$value=((Boolean)$value)||((Boolean)$e.value);})*;
47
48 //middle precedence
49 expr_1 returns [Object value]
```

```
50          : e=expr_2 {$value=$e.value;}
51          ( EQ    e=expr_2 {$value=pM.equals($value,$e.value);}
52          | UEQ   e=expr_2 {$value=!pM.equals($value,$e.value);})*;
53          catch[Exception exc] {throw new RuntimeException(exc);}
54
55  //highest precedence
56  expr_2 returns [Object value]
57          : (e=atom {$value=$e.value;}
58          | NOT   e=atom {$value=!(Boolean)$e.value;})
59          ( GTEQ  e=atom {$value=(pM.compareTo($value,$e.value)>=0)?true:false;}
60          | LTEQ  e=atom {$value=(pM.compareTo($value,$e.value)<=0)?true:false;}
61          | LT    e=atom {$value=(pM.compareTo($value,$e.value)<0)?true:false;}
62          | GT    e=atom {$value=(pM.compareTo($value,$e.value)>0)?true:false;})?;
63
64  //Parse a loop
65  atom returns [Object value]
66          : e=base_atom {$value = $e.value;}
67          (MEMBER_SEPARATOR (f=method_call[$value, new ArrayList<Object>()] {$value=$f.value;}
68          | f=attribute_call[$value] {$value=$f.value;})
69          | LOOP_SEPARATOR e=parse_loop[$value] {$value=$e.value;})*;
70
71  //Basic datatype with method invocation
72  base_atom returns [Object value]
73          : e=parse_item {$value=$e.value;}
74          | f=parse_double {$value=$f.value;}
75          | g=parse_string {$value=$g.value;}
76          | h=parse_int {$value=$h.value;}
77          | i=parse_boolean {$value=$i.value;}
78          | j=parse_char {$value=$j.value;}
79          | k=parse_date {$value=$k.value;}
80          | BRACKET_OPEN e=expr BRACKET_CLOSE {$value=$e.value;};
81
82  //Basic atom, basic datatype
83  parse_loop [Object obj] returns [Object value]
84          : e=FOREACH {$value=((List<?>)obj).size()==pM.evaluateLoop((List<?>)obj,$e.text.
                substring(7));}
85          | e=EXISTS {$value=pM.evaluateLoop((List<?>)obj,$e.text.substring(6))>0;}
86          | e=COUNT {$value=pM.evaluateLoop((List<?>)obj,$e.text.substring(5));};
87
88  //Invoke a method on an Object and collect all parameters.
89  method_call [Object obj,List<Object> params] returns [Object value]
90          : e=parse_method_parameters[$params] {$value=pM.invokeMethod($obj,(String)$e.value,
                $params);};
91          catch [InvocationTargetException exc] {throw new RuntimeException(exc.getCause());}
92          catch [Exception exc] {throw new RuntimeException(exc);}
93
94  //Call an attribute on an item.
95  attribute_call [Object obj] returns [Object value]
96          : e=ITEM{$value=pM.invokeAttribute($obj,$e.text);};
97          catch [Exception exc] {throw new RuntimeException(exc);}
98
99
100 //Add all parameters for an method invokation to the parameters list.
101 parse_method_parameters [List<Object> parms] returns [String value]
102         : e=ITEM {$value = $e.text;}
103         BRACKET_OPEN
104         (f=expr {parms.add($f.value);} (PARAMETER_SEPARATOR f=expr {parms.add($f.value);})*)?
105         BRACKET_CLOSE;
106
107 //Get an item from the memory
108 parse_item returns [Object value]
109         : e=ITEM {$value=pM.getValue($e.text);}
110         | e=DOLLAR_ITEM  {$value=pM.getValue($e.text);};
111
112 //Parse a double value
113 parse_double returns [Double value]
114         : e=DOUBLE {$value=Double.valueOf($e.text);};
115
116 //Parse a int value
117 parse_int returns [Integer value]
118         : e=INT {$value=Integer.valueOf($e.text);};
119
120 //Match and parse a string and remove
121 parse_string returns [String value]
122         : e=STRING {$value=$e.text.substring(1,$e.text.length()-1);};
123
124 //Parse a boolean value
125 parse_boolean returns [Boolean value]
126         : e=BOOLEAN {$value=Boolean.valueOf($e.text);};
127
128 //Parse a char value
129 parse_char returns [Character value]
130         : e=CHAR {$value=Character.valueOf($e.text.toCharArray()[1]);};
131
132 //Parse date
133 parse_date returns [Date value]
134         : e=DATE {$value=pM.parseDate($e.text.substring(1,$e.text.length()-1));};
135         catch[Exception exc] {throw new RuntimeException(exc);}
```

