

2016

Framework for Automated Functional Tests within Value-Added Service Environments

Wacht, Patrick

<http://hdl.handle.net/10026.1/5335>

<http://dx.doi.org/10.24382/4717>

Plymouth University

All content in PEARL is protected by copyright law. Author manuscripts are made available in accordance with publisher policies. Please cite only the published version using the details provided on the item record or document. In the absence of an open licence (e.g. Creative Commons), permissions for further reuse of content should be sought from the publisher or author.

Copyright Statement

This copy of the thesis has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the thesis and no information derived from it may be published without the author's prior consent.

Framework for Automated Functional Tests within Value-Added Service Environments

by

Patrick Sebastian Wacht

A thesis submitted to Plymouth University
in partial fulfilment for the degree of

DOCTOR OF PHILOSOPHY

School of Computing and Mathematics

In collaboration with
Darmstadt Node of the Centre for Security,
Communications and Network Research (CSCAN)

December 2015

Acknowledgements

In the first place I wish to express my sincere thanks to my supervisors Prof. Woldemar Fuhrmann and Dr. Bogdan Ghita for their positive and comprehensive support and guidance throughout this research.

I would like to further express my special appreciation to my supervisor Prof. Ulrich Trick who has always been a tremendous mentor for me throughout my time as a member of the Research Group for Telecommunication Networks at the University of Applied Sciences Frankfurt. His guidance helped me a lot during the research and writing of this thesis. Without his encouragement and support, this research probably would have never been performed.

I would also like to thank all current and former members of the Research Group for Telecommunication Networks for their friendship, support and the great inspiration that I could experience during the last years.

Warm thanks go to the members of both the graduate school and the CSCAN Network at Plymouth University, and special thanks also go to the members of the CSCAN Darmstadt node for their experienced support especially during PhD seminars.

I wish to thank my family and friends for their encouragement and support offered throughout the whole time of this research.

Above all, I owe thanks to my loving wife Kathleen for her great support, understanding and all of the sacrifices that she has made on my behalf.

Author's declaration

At no time during the registration for the degree of Doctor of Philosophy has the author been registered for any other University award without prior agreement of the Graduate Sub-Committee.

Work submitted for this research degree at the Plymouth University has not formed part of any other degree either at Plymouth University or at another establishment.

Relevant scientific seminars and conferences were regularly attended at which work was often presented, and several papers prepared for publication.

Publications:

- Wacht, P.; Lehmann, A.; Eichelmann, T.; Trick, U.; Fischer, M.; Lasch, R.; Toenjes, R. (2010), "Ein neues Verfahren zum automatisierten Testen von Mehrwertdiensten" (translated title: "A novel approach to automated testing of value-added services"), In *Proceedings of the 15th VDE/ITG Fachtagung Mobilkommunikation (Mobilfunktagung 2010)*, pp. 73-80, 2010.
- Wacht, P.; Lehmann, A.; Eichelmann, T.; Fuhrmann, W.; Trick, U.; Ghita, B. (2010), "Integration of Model-Based Functional Testing Procedures within a Creation Environment for Value-Added Services", In *Proceedings of the 6th Collaborative Research Symposium on Security, E-learning, Internet and Networking (SEIN 2010)*, pp. 61-74, 2010.
- Wacht, P.; Eichelmann, T.; Lehmann, A.; Trick, U. (2011), "A New Approach to Design Graphically Functional Tests for Communication Services", In *Proceedings of the 4th IFIP International Conference on New Technologies, Mobility and Security (NTMS 2011)*, IEEE, pp. 1-5, 2011.
- Wacht, P.; Lehmann, A.; Eichelmann, T.; Trick, U. (2011), "ComGeneration: die Dienstbeschreibung als Basis für automatisierte Tests" (translated title: "ComGeneration: a service description as basis for automated tests"), In *Proceedings of the 16th VDE/ITG Fachtagung Mobilkommunikation (Mobilfunktagung 2011)*, pp. 118-123, 2011.
- Wacht, P.; Eichelmann, T.; Lehmann, A.; Fuhrmann, W.; Trick, U.; Ghita, B. (2011), "A new Approach to model a formalised Description of a Communication Service for the Purpose of Functional Testing", In *Proceedings of the 4th International Conference on Internet Technologies & Applications (ITA 2011)*, pp. 262-269, 2011.
- Wacht, P.; Trick, U.; Fuhrmann, W.; Ghita, B. (2013), "A New Service Description for Communication Services as Basis for Automated Functional Testing", In *Proceedings of the Second International Conference on Future Generation Communication Technology (FGCT 2013)*, IEEE, pp. 59-64, 2013.

Presentations and Conferences attended:

- 15th ITG Fachtagung Mobilkommunikation (Mobilfunktagung 2010), Osnabrück, Germany, May 2010
- 2nd International NGN Workshop (ngnlab.eu 2010), Leipzig, Germany, November 2010
- 6th Collaborative Research Symposium on Security, E-learning, Internet and Networking (SEIN 2010), Plymouth, UK, November 2010
- 4th IFIP International Conference on New Technologies, Mobility and Security, Paris, France, February 2011
- 16th ITG Fachtagung Mobilkommunikation (Mobilfunktagung 2011), Osnabrück, Germany, May 2011
- 4th International Conference on Internet Technologies and Applications (ITA 2011), Wrexham, UK, September 2011
- 8th Collaborative Research Symposium on Security, E-learning, Internet and Networking (SEIN 2013), Darmstadt, Germany, November 2013
- 2nd International Conference on Future Generation Communication Technology (FGCT 2013), London, UK, November 2013

Word count of main body of thesis: 71.004

Signed

Date

Framework for Automated Functional Tests within Value-Added Service Environments

Patrick Sebastian Wacht

Abstract

Recent years have witnessed that standard telecommunication services evolved more and more to next generation value-added services. This fact is accompanied by a change of service characteristics as new services are designed to fulfil the customer's demands instead of just focussing on technologies and protocols. These demands can be very specific and, therefore, diverse potential service functionalities have to be considered by the service providers. To make matters worse for service providers, a fast transition from concept to market product and low price of a new service is required due to the increasing competition in the telecommunication industry. Therefore, effective test solutions need to be developed that can be integrated in current value-added service development life-cycles. Besides, these solutions should support the involvement of all participating stakeholders such as the service provider, the test developers as well as the service developers, and, in order to consider an agile approach, also the service customer.

This thesis proposes a novel framework for functional testing that is based on a new sort of description language for value-added services (Service Test Description). Based on instances of the Service Test Description, sets of reusable test components described by means of an applied Statecharts notation are automatically selected and composed to so-called behaviour models. From the behaviour models, abstract test cases can be automatically generated which are then transformed to TTCN-3 test cases and then assembled to an Executable Test Suite. Within a TTCN-3 test system, the Executable Test Suite can be executed against the corresponding value-added service referred to as System Under Test. One benefit of the proposed framework is its application within standard development life-cycles. Therefore, the thesis presents a methodology that considers both service development and test development as parallel tasks and foresees procedures to synchronise the tasks and to allow an agile approach with customer involvement.

The novel framework is validated through a proof-of-concept working prototype. Example value-added services have been chosen to illustrate the whole process from compiling instances of the Service Test Description until the execution of automated tests.

Overall, this thesis presents a novel solution for service providers to improve the quality of their provided value-added services through automated functional testing procedures. It enables the early involvement of the customers into the service development life-cycle and also helps test developers and service developers to collaborate.

Contents

Contents	vii
List of Figures.....	ix
List of Tables	xi
1 Introduction	1
1.1 Aims and Objectives.....	2
1.2 Thesis Structure	3
2 Telecommunications Infrastructure and Value-Added Services.....	5
2.1 NGN (Next Generation Networks).....	5
2.2 SIP and its Utilisation in NGN	10
2.2.1 SIP Architecture and Basic Functionality	11
2.2.2 The Layered Structure of SIP.....	14
2.2.3 SIP-based NGN	16
2.2.4 SIP Application Server.....	18
2.3 Value-Added Services in NGN.....	21
2.3.1 Classifications and Definitions.....	21
2.3.2 Development and Provisioning of Value-Added Services.....	25
2.4 Stakeholders in Value-Added Service Provisioning.....	29
2.5 Conclusion	33
3 The Challenge of Testing Value-Added Services.....	35
3.1 Principles of Functional Testing.....	35
3.1.1 Fundamentals of Testing and Test Processes	36
3.1.2 Schematic Approach to Functional Testing	40
3.1.3 Relevance for Testing of Value-Added Services	43
3.2 Related Work on Current Testing Methodologies.....	45
3.2.1 Test-Driven Development	46
3.2.2 Acceptance Test-Driven Development	49
3.2.3 Behaviour-Driven Development	51
3.2.4 Model-Based Testing	54
3.3 Related Work on Current Research Projects on Functional Testing.....	57
3.3.1 UML 2.0 Testing Profile	57
3.3.2 TT-Medal Test Platform.....	61

3.3.3	Fokus!MBT Test Modelling Environment.....	64
3.3.4	ComGeneration	68
3.3.5	Telling TestStories	73
3.4	Requirements for a New Optimised Solution for Functional Testing of Value-Added Services	77
3.5	Conclusion	83
4	Proposed Framework for Testing of Value-Added Services.....	85
4.1	Preconditions and Tasks of Roles.....	85
4.2	Overall Methodology for Testing Value-Added Services	87
4.3	Framework Architecture and Components.....	93
4.4	Conclusion	99
5	Novel Service Test Specification and Related Specifications	101
5.1	Existing Specification and Description Languages for Services in the Telecommunication Domain	102
5.1.1	Structured Use Case Models	103
5.1.2	Restricted Use Case Modeling (RUCM).....	110
5.1.3	Unified Test Modeling Language (UTML).....	118
5.1.4	Alternative Approaches.....	124
5.2	Proposed Novel Service Test Description	127
5.2.1	Structure of Service Test Description	129
5.2.2	Architectural Perspective	130
5.2.3	Behavioural Perspective	132
5.2.4	Sample Specification with Service Test Description	147
5.3	Comparison of Service and Test Specification Languages	151
5.4	Conclusion	154
6	Reusable Test Modules and Behaviour Model Generation.....	155
6.1	Notation for Behaviour Modelling	157
6.1.1	Evaluation of Potential Modelling Notations.....	157
6.1.2	Relevant Portions of the Selected Modelling Notation	162
6.1.3	Principles of Modelling Service Behaviour with Statecharts.....	169
6.2	Reusable Test Modules.....	175
6.2.1	Test Modules Environment Architecture	175
6.2.2	Identification of Reusability.....	178
6.2.3	Classification of Reusable Test Modules	180
6.2.4	Modelling of Reusable Test Modules	183
6.3	Test Data Integration	194
6.4	Generation of Behaviour Models.....	201
6.5	Conclusion	218

7 Test Case Generation, Execution and Management	221
7.1 Generation of Abstract Test Suite.....	223
7.1.1 From Behaviour Models to Abstract Test Cases.....	223
7.1.2 Test Case Derivation	226
7.2 Test Suite Generation.....	236
7.2.1 Motivation for a TTCN-3-based Approach.....	236
7.2.2 Test Code Generation and Test Suite Building	240
7.3 Test Case Execution.....	262
7.4 Test Evaluation and Management.....	267
7.5 Conclusion	271
8 Framework and Prototype Evaluation.....	275
8.1 Evaluation of the Defined Framework Requirements	275
8.2 Prototype Architecture and Implementation.....	279
8.3 Proof of Proposed Framework Concept.....	291
8.3.1 Description of Example Service Scenario.....	291
8.3.2 SUT Environment and Service Implementation	294
8.3.3 Specification of Chat Service with Service Test Description	297
8.3.4 Test Building and Test Execution	301
8.4 Conclusion	307
9 Conclusions	309
9.1 Achievements of the Research.....	309
9.2 Limitations of the Research.....	313
9.3 Suggestions and Scope for Future Work	314
References	317
Appendix A – Abbreviations.....	331
Appendix B – Own Publications	337

List of Figures

Figure 2.1: Principle structure of an NGN (Trick and Weber, 2009)	7
Figure 2.2: NGN architecture in a strata/layer structure (Trick and Weber, 2009)	9
Figure 2.3: Basic principle of SIP communication	11
Figure 2.4: Basic establishment of a SIP session.....	13
Figure 2.5: Layered structure of SIP (Ding and Liu, 2008).....	14
Figure 2.6: Principle structure of SIP-based NGN (Trick and Weber, 2009).....	17
Figure 2.7: Modes of operation of a SIP Application Server (Trick and Weber, 2009).	19
Figure 2.8: Categorisation of telecommunication services	23
Figure 2.9: Service life cycle (adapted from (OMA OSPE, 2005)).....	26
Figure 2.10: Service Delivery Platform in NGN (Trick and Weber, 2009).....	28
Figure 2.11: Roles in NGN management (adapted from (ITU-T M.3340, 2009))	30
Figure 3.1: Dynamic test processes (ISO/IEC/IEEE 29119-2, 2013).....	37
Figure 3.2: Schematic Approach to Functional Testing (adapted from (Pezzè and Young, 2009))	41
Figure 3.3: Comparison of TDD and Traditional Development (adapted from (Abrahamsson <i>et al.</i> , 2005)).....	47
Figure 3.4: Relevant activities in Acceptance Test-Driven Development life cycle	49
Figure 3.5: Conceptual Model of Behaviour-Driven Development (Solís and Wang, 2011)	52
Figure 3.6: Example BDD scenario description	53
Figure 3.7: Model-Based Test Development (adapted from (ETSI ES 202 951, 2011))	55
Figure 3.8: Example test case specification with U2TP using a UML sequence diagram (adapted from (OMG, 2013a))	60
Figure 3.9: TT-Medal test platform (TT-Medal Consortium, 2005)	62

Figure 3.10: Fokus!MBT test modelling environment (Wendland <i>et al.</i> , 2013)	65
Figure 3.11: Architecture and technology stack of Fokus!MBT (Wendland <i>et al.</i> , 2013)	66
Figure 3.12: ComGeneration methodology (Wacht <i>et al.</i> , 2011b)	68
Figure 3.13: Connectivity Editor for Click2IM service (Wacht <i>et al.</i> , 2011b).....	70
Figure 3.14: Tree-like Test Data Editor (Wacht <i>et al.</i> , 2011b).....	71
Figure 3.15: Behaviour Model for Click2IM service (Wacht <i>et al.</i> , 2011b)	71
Figure 3.16: TTS artefacts overview (Felderer <i>et al.</i> , 2010).....	73
Figure 3.17: Model-driven Testing Process (Felderer <i>et al.</i> , 2011).....	75
Figure 3.18: Test story of routing a call (Felderer <i>et al.</i> , 2010).....	76
Figure 4.1: TeamCom service development (adapted from (Eichelmann <i>et al.</i> , 2010)).	87
Figure 4.2: Methodology with both service and test development	88
Figure 4.3: Proposed overall methodology	91
Figure 4.4: Use case diagram containing the tasks of the stakeholders	93
Figure 4.5: Test Creation Framework architecture	94
Figure 5.1: Interaction between actor and system (Ryndina and Kritzinger, 2005)	104
Figure 5.2: Structured use case metamodel (Ryndina and Kritzinger, 2005)	105
Figure 5.3: Use case model of sample chat service	107
Figure 5.4: Structured use case model definition of “Add User”	108
Figure 5.5: Structured use case model definition of “Send Message”	109
Figure 5.6: RUCM process flow (Yue <i>et al.</i> , 2009).....	111
Figure 5.7: Overview of UTML test modelling process (Feudjio, 2009).....	118
Figure 5.8: Overview of UTML test models (Feudjio, 2011).....	120
Figure 5.9: Test Data View with UTML for SIP messages	121
Figure 5.10: Test Architecture Diagram for sample chat service	122
Figure 5.11: Test Behaviour Diagram for Send Message use case of sample chat service	123

Figure 5.12: Structure of Service Test Description.....	129
Figure 5.13: Dependency of Requirements through Preconditions	134
Figure 5.14: Relationship between Roles and CIs	135
Figure 5.15: SIP multimedia communication terminal (ITU-T Q.3948, 2011).....	136
Figure 5.16: The Role SIP phone with its corresponding CIs	137
Figure 5.17: Significance of channel for Roles and corresponding CIs	139
Figure 6.1: Generation of Behaviour Models based on STD and reusable test modules	155
Figure 6.2: Statecharts basic state example.....	163
Figure 6.3: Hierarchical OR-state example.....	164
Figure 6.4: Hierarchical AND-state example.....	165
Figure 6.5: Labelling of transitions.....	166
Figure 6.6: Specification of timeouts.....	167
Figure 6.7: Light Switch Statechart example.....	168
Figure 6.8: SCXML representation of Light Switch Statechart.....	168
Figure 6.9: Transaction user as mediator between client and server cores.....	171
Figure 6.10: Statechart example with explicit TU involvement	172
Figure 6.11: Simplified Statechart example without explicit TU involvement	173
Figure 6.12: Test Modules Environment architecture.....	176
Figure 6.13: Classification template for reusable test modules	181
Figure 6.14: Example classification template for SIP UAS non-INVITE reusable test module.....	185
Figure 6.15: Behavioural description of SIP UAS non-INVITE reusable test module	186
Figure 6.16: SCXML document of SIP UAS non-INVITE reusable test module	188
Figure 6.17: Example classification template for SIP UAC INVITE reusable test module	190
Figure 6.18: Behavioural description of SIP UAC INVITE reusable test module	191

Figure 6.19: SCXML document of SIP UAC INVITE reusable test module.....	193
Figure 6.20: Structure of abstract data types for test data.....	195
Figure 6.21: Conceptual structure of SIP_Request abstract data type.....	196
Figure 6.22: Predefined copying of message headers.....	198
Figure 6.23: Example XML document of SIP response message “s_Response2xx_6xx”	199
Figure 6.24: Stored data within Test Data Pool.....	200
Figure 6.25: Behaviour models generation process with ACE.....	202
Figure 6.26: Conceptual model of Service Test Description.....	204
Figure 6.27: Test modules instantiation in behaviour model flow chart.....	206
Figure 6.28: Test modules instantiation process example.....	207
Figure 6.29: Variable reading and parameterisation flow chart.....	208
Figure 6.30: Composition algorithm flow chart for Sender Step.....	211
Figure 6.31: Example composition of reusable test module instances with focus on Sender Step.....	212
Figure 6.32: Composition algorithm flow chart for Receiver Step.....	213
Figure 6.33: Example parsing with focus on Receiver Step.....	213
Figure 6.34: Composition algorithm flow chart for Parallel Step.....	214
Figure 6.35: Example composition of reusable test module instances with focus on Parallel Step.....	215
Figure 6.36: Example composition of reusable test module instances with focus on Condition Step.....	216
Figure 7.1: Generation, Execution and Evaluation of Test Cases.....	221
Figure 7.2: Test case derivation from behaviour models.....	224
Figure 7.3: Abstract test case generation from behaviour models by Test Case Derivation Unit.....	225
Figure 7.4: Hierarchy of structural coverage criteria (adapted from (Haschemi, 2009))	226

Figure 7.5: Behavioural description of SIP UAC non-INVITE (with transition marking)	230
Figure 7.6: Test case derivation from SIP UAC non-INVITE	230
Figure 7.7: Behavioural description of SIP UAS non-INVITE (with transition marking)	232
Figure 7.8: Test case derivation from SIP UAS non-INVITE	232
Figure 7.9: Conceptual model of a TTCN-3 test system (Willcock <i>et al.</i> , 2011)	237
Figure 7.10: Generation of executable TTCN-3 test suite based on abstract test cases	240
Figure 7.11: Dynamic test configuration with TTCN-3 test system	242
Figure 7.12: Abstract interface definition in TTCN-3 for SUT and test components	243
Figure 7.13: Example test configuration with two example PTCs	244
Figure 7.14: Mapping between XML representation of test data and resultant TTCN-3 template	246
Figure 7.15: Instantiation of test components in TTCN-3 test case	248
Figure 7.16: Example graph-based test case	249
Figure 7.17: First generated TTCN-3 behaviour function based on abstract test case	250
Figure 7.18: Second generated TTCN-3 behaviour function based on abstract test case	252
Figure 7.19: Starting of behaviour functions on test components	253
Figure 7.20: Concurrency example with two test components	254
Figure 7.21: Example test case with conditions	255
Figure 7.22: Example of conditions within generated TTCN-3 code	256
Figure 7.23: Generation of Executable Test Suite by Test Suite Builder	258
Figure 7.24: Excerpt of test adapter configuration file for compilation process	260
Figure 7.25: Interaction of test system entities during test case execution	263
Figure 8.1: Prototype architecture components illustrated as OSGi bundles	282
Figure 8.2: Apache Karaf architecture (adapted from (Apache Karaf, 2015))	283

Figure 8.3: Screenshot of TFUT web application showing the definition of an STD instance.....	284
Figure 8.4: OSGi service interface provided by the “Test Configuration Unit” bundle	285
Figure 8.5: OSGi service interface provided by the “Test Modules Environment” bundle	285
Figure 8.6: Simplified UML use case diagram of sample chat service	292
Figure 8.7: Basic functionality of login process in sample chat service.....	292
Figure 8.8: Alternative functionality of login process in sample chat service.....	293
Figure 8.9: Basic functionality of message exchange in sample chat service	293
Figure 8.10: Components within Mobicents SIP Servlets application server (adapted from (Mobicents, 2015)).....	295
Figure 8.11: ChatServiceServlet class of proof of concept sample chat service	296
Figure 8.12: Active OSGi bundles in Apache Karaf environment	301
Figure 8.13: Logging from "Automatic Composition Engine" bundle.....	302
Figure 8.14: Created behaviour models by "Automatic Composition Engine" bundle	303
Figure 8.15: Test execution of "Login" process for "[sender]" <i>Role</i>	305
Figure 8.16: Test report for test execution against sample chat service	306

List of Tables

Table 3.1: Overview of the UML 2.0 Testing Profile concepts (Zander <i>et al.</i> , 2005).....	58
Table 3.2: Evaluation of related projects based on derived requirements	82
Table 5.1: RUCM Use Case template (Yue <i>et al.</i> , 2009).....	112
Table 5.2: Restriction rules R1-R16 of RUCM approach (Yue <i>et al.</i> , 2013)	115
Table 5.3: Example RUCM use case of "Send Message"	116
Table 5.4: Excerpt of example STD containing two example Participating Roles.....	135
Table 5.5: Example of specifying CIs in STD	138
Table 5.6: Parameterisation of an example SIP MESSAGE request.....	140
Table 5.7: Instantiation of timers in <i>Parameters</i> field	142
Table 5.8: STD architectural perspective of simplified sample chat service.....	147
Table 5.9: STD Requirement definition for "Send Message" from sample chat service	148
Table 5.10: Comparison of specification languages	153
Table 6.1: Comparison of potential modelling notations.....	162
Table 6.2: Potential server types and their corresponding application layer protocols	180
Table 8.1: Architectural perspective of sample chat service.....	297
Table 8.2: Behavioural perspective for "Login" use case ("Req01").....	299
Table 8.3: Behavioural perspective for "Message Exchange" use case ("Req02").....	300

1 Introduction

The demand for advanced telecommunication services, so-called value-added services, has increased enormously over the last years. This has led to situations in the telecommunication domain where service providers and network operators have to provide a fast transition from concept to market product and have to offer a low price for new value-added services to satisfy their customers. The monopolies in the telecommunication domain have disappeared and accordingly, the fight for market shares between the competitors has become more difficult than ever before. Furthermore, the demand for even more specialised end-user services keeps growing.

In order to face the mentioned challenges, service providers have integrated Service Creation Environments (SCE) to allow their developers to rapidly create real value-added services and bring them to the market. However, relying on the quality of the SCEs and the skills of the developers to create new value-added services is not sufficient in order to provide the services in best quality. Therefore, thorough methodologies to consequently test the value-added services before the deployment and provisioning have to be implemented by the service providers. Then, they are able to assure their customers of a proper execution of the delivered value-added services and that they perform according to the specified requirements.

This research work has been dedicated to find and describe a novel methodology for functional testing of value-added services. It should enable service providers to increase

the quality of their delivered services and should provide both verification and validation of the service's implementation. The detailed aims and objectives of this research work are presented in section 1.1, followed by an outline of the thesis structure in section 1.2.

1.1 Aims and Objectives

The aim of this research is to propose a framework that allows the functional testing of value-added telecommunication services involving the concepts of Next Generation Networks (NGN). It should help test developers during the testing process by means of a novel sort of specification language and reusability aspects and should allow a better involvement of the service customer.

The main objectives of this research can be outlined as follows:

1. To analyse the existing approaches in value-added service development and provisioning and to figure out the possible benefits of the introduction of a novel test framework and methodology.
2. To analyse existing testing strategies and methodologies and related approaches. Based on the deficits and assets, the requirements for a novel framework will be elaborated.
3. To define the architecture and associated methodology of the proposed framework for the functional testing of value-added services, also including their verification and validation.
4. To examine diverse service descriptions or rather specifications of services, resulting in a proposed novel service description language.

5. To analyse recurring behaviour in value-added services and based on the results, define reusable test modules by means of a selected modelling notation. The reusable test modules shall be applied based on the proposed novel service description language.
6. To specify an adequate algorithm to compose the reusable test modules to complex behaviour models based on the proposed novel service description language.
7. To propose and analyse test case derivation and test case generation from the behaviour models and subsequently the execution of the derived test cases against the SUT.
8. To implement and evaluate the proposed framework for functional testing of value-added services by means of a prototype implementation.

The order of objectives declared above corresponds to the general structure of this thesis as presented within the following sections.

1.2 Thesis Structure

Chapter 2 describes the theoretical background of this thesis by introducing the concept of NGN and giving an overview of the SIP architecture and basic functionality. Furthermore, the term “value-added service” is defined as well as the service’s life cycle. The required environment to provide value-added services is depicted. Finally, the stakeholders in value-added service provisioning are introduced. Here, an important aspect is to identify the benefits for each stakeholder from the establishment of a novel test framework.

The state of the art in testing and the standardised test processes are described in chapter 3. Moreover, current testing methodologies are described and further related research is discussed. The essential and final part of this chapter is the definition of requirements for a novel test framework based on the limitations of the given approaches.

In chapter 4, the results of the identified requirements from chapter 3 are used as the starting point to propose a novel enhanced methodology for functional testing of value-added services. Additionally, an architectural overview of the associated new test framework is developed and its components are briefly introduced.

Chapter 5 deals with the proposed novel service specification language (so-called Service Test Description) for value-added services. The relevant components of the Service Test Description is described as well as its application within the test framework architecture.

Chapter 6 defines the structure and definition of the generic and reusable test modules and introduces a novel algorithm to compose the modules based on instances of the Service Test Description in order to generate so-called behaviour models.

Chapter 7 investigates on the test case derivation, generation and execution. The relevant algorithms are described. Finally, the evaluation of the tests is discussed.

Chapter 8 focuses on both the research prototype for the proof of concept and the evaluation of the proposed test framework.

Chapter 9 concludes the research work with a summary of its achievements and limitations. Furthermore, potential areas of future research and development are proposed.

2 Telecommunications Infrastructure and Value-Added Services

This chapter provides the theoretical background of the telecommunication infrastructure this research work is based on. After introducing the Next Generation Networks (NGN) concept and its general architecture specified by international standardisation bodies (section 2.1), the SIP protocol, its basic functionality, its relevance for NGN as well as the concept of SIP Application Servers is described (section 2.2). The third section outlines the term “value-added service”, gives diverse definitions and introduces the life cycle of services (section 2.3). Finally, the chapter closes with the identification of the stakeholders within the telecommunication domain with regard to possible improvements in the development and provisioning of value-added services (section 2.4).

2.1 NGN (Next Generation Networks)

The concept of NGN was introduced initially in the mid-1990s and has become widely accepted within the field of both fixed and mobile telecommunication networks. While telecommunication networks have historically been dominated by a circuit-switched paradigm, the implementation of NGNs led to a conversion towards packet-switched networks. Furthermore, the concept of NGN became popular to face the emerging situation in telecommunications characterised by a lot of different factors (Cochennec, 2002):

- deregulation of the market (followed by the open and international competition among network operators),
- increase of Internet utilisation and accordingly explosion of data traffic,
- strong demand from users for new multimedia services, and
- increasing demand from users for general mobility.

The ITU-T (International Telecommunication Union – Telecommunication Standardization Sector) started its standardisation work in the field of NGN in the year 2000. According to (ITU-T Y.2001, 2004), an NGN is defined as follows:

“A packet-based network able to provide telecommunication services and able to make use of multiple broadband, QoS-enabled transport technologies and in which service-related functions are independent from underlying transport-related technologies. It enables unfettered access for users to networks and to competing service providers and/or services of their choice. It supports generalized mobility which allow consistent and ubiquitous provision of services to users”.

In addition to this definition, (ITU-T Y.2001, 2004), (ETSI TR 180 000, 2006) and (Trick and Weber, 2004) indicate that NGN can be characterised by the following key features:

- packet-based data transport,
- broadband capabilities with end-to-end QoS (Quality of Service),
- support for a wide range of arbitrary services,
- separation of control functions among bearer capabilities, call/sessions and applications/services,
- interworking with legacy networks or other important telecommunication networks, especially access networks,

- Application Server support,
- unrestricted access for users to different networks and service providers,
- support for multimedia services,
- overall unified network management,
- mobility support,
- service-appropriate charging,
- scalability, and
- compliance with all regulatory requirements such as lawful interception and emergency calling features.

Figure 2.1 shows the NGN core consisting of a packet-switched network supporting security and QoS functionalities. Permission to reproduce Figure 2.1 has been granted by the authors of the referenced publication.