```
136
137  //Lexer rules
138  INT                 :       PRAEFIX?DIGIT+;
139  DOUBLE              :       PRAEFIX?DIGIT+'.' DIGIT+;
140  DATE                :       '\''(('0'..'9')('0'..'9')':'('0'..'9')('0'..'9')' ')?␣('0'..'9')('0'..'9
         ')(('.'('0'..'9')('0'..'9')'.')|(' 'MON' '))('0'..'9')('0'..'9')'\'';
141  BOOLEAN             :       'TRUE'|'FALSE'|'true'|'false';
142  //Generated by info.teststories.unicodeTool
143  CHAR                :       '\''(('\u0000'..'\u0026')|('\u002a'..'\u002a')|('\u002c'..'\u002c')|('\
         u002f'..'\u002f')|('\u003a'..'\u007b')|('\u007d'..'\ufffd'))'\'';
144  EQ                  :       '==';
145  GT                  :       '>';
146  GTEQ                :       '>=';
147  LT                  :       '<';
148  LTEQ                :       '<=';
149  UEQ                 :       '!=';
150  AND                 :       '&&';
151  OR                  :       '||';
152  NOT                 :       '!';
153  DOLLAR_ITEM         :       DOLLAR (~DOLLAR+) DOLLAR { setText($text.substring(1,$text.length()-1))
         ;};
154  ITEM                :       IDENTIFIER;
155  STRING              :       '"'.*'"';
156  FOREACH             :       'foreach{'IDENTIFIER'|'~('{'|'}')+'}';
157  EXISTS              :       'exists{'IDENTIFIER'|'~('{'|'}')+'}';
158  COUNT               :       'count{'IDENTIFIER'|'~('{'|'}')+'}';
159  BRACKET_OPEN        :       '(';
160  BRACKET_CLOSE       :       ')';
161  MEMBER_SEPARATOR:           '.';
162  LOOP_SEPARATOR      :       '->';
163  PARAMETER_SEPARATOR
164                      :       ',';
165
166  //Fragments
167  fragment IDENTIFIER
168                      : ((LETTER DIGIT*)'_'*)+;
169  fragment DOLLAR     :       '$';
170  fragment MON        :       'Jan'|'Feb'|'Mar'|'Apr'|'May'|'Jun'|'Jul'|'Aug'|'Sep'|'Okt'|'Nov'|'Dez';
171  fragment PRAEFIX:           ('+'|'-');
172  fragment DIGIT      :       ('0'..'9');
173  fragment LETTER     :       ('a'..'z'|'A'..'Z');
174  NEWLINE             :       '\r'? '\n';
175  WS                  :       ('␣'|'\t')+ {skip();};
176  ERROR               :       .;
```

Listing F.1: Assertion Grammar

F.2 Arbitration Grammar

```
1   grammar Arbitration;
2
3   @header {
4   import org.apache.log4j.Logger;
5   import info.teststories.evaluation.evaluator.TestCollection;
6   import info.teststories.evaluation.evaluator.Verdict;
7   import info.teststories.evaluation.evaluator.parsers.IParserMgr;
8   import java.io.PrintWriter;
9   import java.io.StringWriter;
10  }
11
12  @members {
13  private static Logger logger = Logger.getLogger(ArbitrationParser.class);
14
15  private IParserMgr parserMgr;
16  private TestCollection testCollection;
17
18          public void setParserMgr(IParserMgr parserMgr) {
19                  this.parserMgr = parserMgr;
20          }
21
22          //Return test collection
23          public TestCollection getTC() {
24                  return (TestCollection)this.parserMgr.getValue("TestCollection");
25          }
26
27          @Override
28          public void reportError(RecognitionException e) {
```