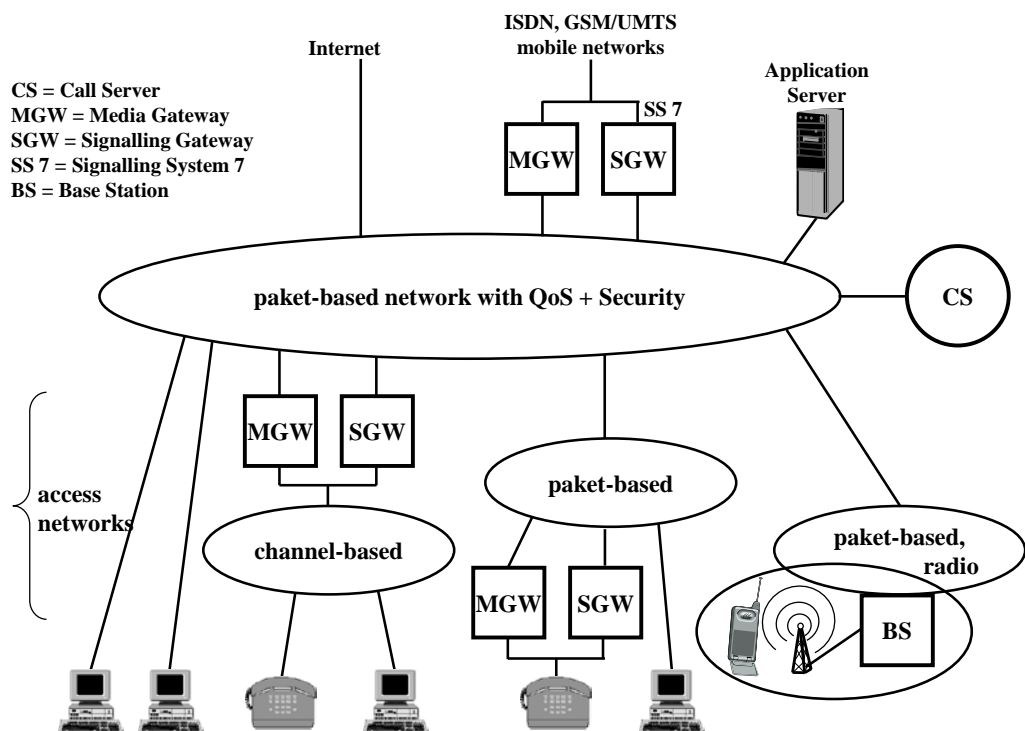


Figure 2.1: Principle structure of an NGN (Trick and Weber, 2009)

The displayed end user equipment, such as telephones, mobile phones or personal computers, can be directly connected to the NGN or via other access technologies, for instance channel- or packet-oriented, fixed or mobile access networks. The connection to other access networks requires *Media Gateways* (MGW) and *Signalling Gateways* (SGW). The role of the *Call Server* (CS) is to handle service requests and to control the MGWs according to a call control model and signalling handling. *Application Servers* (AS) can be involved in order to provide advanced services, so-called value-added services, which play a very relevant role in this research work and will be further specified in the upcoming section 2.3. Besides the mentioned servers and gateways, the NGN also offers access to other networks such as the Internet or to both circuit-switched and packet-switched telecommunication networks by gateways.

Regarding its functional architecture, a basic reference model for NGN was defined in (ITU-T Y.2012, 2010), which implies “the most important novelty introduced with NGN in the telecommunications (i.e. ICT) world – the separation of services and transport in separate so-called stratum” (Janevski, 2014). The transport stratum is concerned with the transfer of information or rather data between terminating endpoints. It also includes further transport functions, such as admission control and mobility management functions. The service stratum is located above the transport stratum and is responsible for the control and management of network services to enable end-users services and applications. Such services may be related to voice, data, or video applications, or alternatively, in some combination in the case of multimedia applications (Obermann and Horneffer, 2013). The main reason for the separation of the transport stratum and the service stratum is “to allow independent evolution of the technologies used in these strata”

(Ding, 2010), meaning that certain evolutions, for instance in the service stratum, will not affect the transport stratum.

According to (ITU-T Y.2012, 2010), applications are outside of the NGN scope. This might be an inadequate approach as applications are predicted to be “one of the main revenue streams in future telecommunication networks” (Lehmann, 2014). Hence, other researches will not leave applications outside the scope of NGN. (Trick and Weber, 2009) and (Magedanz and de Gouveia, 2006) describe a so-called application stratum (or application layer) on top of the service stratum and the transport stratum. The following Figure 2.2 presents the NGN architecture considering all three strata.

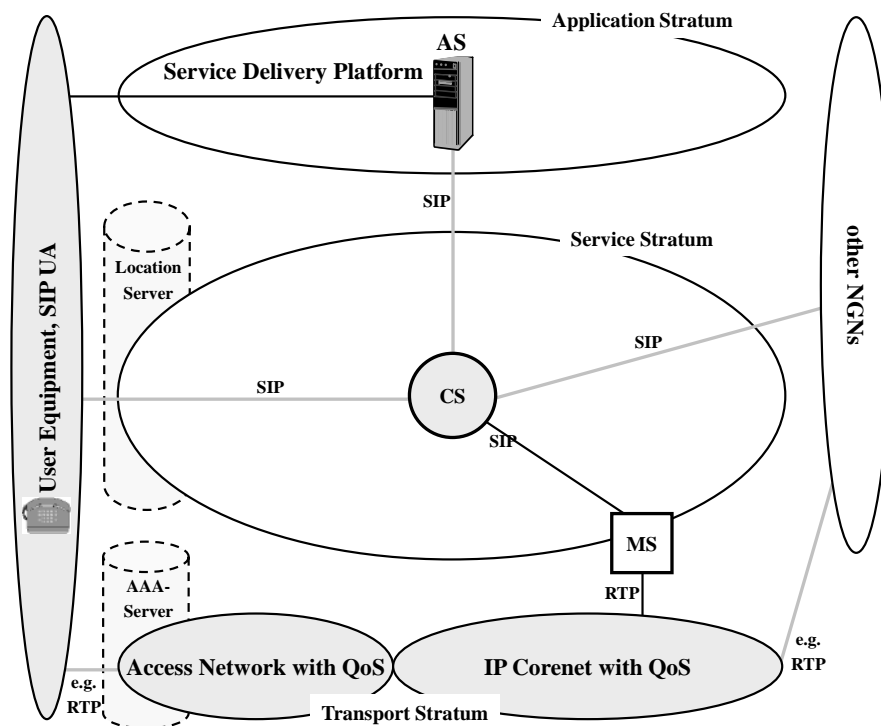


Figure 2.2: NGN architecture in a strata/layer structure (Trick and Weber, 2009)

Permission to reproduce Figure 2.2 has been granted by the authors of the referenced publication. The integrated *Application Stratum* includes a *Service Delivery Platform* (SDP) with at least one *Application Server* in order to provide value-added services. The

SDP concept will be further described in section 2.3.2. Figure 2.2 also illustrates the relevant communication channels between the three strata including the needed protocols, such as the *Real-Time Transport Protocol* (RTP) mostly relevant in the transport stratum to transfer real-time payload, and the *Session Initiation Protocol* (SIP) enabling the controlling of sessions in the service stratum. SIP and its role for NGN will be further described in the upcoming section 2.2.

2.2 SIP and its Utilisation in NGN

The Session Initiation Protocol (SIP) was initially defined by the Internet Engineering Task Force (IETF) and specified in (IETF RFC 2543, 1999). In 2002, a new version of the IETF standard was established (IETF RFC 3261, 2002). Furthermore, diverse extensions and updates are specified by a number of RFCs (Request for Comments). The main purpose of SIP is to initiate, coordinate and tear down real-time communication sessions between endpoints over an IP-based network. While the role of SIP is to set up communication sessions, the Session Description Protocol (SDP) is used to describe the session. Furthermore, the communication endpoints can negotiate the codecs to be used in a VoIP call. Based on the negotiated codecs, the actual media, such as audio, video or other multimedia content, is then exchanged between the session participants by the use of an appropriate transport protocol, e.g. RTP (IETF RFC 3550, 2003). SIP also offers advanced functions, such as instant and presence messaging, and implements several mechanisms, e.g. handshake, retry or timeout mechanisms. It has gained wide industry acceptance and has been determined as standard protocol in the Universal Mobile Telecommunication System (UMTS) Release 5 (ETSI Tdoc RP 030375, 2003).

Within the proposed framework for automated functional testing of value-added services resulting from this research work, the concepts of SIP play a major role. On the one hand, its transactional concept is reused and, on the other hand, most of the value-added services require SIP signalling.

2.2.1 SIP Architecture and Basic Functionality

According to (IETF RFC 3261, 2002), SIP uses a modular architecture that includes the following network components: SIP User Agent (UA), SIP Registrar Server, Location Server and SIP Proxy Server.

A SIP-enabled end user device within a SIP-based telecommunication infrastructure is called a SIP UA. It acts as an agent on the behalf of a user and sends and receives SIP messages to establish, modify and terminate sessions. A SIP UA contains both a client application and a server application. These two parts are designated as User Agent Client (UAC) and User Agent Server (UAS). The UAC is responsible to create and send requests whereas the UAS processes incoming requests and generates appropriate responses. During a session, a SIP UA will operate as both a UAC and a UAS. The concept of UAC and UAS within a SIP UA is shown in Figure 2.3.

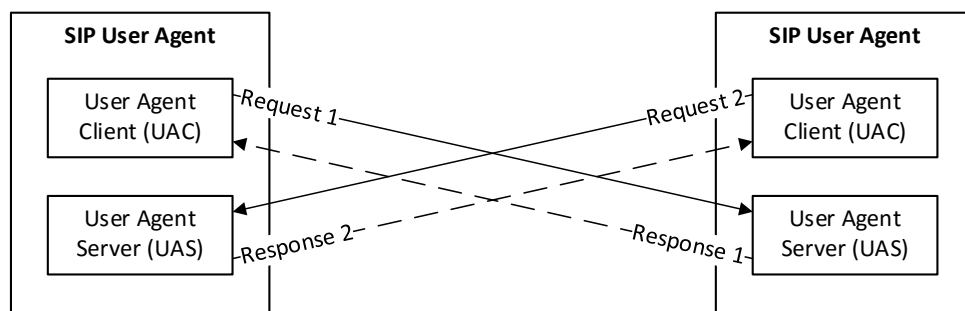


Figure 2.3: Basic principle of SIP communication

Each end-terminal registers its current contact information (such as the IP address and the port of the endpoint) at a SIP Registrar Server using a special SIP message, the SIP REGISTER request. Upon receipt of this message, the SIP Registrar Server transmits the data to the Location Server which will store it in a database for contact information of participating SIP UAs within a specific domain. The interface between the Location Server and other servers is not standardised.

The SIP Proxy Server routes messages between SIP UAs. According to (Trick and Weber, 2015), two different kinds of SIP Proxy Servers exist, so-called “Stateless” Proxy Servers and “Stateful” Proxy Servers. A Stateless Proxy Server acts as an intermediate that simply forwards the SIP request it receives. It does not store any information of the call. Contrary to this, a Stateful Proxy Server keeps track of every request and response it receives by storing the relevant information. It can act as both UAC and UAS and is therefore able to create requests (e.g. “CANCEL”) and responses (e.g. “100 Trying”). The Stateful Proxy Server is also capable of absorbing retransmissions because it knows that it has already received a specific message.

The basic establishment of a SIP session using the described SIP network components is illustrated in Figure 2.4. At first, a SIP User Agent A sends an INVITE request containing the target end-terminals address as SIP URI (e.g., “user@domain.de”) to a locally predefined SIP Proxy Server (see Figure 2.4, step 1). The INVITE request also includes a Session Description Protocol (SDP) (IETF RFC 4566, 2006) message with the proposed streaming media initialisation parameters of SIP User Agent A. After the SIP Proxy Server received the message, it subsequently checks the location database to lookup the location of SIP User Agent B. However, if the target’s SIP URI is within another domain,

a lookup is initialised using the Domain Name System (DNS) (IETF RFC 1034, 1987) (IETF RFC 1035, 1987) (see Figure 2.4, step 2). Then, the message is forwarded to the proper SIP Proxy Server of the other domain (see Figure 2.4, step 3). If SIP User Agent B is located within the same domain, the SIP Proxy Server can locate the target's current contact address by requesting the domain-local Location Server (see Figure 2.4, step 4).

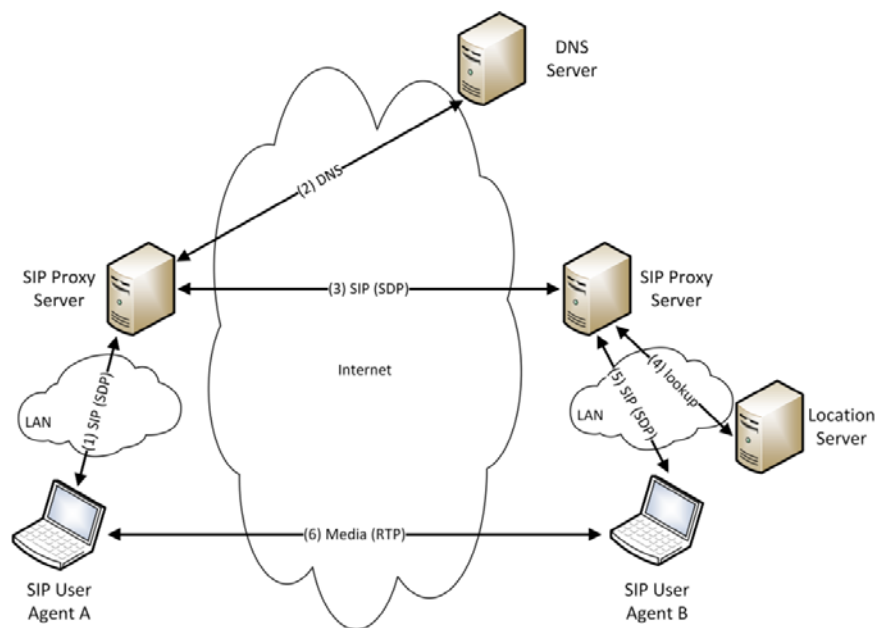


Figure 2.4: Basic establishment of a SIP session

Accordingly, the SIP Proxy Server is able to forward the message to SIP User Agent B (see Figure 2.4, step 5). When SIP User Agent B accepts the call, it sends a message with a response code of “200” that also contains SIP User Agent B’s codec capabilities and the port numbers where it wants SIP User Agent A to send the RTP data to. The final part of the so-called “Three-Way-Handshake” occurs when SIP User Agent A sends an acknowledgement to SIP User Agent B (so-called “ACK” request). By sending the ACK, SIP User Agent A confirms to have received the response from SIP User Agent B. Now, a logical connection-oriented communication state, a so-called SIP dialog, has been

established. The end systems are now ready to exchange media data of arbitrary nature, such as audio and/or video data flows, by making use of RTP (see Figure 2.4, step 6).

2.2.2 The Layered Structure of SIP

SIP is structured as a layered protocol comprising the syntax and encoding layer, transport layer, transaction layer and transaction user (TU) layer. The structure allows different modules within it to function independently with just a loose coupling between each layer (IETF RFC 3261, 2002). The following Figure 2.5 visualises the layered structure of SIP in the application layer and also includes the two lower layers, transport layer and network layer. Permission to reproduce Figure 2.5 has been granted by Springer Publishing.

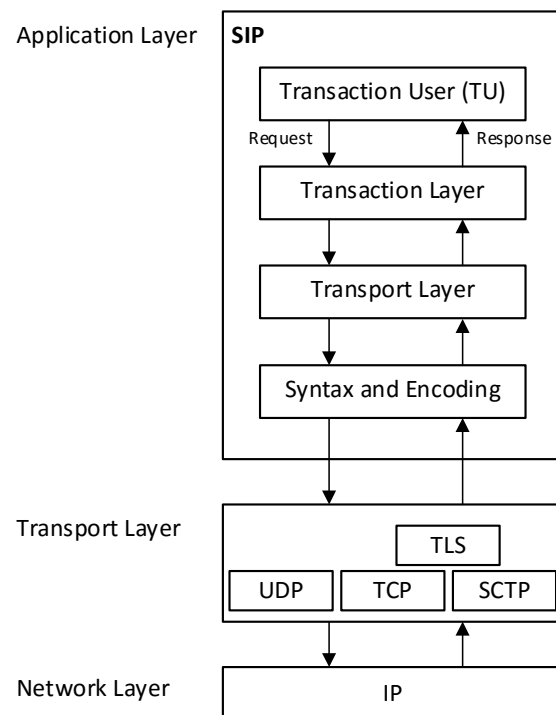


Figure 2.5: Layered structure of SIP (Ding and Liu, 2008)

The lowest layer of SIP is its *syntax and encoding* specifying the format and structure of a SIP message by the use of an augmented Backus-Naur Form grammar (BNF) defined

in (IETF RFC 2234, 1997). Such a SIP message can be either a request from a client to a server, or a response from a server to a client.

The *transport layer* as second layer defines the behaviour of SIP entities in sending and receiving messages over the network. It is responsible for managing persistent connections for transport protocols like UDP (User Datagram Protocol) or TCP (Transmission Control Protocol) and SCTP (Stream Control Transmission Protocol) with or without TLS (Transport Layer Security) over the network. The opened connections are shared between the client and server transport functions.

On top of the SIP transport layer is the *transaction layer*. A transaction, a very fundamental component of SIP, is a request that is sent by a client to a server, along with all responses to that request sent from the server back to the client. All the SIP messages of a transaction share a common unique identifier and traverse the same set of hosts (Toral-Cruz *et al.*, 2011). The transaction layer itself handles application-layer retransmissions, matching of responses to requests by comparing the identifiers, and application-layer timeouts. It uses the transport layer for sending and receiving requests and responses. The transaction layer contains four transaction-state machines each having their own timers, re-transmission rules and termination rules. These state machines are specified in (IETF RFC 3261, 2002):

1. UAS INVITE state machine
2. UAS non-INVITE state machine
3. UAC INVITE state machine
4. UAC non-INVITE state machine

The fourth and topmost layer of the SIP structure is the *transaction user* (TU) that actually creates client and server transactions. When a TU intends to send a SIP request, first it creates an instance of a client transaction and subsequently, it sends the SIP request along with the destination information (destination IP address, port number and transport protocol). Generally, TUs are defined to be both UAC core and UAS core and are part of all SIP entities except for Stateless Proxy Servers. The UAC part of the TU creates and sends requests and receives responses using the transaction layer, whereas the UAS part receives requests and creates and sends responses using the transaction layer. There are two factors that can affect the behaviour of the TU, the method name in the SIP message on the one hand and the state of the request with regard to SIP dialogs on the other hand (Poikselkä and Mayer, 2009).

2.2.3 SIP-based NGN

As mentioned in the previous sections, SIP is a powerful protocol for the control and management of communication sessions between end-users in telecommunication networks. It also includes various methods to modify existing sessions or even to combine them. These aspects clarify why SIP has become widely accepted as the protocol of choice for communication control in NGN. A SIP-based NGN matches the descriptions accentuated in section 2.1 due to the fact that SIP is used as standard protocol to enable the controlling of sessions in the service stratum. Figure 2.6 shows the principle structure of an NGN based on SIP. Permission to reproduce Figure 2.6 has been granted by the authors of the referenced publication.

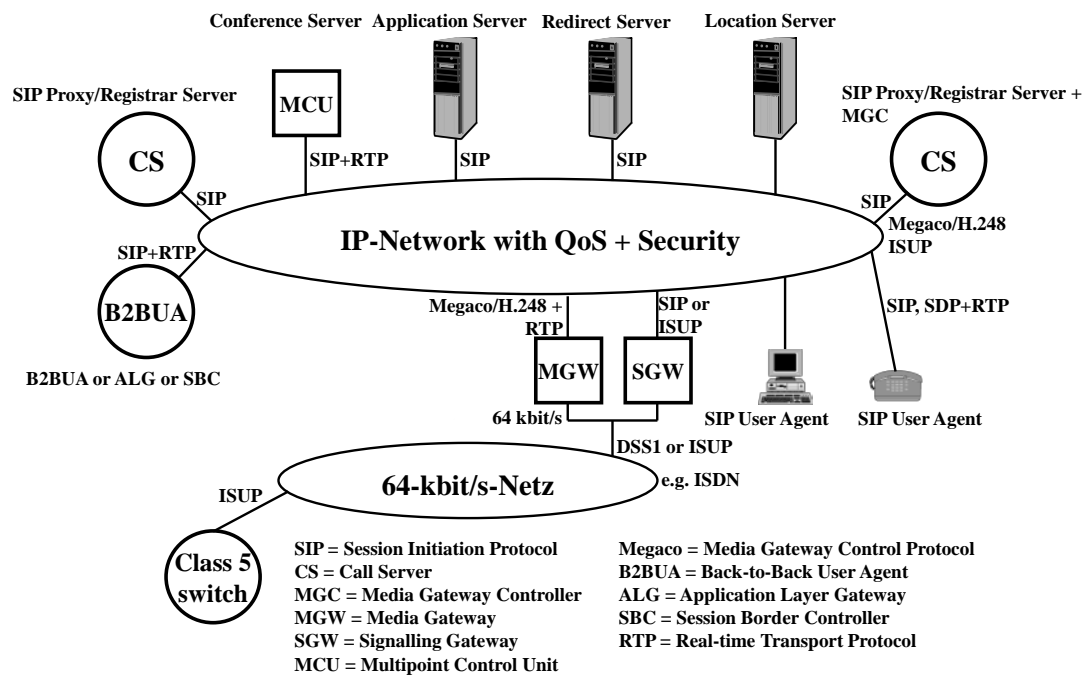


Figure 2.6: Principle structure of SIP-based NGN (Trick and Weber, 2009)

A SIP service provider operates and manages the core infrastructure which includes the *Call Server* (CS) as a centralised component, whose functionality is provided by *SIP Proxy* and *SIP Registrar Servers* (Trick and Weber, 2009). The SIP Proxy Server relies on *Location Servers* in order to find the matching temporary SIP URIs for given permanent SIP URIs. The *Redirect Server* acts as a UAS that generates SIP redirection responses (SIP responses containing status codes from 300 until 399) to SIP requests it receives in order to direct the client to contact an alternate set of SIP URIs. *MGW* and *SGW* enable access to traditional circuit-switched networks, such as the Public Switched Telephone Network (PSTN). The *Multipoint Control Unit* (MCU) or rather Conference Server (IETF RFC 4353, 2006) is implemented in the SIP-based NGN in order to let SIP User Agents take part in conference scenarios. All displayed SIP User Agents consist of the end user equipment.

As far as (Weber, 2012) is concerned, “SIP signaling and media streams can be forced to be routed in parallel via intermediate service layer network elements that are trusted by the SIP service provider”. This aspect might be useful considering particular legal requirements such as lawful interception, the interconnection with other providers and for Network Address and Port Translation (NAPT) traversal. To achieve this, *SIP Back-to-Back User Agents* (B2BUA) are used. According to (3GPP TR 29.962, 2005), a B2BUA is permanently inserted at connections between the SIP-based NGN (e.g. IMS) and a given external network handling all SIP signalling (including session attempts, subscription, instant messages) including signalling where the flows may forward without B2BUA interventions. In general, B2BUAs are implemented in network elements such as *Session Border Controllers* (SBC) or *Application Layer Gateways* (ALG).

Finally, the most relevant component of the SIP-based NGN (see Figure 2.6) regarding this research work, the *SIP Application Server*, will be introduced in the following section.

2.2.4 SIP Application Server

In principle, the main task of a SIP AS within a SIP-based NGN is to enable a fast and cost-efficient provision of value-added services. According to (Trick and Weber, 2015), the SIP AS is a combination of a SIP UA, and/or a SIP Proxy Server, and/or a SIP Redirect Server. In particular, it contains a software platform for services.

A SIP AS requires a SIP communication channel to a corresponding CS in order to allow end users to invoke services that are currently deployed on the SIP AS. The CS routes the SIP messages to the SIP AS based on configured or currently requested filtering criteria.

Based on further filtering criteria, the SIP AS chooses the appropriate software and starts the service execution.

According to (Trick and Weber, 2015), four different modes of operation have been established regarding the SIP AS. These modes are illustrated in the following Figure 2.7. Permission to reproduce Figure 2.7 has been granted by the authors of the referenced publication.

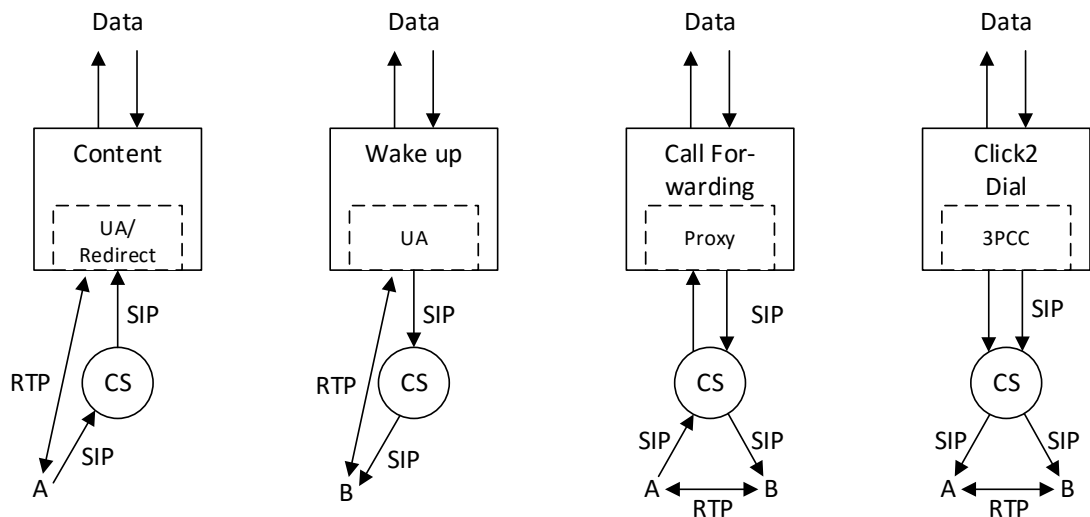


Figure 2.7: Modes of operation of a SIP Application Server (Trick and Weber, 2009)

The first mode of operation, the “Content” mode, determines the SIP AS to either act as SIP UA or SIP Redirect Server. Here, the SIP UA of user A triggers the initiation of the service and subsequently, user data can be transmitted between the AS and the SIP UA of user A. Depending on the interfaces the SIP AS has, the data can be of any kind. An example service can be, for instance, a weather forecast that will be read out to A based on the content of a web site.

The “Wake up” mode indicates the SIP AS to be the initiator of the service. So, the SIP AS builds up a SIP session to the SIP UA of B independently. Usually, the initiation of

the call takes place as soon as some specified condition is fulfilled. A typical example service is the “Wake up” service itself. Here, a call is initialised at a certain time.

In the “Call Forwarding” mode, the SIP AS acts like a standard SIP Proxy Server. First, the SIP AS receives a request from the SIP UA of A forwarded by the CS because of some defined filtering criteria (e.g. “user unknown”). Consequently, the SIP AS determines the relevant data by using its data interface and then provides the CS with the information. Afterwards, the CS forwards the appropriate message to the SIP UA of B and finally, both users can exchange data. An example service can be a location-based search for a restaurant.

In the final “Click 2 Dial” mode of operation, the data interface of the SIP AS or rather some third party (3PCC = Third Party Call Control) triggers the initialisation of a session between the UAs of A and B. In this scenario, the SIP AS is acting as a B2BUA. An example trigger can be, for instance, the clicking of a button on a web site or some other software application. Based on this event, the call is initiated and finally, the SIP UAs of A and B can directly communicate via RTP.

The introduced modes of operation of a SIP AS demonstrate the variety of possibilities regarding the development of value-added services. However, further servers are required in order to support the diversity of services, such as email servers, media servers, web servers or database servers. In general, through the data interface of a SIP AS, a value-added service can make use of any functionality that is provided by the different servers.

The mentioned diversity of services that can be provided by SIP AS gives a further reason why this research work is so relevant. The more complex a value-added service gets the

more relevant is the integration of a test framework to support service developers to program high quality services for the service customers.

2.3 Value-Added Services in NGN

Value-added services are the main object of this research and will be described in the following. This introduction also requires the knowledge of how services in the telecommunication domain are classified in general.

2.3.1 Classifications and Definitions

With regard to the definition of the term “telecommunication” itself, the ITU has the following to say: “Any transmission, emission or reception of signs, signals, writing, images and sounds or intelligence of any nature by wire, radio, optical or other electromagnetic systems” (ITU, 2011). To enable this telecommunication capability, telephone companies provide telecommunication services. To quote (ITU-T T.174, 1996), telecommunication services are “that which is offered by an administration to its customers in order to satisfy a specific telecommunication requirement”. A more detailed definition is given in the words of (Calisti, 2003), where a telecommunication service is “a set of independent functions that are an integral part of one or more business processes. This functional set consists of the hardware and software components as well as the underlying communication medium. The customer sees both as an amalgamated unit. A service can be a service component of another service”. Another quite similar definition of the term is described by the 3GPP, a service is “a component of the portfolio of choices

offered by service providers to a user, a functionality offered to a user.” (3GPP TR 21.905, 2005).

Besides the provided definitions of the term “telecommunication service” or rather “service”, many others exist and the words are often used in several different contexts with somewhat different meanings although they are describing the same, such as in (Kühn, 1991), (ETSI TS 122 228, 2011), (ETSI TS 122 105, 2011), (ETSI TS 122 101, 2011) and (ITU-T I.211, 1993).

In principle, telecommunication services are divided into bearer services, teleservices and supplementary services.

A bearer service is a type of telecommunication service that provides the “capability of transmission of signals between access points” (ETSI TS 122 105, 2011). Typically, bearer services are categorised by their information transfer characteristics, methods of accessing the service, interworking requirements (also to other networks), and other general attributes (Harte *et al.*, 1999). Bearer services cover the lower three layers of the OSI (Open Systems Interconnection) model from physical layer up to the network layer.

A teleservice is a type of telecommunication service that provides the “complete capability, including terminal equipment functions, for communication between users according to protocols established by agreement between network operators” (ETSI TS 122 105, 2011). The teleservices are user end-to-end services (e.g. telephone calls) and cover the full seven layers of the OSI protocol layer model.

Supplementary services modify or supplement basic telecommunication services. Therefore, they cannot be offered to a customer as a stand-alone service and must be

offered in combination with a basic bearer service or basic teleservice. The same supplementary service can be applicable for a number of telecommunication services (ETSI TS 122 105, 2011).

According to (ETSI TS 122 001, 2011) and (ITU-T I.210, 1993), Figure 2.8 illustrates the categorisation of telecommunication services.

telecommunication services	
teleservice	basic teleservice + supplementary service(s)
	basic teleservice
bearer service	basic bearer service + supplementary service(s)
	basic bearer service

Figure 2.8: Categorisation of telecommunication services

Besides the already mentioned classes of services in the telecommunication domain, a further service type, value-added (telecommunication) services, exists. According to (ETSI TS 122 101, 2011), these value-added services can be based on fully proprietary protocols or standardised protocols. With regard to this research work, the following definitions of the term “value-added services” might be suitable. The OMA (Open Mobile Alliance), for instance, defines in (OMA ORG, 2007) that the term stands for a “telecommunication/information service that is offered in addition to and/or in conjunction with a basic telecommunication/data service”. This rather generic definition indicates that every service can be seen as a value-added service if it extends the functionality of a pre-existing basic telecommunication service.

(Glitho *et al.*, 2003) agrees with the OMA. In their opinion, value-added services are “defined as anything that goes beyond two-party voice calls” (Glitho *et al.*, 2003). Furthermore, “Value-added services are usually grouped under two umbrellas: telephony services and nontelephony services. Telephony services interact with call control while nontelephony services do not” (Glitho *et al.*, 2003). The authors also give examples of telephony services such as conferencing, call diversion or telephone voting. Nontelephony services can be special instant messaging services, push-to-talk and multimedia messaging.

A further definition of the term “value-added service” is given by (Guo *et al.*, 2009) who point out that value-added services “add value to the standard service offering, spurring the subscribers to use their phone more and allowing the operator to drive up their ARPU (Average Revenue per User)” (Guo *et al.*, 2009). They also state that “Both the academic and the industrial communities have paid much attention on the subject how to design and implement the personalized service and shorten the time to market” (Guo *et al.*, 2009). Here, the authors denote the potential of value-added services, especially emphasising the economic benefits for service providers and network operators and the need for mechanisms in order to provide the services fast, custom-made and in high quality.

The most appropriate definition of value-added services relating to this research work is provided in (Lehmann, 2014): “Value-added Services (VAS) are functional properties which will offer certain comfort to consumers. Consumers will recognise additional benefit by value-added services”. Regarding the composition of value-added services, (Lehmann, 2014) discusses that they “are based on a combination of one or more bearer

services and one or more teleservices, and optionally, one or more supplementary services.” The author also states that value-added services can be an extension to basic teleservices and they can sometimes stand-alone (e.g. non-call related services). “VAS also have a certain time dimension associated with them. A value-added service today can become a basic service in the future when it becomes sufficiently common place and widely deployed, and for example, is no longer used as a differentiation feature among operators” (Lehmann, 2014). The author describes the positioning of value-added services within the telecommunication domain and predicts that the provisioning of them will play a major role for the operators in future.

As already mentioned in the sections 2.1 and 2.2.3, unlike basic telecommunication services being provided in the service stratum of the NGN by Call Servers, the value-added services are provided by Application Servers. According to (Trick and Weber, 2009), the handover of the service intelligence from the Call Servers to the Application Servers leads to a significantly low dependency between the network and the value-added services. This aspect makes it possible to quickly and easily provide new value-added services.

2.3.2 Development and Provisioning of Value-Added Services

As stated in section 2.2.4, value-added services can be complex because of the diversity of functionality that can be applied through a SIP AS. As in the development of complex software systems, the development and provisioning of value-added telecommunication services requires “expertise on system architecture, software design, communication protocols, and [possibly] knowledge of legacy systems” (Ling *et al.*, 2009). A major

challenge for service designers is the complexity and heterogeneity of the network infrastructure which always has to be considered at both system and application levels.

With regard to the traditional service life cycle, the OMA has specified in (OMA OSPE, 2005) the different stages a telecommunication service has to go through. The following Figure 2.9 illustrates the service life cycle phases.

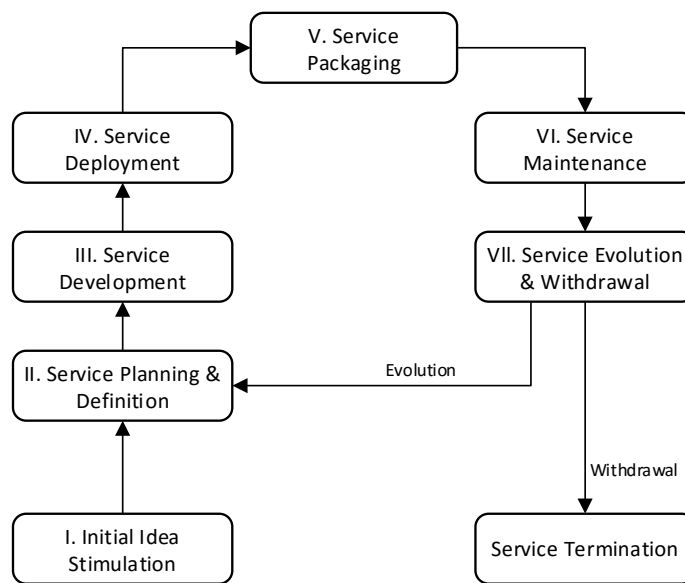


Figure 2.9: Service life cycle (adapted from (OMA OSPE, 2005))

Initially, the service life cycle starts with the vague idea of a new service demand. Such an idea is usually inspired either by the analysis of market needs performed by the service providers or, in most cases, by a customer’s desire of what a new service is supposed to do. The idea can also be derived from successful service ideas in other networks initiated by other service providers. Following the phase of idea generation a feasibility study is done in the “Service Planning & Definition” phase identifying if the service is found to be commercially feasible. This phase also includes the capturing of the service’s requirements in order to establish a service specification which includes a detailed analysis of the service’s functionality, necessary data and desired output. Among all the

phases within the service life cycle, the upcoming third phase, “Service Development”, is the most abstract and general of all phases, since there are diverse approaches on how to structure the different stages within the phase. Generally, the phase refers to the process of implementing the service logic and testing its functionality. In the upcoming fourth phase “Service Deployment”, the implemented and tested service is actually deployed in the service provider’s environment. This process includes every step from the initial installation of the service until its activation. Afterwards, the service can be offered to the customer base by defining commercial packages or bundles in the “Service Packaging” phase. This phase is followed by the active use of the service by customers who subscribed the service in advance. The service provider has to maintain the service’s functionality (“Service Maintenance”) and may find some needs to influence and adjust the service to changing requirements. This aspect might necessitate the evolution of the service which leads to an adaptation of the service specification and the phases until phase six have to be repeated. Alternatively, the service provider decides to withdraw the service (“Service Termination”) possibly due to its weak technical or commercial performance.

In order to manage the life cycle of services within the service provider’s environment and especially with respect to the provisioning of value-added services, service providers use a scalable and standardised platform for the creation, deployment, execution, orchestration and management of these value-added services, the so-called Service Delivery Platform (SDP). The SDP is located in the application layer and is connected to the NGN service and transport layer through abstract interfaces (Trick and Weber, 2009). It can contain multiple Application Servers and Media Servers and provides interfaces to an environment, the Service Creation Environment (SCE), in which service developers can efficiently develop new value-added services by combining the capabilities of

existing basic telecommunication services and other value-added services (Menkens, 2010). This SCE enables the development of a new value-added service either from scratch or from predefined modules. Generally, the SCE provides graphical tool support for the service developer in order to simplify and accelerate the service development process. The already mentioned connection between the SCE and the SDP enables an immediate provisioning of value-added services (Trick and Weber, 2009). An illustration of the overall concept of the SDP in NGN is given in Figure 2.10. Permission to reproduce Figure 2.10 has been granted by the authors of the referenced publication.

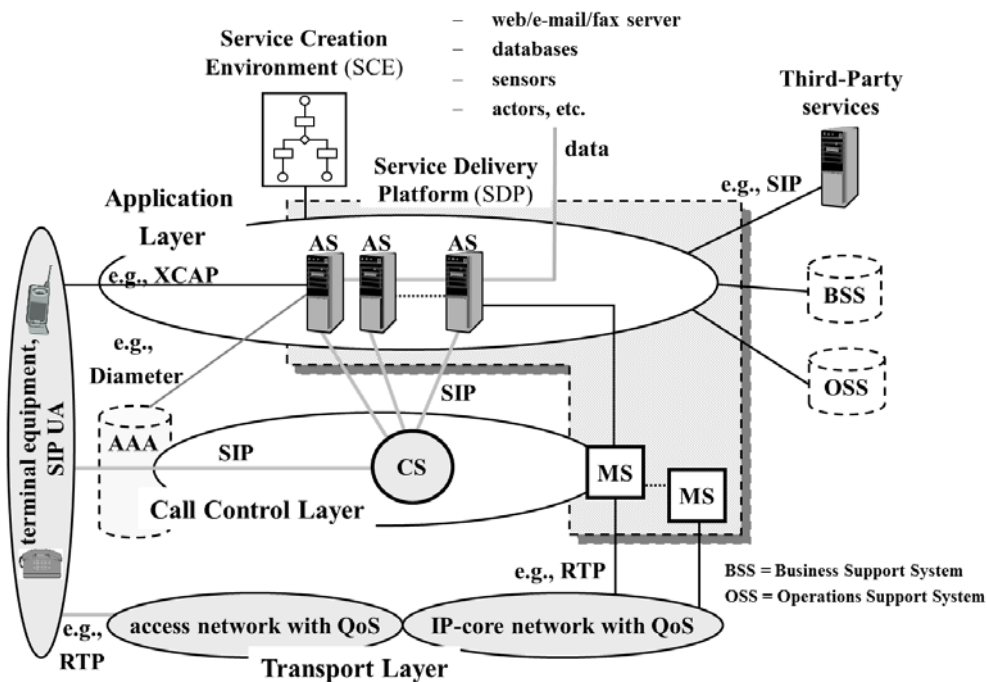


Figure 2.10: Service Delivery Platform in NGN (Trick and Weber, 2009)

To sum up, the relevance of SDPs in combination with SCEs is very high for service providers in order to provide value-added services to their customers in a standardised approach and within a short time period. However, there is to date no standardised and robust Test Execution Environment (TEE) defined specifically within an NGN. To our knowledge, the major focus of testing in the NGN field is related to the testing of NGN

protocols, so-called conformance testing (ITU-T Q.3946.2, 2013), and to the testing of NGN equipment for compatibility and interoperability (ITU-T Q.3948, 2011). Therefore, the establishment of a TEE in combination with an equivalent environment to SCEs but focusing on test creation would make an appealing framework, possibly called “Test Creation Framework” (TCF). Such a TCF would improve the quality of developed value-added telecommunication services on the one hand and would verify that the customer’s desire of what a value-added service has to do is fulfilled on the other hand.

2.4 Stakeholders in Value-Added Service Provisioning

In order to identify the benefits of a TCF especially defined for the process of functional testing of newly developed value-added telecommunication services, the stakeholders in service development and provisioning have to be introduced. Of course, the proposed TCF will be part of the service development and provisioning process and is therefore situated in the service provider environment. Nevertheless, its implementation might have potential positive effects for each stakeholder. This assumption will be analysed in the following.

According to (ITU-T M.3340, 2009), the relevant roles (respectively stakeholders) in an NGN environment are as illustrated in Figure 2.11. Permission to reproduce Figure 2.11 has been granted by ITU.

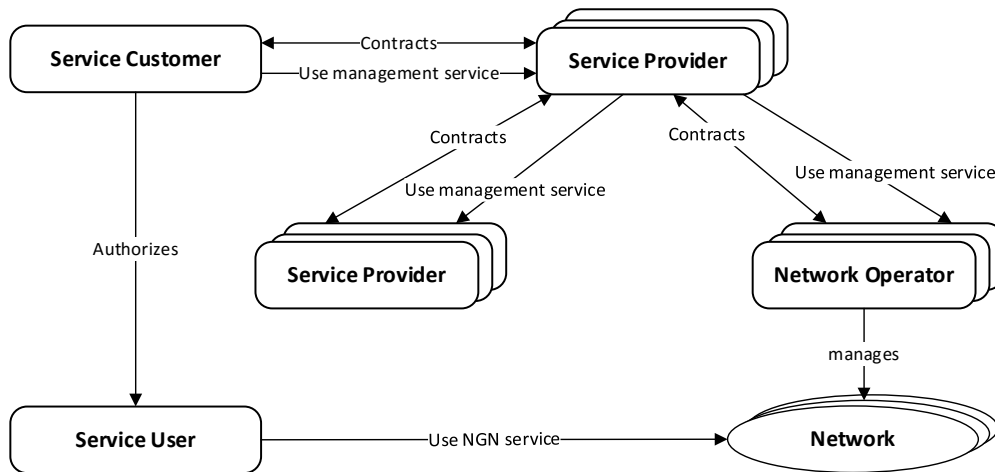


Figure 2.11: Roles in NGN management (adapted from (ITU-T M.3340, 2009))

Besides the network operators, service providers and service customers, also the service users or rather end-users are shown including the relationship between one another.

The network operators are organisations that enable the transport stratum in the NGN architecture illustrated in Figure 2.2. They operate the network and take responsibility for providing the required end-to-end connectivity to the service providers using their network (Salina and Salina, 2007). The establishment of a novel TCF within the service provider environment would not directly concern the network operators. However, the erroneous function of a newly deployed value-added service could also be due to some problems (e.g. lack of performance) within the network.

The service providers offer basic and value-added telecommunication services through their service provider environment to service customers. As far as (ITU-T M.3340, 2009) is concerned, the service providers may or may not operate a network themselves and may or may not be a customer of another service provider. Additionally, the service providers have to deal with the following tasks (Menkens, 2010), (ITU-T M.3340, 2009), (Salina and Salina, 2007), (ITU-T M.3050.1, 2007):

- Managing and administrating the SDP and its associated applications, components and configuration logic.
- Automating their customer care, service and network management processes.
- Installing and testing new services and supporting functions in the SDP as well as investigating and resolving service related issues (which may be experienced by a service user).
- Ensuring that newly deployed services do not impact existing services.
- Administrating the life cycle management of value-added services.
- Aggregating generic service capabilities to create high-value combinational services that enrich the user experience, e.g. applying an SCE.
- Moving to more of an end-to-end process management approach developed from the service customer's point of view.

Apparently, the service providers would benefit most from the establishment of a novel TCF. As they have to face enormous challenges, such as more demanding customers, increased competition, ever-growing regulatory requirements and time-to-market pressure, the service providers have to offer value-added telecommunication services in the best possible quality. This aspect will lead to satisfied service customers.

The third role, the service customer (ITU-T M.3050.1, 2007), can be a person or an organisation that has a contractual relationship with a service provider. The customer is responsible for ordering and paying for the products of a service provider. Additionally, the service customer can act as service user by actually consuming a service provided by the network. Alternatively, a service customer can act as a wholesale customer that resells

the service provided, possibly with some further value. Relating to the establishment of a TCF in the service provider environment, the service customers will also benefit from thoroughly tested value-added services as they are either the direct consumers or alternatively wholesale customers who can provide their service users with high quality products.

Another relevant entity, not being mentioned in the NGN environment but playing an important role in the value-added service development, is the group of service developers. Generally, the service developers are working for service providers and develop the applications and business logic that allows the service providers to offer their services to the service customers. The used development platform within the SDP of the service provider (e.g. SCE) needs to have a lower barrier of entry for the service developers. It should be easy to use, easy to maintain and self-descriptive (OMA OSPE, 2005). Additionally, the development platform needs to be state-of-the art with well-known programming languages and easy to learn Software Development Kits (SDK) and Application Programming Interfaces (API). Similarly to the service providers, the service developers would also benefit from the establishment of a proper TCF. During development and after having developed a new value-added service, a service developer might receive feedback if the service is correct and if it meets the requirements of the service customer. Also, the maintenance phase especially after just having deployed a service in the service provider environment can be shortened.

2.5 Conclusion

Within this chapter, the general environment of this research work was introduced. Starting with the discussion of the NGN concept as defined by ITU-T and ETSI in section 2.1, the following section 2.2 focused on the architecture and structure of the SIP protocol as well as its basic functionality. This section also mentioned the relevance of the SIP protocol for the NGN environment and also for the research work as one major component.

The main object of this research work, the value-added telecommunication services, was introduced in section 2.3. Several definitions of the term “value-added services” were mentioned and discussed. A standard life cycle of services was described as well as approaches for service providers to develop new services and provide them in their environment in order to be consumed by their customers. Based on the information given, the lack of a proper test framework for functional testing of services in addition to the existing concept of SDP for service development and provisioning was identified.

Completing this chapter in section 2.4, the stakeholders in value-added service development and provisioning were introduced. It was also depicted how they would benefit from the installation of a novel TCF.

3 The Challenge of Testing Value-Added Services

This chapter introduces the foundations of functional testing and investigates current testing methodologies and research projects with reference to the research field. Based on the identified insufficiencies of the related projects, a list of criteria will be defined which have to be met by the proposed novel test framework.

An introduction into the field of functional testing will be given in section 3.1. Subsequently, the current state-of-the-art testing methodologies will be described and evaluated in section 3.2. Related projects as well as technologies are depicted in section 3.3 and the final section 3.4 summarises the requirements for a novel test framework for value-added services and includes the list of relevant criteria.

3.1 Principles of Functional Testing

The focus on functional testing within this research work requires the understanding of how testing in general is defined.

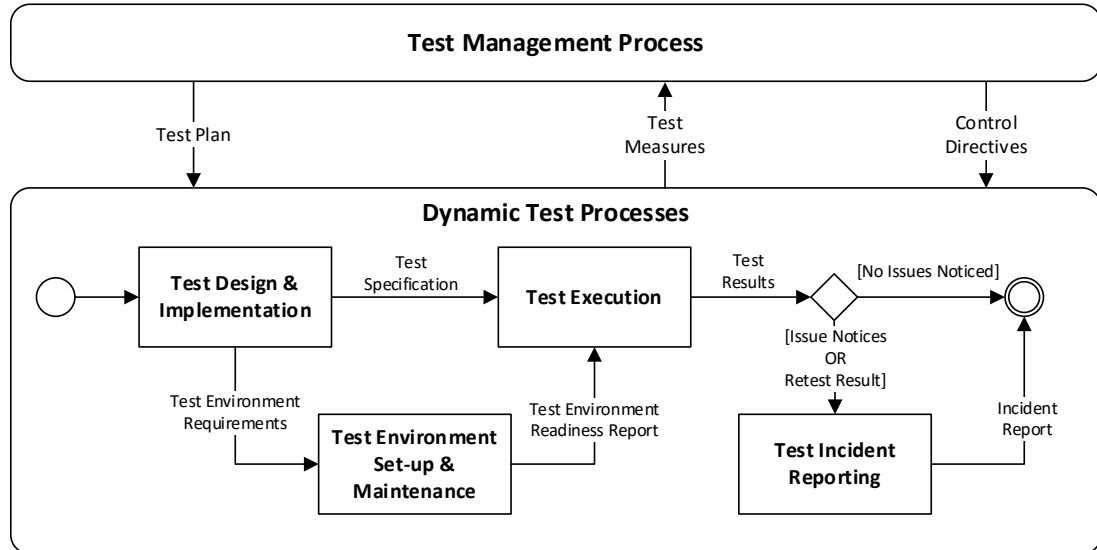
3.1.1 Fundamentals of Testing and Test Processes

It is widely accepted that it is impossible to implement perfect software, and as value-added telecommunication services are software-based, it is necessary to do tests in order to reduce the risk of errors during service development which cause failures when the service is consumed. According to (IEEE Std 610.12, 1990), testing is defined as “the process of analyzing a software item to detect the differences between existing and required conditions (that is, bugs) and to evaluate the features of the software items”. Other sources, such as (Amman and Offutt, 2008), define testing as the process of “evaluating a system by observing its execution”. To sum up, testing can be used to detect failures in the observed system or service, which will be further referred to as System/Service under Test (SUT). The process is carried out by executing defined test cases against the SUT in order to check the system’s behaviour.

When it comes to testing, two independent procedures have to be considered, verification and validation. As far as (IEEE Std 1490, 2011) is concerned, verification is “the evaluation of whether or not a product, service, or system complies with a regulation, requirement, specification, or imposed condition. It is often an internal process”. The definition indicates that “verification” is intended to prove a software (or telecommunication service) meets a set of functional specifications. This set is usually part of a document, the System Requirements Specification (SRS), and is derived from the customer’s demands by members of the development team or a business analyst. In contrast to verification, validation is “the assurance that a product, service, or system meets the needs of the customer and other identified stakeholders. It often involves acceptance and suitability with external customers” (IEEE Std 1490, 2011). The focus of

validation is to confirm that the software (or telecommunication service) will fulfil its intended use. The tests have to be executed by the customers or end-users because they have to accept the behaviour of the software.

Historically, testing was mainly used as debugging to verify that the implemented software performed as intended. There was no separate and well-defined process defined in the software development life cycle. Today, standards exist in order to describe the process of testing. The following Figure 3.1 illustrates how dynamic test processes interact and shows the relationship with the test management process. This methodology is taken from the standard (ISO/IEC/IEEE 29119-2, 2013), a document published by the International Organization for Standardization (ISO), the International Electrotechnical Commission (IEC) and the Institute of Electrical and Electronics Engineering (IEEE).



© 2013 IEEE

Figure 3.1: Dynamic test processes (ISO/IEC/IEEE 29119-2, 2013)

The test management process is an overseeing process that initialises the dynamic test process by delivering the test plan. This document should be based on the IEEE standard

for test plan specification (IEEE Std 829, 2008) and describes the scope of the test, the testing approach and the resources and schedule of intended testing activities. Furthermore, it identifies test items to be tested and test items not to be tested. Test items represent individual elements and can either be a document, a class, a whole program, a component of a system or even the whole system. Besides the triggering of the dynamic test processes, the test management process also monitors the progress (through test measures) and may require further tests (through control directives) to be designed and run until a specified completion criterion is achieved (ISO/IEC/IEEE 29119-2, 2013).

For any specified test, the dynamic test process will execute in the order presented in Figure 3.1. The initial phase, the “Test Design & Implementation”, is used to specify the test specification. Here, the tester as primarily responsible person has to apply one or more test design techniques to derive test cases and test procedures with the aim of achieving the test completion criteria which are defined in the test plan. It is possible that the “Test Design & Implementation” phase is exited and re-entered afterwards, if some additional test cases are required after the first execution of a test procedure. Besides the test specification as output of the phase, relevant test data and test environment requirements are identified by the tester.

A following phase within the dynamic test process, the “Test Environment Set-Up & Maintenance”, is used to establish and maintain the environment in which the specified test cases are executed against the SUT. The person responsible for the maintenance of the test environment may configure a set of parameters that are required for the testing of the specific SUT. If a test environment based on the Testing and Test Control Notation (TTCN-3) (ETSI ES 201 873-1, 2015) is used, the responsible person for instance has to

load the appropriate codecs for the protocol communication with the SUT and has to set the relevant parameters in order to access the SUT. After the setting up of the test environment is finished, all relevant stakeholders are informed through a so-called test environment readiness report.

After the test environment is ready, the “Test Execution” follows. This phase contains the execution of the test procedures generated as a result of the “Test Design & Implementation” phase on the prepared test environment. Although it is not defined explicitly in Figure 3.1, it may be required to perform the execution a number of times as all the available test procedures may not be executed in a single iteration. If an occurred issue is fixed in the SUT, it should be retested by re-entering the “Test Execution” phase. As a result of carrying out this phase, the test results and the test execution log are produced.

The final phase of the dynamic test process is the “Test Incident Reporting” phase that provides the reporting of test incidents. This phase will be entered if test failures were identified, unexpected behaviour took place or if retests passed. The main purpose of the phase is to report the stakeholders emerging incidents which require further action.

With reference to the overall dynamic test process illustrated in Figure 3.1, it should be noted again that it is shown as a pure sequential process, however, in practice it may be carried out in iterative steps (ISO/IEC/IEEE 29119-2, 2013).

3.1.2 Schematic Approach to Functional Testing

The described concept of dynamic test processes can be applied to any particular phase of testing (such as unit, integration, system and acceptance testing) or type of testing (such as performance testing, security testing, usability testing and functional testing) (ISO/IEC/IEEE 29119-2, 2013). The focus of this research work is the functional testing of value-added telecommunication services and one of the major objectives is that both verification and validation are supported by the proposed approach. This section focusses on black-box or specification-based testing is therefore only related to the verification process and not to the validation process.

Functional testing is an essential activity in most software development projects and is also significant during and after the process of developing new value-added services in service provider environments. The term itself describes the process of verifying the functions in a system to assure that they meet the specified requirements. Furthermore, every software system can be seen as a black box, where a tester selects valid and invalid inputs and determines the correct output. In functional testing, a tester does not need to know the internals of the SUT as the focus is to evaluate the functional correctness of a given system, independently of its internal implementation (Pezzè and Young, 2009).

(Pezzè and Young, 2009) describe a schematic approach to functional testing which is presented in Figure 3.2. Permission to reproduce Figure 3.2 has been granted by the authors of the referenced publication.

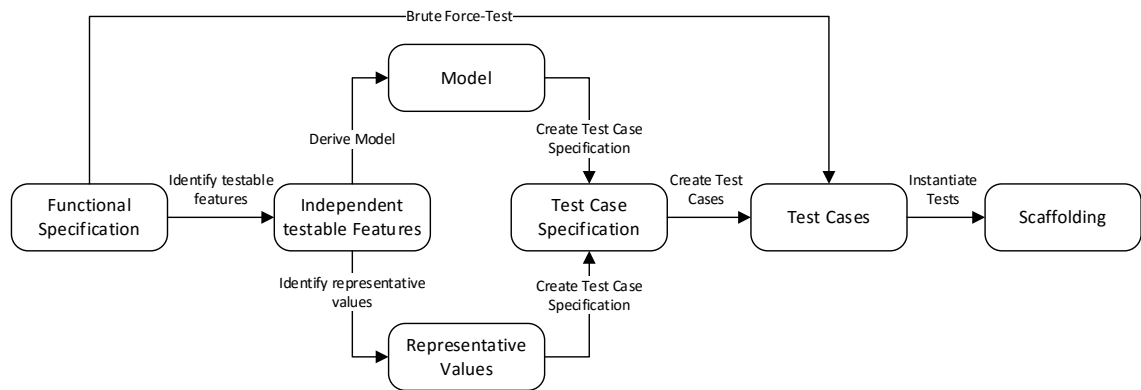


Figure 3.2: Schematic Approach to Functional Testing (adapted from (Pezzè and Young, 2009))

Initially, there is an existing functional specification (see Figure 3.2) describing the requested behaviour of a system or service. It typically contains what is needed by the system/service user as well as all the relevant properties of inputs and outputs. Based on the functional specification, test cases can be directly defined by an experienced test designer using a Brute Force method (Mathur, 2008). Here, the test cases will be created without consideration of any criteria and it is nearly impossible to measure the quality of the test cases. Moreover, the use of the Brute Force method depends only on the expertise of the test designer and it costs him a lot of time to repeat the process later on. Because of these limitations, this process is rather inefficient and ineffective.

Alternatively, a systematic approach can be followed. It simplifies the whole process by separating it into basic automated steps and steps that require intellectual work. The first step in this approach is the identification of the independent testable features (see Figure 3.2) from the functional specification. These testable features are parts of a system or service that can be tested separately. In order to group these features, so-called logical units are defined comprising related use cases. Then, the test designer has to define all possible input parameters for the specified logical units (Pezzè and Young, 2009).

After the logical units based on the independent features have been identified, a test designer can choose two alternative methods to generate a test case specification. Firstly, he can identify an amount of representative values (see Figure 3.2) for each derived logical unit. According to (Pezzè and Young, 2009), these representative values should be inputs for the logical units that are especially valuable. In general, valuable inputs can be identified by choosing representatives of equivalence classes that are apt to fail often or not at all. The equivalence classes can be derived by examining the input conditions from the functional specification. Each input condition induces an equivalence class with valid and invalid inputs. Of course, inputs can also be generated randomly, but this approach is less likely to cover all parts of the functional specification (Gutjahr, 1999). An example application for the representative values approach can be a ZIP code lookup. A user has to input a ZIP code (e.g. “12345”) into a form and the list of cities in the ZIP code are listed after actuating a button. Now, the tester first defines the valid inputs, consequently 5-digit ZIP codes. The representative selected ZIP codes have different impact on the output. The first group returns 0 cities, the second just one city and the third many cities. Afterwards, the invalid ZIP codes are defined, such as empty inputs, ZIP codes with less than 5 digits, ZIP codes with more than 5 digits or ZIP codes that contain characters instead of numbers. For each invalid input, one representative value is selected.

A second approach to derive inputs for the logical units is to derive a formal model (see Figure 3.2) that specifies software behaviour. Such a model can already be a part of a functional specification, but more commonly, the test designer has to create the model by himself. Typical models come as finite state machines (FSM) containing already implicit information of the possible input values. Comparing this method to the alternative identification and definition of representative values, the definition of a formal model has

several advantages. Although the definition of representative values might be easier to handle than a complex formal model, the model is generally much more cost-effective in the long term. It enables flexibility regarding the amount of test cases to be derived and can easily be adapted to possible changes in the functional specification (Pezzè and Young, 2009).

The test case specification (see Figure 3.2) in the systematic approach can be derived by enumerating the input values for each logical unit from the previous step. Afterwards, the input values have to be combined. It must be pointed out that invalid combinations of values have to be eliminated. Depending on the complexity of the functional specification, the derived test case specification can become quite comprehensive. If using the formal model in the previous step, an adequate test selection algorithm has to be chosen to prevent a test case explosion (Pezzè and Young, 2009).

In the next step, the test case specification is converted to an amount of test cases (see Figure 3.2). In order to instantiate the test cases, the appropriate drivers and stubs have to be installed and loaded. This process is called scaffolding (see Figure 3.2). Especially for effective testing of higher level components, scaffolding is required. Afterwards, the functional test cases can be executed against the SUT (Pezzè and Young, 2009).