```
29                  if(e instanceof NoViableAltException) {
30                          throw new RuntimeException("No alternative found at line: "+e.line+"
                                position: "+e.charPositionInLine+" token: "+e.token.getText()+"...")
                                ;
31                  }
32                  StringWriter stackTrace = new StringWriter();
33                  e.printStackTrace(new PrintWriter(stackTrace));
34                  throw new RuntimeException("Parser error...\n"+stackTrace.toString());
35      }
36  }
37
38  //Grammar entry rule
39  arbitration returns [boolean value]
40          : e=expr {$value=$e.value;} EOF
41          ;
42
43  expr returns [boolean value]
44          : e=basic_expr {$value=$e.value;}
45          (OR     e=basic_expr {$value=((Boolean)$value)||((Boolean)$e.value);}
46          | AND   e=basic_expr {$value=((Boolean)$value)&&((Boolean)$e.value);})*;
47
48  basic_expr returns [boolean value]
49          : e=verdict {$value=$e.value;}
50          | e=time {$value=$e.value;}
51          | BRACKET_OPEN e=expr BRACKET_CLOSE {$value=$e.value;}
52          | NOT e=basic_expr {$value=!(Boolean)$e.value;};
53
54  verdict returns [boolean value]
55          : e=pass {$value=$e.value;}
56          | e=inconc {$value=$e.value;}
57          | e=fail {$value=$e.value;};
58
59  pass returns [boolean value]
60          : PASS
61          (PROC e=eval[getTC().getPercent(Verdict.PASS)] {$value=$e.value;}
62          | COUNT e=eval[getTC().getCount(Verdict.PASS)] {$value=$e.value;});
63
64  inconc returns [boolean value]
65          : INCONC
66          (PROC e=eval[getTC().getPercent(Verdict.INCONCLUSIVE)] {$value=$e.value;}
67          | COUNT e=eval[getTC().getCount(Verdict.INCONCLUSIVE)] {$value=$e.value;});
68
69  fail returns [boolean value]
70          : FAIL
71          (PROC e=eval[getTC().getPercent(Verdict.FAIL)] {$value=$e.value;}
72          | COUNT e=eval[getTC().getCount(Verdict.FAIL)] {$value=$e.value;});
73
74
75  eval [Object obj] returns [boolean value]
76          : EQ    e=atom {$value=this.parserMgr.equals($obj,$e.value);}
77          | UEQ   e=atom {$value=!this.parserMgr.equals($obj,$e.value);}
78          | GTEQ  e=atom {$value=(this.parserMgr.compareTo($obj,$e.value)>=0)?true:false;}
79          | LTEQ  e=atom {$value=(this.parserMgr.compareTo($obj,$e.value)<=0)?true:false;}
80          | LT    e=atom {$value=(this.parserMgr.compareTo($obj,$e.value)<0)?true:false;}
81          | GT    e=atom {$value=(this.parserMgr.compareTo($obj,$e.value)>0)?true:false;};
82
83  time returns [boolean value]
84          : e=min_time {$value=$e.value;}
85          | e=max_time {$value=$e.value;}
86          | e=avg_time {$value=$e.value;};
87
88  min_time returns [boolean value]
89          : MIN_TIME
90          (PASS e=eval[getTC().getMinTime(Verdict.PASS)] {$value=$e.value;}
91          | FAIL e=eval[getTC().getMinTime(Verdict.FAIL)] {$value=$e.value;}
92          | INCONC e=eval[getTC().getMinTime(Verdict.INCONCLUSIVE)] {$value=$e.value;});
93
94  max_time returns [boolean value]
95          : MAX_TIME
96          (PASS e=eval[getTC().getMaxTime(Verdict.PASS)] {$value=$e.value;}
97          | FAIL e=eval[getTC().getMaxTime(Verdict.FAIL)] {$value=$e.value;}
98          | INCONC e=eval[getTC().getMaxTime(Verdict.INCONCLUSIVE)] {$value=$e.value;});
99
100 avg_time returns [boolean value]
101         : AVG_TIME
102         (PASS e=eval[getTC().getAvgTime(Verdict.PASS)] {$value=$e.value;}
103         | FAIL e=eval[getTC().getAvgTime(Verdict.FAIL)] {$value=$e.value;}
104         | INCONC e=eval[getTC().getAvgTime(Verdict.INCONCLUSIVE)] {$value=$e.value;});
105
106 atom returns [Object value]
107         : DOUBLE PROC?  {$value=Double.valueOf($DOUBLE.text);}
108         | INT PROC?     {$value=Integer.valueOf($INT.text);};
109
110 //Lexer rules
111 PASS            :       'PASS'|'pass';
112 FAIL            :       'FAIL'|'fail';
113 INCONC          :       'INCONC'|'inconc';
114 PROC            :       '%';
```

```
115  COUNT              :       'COUNT'|'count';
116  MIN_TIME           :       ('MIN'|'min')' 'TIME;
117  MAX_TIME           :       ('MAX'|'max')' 'TIME;
118  AVG_TIME           :       ('AVG'|'avg')' 'TIME;
119  BRACKET_OPEN       :       '(';
120  BRACKET_CLOSE      :       ')';
121  EQ                 :       '=='|'=';
122  GT                 :       '>';
123  GTEQ               :       '>=';
124  LT                 :       '<';
125  LTEQ               :       '<=';
126  UEQ                :       '!=';
127  AND                :       'AND'|'and';
128  OR                 :       'OR'|'or';
129  NOT                :       'NOT'|'not';
130  DOUBLE             :       DIGIT+'.'DIGIT*;
131  INT                :       DIGIT+;
132  NEWLINE            :       '\r'? '\n';
133  WS                 :       (' '|'\t')+ {skip();};
134  ERROR              :       .;
135
136  //Fragments
137  fragment TIME      :       'TIME'|'time';
138  fragment PRAEFIX:          ('+'|'-')?;
139  fragment DIGIT     :       ('0'..'9');
```

Listing F.2: Arbitration Grammar