3.1.3 Relevance for Testing of Value-Added Services

The fundamentals of testing processes and functional testing approaches has been introduced in the previous sections 3.1.1 and 3.1.2. Now, it has to be elaborated why especially value-added services require a distinct approach to testing. The following characteristics have been identified:

1. The provisioning of value-added services in NGNs or SIP-based IP networks is a very difficult and also error-prone task. On the one hand, this has to do with the various service architectures. In principle, a proper consumption of a value-added services does not depend only on the SIP Application Server where the service is deployed on. Furthermore, other servers as part of the SDP might be involved in the service consumption such as web servers, media servers or database servers. So, the SUT can be characterised as distributed and complex which usually requires a thorough testing approach to validate its functionality.
2. According to (Fischer *et al.*, 2011), “the complexity of the protocols for NGN networks poses a vast number of possibilities for mistakes during the development of new services”. Especially the structure of the SIP protocol can get quite complex. In fact, over 60 different headers have been defined and standardised for SIP requests and SIP responses (IETF RFC 3261, 2002). If one of these headers contains errors or misses required fields, the functionality of a service can be affected.
3. Another important aspect which makes testing of services relevant is the heterogeneity of services. Due to the requirements mentioned in chapter 1 of this thesis, the demand for more specialised and individual services keeps growing and has to be fulfilled by the service providers. The development and provisioning of individual services is much more demanding for service developers because they might have to solve issues they are not facing regularly. This oftentimes leads to errors in the service logic.
4. The execution of value-added services might produce unwanted side effects in other service executions. Especially changes in data or state caused by service

invocations can interfere with other service compositions. This produces unwanted changes.

5. Services are often consumed by end-user terminals (such as VoIP phones, smartphones) which have implemented standardised protocol behaviour. This leads to the fact that services have to follow the standards of protocol-specific communication.

Besides these specific reasons for establishing a testing process especially for value-added services there are of course general reasons. A distinct approach to testing, for instance, ensures the quality of the product. To deliver a quality product to customers helps in gaining their confidence.

3.2 Related Work on Current Testing Methodologies

An important aim of this research work is to define a new framework for functional testing of value-added services. For the development of software and services, many state-of-the-art methodologies include the process of testing, such as:

- Test-Driven Development (Karleysky *et al.*, 2006) and (Yenduri and Perkins, 2006)
- Acceptance Test Driven Development (Adzic, 2011) and (Gärtner, 2012)
- Behaviour-Driven Development (Solís and Wang, 2011)
- Model-Based Testing (ETSI ES 202 951, 2011) and (Utting and Legeard, 2006)

The first three methodologies are typical agile testing approaches. Especially enhanced agile concepts have been taken into consideration because they involve the customer at

frequent intervals within the development and test process. This involvement usually has a good impact on the service quality, because misunderstanding between the service provider and the customer can be eliminated quickly. Model-Based Testing (MBT), however, is a standard approach to realise black box testing. One major advantage of MBT over most other testing approaches is the possibility to generate a lot of tests within a short amount of time. Furthermore, MBT approaches enable tests to be linked directly to requirements through the model. So, a traceability of requirements is supported.

All of the approaches will be evaluated in general and regarding their potential to be applied as methodology for this research.

3.2.1 Test-Driven Development

Test-Driven Development (TDD) is an agile software development technique that relies on the repetition of a very short development cycle. It prescribes that test cases have to be programmed before the functional code is implemented that has to pass the tests. The main objectives of TDD are on the one hand to be able to test the software at any time under automation (Karleysky *et al.*, 2006) and to achieve immediate input through the test cases and thereby construct a program (Yenduri and Perkins, 2006) on the other hand. The process of defining test cases prior to the implementation code is termed as “Test First” approach. In traditional software development approaches (such as in the waterfall model), testing is often left “to the end of a project where budget and time constraints threaten thorough testing. TDD systematically inverts these patterns” (Karlesky *et al.*, 2006). The following Figure 3.3 demonstrates the differences between traditional

development and TDD. Permission to reproduce Figure 3.3 has been granted by Springer Publishing.

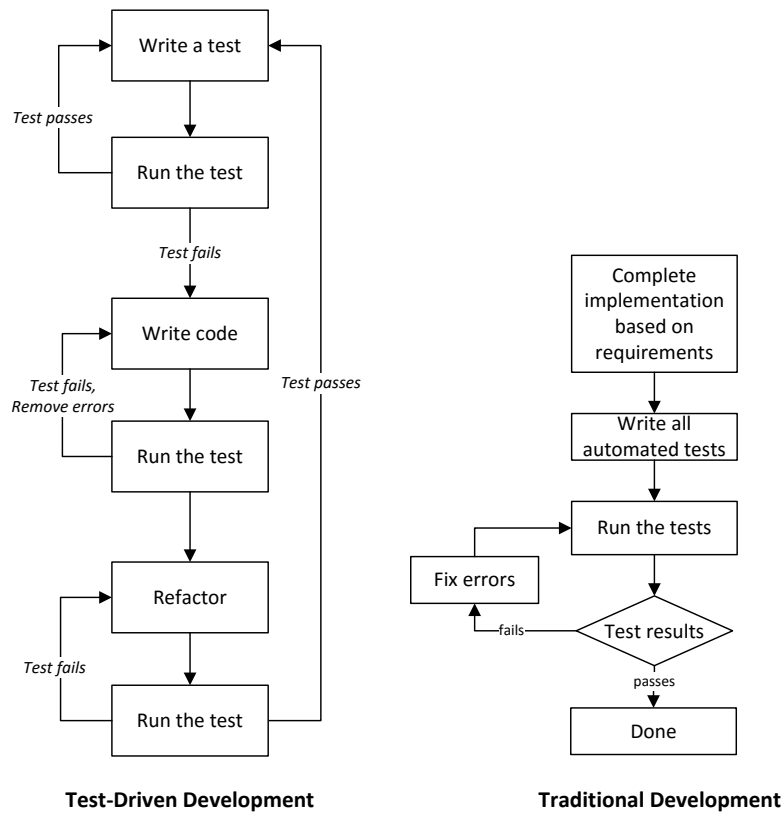


Figure 3.3: Comparison of TDD and Traditional Development (adapted from (Abrahamsson *et al.*, 2005))

The traditional development approach (see Figure 3.3, right part) shows that after completing the implementation phase, all the tests are implemented and executed against the implementation. If the tests fail, the emerging errors have to be fixed until the test execution succeeds with no errors.

Compared to the traditional development, a developer in TDD (see Figure 3.3, left part) initially adds a new test for a piece of system functionality to implement (such as a single function or a method). As there is no implementation present, a first test invocation should fail. Afterwards, the developer writes the implementation code for the piece of system

functionality and restarts the test. The implementation code now has to be reworked until all the tests pass. In a final step after the tests succeeded, the implementation code has to be refactored and tested again. Now, the piece of system functionality is implemented and tested and the developer can continue with other test definitions.

The proponents of TDD claim that it leads to faster development and that the implementation code is of better quality. Developers are forced to implement modular software which makes the implementation code easier to maintain and refactor. TDD makes collaboration between team members easier and more efficient and they can edit each other's code with confidence because the predefined tests will inform them if the changes are making the code behave in unexpected ways.

Some studies come to the conclusion that TDD has several shortcomings or disadvantages such as lack of design (Pancur *et al.*, 2003), problems with applying unit tests, lack of documentation (van Deursen, 2001), reliance on refactoring and dependence on the skills of the developer (or programmer) (George and Williams, 2004). A further limitation of TDD is that the developer and tester is one and the same person. Although the developer can be a highly effective tester, he should not be the tester of the features he has implemented. With a separate tester involved in the process, the tests are much better at finding expectations the developer did not take into consideration. Another negative aspect of TDD is that it is not covering the validation of the software. Even if the functional specification is the amount of test cases constructed by the developer, there is no validation whether the functional specification meets the requirements mentioned by the service customer.

3.2.2 Acceptance Test-Driven Development

Acceptance Test-Driven Development (ATDD) is an agile software requirements specification process that emphasises the automation of acceptance tests as well as the specification of customer-readable requirements through concrete examples. Hence, this approach is also referred to as “specification by example” (Adzic, 2011).

The focus of ATDD is to keep all participants of the development process on the goals of the software project, whether it is the customer, the developers or the testers. During the project, customer-readable requirements are established as well as relating acceptance tests in order to improve the communication between the participants. The collaboration aspect in ATDD is essential in order to produce testable requirements that enable higher quality software more rapidly (Gärtner, 2012).

According to (Gregory and Crispin, 2015), the ATDD life cycle comprises four main activities which have to be managed by the participants or rather stakeholders. The following Figure 3.4 illustrates the dependency between these activities.

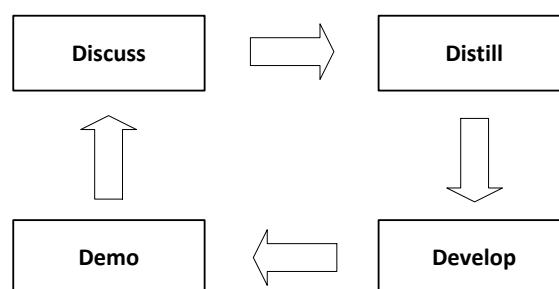


Figure 3.4: Relevant activities in Acceptance Test-Driven Development life cycle

In the initial phase “Discuss”, the customer, testers and developers work together and define tests that outline expected behaviour to a requirement. All possible variants of the behaviour are specified through user stories, concrete scenarios, with clearly defined

input and output. It is important that the customer also understands the documented tests, possibly by using tables of example data (Cohn, 2004).

The participating roles in the second phase “Distill” are the developers and the testers who will transform the documented tests from the previous phase in a format that can be applied to the used test framework. Here, also further tests can be added based on the improved understanding of the project goals.

In the “Develop” phase of ATDD, the concept of TDD is applied. The developer follows the “Test First” approach and executes the defined test from phase “Distill” while implementing the code. Potentially, the developer might find new scenarios that have not been identified before. In that case, the new tests have to be added to the previous set and shared with the other project participants (testers and customers). The role of the tester in this phase is to work with developers to automate the tests. Furthermore, the testers conduct exploratory testing and run acceptance tests.

In the final phase “Demo”, the developers will meet the customer to show them the final implementation containing all the programmed and tested user stories. The customer is able to validate the required functionality by running the tests within a live environment.

As illustrated in Figure 3.4, the ATDD process is applied iteratively. Each iteration step starting from the “Discuss” phase until the “Demo” phase can then contain acceptance tests for specific requirements.

As with TDD, the tests in ATDD are no longer at the end of the development cycle but at the beginning. The focus on defining acceptance tests increases the shared understanding of requirements because they are a product of direct interaction between customers,

developers and testers. Another positive aspect is that the software delivery is now dependent on all acceptance tests passing which also defines the end of a project. Simultaneously, the percentage of passed acceptance tests is a clear indicator of the project progress.

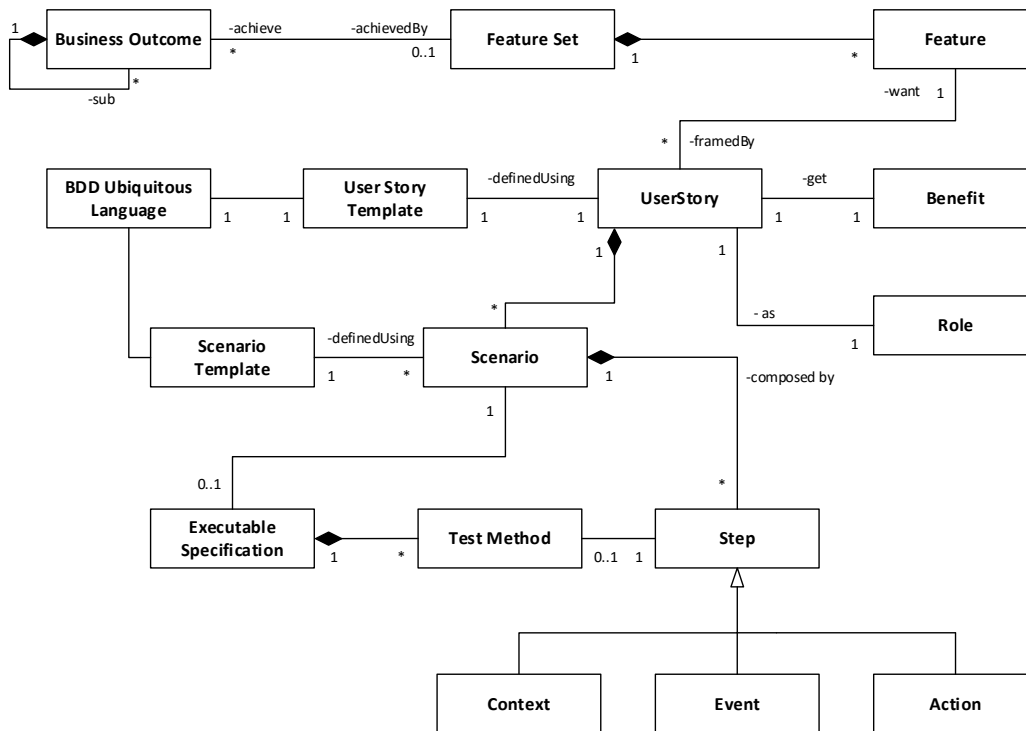
There are also some ATDD drawbacks. First, the process requires the customer to play an active role which might prove to be difficult due to time constraints. Accordingly, the project progress might be slower because of the additional effort. Another problem might be that the developers take part in the process of defining and implementing the acceptance tests. Their influence in the process might end up with a wrong understanding of certain requirements or user stories and therefore also a wrong implementation. Here, the acceptance test will pass but the customer will not get a valid product.

3.2.3 Behaviour-Driven Development

Behaviour-Driven Development (BDD) is an agile software development technique that is generally regarded as the evolution of TDD and ATDD. It focusses on defining fine-grained specifications of the behaviour of the system or service, in a way that they can be developed (Solís and Wang, 2011). BDD has adopted the concept of ubiquitous language from Domain-Driven Design (DDD) (Evens, 2003) that minimises miscommunication and ensures that all stakeholders, such as developers, analysts, testers and managers, are using the same words to describe certain behaviour.

Generally speaking, it is often difficult for developers to find a starting point to communicate with customers during the gathering of requirements for a system. Therefore, the communication should be focused on the business value the system

delivers. However, it is very hard to make business value explicit. In BDD, the initial step is to identify the expected behaviour of a system. This behaviour can be directly derived from the *business outcomes* the system intends to produce (see Figure 3.5). Afterwards, the business outcomes are specified further and *feature sets* are defined. These feature sets contain *features* each capturing a *user story*.



© 2011 IEEE

Figure 3.5: Conceptual Model of Behaviour-Driven Development (Solís and Wang, 2011)

User stories in BDD provide the context of the features delivered by the system. As the name indicates, user stories are user-oriented and describe interactions between users and a system. For one user story, there can be different versions and different contexts. These variations of a user story are called *scenarios*. The specific contexts and outcomes a scenario describes should be provided by the customer. In BDD, the scenarios are used

as acceptance criteria. The described decomposition process should be performed iteratively.

The implementation of acceptance tests in the process can be done by the tester who can lean on the scenario specifications. This process can also be automated because the scenarios are described by means of a ubiquitous language or rather Domain Specific Language (DSL). Figure 3.6 shows an example description of a scenario to login a user on a web site implemented with “Given-When-Then” steps.

```
Scenario: Login Successfully

Given I am on the home page
When I enter the username 'admin'
AND I enter the password 'test'
AND I click 'login'
Then I should be logged in
```

Figure 3.6: Example BDD scenario description

The “Given” part describes the state of the system before the behaviour starts whereas the “When” section actually contains the behaviour. Finally, the “Then” section describes the changes that are expected due to the specified behaviour. In between, concatenations can be realised by using the “And” statement. All in all, the description is quite easy to follow and possibly understandable for each stakeholder taking part in the process (Solís and Wang, 2011).

The concept of BDD has many advantages and also a few drawbacks. As BDD forces the development team to specify the scenarios in collaboration with the customer, it helps to avoid wasted effort by helping teams focus on features that are aligned with business goals. The stakeholders have a “living documentation” throughout the project which

makes it considerably easier to handle changes or extension in the application. Furthermore, as testers are not required to carry out long manual testing sessions before each new release of the application, they can use the automated acceptance tests as starting point. This leads to faster releases and satisfied customers.

A major drawback of BDD is that the process might be very time consuming, especially for the customer who may be unwilling or unable to engage in conversations and collaboration. Another aspect can be poorly written tests by the developers and testers. This drawback is mainly caused by the possible ambiguities that can be specified when the scenarios are described by means of the ubiquitous language. In the middle term, this aspect leads to higher test-maintenance costs.

3.2.4 Model-Based Testing

Model-Based Testing (MBT), also known as Model-Driven Testing, means that testing is based on some form of a formal (computer-readable) model that describes the desired behaviour of the system to be tested. After the formal model is complete, tests can be generated from it by means of an automatic or semi-automatic approach.

According to (ETSI ES 202 951, 2011), the methodology in MBT (see Figure 3.7) starts with a test designer receiving a set of requirements of the system to be tested, generally given in a specification written in natural language. Then, the test designer authors a model using a specific modelling notation that fulfils the requirements stated in the document. The model encodes the requirements and specifies the aspects of the functional behaviour and the relevant interfaces via which these are tested.

Afterwards, the model is utilised for the purpose of test case generation by adding or choosing test selection criteria, e.g. coverage goals. It is necessary to specify the test selection in order to reduce the amount of test cases that will be derived from the model. Then, an abstract test suite is automatically generated that complies with the chosen test selection criteria. In order to enable test execution against the SUT, the abstract test suite may need to be adapted. Permission to reproduce upcoming Figure 3.7 has been granted by ETSI.

Figure 3.7 has been removed due to Copyright restrictions.

Figure 3.7: Model-Based Test Development (adapted from (ETSI ES 202 951, 2011))

In MBT, two different testing approaches exist, either offline or online testing. In offline testing, the test generator is not connected to the SUT and the generated test suite can be executed against the SUT after it has been built completely. There is also the possibility to optimise the test suite after its creation. In online MBT, the test generator and the SUT are connected and all commands are executed directly on the SUT. Here, the test cases are usually generated and executed one after another which does not allow further optimisations of the test suite (Utting and Legeard, 2006).

The MBT methodology in Figure 3.7 continuously delivers feedback for the involved artefacts on multiple levels. Firstly, the process of creating the model provides feedback for the consistency of the system specification. This can be measured before any test is executed. Secondly, the examination of generated test cases and feedback from model analysis can reveal certain issues either in the system specification or the model. Thirdly, issues can be found in the SUT, in the system specification and in the model when the tests are finally executed (ETSI ES 202 951, 2011).

Regarding the evaluation of MBT, several advantages and disadvantages exist. One of the most attracting benefits of MBT is that it automatically generates relevant test cases from the formal model so a better test coverage is guaranteed. The higher level of abstraction in the model helps to concentrate on the right things as the irrelevant details are hidden. Another positive aspect of models is that they can be visualised easier than code. Several studies such as (Pretschner *et al.*, 2005) and (Baker *et al.*, 2007) show that MBT works better at detecting faults in SUTs than manually designed tests. However, this ability depends on the skills and experience of the test designer. A further advantage of MBT is traceability throughout the whole process. Each test case can be related to the model, to the test selection criteria and even to the informal system specification.

Besides the advantages of MBT, several limitations have to be faced. Firstly, the concept of MBT is not an agile method, so it follows the methodology of traditional test development. It might be difficult for the developer to figure out all the errors in the system because there has not been an iterative process. Furthermore, as requirements of customers can change, also the informal requirements the model is based on might become out of date. In this case, a wrong model will be built and the test case execution

will yield a significant amount of errors (Utting and Legeard, 2006). Another disadvantage of MBT is that its quality is totally depending on the skills of the test designer to build models. He must be able to abstract and design the models and has to be an expert in the application area. This requires training costs and an initial learning curve when starting to use MBT (Utting and Legeard, 2006). A further well-known issue of MBT is the state space explosion. Models of any non-trivial system functionality can grow beyond manageable levels. In this scenario, all tasks within MBT are affected such as model maintenance, checking, reviewing and non-random test generation (El-Far and Whittaker, 2001).

3.3 Related Work on Current Research Projects on Functional Testing

In this section, related testing approaches and current research projects in the field of testing are introduced. It shall be analysed whether the solutions can be applied in order to verify and validate value-added telecommunication services. Also advantages and disadvantages of the approaches are discussed.

3.3.1 UML 2.0 Testing Profile

The UML 2.0 Testing Profile (U2TP) defines a modelling language for designing, visualising, specifying, constructing and documenting artefacts of test systems and is an Object Management Group (OMG) standard (OMG, 2013a). According to (Zander *et al.*, 2005), U2TP can be applied to test systems in various applications and can be either used

stand alone for the handling of test artefacts or in an integrated manner with UML 2.0 (OMG, 2011a) for handling of both system and test artefacts. In principal, U2TP enhances UML 2.0 with test-specific concepts such as test architecture, test behaviour, test data and time concepts (see Table 3.1). Permission to reproduce Table 3.1 has been granted by Springer Publishing.

Table 3.1: Overview of the UML 2.0 Testing Profile concepts (Zander *et al.*, 2005)

Architecture concepts	Behaviour concepts	Data concepts	Time concepts
SUT	Test objective	Wildcards	Timer
Test components	Test case	Data pools	Time zone
Test context	Defaults	Data partitions	
Test configuration	Verdicts	Data selectors	
Arbiter	Test Control	Coding rules	
Scheduler			

The test architecture group covers the concepts related to test structure and test configuration such as specifying test components, their interfaces, possible connections among test components and between test components and SUT. The test behaviour group embodies dynamic aspects of test procedures and addressing observations and activities during the test. Test behaviours can be defined by any behavioural diagram of UML 2.0, e.g. interaction diagrams or state machines. The test data group includes concepts for specifying test data used in test procedures, such as the structures and meaning of values to be processed in a test. Finally, the time group covers concepts for a time quantified definition of test procedures, e.g. the time constraints and time observation for test execution (Zander *et al.*, 2005) (OMG, 2013a).

Based on the U2TP concepts, a given design model specified in UML notation can be extended with test-specific information. According to (Dai *et al.*, 2004), a tester first has

to define a new UML package as the test package of the system. Then, he imports the implementation code, all classes and interfaces, and starts with the specification of the test architecture and test behaviour. Oftentimes, the test data and time aspects are already comprised in either the test architecture or test behaviour. In order to define a complete test model, the following steps within the test architecture have to be performed:

1. Assigning the system components to be tested (SUT).
2. Depending on their functionality, the test components have to be defined. The test components should be grouped to the system components of the design model.
3. Specifying a test suite class that lists the test attributes and test cases as well as test control and test configuration.

Besides the test architecture definitions, the test behaviour includes the designing of the test cases. Here, the given interaction diagrams of the design model can be reused, but the instances have to be assigned with stereotypes of U2TP according to their functionalities. At each test case specification, verdicts (such as pass, fail or inconclusive) have to be assigned.

As soon as the test model is final, the test cases still have to be generated and executed against the SUT. Actually, U2TP provides two mappings towards test execution environments or rather technologies, either JUnit (JUnit, 2015) or TTCN-3 (ETSI ES 201 873-1, 2015). Both technologies allow the execution of tests and deliver an evaluation report.

In the following, an example test case specification using U2TP is demonstrated. The specific test case concerns the use of a bank Automated Teller Machine (ATM), especially the verification of an entered pin number. It shall be tested how the ATM reacts

if a wrong pin number is entered after a bank card is inserted. In order to test this, a “hardware emulator” (HW Emulator) is defined as test component as well as an external component “current” which determines whether the pin number is correct for the given bank card or not. The test case specification is illustrated as a UML sequence diagram in the following Figure 3.8. Permission to reproduce Figure 3.8 has been granted by OMG.

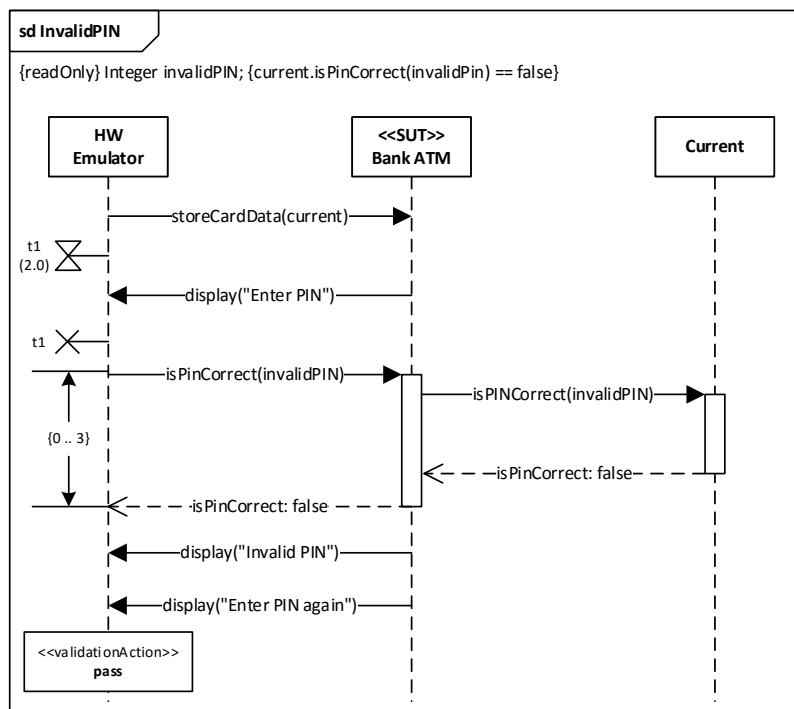


Figure 3.8: Example test case specification with U2TP using a UML sequence diagram (adapted from (OMG, 2013a))

A significant information within the sequence diagram is defined as soon as the HW Emulator is terminated, the so-called validation action. If the two messages “Invalid PIN” and “Enter PIN again” are displayed on the HW Emulator after the pin number has been checked, the test case passes.

The proponents of U2TP claim that the approach is standardised by the OMG and is based on standard UML notations. Actually, U2TP has a lot of features, such as the support for

domain-independent test modelling, test case specification and test data specification. The close connection with UML enables also the combination with other standardised profiles like the Systems Modeling Language (SysML) (OMG, 2012a) or the Service oriented architecture Modeling Language (SoaML) (OMG, 2012b).

The negative aspects of U2TP correlate with the negative aspects of Model-Based Testing (MBT). A lot of training is required for the testers to build adequate U2TP test models with complete test architecture and test behaviour configurations. As the domain model and the test model are closely coupled, possible changes triggered by the customer of a system or service lead to changes in the domain model and in the test model. Another restriction of U2TP is that many important aspects relevant for testing are not covered, such as a proper test management, audits and reviews or a thorough test methodology. Finally, agile concepts cannot be applied easily, as the system model always has to exist before the test model can be defined.

3.3.2 TT-Medal Test Platform

The Information Technology for European Advancement (ITEA) project TT-Medal (TT-Medal Consortium, 2005) has focused on developing the methodologies, tools and industrial experience in order to allow the testing process of software intensive systems to be made more effective and efficient. Special accentuation has been given to standardised test technologies and notations, such as TTCN-3 and U2TP. Because of its high maturity, wide applicability and existing tool support, TTCN-3 became the central focus of the TT-Medal project.

Although TTCN-3 is a powerful testing technology, in isolation it only provides one piece of a complete testing solution. TT-Medal proposes a tool chain that supports testers during all phases of the testing process. Its resulting toolset is called TT-Medal test platform and is illustrated in Figure 3.9. Permission to reproduce Figure 3.9 has been granted by the copyright owner of the white paper.

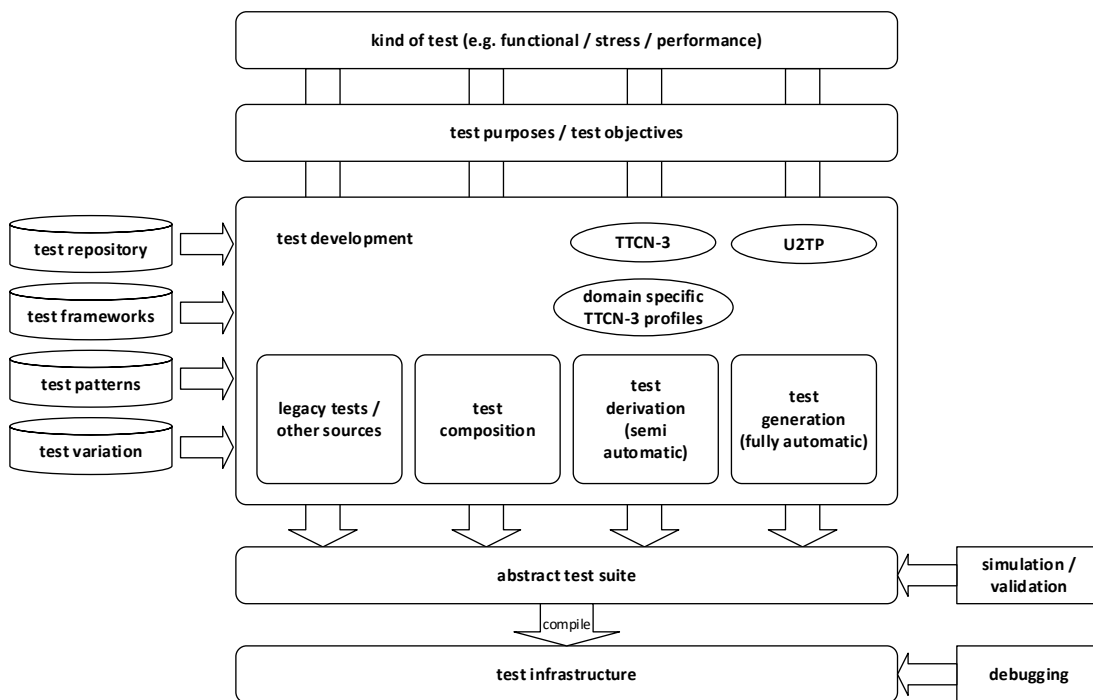


Figure 3.9: TT-Medal test platform (TT-Medal Consortium, 2005)

TT-Medal considers requirements from the automotive, railway, financial and telecom domains to find and demonstrate test solutions that are all based on one standardised test notation. Furthermore, it defines a TTCN-3 test infrastructure that focuses on the test execution phase and which is applicable for all the mentioned industrial domains. Furthermore, it included approaches for the development of tests within the test infrastructure. However, these methods are not further specified in the given sources. (TT-Medal Consortium, 2005) only state that existing test generation tools from external

sources, or imports and mappings from other specification and programming techniques have been adopted. The resulting toolset is called TT-Medal test platform and is illustrated in. Permission to reproduce Figure 3.9 has been granted by the copyright owner of the white paper.

The test platform offers components dedicated to the synthesis, validation and analysis of tests. First, tests can be developed along different types, such as functional, interoperability, performance or load tests. Then, specific purposes (or rather test objectives) are assigned to the testing types. TT-Medal supports tests that are specified in either TTCN-3 or U2TP. The results of the test development are abstract test suites (ATS) that need to be compiled into a target programming code (e.g. Java) before the tests can be performed using a test management tool (TT-Medal ITEA, 2005). In TT-Medal, the tools for compiling (TTthree) and execution (TTman) are integrated within an Eclipse-based TTCN-3 toolset, the TTworkbench (TTworkbench, 2015). The results of the test execution using the test manager are test logs, which are the basis for determining the final test results. They can be visualised using diverse presentation formats.

The advantages of TT-Medal are the use of standardised tools such as TTCN-3 and U2TP and the focus on a wide spectrum of support for diverse domains, even for the telecom domain. However, the main target of TT-Medal is not to deliver thorough methodologies for all domains to help deriving test cases from requirements specifications, but to support certain domain-specific protocols. As an example, a SIP protocol codec was implemented for TTCN-3 in order to realise SIP conformance testing. Furthermore, the project results of TT-Medal aimed more on delivering a training and experience package. The training aspect should show project partners and others in European industry how to use new

testing technologies (TTCN-3) to effectively test their business process. The experience aspect in contrast should describe where the technologies should be applied and why. Finally, as with U2TP, TT-Medal does not support agile testing concepts.

3.3.3 Fokus!MBT Test Modelling Environment

Fokus!MBT (Fokus!MBT, 2015) is an integrated test modelling environment that supports test model authoring by guiding the tester through methodology-specific support (Wendland *et al.*, 2013). Fokus!MBT uses U2TP as language for expressing test models. Its main goal is to provide domain and testing experts with an integrated modelling environment helping them to perform their work quickly, easily and free of errors. Especially in the area of MBT, authoring tools are important in order to avoid the domain experts of getting easily frustrated with the complexity or the granted degree of freedom a modelling tool might provide.

The Fokus!MBT test modelling environment is illustrated in the following Figure 3.10.

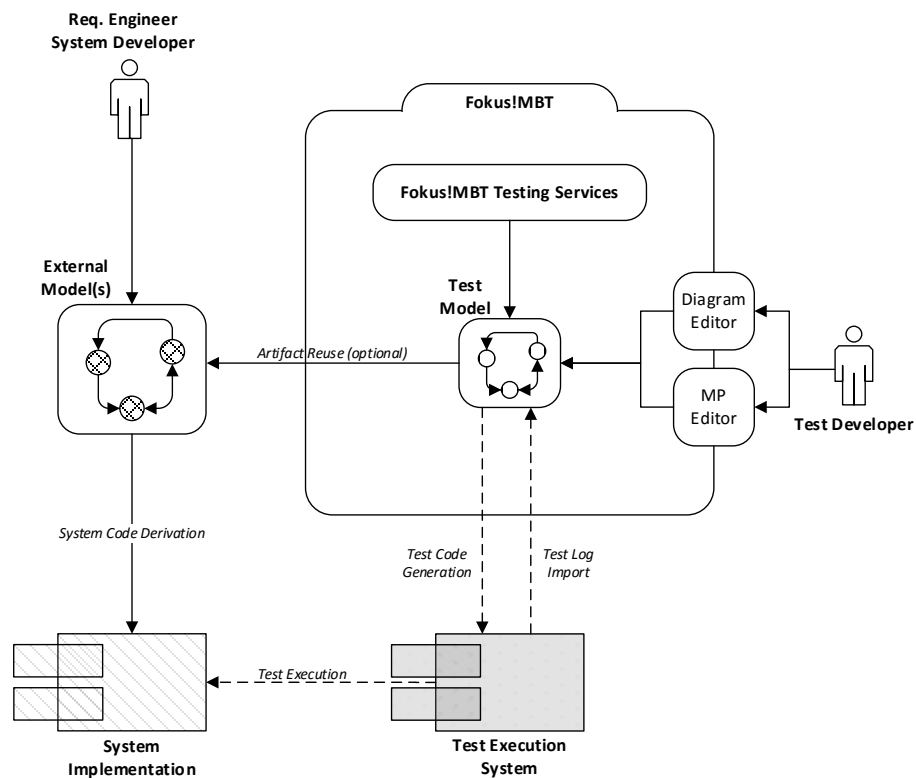


Figure 3.10: Fokus!MBT test modelling environment (Wendland *et al.*, 2013)

Permission to reproduce Figure 3.10 has been granted by ACM (Association for Computing Machinery, Inc.). Within the approach, a separate test model is created and authored independently from the system specification. This separate test model is created by a test developer by means of a Diagram Editor or a MP Editor. So, the test model can be either based on a formal model or a document. The compilation process of the test model can be simplified if an existing system model (Artifact Reuse) is available and accessible, because Fokus!MBT allows the reusing of certain aspects of the system. This aspect shows the similarities the approach has to the standard U2TP approach. However, Fokus!MBT seems to be more flexible as it enables the inclusion of external testing services, such as test case or test report generators. The test code generation, for instance, is a service that is part of the Fokus!MBT (Testing) Services environment. The test execution system is not specified any further.

Generally speaking, Fokus!MBT is designed to be flexible enough to be integrated into various testing tools and process landscapes. It can be decomposed into a core component that is framed by three logical layers which is shown in Figure 3.11. Permission to reproduce Figure 3.11 has been granted by ACM.

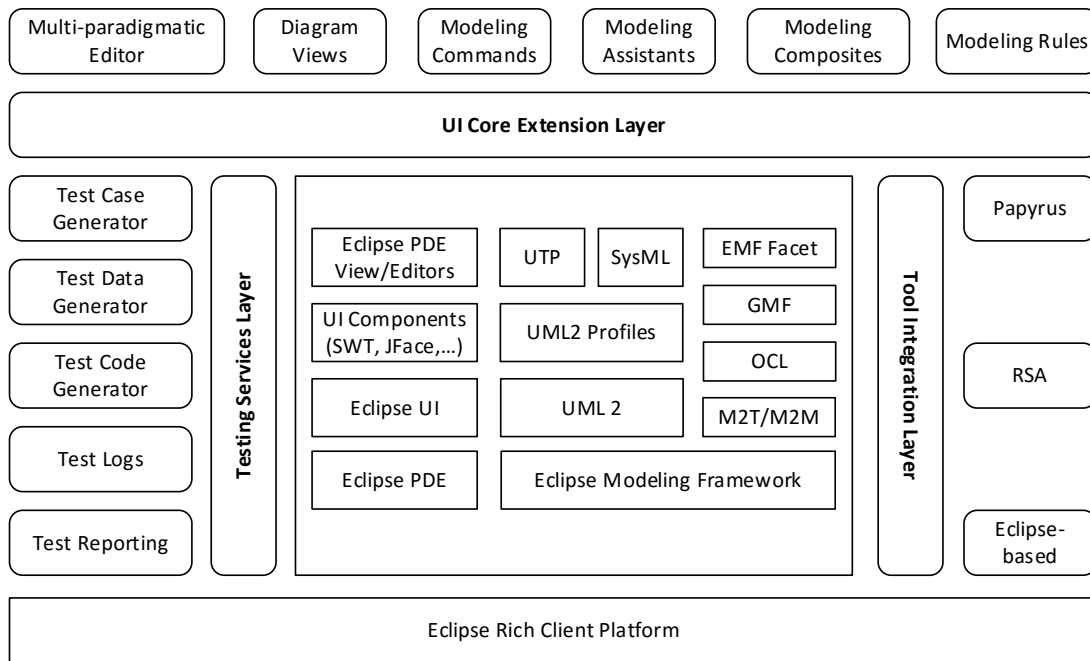


Figure 3.11: Architecture and technology stack of Fokus!MBT (Wendland *et al.*, 2013)

It should be mentioned at this point that the authors chose to illustrate two of the three layers vertically. The core component relies on the key technologies and provides fundamental capabilities for implementing and registering test-related services, certain UI extensions as well as the integration with Eclipse-based modelling environments. The main task of the core component is to guarantee that both the syntactical and semantical methodology is respected. The logical layers encapsulate technologies and concepts that are specific to concrete services and implementations of modelling environments. The purpose of the three layers is as follows (Wendland *et al.*, 2013):

- Testing Service Layer – Here, testing-related interfaces are provided as services, such as test case generation or test report generation.
- UI Core Extension Layer – The main objective of Fokus!MBT is to provide an authoring system for the test developers and engineers. Here, several service extension points are defined which realise the idea of multi-paradigmatic test modelling. Furthermore, the service extension points allow tailoring the UI for different purposes and stakeholders.
- Tool Integration Layer – Any modelling environment-specific implementation can be encapsulated from the core component.

The focus of Fokus!MBT to simplify the test process for test developers and test engineers is promising. Especially MBT approaches are oftentimes not embraced by the testers because of the missing support of the modelling tools. The aspect of highest possible flexibility in order to integrate the platform into a given process landscape can also be estimated positively, however, this is also fraught with risk. The tools behind important testing-related interfaces might not be supported or even error-prone. All in all, Fokus!MBT is currently developed in the third generation and still has to undergo a lot of changes. The methodology has not yet proven to be feasible as there is no published case study so far. Another problem of the approach is that it intends to be an overall solution for any present problem in model-based testing. This flexibility is, however, a problem as there is no standard approach or case study published demonstrating the deriving of test cases or the execution of test cases against a SUT.

3.3.4 ComGeneration

The ComGeneration approach described in (Wacht *et al.*, 2011a) and (Wacht *et al.*, 2011b) defines a methodology that can be specifically applied to functional testing of value-added services. It considers both service implementation and functional testing of the service (see Figure 3.12). Permission to reproduce Figure 3.12 has been granted by the copyright owners of the referenced publication.

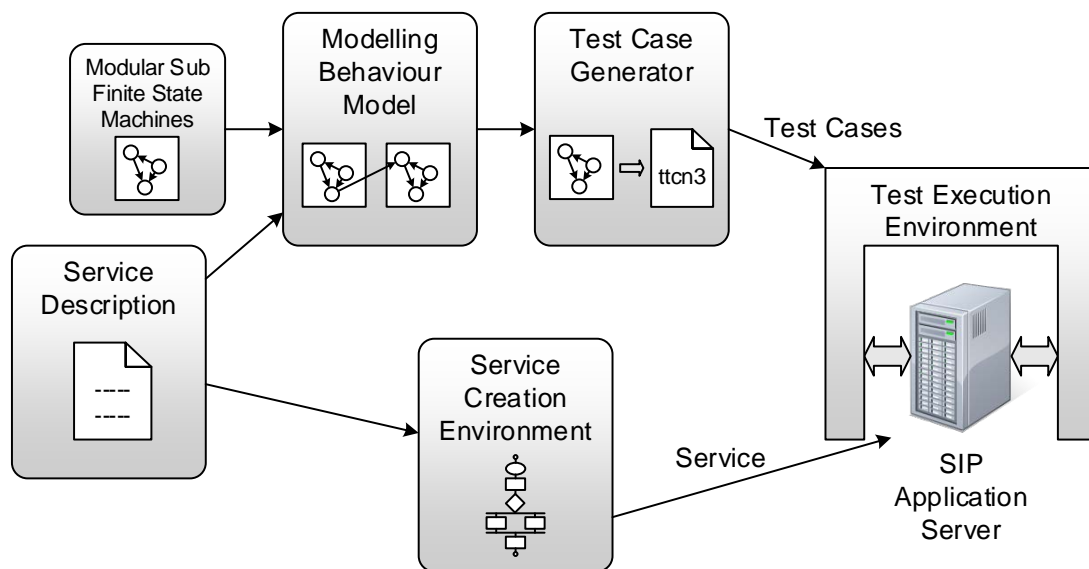


Figure 3.12: ComGeneration methodology (Wacht *et al.*, 2011b)

Initially, a so-called service description is specified. This is a natural language-based document that can be understood as the requirements specification and is created by the service provider in consultation with a service customer.

After the service description document has been completed, both service developer and test developer can start with the development. The service developer can use a Service Creation Environment (SCE) to develop and subsequently deploy the service on a SIP Application Server within a Service Delivery Platform (SDP).

Just like the service developer, the test developer has to extract the relevant service information for the test purpose from the service description. First, he chooses the service-related characteristics out of a repository of so-called predefined modular extended finite state machines (EFSM). The EFSMs cover basic service characteristics like protocol sequences for SIP (IETF RFC 3261, 2002) or HTTP (Hypertext Transfer Protocol) (IETF RFC 2616, 1999). By composing the chosen predefined modular EFSMs, the test developer creates a behaviour model describing the behaviour of a value-added service. Once the behaviour model is created, it is passed to the Test Case Generator which contains an algorithm to automatically generate the service-specific abstract test cases by identifying every possible path through the EFSM. Afterwards, the abstract test cases are converted into TTCN-3 test cases by using a special mapping concept (Wacht *et al.*, 2011a). Finally, the TTCN-3 test cases are combined to a test suite that can be executed within a TTCN-3 test execution environment. In the approach, the TTworkbench (TTworkbench, 2015) was used.

In the following, an example demonstrates how the behaviour model is established. The service to be tested is a standard Click-to-Instant-Message (Click2IM) service. The input is a SIP URI and a text message. Both have to be set on a web site. By actuating a “Send” button, a SIP message will be send to the entity that is reachable through the SIP URI. The message, of course, contains the specified text message from the web site. In the first step to establish the behaviour model, the test developer has to configure the test architecture through the so-called Connectivity Editor (see Figure 3.13). Permission to reproduce Figure 3.13 has been granted by the copyright owners of the publication.

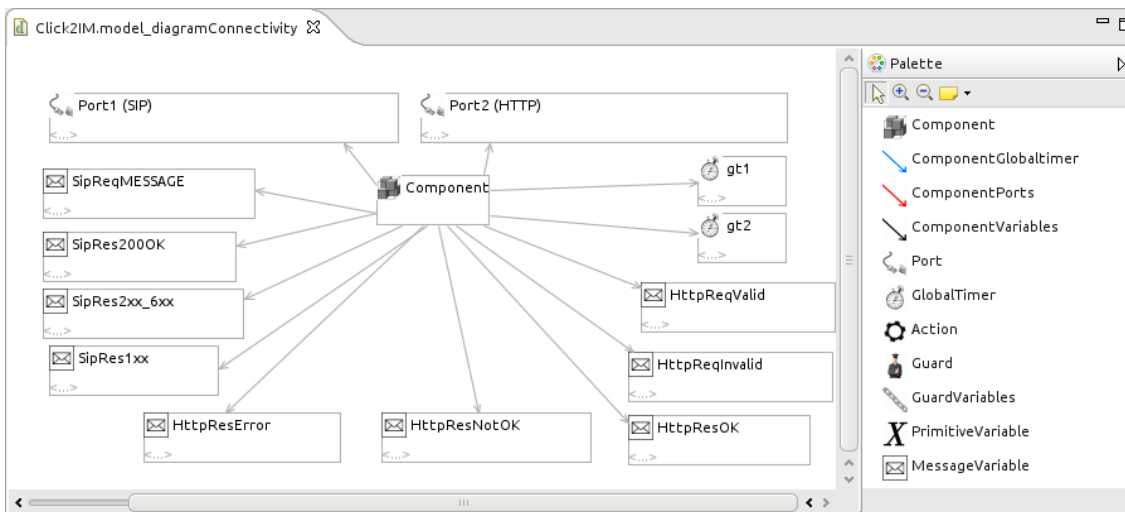


Figure 3.13: Connectivity Editor for Click2IM service (Wacht *et al.*, 2011b)

The Connectivity Editor contains the protocols that have to be used (SIP, HTTP), specifies certain timers that have to be integrated and includes all required messages (SIP requests and responses as well as HTTP requests and responses). The example also considers certain mistakes a service customer could do while using the service, such as forgetting to include the SIP URI in the text field. In the approach, such considerations require to specify long lists of defined possible messages. In the next step, the messages have to be further specified. In the approach, complex data types have been defined that represent example protocol messages. By using a so-called Test Data Editor, the test developer can determine the test data for each defined message. The following Figure 3.14 shows how a SIP request message can be specified through a tree-like view. Permission to reproduce Figure 3.14 has been granted by the copyright owners of the referenced publication.

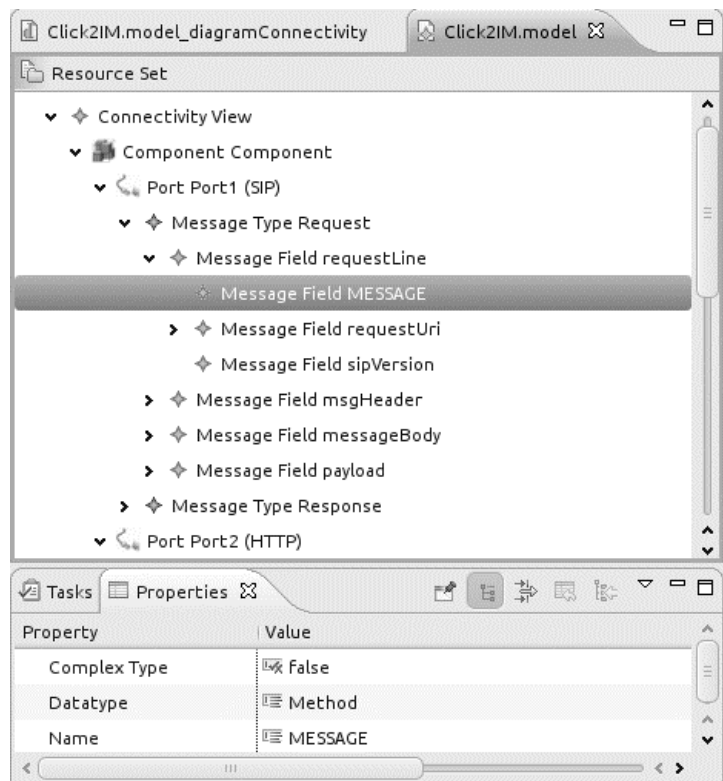


Figure 3.14: Tree-like Test Data Editor (Wacht *et al.*, 2011b)

The final step of the modelling process is the design of the behaviour model itself. Here, an EFSM is applied and the main information is included on the transitions (see Figure 3.15). Permission to reproduce Figure 3.15 has been granted by the copyright owners of the referenced publication.

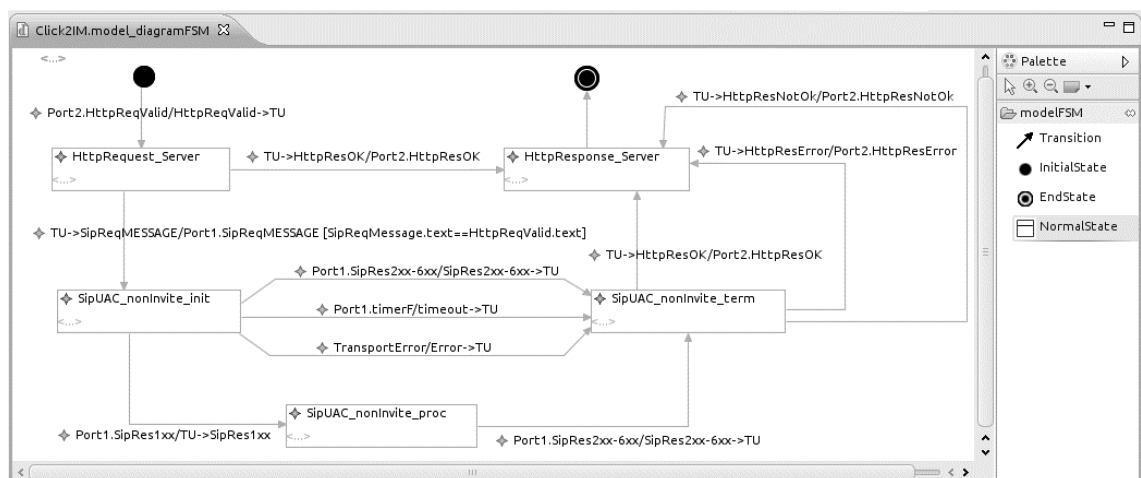


Figure 3.15: Behaviour Model for Click2IM service (Wacht *et al.*, 2011b)

The illustrated Behaviour Model in the ComGeneration approach contains five states, each representing a predefined EFSM, either SIP-based or HTTP-based. The transitions between the states either describe events that might occur or specify actions that take place as soon as the events happened. Finally, based on the complete EFSM-based Behaviour Model, a test generation transition coverage algorithm has been implemented to derive test cases and execute them against the value-added service running on a SIP AS.

To the author's knowledge, the described methodology of the ComGeneration project is the only test approach that has been specifically applied to the field of NGN or rather SIP-over-IP-based environments. It can be understood as the foundation of this research. A few aspects of the ComGeneration project have been adopted in this research, however, mainly aspects the author has established during the project work. In this connection, publications have been done, such as (Wacht *et al.*, 2010), (Wacht *et al.*, 2011a), (Wacht *et al.*, 2011b) and (Wacht *et al.*, 2011c).

Regarding the ComGeneration approach, the separation of the development process and the test process enables a thorough verification based on the service description. Unfortunately, the role of the service customer in this approach is only relevant at the project start. This leads to the question, if the ComGeneration approach also validates the value-added services. Furthermore, the approach lacks an efficient test case derivation algorithm from the behaviour model to avoid the well-known combinatorial explosion issue in EFSMs. Also in the ComGeneration approach, the agile concepts have not been considered.

3.3.5 Telling TestStories

(Felderer *et al.*, 2010) and (Felderer *et al.*, 2011) describe a tool-based methodology for model-driven system testing of service-oriented systems called Telling TestStories (TTS). This methodology is based on a separated system and test model.

Figure 3.16 shows the existing system and testing artefacts of the methodology. The informal requirements are written or non-written capabilities or rather properties of the system whereas the SUT provides the services that are callable by the test controller. The test execution is not discussed in the papers as it is not a focus of the methodology.

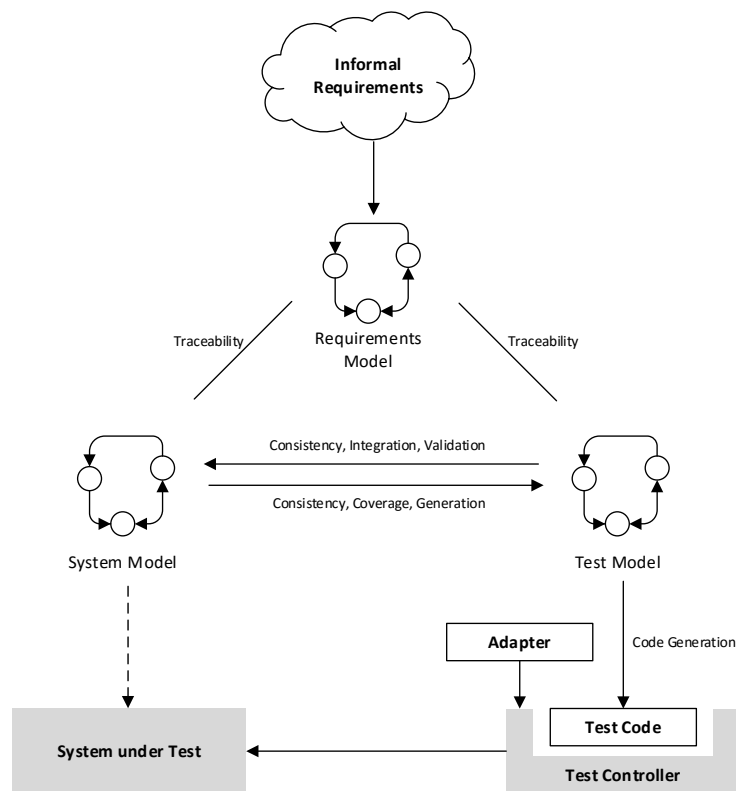
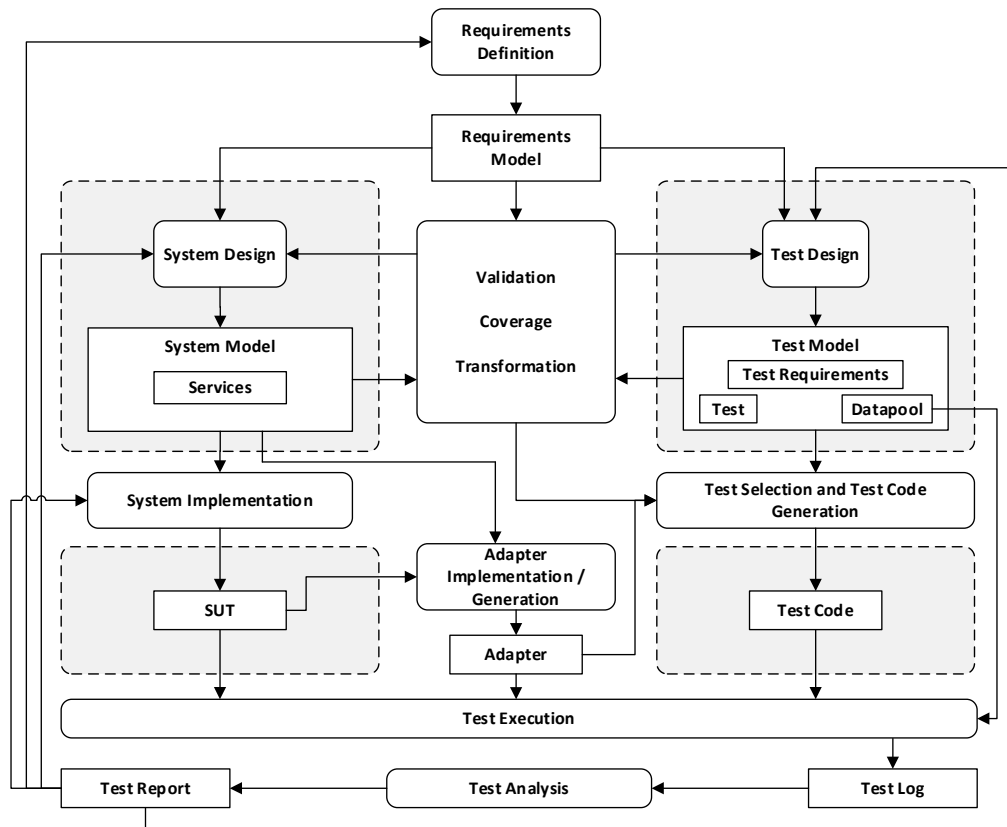


Figure 3.16: TTS artefacts overview (Felderer *et al.*, 2010)

The requirements model contains the requirements for both system development and testing. It provides a way to integrate the textual descriptions of requirements that are

needed for communication with non-technicians into a modelling tool. The system model describes the structure of the system and its behaviour in a platform independent way. Its static structure is based on the notions of services, components and types. Each service operation call is assigned to specific use cases, actors correspond to components providing and requiring services. The domain types also correspond to types. (Felderer *et al.*, 2011) assume that “each service in the system model corresponds to an executable service in the running system to guarantee traceability”. The test model in contrast defines the test data and the test scenarios as so called test stories. The concept of the test stories mentioned can be compared to user stories in ATDD or BDD. The test stories are controlled sequences of service operation invocations exemplifying the interaction of components. Principally, test stories can be modelled by means of UML sequence diagrams or activity diagrams. To gain traceability between the requirements and the test model, each test story has to be linked to a requirement (Felderer *et al.*, 2011).

In Figure 3.17, the model-driven testing process of TTS is presented. It consists of a design, validation, execution and evaluation phase and is processed iteratively.



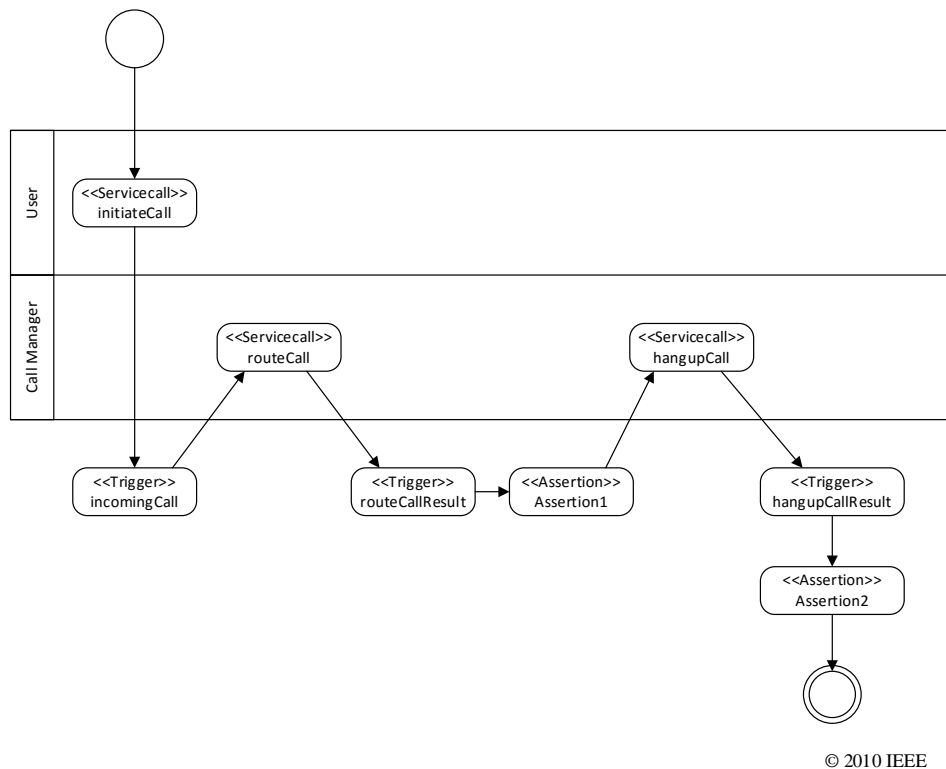
© 2011 IEEE

Figure 3.17: Model-driven Testing Process (Felderer *et al.*, 2011)

Initially, from the defined requirements, the system model containing services and the test model containing tests are designed. The test design includes the data pool definition as well as the definition of the test requirements. The system model and the test model can be checked for consistency and coverage based on Object Constraint Language (OCL) (OMG, 2014) queries. This enables an iterative improvement of both the system and the test model quality. The methodology does not consider the system development itself but is based on services offered by the SUT. As soon as adapters are available for the system services, the process of test code generation can take place. Subsequently, the generated test code is automatically compiled and executed by a test execution engine which also logs occurring events into a test log. The test result can be used to validate the

system model. Finally, a test analysis tool realises the test evaluation and generates the relevant test reports (Felderer *et al.*, 2010) (Felderer *et al.*, 2011).

A practical example of a test story can be, for instance, the routing of a call. This test story is a sequence of activities and is illustrated as an example (see Figure 3.18).



© 2010 IEEE

Figure 3.18: Test story of routing a call (Felderer *et al.*, 2010)

After the call is initiated by the user, the service routes the call and terminates it. The results of these calls are triggered on the test controller. The assertions check whether the result provided matches the expected one (Felderer *et al.*, 2010).

The authors have defined a methodology that includes a very practical form of specification, test stories or rather user stories. An advantage of this approach is that theoretically, the traceability between the test model, the system model and the requirements model is provided. However, the authors do not mention the test case

derivation and they do not follow a standardised approach to execute tests. In fact, the technologies RMI (Remote Method Invocation) (RMI, 2015) and CORBA (Common Object Request Broker Architecture) (OMG, 2012c) have been applied. Unfortunately, there is no standardised test execution technology used such as JUnit or TTCN-3. This makes it difficult to maintain the environment or to enhance it.

3.4 Requirements for a New Optimised Solution for Functional Testing of Value-Added Services

The previous sections 3.2 and 3.3 introduced current testing methodologies as well as related projects in the field of functional testing. All of these approaches include their relative strengths and weaknesses that have been mentioned at the end of each section. The target of this chapter is to derive requirements for a new optimised solution in order to do functional testing of value-added services. This means that these requirements will also be the basis for the proposed Test Creation Framework (TCF) which will be presented in chapter 4.

- Test Execution – The fully automatic execution of tests is one major requirement that has to be met by any test framework. The test developer does not have to do any manual actions.
- Test Report – The test execution environment shall deliver a thorough test report that a test developer can interpret easily.

Regarding the related projects, the TT-Medal Test Platform as well as the ComGeneration approach use the TTworkbench, a TTCN-3-based test execution environment in order to

execute tests. The TTPworkbench also supports the generation of report results. The Telling TestStories project includes a proprietary approach to execute tests based on CORBA and RMI. U2TP does not support test execution and Fokus!MBT requires an external tool that is not specified.

A further criterion which could be derived from agile approaches such as ATDD and BDD is that the needs of the service customer should always be the centre of attention in the testing and development process. Based on this aspect, the following requirements can be determined by means of keywords:

- Collaboration and support for agile principles – An optimised solution is required to integrate all stakeholders in the test process, such as service developers, test developers and service customers.
- Comprehension – All stakeholders shall always have the chance to get an overview of the project progress (both testing and development), especially the service customer if he is interested.

None of the current related projects directly support the collaboration or comprehension. The ComGeneration approach lets the service customer participate in the compilation of the Service Description, a contract document between the service customer and the service provider. The Telling TestStories project includes the compilation of test stories which can be compared to approaches in ATDD and BDD. So, a minimal support for agile principles can be identified.

The next set of requirements refers to the usability of a test framework. Here, the following keywords have been defined:

- Manageability and time exposure – It is important that the framework concepts and methodologies do not overburden the stakeholders and are quite easily manageable in a reasonable timeframe.
- Tool support – The framework shall provide tools especially for the test developer to maintain the test process.

U2TP is very well documented as it is also a test specification standard and directly connected to UML. There is also a tool that uses components of U2TP (e.g. Eclipse Test & Performance Tools Platform Project) (Eclipse TPTP, 2015). For the ComGeneration approach, an Eclipse Modeling Framework (EMF)-based solution exists, but it lacks relevant documentation. The use of the tool is manageable, but not straightforward. Fokus!MBT is very complex, as it involves many types of applications depending on the functionality to apply (e.g. one tool for test data generation). For test modelling, U2TP is used. The Telling TestStories approach provides a good documentation and a tool is shipped as a bundle of Eclipse plugins (Telling TestStories, 2015). It is quite easily manageable, but the functionality is also very limited. For the TT-Medal project, there is no existing tool that can be used.

- Traceability of requirements – It shall be possible to detect the specified requirements throughout the whole testing process.

The traceability of requirements is supported by the Fokus!MBT and Telling TestStories approaches. The other related projects do not mention the support.

The upcoming set of requirements is directly derived from the aims and objectives of this research:

- Reusability – It shall be possible to reuse certain aspects or components within the test process in order to save time in future projects.
- NGN-compliance or support for general SIP-based IP networks – The framework either shall consider the NGN-related artefacts such as possible SCEs, SDPs and SIP AS or standard SIP-based IP networks.
- Verification and Validation – The framework shall provide both verification and validation through test processes. Especially the validation of a value-added service requires an intense involvement of the service customer in the test process.

Regarding the reusability, the ComGeneration approach defines reusable EFSMs that describe common behaviour. Fokus!MBT and U2TP specify reusable test patterns which refer to recurring test architectures, but no recurring behaviour is specified. Regarding the NGN-compliance or the support for SIP-based IP networks, ComGeneration is the only project to support this. As all approaches are MBT-based, the verification should also be supported by them. Because of the missing involvement of the service customer in the processes of the related projects, the validation is not supported by any mentioned project.

- Effectivity and efficiency of generated test cases – The framework shall generate an amount of test cases that is feasible. Furthermore, these test cases shall be sufficient enough to prove that the SUT has been implemented completely towards the specified requirements.

Fokus!MBT and U2TP both apply the test generation methods mostly based on UML sequence diagrams. This is a rather efficient method, because the amount of test cases is manageable. However, it does not prove that the test cases are sufficient enough. The ComGeneration approach is not efficient as it includes the well-known state explosion

problem, but it covers all possible behaviours that might occur in value-added service consumption. The authors of Telling TestStories claim that their approach is efficient because the test can be defined on an abstract visual level with tool support (Felderer *et al.*, 2011). There is no explicit information given regarding the effectivity.

- Expandability – It shall be possible to expand the functionality of the framework or rather to widen the support for further technologies (such as further protocols that can be tested).

U2TP is based on UML and because of the object-oriented concept of modularity and code reuse, this concept should also be provided by concepts that are based on U2TP (so, also for Fokus!MBT). Principally, this is also possible for ComGeneration, because for further support of technologies, new modular finite state machines have to be defined. Telling TestStories includes expandability through the possibility of automatically generating adapters for the communication with the SUT.

The following Table 3.2 illustrates a list of the related projects with the evaluation regarding the derived requirements.

Table 3.2: Evaluation of related projects based on derived requirements

Requirements	Current related test projects				
	U2TP	TT-Medal	Fokus!MBT	ComGeneration	Telling TestStories
Test Execution	-	+	o	+	+
Test Report	-	+	o	+	+
Collaboration and support for agile principles	-	-	-	o	o
Comprehension	-	-	-	-	-
Manageability and time exposure	o	-	-	o	+
Tool support	+	-	+	o	o
Traceability of requirements	-	-	+	-	+
Reusability	o	-	o	+	-
NGN-compliance or support for general SIP-based IP networks	-	-	-	+	-
Verification	+	+	+	+	+
Validation	-	-	-	-	-
Effectivity and efficiency of generated test cases	o	-	o	o	o
Expandability	+	-	+	+	+

Considering the specified requirements, a novel framework for functional testing of value-added services will be proposed in chapter 4 and the underlying concept will then be explained in the upcoming chapters 5, 6 and 7.

3.5 Conclusion

This chapter introduced the fundamentals of testing and functional testing and its application for value-added telecommunication services. In addition, the difference between verification and validation of a system (or service) was discussed (section 3.1).

Section 3.2 introduced the state-of-the-art testing methodologies, especially agile concepts. The advantages and limitations of the approaches were discussed and it was concluded, that the development and the testing processes have to be performed by different persons (service developers and test developers). Mentionable is also that most agile concepts focus on a close collaboration between developers, testers and customers of a system or service.

Section 3.3 described related testing approaches, tools and methodologies. Most of them refer to some kind of MBT approach either focussing on enhancing system models with test-related parameters by using U2TP in order to automatically generate TTCN-3 test cases or by supporting a tester to create models from which the tests are directly derived. In principle, the approaches in literature lack the definition of a proper testing methodology from the definition of the requirements of a value-added service until the generation and subsequent evaluation of functional tests.

The main outcome of this chapter is the evaluation of the current related projects based on requirements. The requirements have been derived from the weaknesses and strengths of the testing methodologies and related projects and represent the major criterion for the proposed novel framework for testing of value-added services described in the upcoming chapter.

4 Proposed Framework for Testing of Value-Added Services

This thesis proposes a novel framework that fulfils the requirements stated in section 3.4 and fills the gap of a thorough solution for service providers to provide well-tested value-added telecommunication services to their service customers. This chapter begins by defining the preconditions and tasks to be considered when the novel framework is established (section 4.1), followed by the introduction of the overall novel methodology to enable a more service customer-centric approach (section 4.2). Subsequently, section 4.3 will describe the framework architecture and its components.

4.1 Preconditions and Tasks of Roles

For service providers, network operators and for their customers, the introduction of a thorough test process for the provisioning of value-added services in service provider environments requires a rearrangement of the participating roles. As mentioned in section 2.4, many service providers still try to save costs by letting the developer of a service figure out himself through manual tests whether a developed value-added service meets the requirements of a service customer. However, due to the possible increasing complexity of new value-added services, even an experienced service developer will not be able to locate possible errors of the services.

Due to the limitations of the current approaches in value-added service provisioning, a new role is introduced, the service analyst. In general, service analysts work for service providers and they represent the communication link between service customers on the one hand and the service developers and test developers on the other. Further tasks of the service analysts will be discussed later in this section.

Focussing on the test developer, his role is of course derived from the traditional role of a service tester; however, besides the general requirements testers have to fulfil, further tasks are imposed on the test developer:

- The test developers have to cope with changes to service customer needs, for example, if changes in the implementation of the value-added services have to be done.
- Test developers must improve their social skills. It might be possible that they need to talk to the service customers in a language that they can understand.
- Test developers have to be able to face new technologies and have to be able to understand and work with formal models.
- Test developers must be able to help stakeholders to express their requirements, even if these requirements are rather complex.

Besides the requirements imposed on the service analysts and test developers, also the service developers and service customers have to reorganise their work.

Just as test developers, the service developers also have to improve their social skills as they also will get in contact with the service customer more regularly. As service developers probably implement value-added services by means of a Service Creation

Environment (SCE), they should be able to map specified requirements onto their implementation.

Service customers need to be open to regularly attending project meetings.

In the following section, an overview of the proposed solution framework is given. Based on the described preconditions for the framework and tasks the roles have to do, the functional principle can be performed.

4.2 Overall Methodology for Testing Value-Added Services

The proposed novel methodology assumes that a service provider has an SCE in his environment to enable the service developers to rapidly create new value-added services and bring them to market. Figure 4.1 presents an example SCE published by (Eichelmann *et al.*, 2010) and (Lehmann *et al.*, 2009). Permission to reproduce Figure 4.1 has been granted by the copyright owners of the referenced publication.

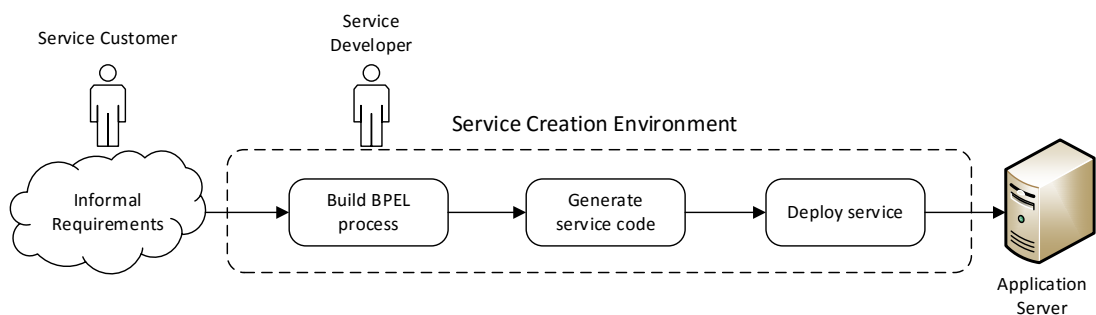


Figure 4.1: TeamCom service development (adapted from (Eichelmann *et al.*, 2010))

First, the service customer writes a non-technical, informal and natural language-based description of the service. The description should contain the idea of what the service should deliver. Based on this information, the service developer creates a formal service

description (here a Business Process Execution Language (BPEL) process) (OASIS, 2007) which is used as basis to generate the service implementation code. Finally, the service is deployed on an Application Server.

In order to integrate functional testing on top of the described service development methodology, a separate test development path is proposed (see Figure 4.2).

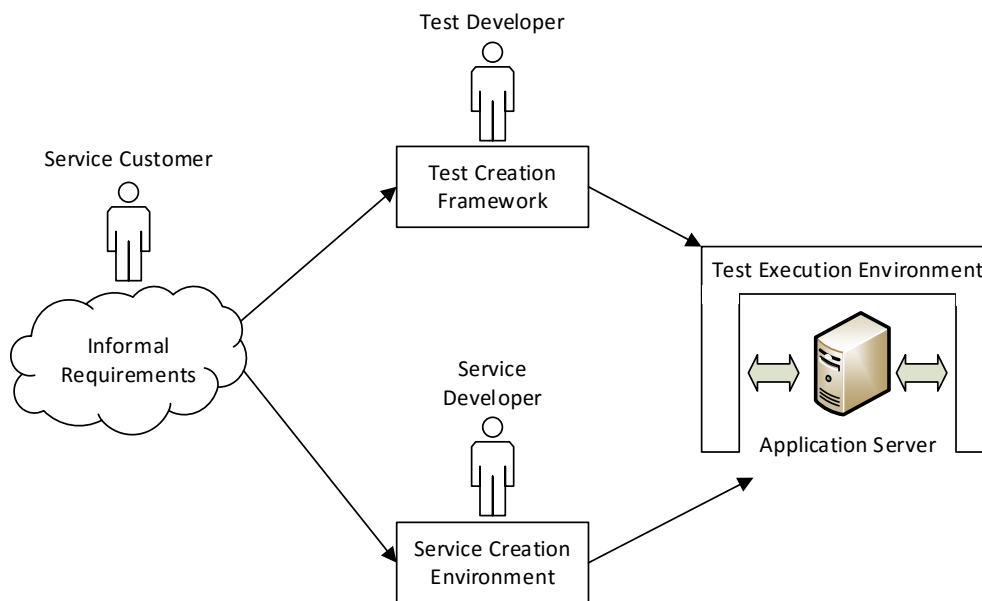


Figure 4.2: Methodology with both service and test development

The service development path in this approach was abstracted. Here, the methodology assumes that any given SCE applied by a service developer can be integrated in the process if it deploys the service on the Application Server at the end. The new test development path includes a so-called Test Creation Framework (TCF) which has to be used by the test developer. Just as the service developer, the test developer also initially receives an informal description of a value-added service's functionality. Based on this information, he can use the tools provided in the TCF in order to create tests that can be

delivered to the Test Execution Environment (TEE) where they can be executed against the SUT, the deployed service running on the Application Server.

Independent of the TCF functionality, which will be introduced in section 4.3 and thoroughly described in the upcoming chapters 5, 6 and 7, the methodology illustrated in Figure 4.2 has some major drawbacks. First, the interpretation of the informal requirements by both the service developer and the test developer will definitely show a high probability to be different. This leads to the fact that the generated tests will most likely never pass because they will not match with the deployed service. Furthermore, the methodology is strictly based on a test-last approach. The testing of the service can only be done when the service is completely developed by the service developer. Hereby, a lot of project time is wasted because the test developer can only start with the test case execution at the end of the project.

In order to solve these issues, the methodology requires another new role, the service analyst. As mentioned before in section 4.1, the service analyst is the communication link between the service customer, the service developers and test developers. When a service customer commissions a service provider to develop a new value-added service for him, he will first get in contact with the service analyst who is working for the service provider. The service customer will tell the service analyst about his service idea and the requirements. Based on the informal information, the service analyst will create a document that contains all the relevant requirements in a structured way. In the best case, the document will consist of textual use cases. Throughout this thesis, this document will be referred to as “Structured Requirements”. Intentionally, no example guideline will be

specified for the compilation of the “Structured Requirements” document, however, it has to fulfil the following requirements:

1. Each use case specified in the “Structured Requirements” document shall have a unique identifier or name. Additionally, all use cases shall be numbered (such as “Req01” for the initial specified use case).
2. If there are any dependencies between use cases they shall be specified.
3. For each use case, the actors shall be named and it shall be clear how the actors interact with the service to be specified.
4. The use case specification shall include successful scenarios as well as exceptions or alternative courses of actions.

An example specification language that fulfils all of these requirements is discussed in 5.1.2 and an example is illustrated in Table 5.3. There are many other possible related approaches that can be applied. A well-known and recognised approach is documented in (Cockburn, 2000).

Coming back to the methodology, as soon as the service analyst finalised the “Structured Requirements” document, it has to be accepted by the service customer and subsequently distributed to the service developer and to the test developer. Based on the specified use cases, both developers can start defining either the test process or the service process. A further positive aspect of the “Structured Requirements” document is that all stakeholders can rely on this document. It also enables an agile approach. Based on the requirements, both test developer and service developer can, for example, develop the service and test in order to fulfill the requirements for “Req01”. They should have an opportunity (e.g.

tool support) to notice each other's progress based on the defined requirements. So, they are able to test parts of a service even if it has not been implemented completely.

In section 4.3, it will be discussed that due to the structure of the tools and components in the TCF, it is indeed possible to actually “synchronise” the processes of both service developer and test developer. In Figure 4.3, the proposed methodology is illustrated incorporating the “Structured Requirements” and the so-called “Service Quality Group” (SQG).

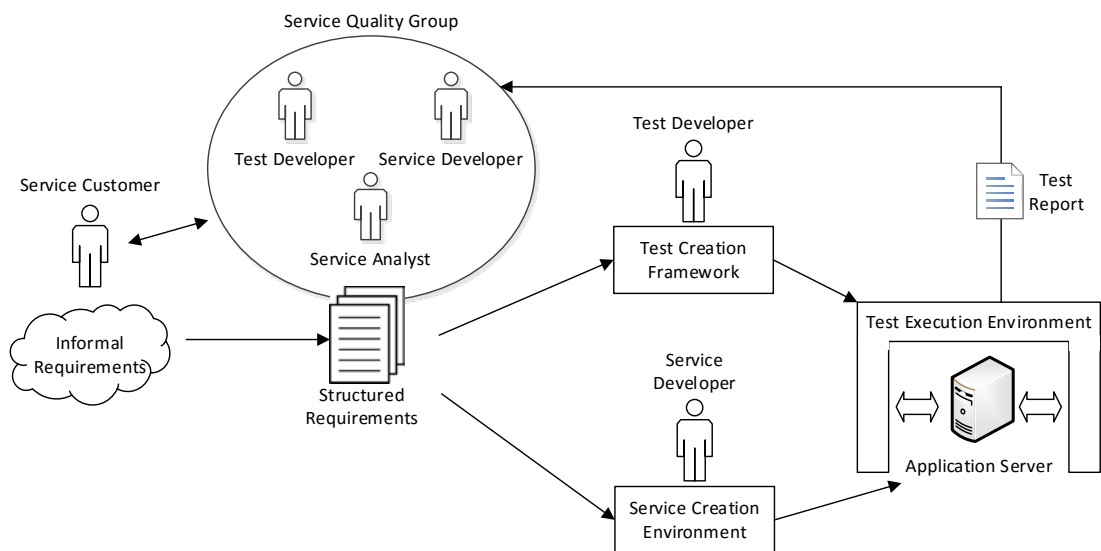


Figure 4.3: Proposed overall methodology

The concept behind the SQG is to handle the occurrence of errors due to the testing process. Of course, the members of the SQG are informed about the results (“Test Report”) of the testing process by the TEE as soon as the test case execution has terminated. Theoretically, occurring errors within the “Test Report” can have many reasons. The first possibility is that the test developer and the service developer have a different understanding of a certain aspect of the “Structured Requirements” document. This might need clarification. As the service analyst, the test developer and the service

developer are members of the SQG, especially the service analyst can discover the misunderstanding. If even he is not capable of clarifying the problem, the service customer has to be consulted. Minor misunderstanding can be clarified by direct contact (for instance by Email or telephone call). More serious issues have to be solved during a project meeting with the participating members of the SQG and the service customer. Actually, consulting the customer to clarify issues is a positive aspect of the methodology. Although there is an accepted requirements document still misunderstandings can arise. In a case like this, it is very important that the problems are discussed in an early stage and that the service customer is involved. Another reason for an occurring error can be that either the test developer or the service developer did something wrong in their development or missed a step during the process. This can also easily be clarified by the SQG. Obviously, it is not always required to arrange meetings where every member of the SQG is present, but agreements between two parties at least have to be documented in short. This also complies with methods of agile development.

The tasks the stakeholders have to perform were discussed. The following UML use case diagram (see Figure 4.4) illustrates the relevant activities they have to perform, either on their own or together with the other stakeholders. As shown in the use case diagram, the final project meeting is defined as a separate meeting where the project is validated. The service customer will have the change to attend a demonstration of the functionality of the service performed by the service developer as well as a demonstration of the test case execution and the results of the tests performed by the test developer.

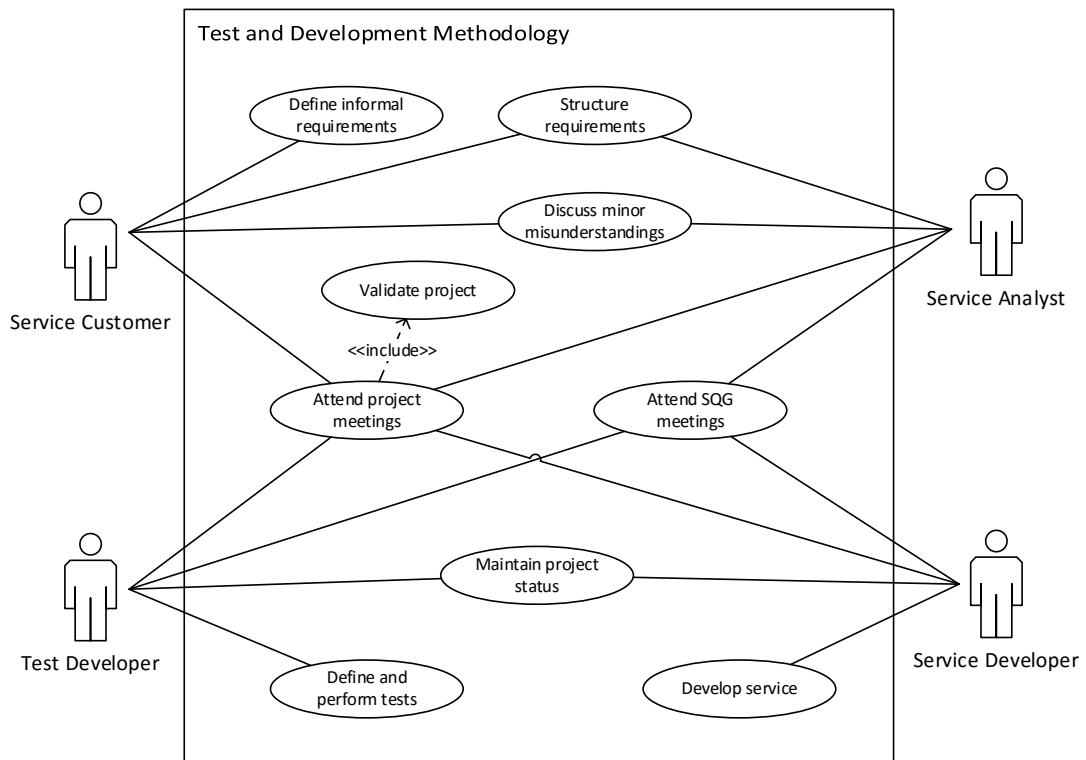


Figure 4.4: Use case diagram containing the tasks of the stakeholders

To sum up, the demonstrated methodology and concepts are oriented towards agile development. One of the highest priorities is that the service customer is satisfied, e.g. by continuously and early provisioning of usable services. Theoretically, the presented approach supports rapid prototyping. Such a prototype can claim to just support a selection of requirements specified in the “Structured Requirements” document. As the test developer and the service developer can synchronise their processes, the provided prototype can even be tested before. This aspect will be further analysed in section 7.4.

4.3 Framework Architecture and Components

Up to now, the Test Creation Framework (TCF) was treated as a black box that is applied by the test developer in order to derive tests for a specific service based on the “Structured

Requirements” document compiled by the service analyst in cooperation with the service customer. The proposed TCF is based on an architecture that is presented in the following Figure 4.5.

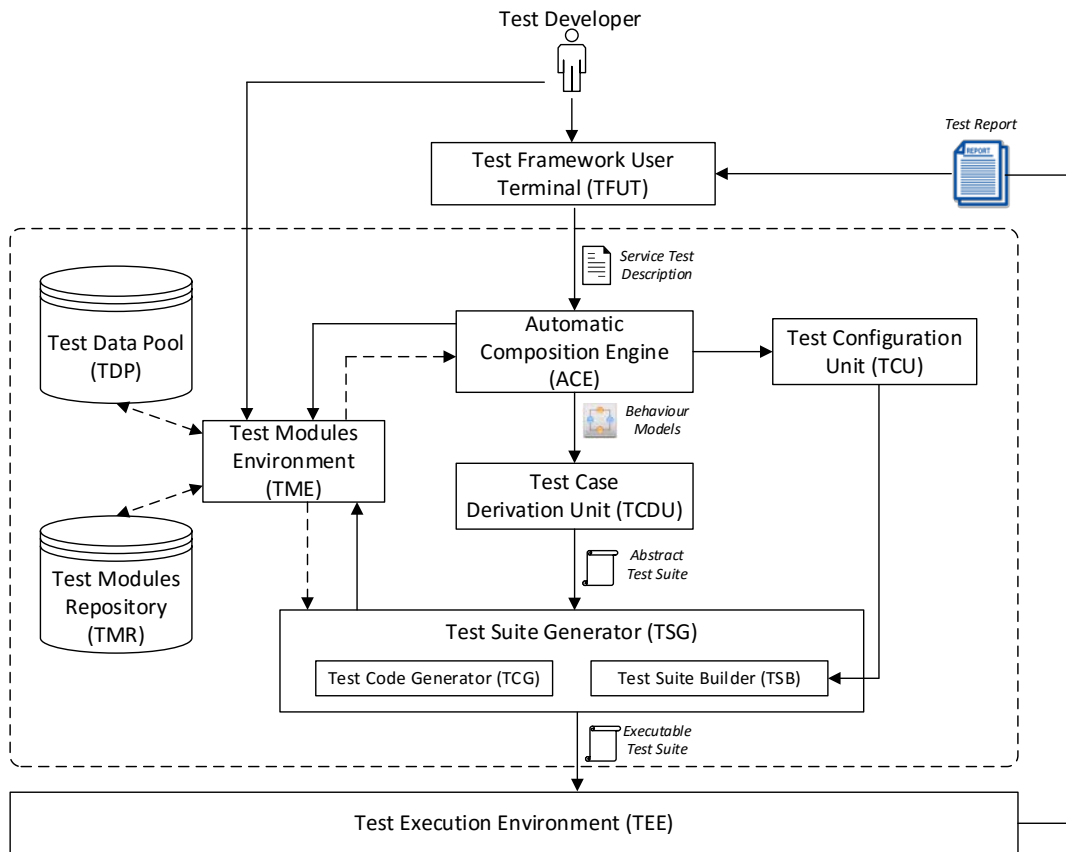


Figure 4.5: Test Creation Framework architecture

The workflow of the methodology within the TCF architecture is started by the test developer who has access to the Test Framework User Terminal (TFUT) and to the Test Modules Environment (TME).

Test Framework User Terminal

With the help of the TFUT (see Figure 4.5), the test developer can actually plan the testing process. By using the terminal website, the test developer can manage all of the projects

he is currently working on. He also sees the current progress of the service developer and can compare the outcomes. Basically, the TFUT is the entrance point for the testing where he is also able to define instances of the so-called Service Test Description (STD) based on the “Structured Requirements” document. As soon as an instance is established, the test developer is able to start the testing process.

Test Modules Environment

The TME (see Figure 4.5) enables a test developer to create, modify or erase so-called reusable test modules through a graphical user interface. Furthermore, it allows the test developer to get access to the relating test data templates and data structures that are connected to the appropriate reusable test modules. To store the relevant data, the TME uses two different databases, the Test Modules Repository (TMR) (see Figure 4.5) as well as the Test Data Pool (TDP) (see Figure 4.5). The reusable test modules including further related metadata is stored within the TMR, whereas the test data templates that are related to the reusable test modules and instances of these are stored within the TDP. When a test developer defines new reusable test modules, it is important to define the metadata which is needed to specify the test module so that a process can automatically select the test module. In this approach, the reusable test modules are modelled by means of a Statecharts-based notation and cover typical service characteristics such as sequences of multimedia protocols like SIP or RTP (Real-Time Transport Protocol) and other important protocols, such as HTTP. The test modules usually define a protocol-specific behaviour of a certain use case, e.g. the sending of an instant message by using the SIP protocol, and cover both standard behaviour as well as possible alternative behaviour like

timeouts. To sum it up, the test modules define the standard compliant behaviour of a certain use case.

Service Test Description

The STD (see Figure 4.5) is a novel type of specification or rather service description language that comprises elements of test specifications and service specifications. Furthermore, it contains architectural definitions describing the participating roles involved in the consumption of a value-added telecommunication service and their relationships as well as dynamic behavioural definitions specifying use-case related requirements. In the compilation phase, the test developer has to follow a guideline to define a STD for a service. The specification of the behaviour definitions will be done with an applied pi-calculus approach. Within the methodology, this is the only task being carried out by a human, the subsequent process performs automatically. One positive aspect of the STD besides others mentioned in section 5.3 is that the specified requirements within the STD can be directly mapped to the definitions in the “Structured Requirements” document.

Automatic Composition Engine

The Automatic Composition Engine (ACE) (see Figure 4.5) gets as input the STD after the test developer has defined it completely. The main task of the ACE is the generation of behaviour models, which are complete formal models based on Statecharts notation which describe the desired possible behaviour of the specified value-added telecommunication service. As a first step, the ACE parses the architectural definitions from the STD and forwards them to the Test Configuration Unit (TCU) (see Figure 4.5).

Afterwards, the ACE continues parsing the behavioural perspective of the STD and identifies the participating entities (or roles) (see section 5.2.2) within the specified requirements to select the suitable reusable test modules from the Test Modules Repository (TMR) via the service interface of the TME. Afterwards, the ACE connects to the Test Data Pool (TDP) in order to read the corresponding variables that are related to the selected reusable test modules. Then, new variables are instantiated and created for the instances of the reusable test modules and these are parameterised from the inputs of the STD. In the next step, the ACE starts with the composition of the reusable test module instances. Each reusable test module has interfaces which are linked to the existing states within the underlying Statecharts notation of a reusable test module. If two reusable test module instances have to be combined, the originating reusable test module instance and the destination reusable test module instance are connected with a new transition between them. The whole process is thoroughly described in section 6.4.

Test Configuration Unit

The TCU (see Figure 4.5) receives the architectural definitions from the ACE and thereupon extracts the relevant information for the Test Code Generator. Relevant information is for instance the SUT addressability and the participating test components. It is relevant for the TCU to identify which protocol is applied in order to deliver a test adapter configuration to the Test Suite Builder.

Test Case Derivation Unit

The behaviour model delivered from the ACE is the input for the Test Case Derivation Unit (TCDU) (see Figure 4.5). It contains a test case finder which uses an algorithm and

follows selected coverage criteria to enable the derivation of abstract test cases from the behaviour model. Depending on the coverage criteria, the amount of derived test cases differs significantly. The output of the TCDU is an abstract test suite which includes abstract test cases for each behaviour model.

Test Suite Generator

The Test Suite Generator (TSG) (see Figure 4.5) creates a valid TTCN-3 test suite that can be imported into a TTCN-3 test execution environment. To achieve this, the abstract test cases have to be translated into TTCN-3 test cases by means of the Test Code Generator. The Test Suite Builder will enhance the TTCN-3 code with specific test modules and includes also the configuration of TTCN-3 codecs and adapters. Furthermore, the Test Suite Builder includes the TTCN-3 compilation as well as the Java compilation process in order to generate an Executable Test Suite (ETS).

Test Execution Environment

The final step of the framework's underlying methodology takes place within the Test Execution Environment (TEE). The generated ETS can be executed due to the control part of the main TTCN-3 module. Of course, the TEE has to be installed into the service provider's test environment in advance in order to be able to address the deployed service.

Test Report

The test report (see Figure 4.5) is the document the test developer and all the other members of the SQG receive after the test execution took place. Based on the results, the test developer has to maintain the project status according to the evaluation of the specified requirements.

The framework components will be further described and introduced in the following chapters. Chapter 5 discusses the structure of the Service Test Description (STD) as well as the underlying concepts. The concept of the test modules, the Automatic Composition Engine (ACE) algorithms to compose them based on the STD as well as the generation of the behaviour model will be described in chapter 6. The other test-specific aspects, such as test case derivation from the behaviour model, the transformation from abstract test cases to TTCN-3 test cases and the test execution itself against the SUT will be part of chapter 7.

4.4 Conclusion

This chapter has introduced a novel methodology for functional testing of value-added services considering current development life cycles in service provider environments.

First, several preconditions have been introduced as well as the new tasks of the roles participating in the service development and service testing process. Both the test developer and the service developer have to improve their social skills in order to get in contact with the service customer and support him during the development and testing phases. Finally, a new role has been defined, the service analyst, who is acting as the communication link between the service customer and the service and test developers (see section 4.1).

Furthermore, section 4.2 has proposed an overall methodology that allows a better involvement of the service customer within the development and testing process. Here, also the establishment of the novel Service Quality Group (SQG) with all the tasks for

the participating roles (service analyst, test developer and service developer) has been discussed.

Finally, section 4.3 has described one of the most relevant aspects of this research, the architecture of the proposed Test Creation Framework (TFC) with all its components.

The following chapter deals with the introduced STD and will include related work done in the field of specifying the functionality of services, especially telecommunication services.

5 Novel Service Test Specification and Related Specifications

A well-defined specification of a value-added service is the basis from which functional tests can be derived. In literature, a number of service description languages and specification languages are presented and developed, mainly with the endeavour to automatically build the source code of the services and deploy them within service provider environments. To our knowledge, there is no existing service description for the purpose of functional testing that has been specifically defined for value-added communication services. To close this gap, this chapter will introduce a new kind of service description language, the Service Test Description. Firstly, section 5.1 will introduce existing service description languages and specification languages that can be applied to specify the functionality of value-added services. In section 5.2, the requirements on a service description language for the TCF described in chapter 4 will be documented and the Service Test Description will be presented. The existing specification and description languages and the new Service Test Description will be compared against each other in section 5.3.

5.1 Existing Specification and Description Languages for Services in the Telecommunication Domain

This section describes a selection of service description languages and service specifications in order to derive specifications for services. All of these approaches have the ability to describe the behaviour especially for value-added services in the telecommunication domain. A major requirement the specifications and descriptions have to fulfil is the possibility to apply them for automatic test case generation, even if they have not been considered for this purpose in the first place. Furthermore, the specifications should support the traceability of requirements. As the “Structured Requirements” document is based on standard UML use cases, it would be good to have a mapping to some use case-specific description in the specification languages. For this reason, mainly use case-based specifications have been taken into consideration.

The following specifications and methodologies will be discussed:

- Structured Use Case Models (Ryndina and Kritzinger, 2005)
- Restricted Use Case Modeling (Yue *et al.*, 2009)
- Unified Test Modeling Language (Feudjio, 2011) and (Feudjio, 2009)

The properties and also drawbacks of the specifications and descriptions will lead to a novel Service Test Description Language which is used for implementation within the proposed framework.

5.1.1 Structured Use Case Models

The research work of (Ryndina and Kritzinger, 2004) and (Ryndina and Kritzinger, 2005) provide an enhanced requirements specification methodology especially for complex systems and communication services by improving standard use case modelling (Jacobson *et al.*, 1992). The authors point out that “use case models lack structure and exact semantics, which makes rigorous analysis of such models impossible” (Ryndina and Kritzinger, 2005). Consequently, they suggest supplementing traditional use case models with a formal structure and semantics such that the use cases are suitable for automated formal analysis. This procedure allows one to discover logical errors and missing requirements early in the development cycle and provides developers with a better understanding of the defined models (Ryndina and Kritzinger, 2004). The authors declare their enhanced use case approach as “structured use case models”.

In general, use case models specify functional requirements for a system or rather service by defining scenarios of interaction between the service and its environment. The main elements used in the models are *actors* and *use cases*. The actors represent entities that actually interact with the service whereas use cases define the functionality the service has to provide. Similar to the standard use case modelling, the approach described in (Ryndina and Kritzinger, 2005) focusses on treating the system (or service) under consideration as a “black box”.

The following Figure 5.1 illustrates the perspective on actor-system interaction, which is fundamental to the technique. Permission to reproduce Figure 5.1 has been granted by the authors of the referenced publication.

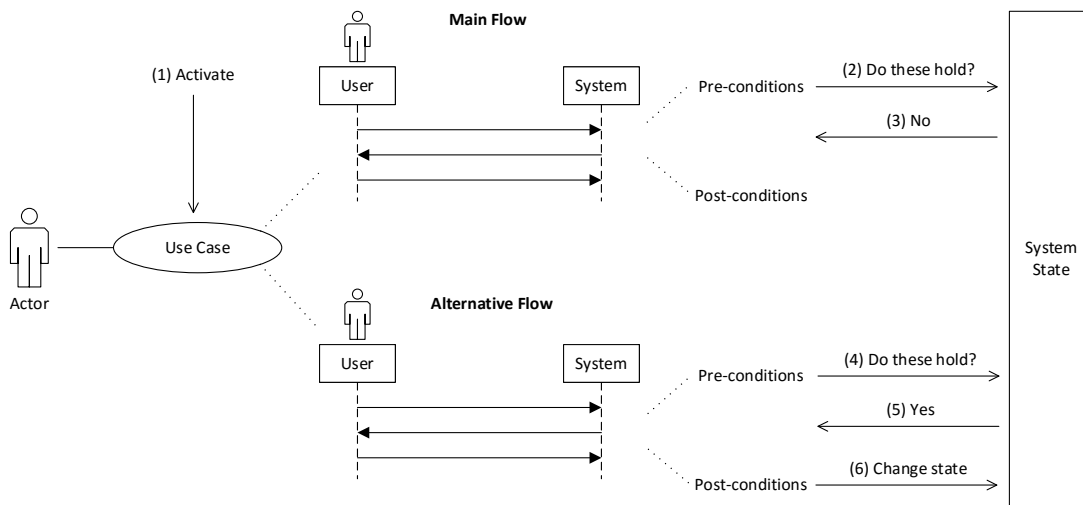


Figure 5.1: Interaction between actor and system (Ryndina and Kritzing, 2005)

Initially, the actor can call upon the system by activating the defined use cases. The system itself is described by the system state. This system state holds a set of conditions and can change throughout the model execution. Each defined use case within the specification is associated with a main flow and an unspecified number of alternative flows. Every type of flow has pre- and post-conditions. As soon as a use case is activated by an actor (1), the pre-conditions of the use case's main flow are queried against the current system state (2). If the pre-conditions do not hold (3), the alternative flow of the use case is considered. Analogous to step 2, the pre-conditions of the alternative flow are queried (4) and in the example shown in Figure 5.1, they are satisfied in the system state (5). This leads to the post-conditions of the alternative flow changing the system state (6). The activation of a use case is said to be successful when the pre-conditions for one of the defined flows hold.

In order to define the structured use case models for systems or rather communication services, a metamodel was defined by the authors (see Figure 5.2). Permission to reproduce Figure 5.2 has been granted by the authors of the referenced publication.

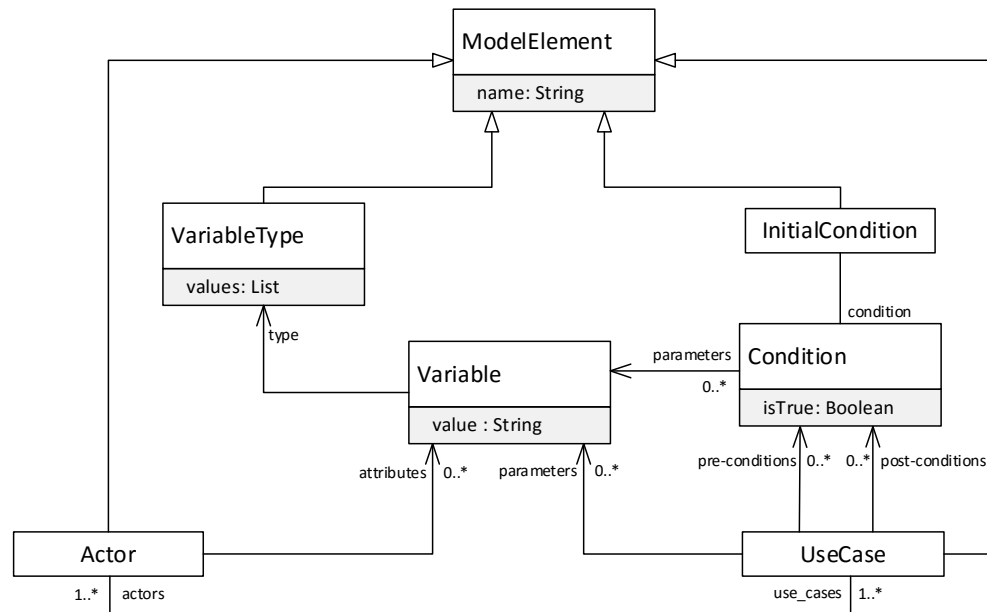


Figure 5.2: Structured use case metamodel (Ryndina and Kritzing, 2005)

The displayed metamodel shows that a structured use case model contains four different types of elements: *actors*, *use cases*, *conditions* and *variable types*. Each element comprises a number of properties that capture information related to that element. With reference to the definition of the metamodel elements, actors and use cases as well as their associations are depicted in a graphical representation (see Figure 5.1). For each actor and use case in the diagram, the textual properties can be defined in addition. The other elements, conditions and variable types, are completely textual and do not have graphical representations.

The actor element in the metamodel has two properties, a name and a list of attributes. The attributes include important information about an actor. In order to deliver services to the actor represented by use cases, a system has to be able to access these particulars. For example, an actor willing to setup a call to a system has an attribute called *Username* that he needs to provide to the system in order to consume the service. Due to the metamodel in Figure 5.2, each attribute is regarded as a *variable*, and each variable has

an associated variable type. A variable type is associated with a number of symbolic values, which are mainly string literals that can only be compared for equivalence (Ryndina and Kritzinger, 2004). Two defined variables are declared as equal if their values are set to identical string literals.

A further important element of the metamodel, the condition, is used to either describe the global state of the system or to declare use case pre- and post-conditions. The condition element has three properties: a name, a list of parameters and a *truth-value* (“isTrue”). A condition becomes a condition instance, as soon as values are assigned to all its parameters. The condition instance is either “true” or “false”. A special type of condition is the *InitialCondition* which is used to describe the state of the system before any interaction between actors and the system occurs (Ryndina and Kritzinger, 2005).

The final element in the metamodel, the use case, has five properties: a name, the list of actors playing a role in the use case, a parameter list, a list of pre-conditions and a list of post-conditions. The use case parameters describe information that is required to provide the corresponding service. As soon as a use case is activated, a value for each of its parameters needs to be passed to the system. The list of pre-conditions specifies that certain aspects about the system state must hold so that a use case activation can be successful. On the contrary, the post-conditions describe the change of the system state after a successful activation of the use case.

Based on the introduced metamodel for structured use case models, a simple chat service, will be described and specified as an example by means of the methodology. Initially, the general functions the simple chat service provides will be documented in a standard use

case diagram which also includes the participating actors. Figure 5.3 illustrates the sample use case diagram.

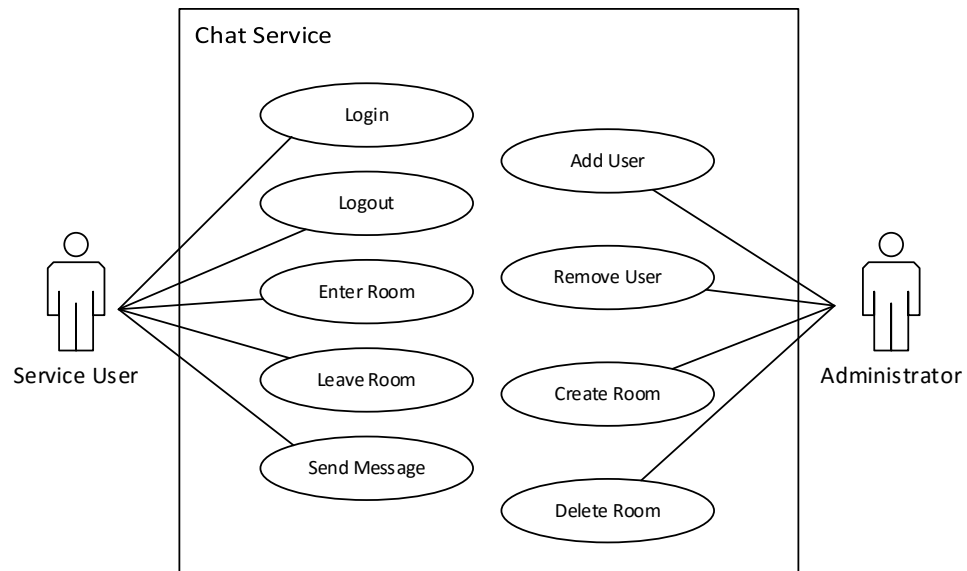


Figure 5.3: Use case model of sample chat service

The service chat usage includes two involved actors, an *Administrator* and a *Service User*. The Administrator can register new Service Users (“Add User”) and deregister existing ones (“Remove User”). He is also able to create new chat rooms for the Service Users (“Create Room”) or to erase old rooms that might not be used anymore (“Delete Room”). The other actor, the Service User, initially has to log in (“Login”) to use the provided functions, such as entering a new chat room (“Enter Room”), leaving the chat room after having entered (“Leave Room”) and sending messages to users in the same room (“Send Message”). Finally, the Service User can log out (“Logout”).

Each use case illustrated in Figure 5.3 can be defined by means of a special language presented in (Ryndina and Kritzinger, 2005). Exemplarily, the use case “Add User” of the actor Administrator is shown in the following Figure 5.4.


```
USE CASE 1
name: Add User
actors: Administrator
parameters: Username of type UserLogin
pre-conditions: UserExists (#uc Username) is false
post-conditions: UserExists (#uc Username) is true

VARIABLE TYPE 1
name: UserLogin
values: chatUser1

CONDITION 1
name: UserExists
parameters: Username of type UserLogin
```

Figure 5.4: Structured use case model definition of “Add User”

The definition of the “Add User” use case is quite straightforward and indicates that the Administrator is the only actor that can activate this use case. Furthermore, the *Username* of the Service User to be added needs to be provided to the system as a use case parameter. Each use case parameter has an associated variable type which defines a finite set of symbolic values. Here, the variable type *UserLogin* holds the value “chatUser1”. The pre-condition for the “Add User” use case states that the activation is successful if the user with the provided *Username* does not exist. In addition, the post-condition indicates that after successful activation of the use case, a Service User with the given *Username* exists. It is required that the pre- and post-condition defined within a use case correspond to a condition declaration within the model, where the name and the type of condition parameters are determined. In this example, the *UserExists* condition is declared. It indicates that the condition has one parameter of variable type *UserLogin*. The *#uc* prefix in the pre- and post-conditions states that at the time of activation, the value of the use case parameter *Username* should be used for the evaluation of this condition.

Most of the other use cases in Figure 5.3 can be defined similarly to the “Add User” use case. There is usually one action performed by an actor that causes a change of the system state, such as logging in and out or creating a new chat room. Through the pre- and post-

conditions it can be easily verified if the action had the desired effect. The sending of a chat message is more complex because more parameters and conditions have to be checked. The following Figure 5.5 shows the definition of the “Send Message” use case.

```
USE CASE 2
name: Send Message
actors: Service User
parameters: Username of type UserLogin, Message of type MessageContent
pre-conditions: LoggedIn (#forall UserLogin) is true,
                RoomEntered (#forall UserLogin, #room1) is true
post-conditions: MessageReceived (#uc Username, #uc Message) is true

ACTOR 1
name: Service User
attributes: Username of type UserLogin

VARIABLE TYPE 2
name: MessageContent
values: Hello, how are you?

VARIABLE TYPE 3
name: RoomDeclaration
values: room1

CONDITION 2
name: LoggedIn
parameters: Username of type UserLogin

CONDITION 3
name: RoomEntered
parameters: Username of type UserLogin, Room of type RoomDeclaration

CONDITION 4
name: MessageReceived
parameters: Username of type UserLogin, Message of type MessageContent
```

Figure 5.5: Structured use case model definition of “Send Message”

In the use case model definition of “Send Message”, the actor role of the Service User is explicitly specified and enhanced by an attribute *Username* of the type *UserLogin*. Two parameters are required to activate the use case, the *Username* of the user who is about to receive the message and the *Message* itself. There are two pre-conditions that have to hold in order to activate the use case. On the one hand, all participating users have to be logged in; on the other hand, all users also have to have entered the chat room *#room1*. In order to verify both cases, the conditions *LoggedIn* and *RoomEntered* have been defined in the use case model definition. Besides, the *#forall* option allows checking that the conditions hold for all values of a particular variable type. Finally, the post-condition

of the “Send Message” use case states that the user with the *Username* of the use case receives the *Message*. Therefore, an additional condition *MessageReceived* is defined.

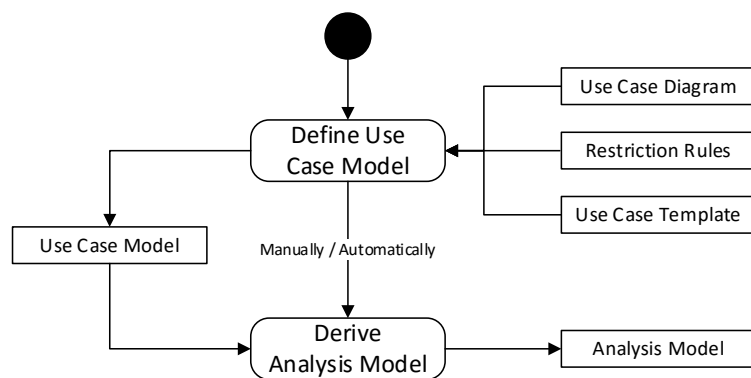
To sum up, the solution to structure standard use case models by applying the well-defined metamodel enables models that are far more consistent and correct. Nevertheless, important properties are missing, for instance the actions that actually take place within the use cases are not precisely specified. Tests cannot be generated from use cases if it is not defined how to trigger the system or rather service. Only the states of the system before the action and after the action are determined. It is also very hard to imagine how the conditions can be applied to the testing process. This can only be done if the conditions are predefined as reusable test components that have to be parameterised by the inputs defined within the use case models. This aspect would limit the possibilities to specify diverse types of services. It is possible that for each condition occurring in structured use case models for specified services, a proper test component first has to be developed by a test developer.

5.1.2 Restricted Use Case Modeling (RUCM)

Restricted Use Case Modeling (RUCM) is a use case modelling approach developed by (Yue *et al.*, 2009). In general, standard use case modelling includes use case diagrams and use case textual specification and is commonly applied to structure and document requirements (Jacobson *et al.*, 1992). However, it is well-known and also stated by the developers of RUCM that standard use case modelling based on textual descriptions inevitably contains ambiguities and tends to be imprecise and incomplete. To overcome this obstacle, RUCM is composed of a use case template to structure the use case

specifications and a well-defined set of restriction rules to restrict the way users write use case specifications. The developers of RUCM conducted a controlled experiment with human subjects and results indicate that RUCM, although it enforces a use case template and restriction rules, “has enough expressive power, is easy to use, and helps improve the understandability of use cases” (Yue *et al.*, 2011).

The RUCM approach can be applied during the requirements elicitation phase of the software or rather service development. The following activity diagram in Figure 5.6 illustrates the process flow.



© 2009 IEEE

Figure 5.6: RUCM process flow (Yue *et al.*, 2009)

In order to define a use case model, the approach requires the input of a use case diagram of the service to be specified, the restriction rules as well as the use case template. After the use case model is documented complying with the rules and structure, an analysis model can be derived either manually performed by system analysts or automatically if the inputted use case models are less ambiguous and automated analysis can be facilitated.

In the following Table 5.1, the structure of the RUCM use case template presented in (Yue *et al.*, 2009) and (Zhang *et al.*, 2013) is illustrated.

Table 5.1: RUCM Use Case template (Yue *et al.*, 2009)

Use Case Name	The name of the use case. It usually starts with a verb.	
Brief Description	Summarises the use case in a short paragraph.	
Precondition	What should be true before the use case is executed.	
Primary Actor	The actor which initiates the use case.	
Secondary Actor	Other actors the system relies on to accomplish the services of the use case.	
Dependency	Include and extend relationships to other use cases.	
Generalization	Generalisation relationships on other use cases.	
Basic Flow	Specifies the main successful path, also called “happy path”.	
	Steps (numbered)	Flow of events.
	Postcondition	What should be true after the basic flow executes.
Specific Alternative Flows	Applies to one specific step of the basic flow.	
	RFS	A reference flow step number where flow branches from.
	Steps (numbered)	Flow of events.
	Postcondition	What should be true after the alternative flow executes.
Global Alternative Flows	Applies to all the steps of the basic flow.	
	Steps (numbered)	Flow of events.
	Postcondition	What should be true after the alternative flow executes.
Bounded Alternative Flows	Applies to more than one step of the basic flow, but not all of them.	
	RFS	A list of reference flow steps where flow branches from.
	Steps (numbered)	Flow of events.
	Postcondition	What should be true after the alternative flow executes.

© 2009 IEEE

The use case template contains eleven so-called first-level fields (first column of Table 5.1) from which the last four fields are decomposed into second-level fields. The first seven fields contain general information about the use case (*use case name*, *brief description*), its state before activation (*precondition*), the actors who are participating in the use case (*primary actor*, *secondary actor*) and relations to other use cases (*dependency*, *generalization*). Additionally, the use case template contains one *basic flow* and can have one or more types of *alternative flows*: specific, global, and bounded alternative flows.

The basic flow describes the main successful path in the use case and is composed of a sequence of *steps* and a *postcondition*. In the approach, five different types of interactions have been reused from (Cockburn, 2000) for each defined step:

- Primary actor → system: the primary actor sends a request including data to the system.
- System → system: the system validates a request and data.
- System → system: the system alters its internal state, for instance by recording or modifying something.
- System → primary actor: the system replies to the request of the primary actor with a specific result
- System → secondary actor: the system sends a request to a secondary actor.

Furthermore, the steps are numbered sequentially so that each step is completed before the next one is started. Conditions, iterations and concurrency can be defined within the steps by specific keywords that are included in the RUCM restriction rules (Yue *et al.*, 2009).

In contrast to the basic flow describing the main successful part, the alternative flows describe scenarios, both success and failure. According to (Yue *et al.*, 2009), an “alternative flow always depends on a condition occurring in a specific step in a flow of reference, referred to as *reference flow*, and that reference flow is either the basic flow or an alternative flow itself”. The classification of the alternative flows has been taken from (Bittner and Spence, 2002). A *specific alternative flow* is an alternative flow referring to a specific step in the reference flow. An alternative flow that refers to more than only one step in the reference flow is called *bounded alternative flow*. Finally, the *global*

alternative flow refers to any step in the reference flow. It is important to mention that each flow, both basic and alternative, must have a defined postcondition which describes a constraint that must be true when the use case terminates.

In order to define the use cases complying with the RUCM use case template, the usage of the set of restriction rules is important. Basically, the restriction rules are classified into two groups: restrictions on the use of natural language, and restrictions enforcing the use of determined keywords for specifying control structures.

The first group of restrictions contains sixteen rules from which the first seven apply only to action steps (see Table 5.2) and not to steps that contain conditions or preconditions and postconditions. The rules R8-R16, however, apply to all sentences in a use case specification, also including the brief description. Mainly, the focus of the sixteen rules is to reduce ambiguity and also to facilitate automated generation of analysis models. It is meant to be a guideline for writing clear and concise use case specifications, for instance by using the appropriate grammatical tense (present tense), avoiding negative adverbs, negative adjectives, and participle phrases that are very difficult to parse by natural language parsers (Yue *et al.*, 2013).

Table 5.2: Restriction rules R1-R16 of RUCM approach (Yue *et al.*, 2013)

#	Description	Explanation
R1	The subject of a sentence in basic and alternative flows should be the system or an actor.	Enforce describing flows of events correctly. These rules conform to the RUCM use case template (five interactions).
R2	Describe the flow of events sequentially.	
R3	Actor-to-actor interactions are not allowed.	
R4	Describe one action per sentence.	Otherwise it is hard to decide the sequence of multiple actions in a sentence.
R5	Use present tense only.	Enforce describing what the system does, rather than what it will do or what it has done.
R6	Use active voice rather than passive voice.	Enforce explicitly showing the subject and/or object(s) of a sentence.
R7	Clearly describe the interaction between the system and actors without omitting its sender and receiver.	
R8	Use declarative sentence only.	Commonly required for writing UCSs.
R9	Use words in a consistent way.	Keep one term to describe one thing.
R10	Don't use modal verbs (e.g., <i>might</i>)	Modal verbs and adverbs usually indicate uncertainty.
R11	Avoid adverbs (e.g., <i>very</i>)	
R12	Use simple sentences only.	Facilitate automated NL parsing and reduce ambiguity.
R13	Don't use negative adverb and adjective (e.g. <i>hardly, never</i>), but <i>not</i> or <i>no</i> is allowed.	
R14	Don't use pronouns (e.g. <i>he, this</i>)	
R15	Don't use participle phrases as adverbial modifier.	
R16	Use "the system" to refer to the system under design consistently.	Keep one term to describe the system; therefore reduce ambiguity.

© 2013 IEEE

The second group of restrictions contains rules constraining the use of control structures by keyword. These keywords are applied within steps of basic or alternative flows. The most important keywords specify conditional sentences (*IF-THEN-ELSE-ELSEIF-ENDIF*), concurrency sentences (*MEANWHILE*), condition checking sentences (*VALIDATES THAT*), and iteration sentences (*DO-UNTIL*). Further rules specify that alternative flows end with a step either using the keyword *ABORT* or *RESUME STEP*. The later signifies that the flow returns back to the reference flow.

The important parts of RUCM, the use case template as well as the restriction rules, have been explained briefly. In the following, the example use case “Send Message” of the sample chat service (see Figure 5.3) will be described by applying the RUCM method.

Table 5.3 illustrates the “Send Message” RUCM use case.

Table 5.3: Example RUCM use case of "Send Message"

Use Case Name	Send Message		
Brief Description	User sends a text message to an Endpoint participating in a chat room.		
Precondition	User and Endpoint are logged into the system and have both entered a chat room.		
Primary Actor	User	Secondary Actor	Endpoint
Dependency	None	Generalization	None
Basic Flow	<ol style="list-style-type: none"> 1) User sends a text message to the system including the name of the Endpoint as target. 2) The system VALIDATES THAT User and Endpoint are in the same chat room. 3) The system forwards the text message to the Endpoint. 4) The system VALIDATES THAT it receives an acknowledgment response from the Endpoint. 5) The system sends an “OK” text message to the User. <p>Postcondition: The system is idle.</p>		
Specific Alternative Flow (RFS Basic Flow 2)	<ol style="list-style-type: none"> 1) The system sends the message “The user is not in the chat room.” to the User. 2) ABORT <p>Postcondition: The system is idle.</p>		
Specific Alternative Flow (RFS Basic Flow 4)	<ol style="list-style-type: none"> 1) The system sends the message “Message not received.” to the User. 2) ABORT <p>Postcondition: The system is idle.</p>		

The most relevant information apart from the flow definitions in the shown example use case is the precondition determining the current state before the use case can be activated (both the *User* and *Endpoint* should be logged in and should have entered the chat room) and the mentioning of the participating actors (here: *User* and *Endpoint*). Only these actors as well as the system can be mentioned as subjects within the steps of the flow definitions. The basic flow contains the sending of the text message from the *User* to the system (step 1) as well as the forwarding of the message to the *Endpoint* (step 3). In addition, a notification is sent to the *User* that the message transmission was successful

(step 5). In between, the system inspected if both *User* and *Endpoint* are in the same chat room (step 2) and if the system received an acknowledgement message from the *Endpoint* after sending the message (step 4). The inspections are detected by means of the keyword *VALIDATES THAT* and automatically lead to the defined specific alternative flows. If the validation process of the system fails, then the relevant specific alternative flow is activated.

In summary, the RUCM method allows the definition of well-structured use cases. In the shown example, the interaction between the system (or rather service) and the participating actors is described clearly. This is very important when it comes to testing because the role of the participating actors will be performed by test components in the test execution process. These test components have to know how to trigger the SUT (System under Test) and what kinds of messages or notifications to expect from it. Although the interactions are thoroughly described in the RUCM method and the use of language is strictly regulated by restriction rules, still natural language might lead to problems as the processing and parsing of it is still very error-prone. A general issue of natural language is its ambiguity. To solve this issue, the authors of (Yue *et al.*, 2011) present a solution to transform the RUCM use cases to UML state machine diagrams by means of natural language parsers. Nevertheless, the method has not been proven to be applied to more complex systems or rather communication services. It is also questionable how reusability of tests can be integrated into the concept. For every system or service, the flow definitions and functionality has to be performed from scratch. The approach also lacks a possibility to define parameters (e.g. text message properties such as text message content, sender and target) explicitly.

5.1.3 Unified Test Modeling Language (UTML)

(Feudjio, 2011) and (Feudjio, 2009) propose a language dedicated to Model-Driven Test Engineering (MDTE) that reuses and extends concepts of U2TP (see chapter 3.3.1). This language is called Unified Test Modeling Language (UTML) and its compilation is carried out during a specified test modelling process. The design of test automation in UTML is based on various principles of abstraction which guide the whole test modelling process to ensure that the resulting model remains concise. The process including the various phases it implies is illustrated in Figure 5.7.

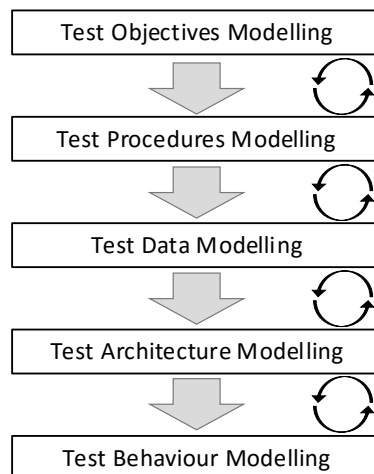


Figure 5.7: Overview of UTML test modelling process (Feudjio, 2009)

The initial phase, *Test Objectives Modelling*, includes the identification of relevant test objectives. These test objectives can be grouped into diverse categories such as functional correctness or usability. They have to be defined manually by design experts or can be generated automatically if the requirements on the system are expressed in a machine-processable notation. (Feudjio, 2011) does not mention an example notation that can be applied.

Within the *Test Procedure Modelling* phase, test procedures are modelled as sequences of test steps. Each test step represents an action or an observation to be performed on one or more elements in the test setup. The test steps can be described by using natural language. The grade of formality increases during the test modelling process (Feudjio, 2011).

The upcoming *Test Data Modelling* phase, for instance, requires the formal description of data that will be exchanged between elements of the test environment and the SUT. The identification of the relevant data results from the designed test procedures from the previous phase. Possible data can be stimuli (e.g. protocol messages that will be sent to the SUT) or potential protocol responses from the SUT. As the description of the messages is depending on the used protocol, a static description of the protocol is required so that the modeller knows how to add data templates to the UTML test model. The protocol descriptions can be available either as a plain document (e.g. IETF RFC), an XML Schema Descriptor (XSD) (W3C, 2012) file or other data description mechanisms such as Abstract Syntax Notation One (ASN.1) (ITU-T X.680, 2015) or Interface Description Language (IDL) (OMG, 2002).

Within the *Test Architecture Modelling* phase, the topology of the test system as well as a collection of test configurations is defined by means of a formal model. This includes the setup of the test system consisting of parallel test and system components interconnected with each other via ports. These ports are used to communicate between the components either synchronously (request/reply) or asynchronously (message-based) and to exchange messages.

In the final *Test Behaviour Modelling* phase, the semantic requirements on the system (or service) can be expressed by means of UTML test behaviour models. These test behaviour models have a graphical representation, the UTML test behaviour diagrams, which are built upon the test architecture configuration from the previous phase. The diagrams are similar to standard UML sequence diagrams and describe how each of the defined test procedures can be illustrated in terms of actions on and between the entities being part of the test configuration.

From each of the described phases, a specific model is generated. The sum of all these models provides the foundation for the UTML test model that is illustrated as *TestModel* in the following UML class diagram in Figure 5.8. Permission to reproduce Figure 5.8 has been granted by the author of the referenced publication.

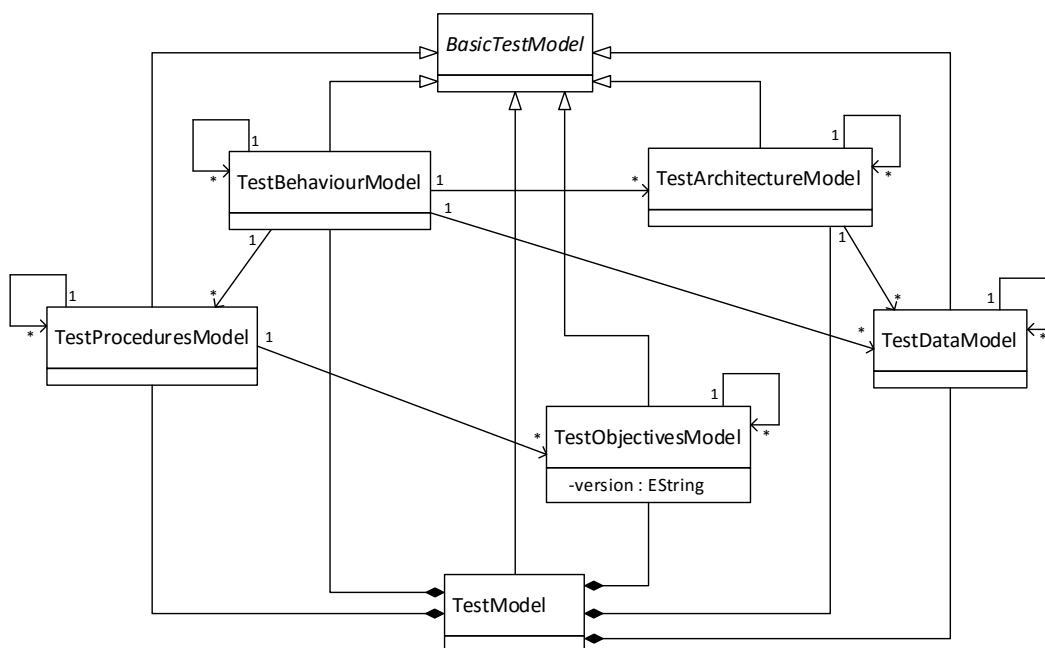


Figure 5.8: Overview of UTML test models (Feudjio, 2011)

The models from the diverse phases deal with specific aspects of test design and extend the abstract *BasicTestModel* class. The relations between the categories of test models are

also depicted in the class diagram. In order to define complete and deliberate UTML test models, the author proposes in (Feudjio, 2009) to use the tool set MDTester (UTML, 2015), which guides test modellers in defining the different categories of test models.

In the following, the UTML test model approach will be applied to the “Send Message” use case from the sample chat service (see Figure 5.3). UTML is, however, not based on standard use case descriptions and a mapping concept is not provided. The requirements stated in the “Send Message” use case (see Table 5.3) have to be broken down to several test procedures in UTML covering the successful path (e.g. “message was received successfully”) as well as the exceptional paths (e.g. “both users are not in the same chat room”). The test objective will be to evaluate the “functionality” of the “Send Message” use case. Afterwards, the test procedure will contain the documentation of the test steps to describe the successful path in natural language. Then, the definition of the test data will be done in the corresponding phase. The tool set MDTester provides a “Data View” that enables the test modeller to create test data templates from predefined data models. Here, every message being exchanged between entities of the test system and the SUT can be defined. The following Figure 5.9 shows a tree view of MDTester that allows the definition of templates of request types for the SIP protocol.

- ❖ Test Data Group SipDataTypes
 - ❖ Message Test Data Type SIP_RequestType
 - ❖ Data Type Field req_type
 - ❖ Data Type Field to_header
 - ❖ Data Type Field from_header
 - ❖ Data Type Field callID_header
 - ❖ Data Type Field via_header
 - ❖ Data Type Field body

Figure 5.9: Test Data View with UTML for SIP messages

In the shown example, the test modeller has the possibility to create instances (templates) of SIP requests and set the underlying header fields of the SIP message according to the information described within the test procedure model. Additionally, SIP responses can be defined.

After having identified and set all the relevant messages, the topology and test configuration for the test procedure has to be determined. The following test configuration (see Figure 5.10) shows two involved so-called Parallel Test Components (PTCs) and the SUT specifying the chat service deployed on an application server.

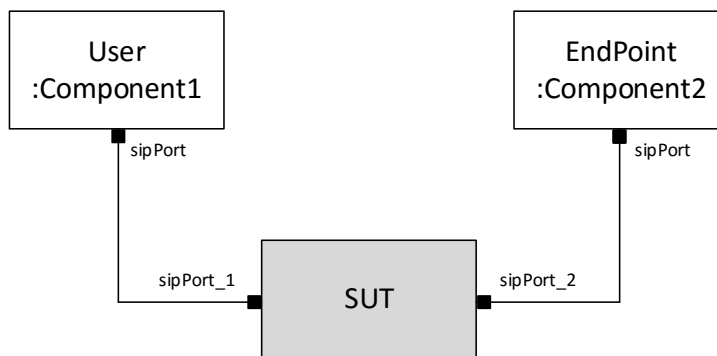


Figure 5.10: Test Architecture Diagram for sample chat service

The PTCs are called *User* and *Endpoint* and have the same role as the actors described in the RUCM use case from Table 5.3. Both are connected to the SUT via their SIP ports so that SIP messages can be exchanged between the corresponding entities.

The specification of the message exchange is part of the Test Behaviour Model. The following UTML Test Behaviour Diagram (see Figure 5.11) shows the involving entities (both PTCs and the SUT) and their expected message exchange regarding to the successful path of the “Send Message” use case. First, the *User* PTC sends a SIP message request containing the chat message that it is about to send to the *Endpoint* PTC. This

chat message is embodied within a SIP request template called *initChatMessage* that has already been defined in the Test Data Model. After sending the chat message, the SUT should then acknowledge the receipt of the SIP message by first sending a SIP response message (*recvResponse_OK*) to the *User* PTC and then initiate the transmission of the SIP message *chatMessage* to the *EndPoint* PTC.

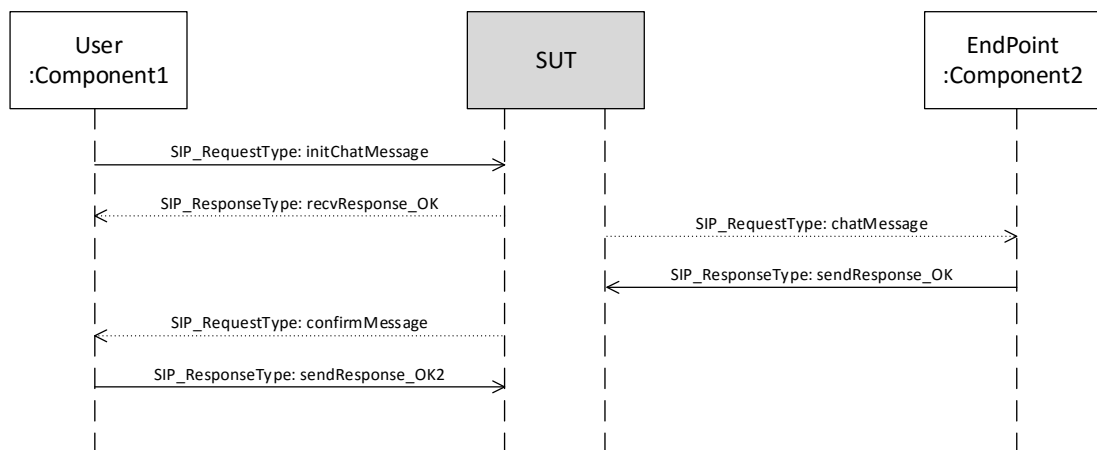


Figure 5.11: Test Behaviour Diagram for Send Message use case of sample chat service

The successful receipt will now be acknowledged by the *EndPoint* PTC and the SUT can accordingly send the “OK” message *confirmMessage* to the *User* PTC to confirm that the initial chat message transmission has been successful. Finally, the SUT will receive an acknowledgement that the message was received by the *User* PTC.

To sum up, the UTML test modelling process structures and simplifies the derivation of a test specification for a given system and can be applied to value-added communication services. According to (Feudjio, 2011), there is already a defined methodology to generate TTCN-3 test cases on the basis of UTML test models. In spite of the positive aspects mentioned, UTML still lacks some properties in order to be an appropriate language for the proposed test framework in this research work (see Figure 4.5). First, UTML does not

specifically refer to use cases. This makes the transition harder for the test developer who would have to build the UTML test models based on the “Structured Requirements” document. Second, a synchronisation of the service development process and the test development process is also not possible or very hard to manage. Third, it is also questionable how reusability of tests can be integrated into the UTML approach. In fact, the test developer has to spend a lot of time to build the test models, especially the Test Behaviour Diagrams. For each alternative behaviour within one use case, a new diagram has to be created. Finally, the author of (Feudjio, 2009) states himself that although first results indicated that the development cycle shortened significantly by applying the UTML methodology, there is still a problem in the context of model consistency between the inter-dependent test models.

5.1.4 Alternative Approaches

Besides the precisely described approaches presented above, there are also other approaches.

Requirements Acquisition and Specification for Telecommunication Services

The first approach is described in (Eberlein, 1997), (Eberlein *et al.*, 1997) and (Eberlein and Halsall, 1997) and is named Requirements Acquisition and specification for Telecommunication Services (RATS). The underlying methodology introduces three different types of scenario representations, textual (natural language-based), structured (textual, but also with preconditions, postconditions and flow conditions) and formalised (structured text and component-centered). The idea behind these representations is to allow a smooth and straight transition from a service description based on natural

language to a formal specification described in the Specification and Description Language (SDL) (ITU-T Z.100, 2007). In the scenarios, different aspects of behaviour is described, such as normal, alternative, and exceptional behaviour. These groupings support the developers to first focus on the common behaviour and afterwards concentrate on the less common service functionality. Most of the scenarios are abstract and linear, but there also so-called overall scenarios capturing multiple scenarios, with a causal ordering. Overall, the RATS methodology is an interesting approach to requirements elicitation, but it is significantly depending on its implementation, the RATS tool. This expert system contains a large knowledge database that has to be updated all the time. Besides, the publications mentioned do not go in depth into the construction of the SDL models, but focus more on the acquisition and the specification of requirements. This leads to the fact that RATS does not provide an output that can be applied to generate test cases because the grade of formality is not sufficient enough.

Telecommunication Modelling Library

Another alternative approach is called Telecommunication Modelling Library (TelcoML) (OMG, 2013b). This language is built on top of SoaML (OMG, 2012b), a standard extension to UML 2.0 that focuses on modelling of services following the Service-Oriented Architecture (SOA) paradigm. TelcoML itself defines a UML profile for advanced and integrated services and provides “a common abstraction to all existing communication services standards so that tools can be built” (OMG, 2013b) for service providers to be able to model service variations in a consistent manner. TelcoML is composed of the TelcoML Enabler Library and the TelcoML Service Composition Profile. The TelcoML Enabler Library is a set of service interfaces representing telecom-

specific facilities such as “Generic Messaging”, “Click-to-Call”, “Synchronisation”, “Voice Recognition and Text-to-Speech” and “Privacy”. On the other hand, the TelcoML Service Composition Profile enables the specification of composite services based on the predefined service interfaces. A UML state machine-based approach can be applied to define the compositions. To sum up, TelcoML includes a very specific way of describing communication services based on a very small number of reusable service interfaces. Due to this limitation only a few services can be modelled based on TelcoML. Although it is based on UML class diagrams and state machines, it does not contain any relation to UML use cases. This makes it more difficult for the test developer to derive a TelcoML specification based on the “Structured Requirements” document. Additionally, test-specific parameters (such as wild cards for data sent from the SUT to the test components) are not part of the language.

AGEDIS Modeling Language

Finally, the AGEDIS Modeling Language (AML) is introduced which is based upon the UML meta-model. Within the AGEDIS methodology presented in (Hartman and Nagin, 2004) and (Craggs *et al.*, 2003), AML serves as behavioural modelling language. Initially, AML includes the structure of the SUT by means of UML class diagrams with their associations. Here, the approach bears resemblance to other methodologies, such as U2TP (see section 3.3.1) and UTML (see section 5.1.3). These methodologies also initially include the definition of the structure or rather architecture of the SUT. The behaviour of each class within the diagrams is defined in corresponding state diagrams. The actions within the state diagrams are individually specified by means of the IF language (Bozga *et al.*, 1999), a specification language that is based on timed automata and extended with

discrete data variables, various communication primitives and dynamic process creation. After the class diagrams and state machines are described, an instance of AML is created. This creation process requires a very deep knowledge in the modelling of UML class diagrams and state diagrams and, additionally, the specification of the actions within the state diagrams requires the use of the IF language which bears resemblance to programming languages. Another missing aspect of AML is the absence of useful concepts that test teams may need, especially if they want to apply agile principles.

5.2 Proposed Novel Service Test Description

As described in the sections 5.1.1, 5.1.2, 5.1.3 and also 5.1.4, the introduced service description languages, service specifications and test specifications are not sufficient enough to be used within the proposed Test Creation Framework for value-added services (see section 4.3). Therefore, a novel description language has been developed within this research that fulfils all the relevant requirements, which are listed below:

- Machine readable and parsable – the output of the novel language is parsable for the Automatic Composition Engine (ACE) to enable the building of the formal behaviour model.
- Usability – the definition of instances of the novel description language is manageable and relatively easy to understand for the test developer.
- Traceability of use cases – to support the agile aspects (such as the possibilities of rapid prototyping and a better involvement of the service customer) of the overall methodology (see Figure 4.3), a mapping to the use cases specified in the “Structured Requirements” document is provided.

- Preciseness – the behaviour, such as potential actions and events that might occur, is determinable in a precise manner.
- Functional specification – within the use case-based specifications, a complete description of possible behaviours exists (both basic and alternative flows).
- Reusability aspects – the description language contains components that can be applied to various scenarios and are reusable for different kinds of value-added services in order to fasten and simplify the definition process.
- Test data integration and parameterisation – the description language supports the usage of appropriate test data and allows parameterisations of reusable test data templates.
- SUT interface description – the execution of test cases within a test execution environment requires knowledge about the SUT (e.g. Service Access Point (SAP)). This information is included within the novel description language.
- Extension support – the description language shall support later enhancements (such as including new reusable components).
- Compliant to value-added communication services – the description language contains typical value-added service-related aspects (e.g. integration of multimedia protocols).

The upcoming section introduces the proposed novel description language called *Service Test Description*.

5.2.1 Structure of Service Test Description

This research work led to the definition of a new description language, the “Service Test Description”, which is abbreviated as STD in the following. The term itself implies that the description language contains both service-specific (“Service”) and test-specific (“Test”) properties. In fact, the STD can be regarded as being a combination of a service specification defining service-related information and behaviour, and a test specification including the determination of test components and test data.

The general overview of the structure of the STD is displayed in the following Figure 5.12.

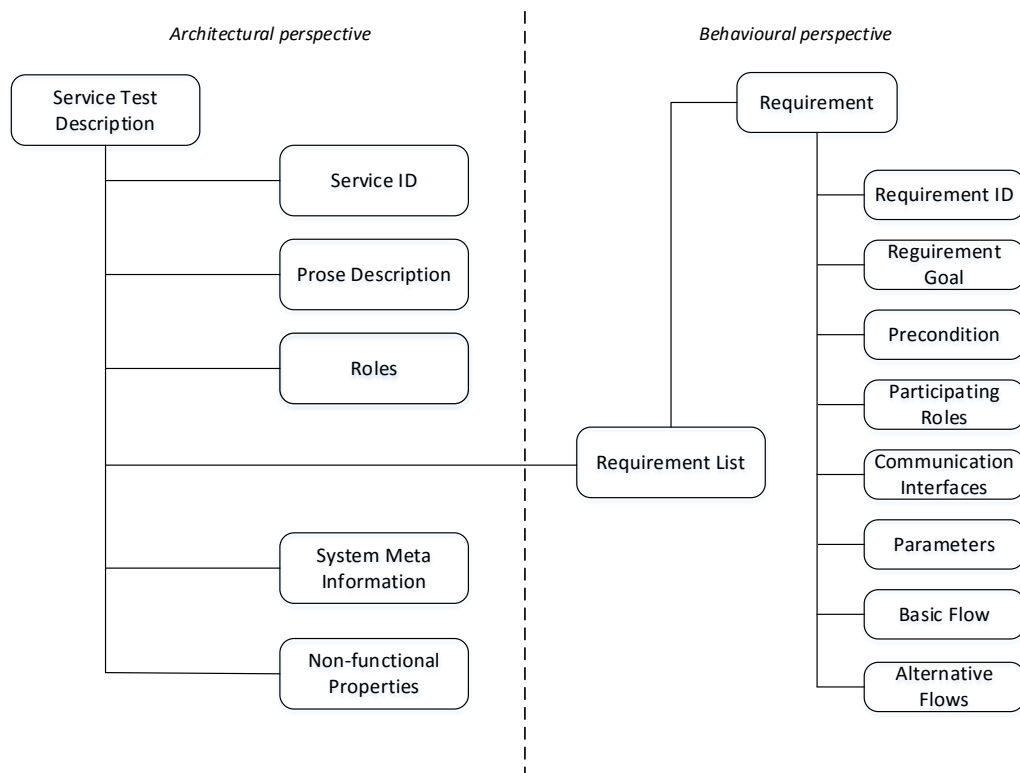


Figure 5.12: Structure of Service Test Description

On the basis of the illustration, also the differentiation between the *architectural perspective* and the *behavioural perspective* within the STD becomes evident. This principle has been derived from both UTML (see section 5.1.3) and U2TP (see section 3.3.1). However, the focus of what is part of the two perspectives and how these parts are described differs from the proposed STD. Also, UTML and U2TP are typical approaches to define test specifications or rather test models whereas STD, besides including test-specific parameters, also contains information that are usually related to service specifications such as, for instance, the description of requirements.

5.2.2 Architectural Perspective

The architectural perspective of the STD illustrated in the treelike Figure 5.12 contains at first the *Service Test Description* element, which is the root for every instance of an STD. Underneath this element, there are the five fields *Service ID*, *Prose Description*, *Roles*, *System Meta Information* and *Non-functional Properties*.

The *Service ID* is an identifier for the value-added service that is about to be tested and is a term containing a series of alphanumeric characters. It should already be defined within the “Structured Requirements” document by the service analyst. The test developer can just reuse the given term and select it as *Service ID* in the STD. This is important because the service developer will also use the term as code name for his development project. Using the same identifier, the test and development processes can be easier matched. This allows an easy test management throughout the development phase. The *Service ID* will also be used for further processing regarding the naming of the derived behaviour models, test cases and the test suite.

The *Prose Description* documents the value-added service's functionality from the potential user's point of view. The description is written by the test developer and should serve him as a reminder of the service's characteristics and main functionality. It also helps him to distinguish between diverse projects that he might have to deal with. There is no defined guideline how to write the *Prose Description*, but it should be brief and concise and it should not contain any pronouns to avoid misunderstandings. The *Prose Description* does not play a role in the further formal processing of the STD.

The most important part within the architectural perspective of the STD is the *Roles* field. It stands for a list of participating entities that communicate with the value-added service by exchanging signals and data on the one hand and that are external to the service environment (e.g. application server) on the other. The *Roles* most likely bear a resemblance to actor elements known from traditional use case modelling (Jacobson *et al.*, 1992), "Structured Use Case Models" (see section 5.1.1), RUCM (see section 5.1.2) and other use case-based approaches. Like actors in the UML context, the *Roles* in STD define a potential behaviour that has to be further specified. However, there is a difference between actors and *Roles*. According to (OMG, 2011a), actors "may represent human users, external hardware, or other subjects". This is a very general statement and allows diverse assumptions. Contrary to the view on actors in UML, the *Roles* in STD represent only specific external hardware or software that can interact with the value-added service via communication protocols such as HTTP, RTP or SIP. The importance of the *Roles* for the STD lies in the fact that based on the chosen *Roles* for a value-added service, sets of predefined test modules for the test execution environment can be automatically derived and afterwards instantiated. This is one major aspect of reusability in the proposed approach and will be further described in the upcoming section. Examples for *Roles*

applied in the STD can be, for instance, a SIP phone (or rather VoIP phone) or a web browser. A SIP phone can either be existent as hardware or software whereas the web browser is, of course, software-based. Both example *Roles* are able to communicate via their applied communication protocols (SIP for SIP phone and HTTP for web browser) with value-added services.

The next field within the architectural perspective, the *System Meta Information*, contains properties for the configuration of the SUT. This information is relevant for the TTCN-3 test suite that will be automatically generated based on the STD input. Each TTCN-3 test suite requires a test configuration containing parameters of the addressability of a service, such as the service URI, IP addresses, port numbers and the transport protocol (such as UDP, TCP or SCTP) used for the message transfer.

The final field determined in the architectural perspective of the STD is the *Non-functional Properties*. Analogous to the *Prose Description*, this information does not have a direct influence on the formal processing. It allows the test developer to capture information that is important for the service customer (such as quality of service experience, performance and usability) but does not describe specific functions.

5.2.3 Behavioural Perspective

As illustrated in Figure 5.12, the behavioural perspective of the STD comprises a list of requirements (*Requirement List*) to specify the functionality a value-added service has to accomplish. One *Requirement* as part of the STD defines one function of a service and generally includes a set of inputs, the relevant behaviour as well as expected outputs.

Requirement ID

In the specification of a *Requirement*, well-defined fields have to be determined. First, a *Requirement* needs a unique identifier. The naming, which is subjected to a special naming convention, can be done in the field *Requirement ID*. Starting from the first specified *Requirement*, the first unique identifier will be named “Req01”. Further requirements will be named by the prefix “Req” followed by the incremented number of *Requirement*.

Requirement Goal

The next field is called *Requirement Goal* and is comparable to the *Prose Description* field within the architectural perspective of the STD. Here, the test developer can specify in a very short natural language-based prose text the main objective of the corresponding *Requirement*.

Precondition

The *Precondition* field in the STD is comparable to the preconditions known from traditional use case specifications, but the notation applied here formally defines the statement as it is not based on natural language. In general, precondition statements indicate what has to have happened before the current function (or *Requirement*) is activated. In the context of STD, the *Precondition* enables the establishment of dependencies between *Requirements*. Figure 5.13 demonstrates how this can be visualised.

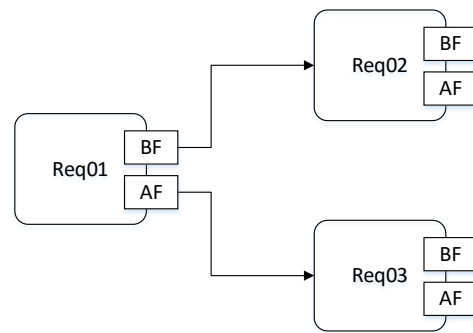


Figure 5.13: Dependency of Requirements through Preconditions

A *Requirement* in the STD contains flows of behaviour, exactly one *Basic Flow* (BF) and at least one *Alternative Flow* (AF). This is exemplified in Figure 5.13 with three *Requirements*, “Req01”, “Req02” and “Req03”. The connection originating from the BF of “Req01” to the target “Req02” determines that “Req02” actually depends on the BF of “Req01”. Similarly, the second connection in the figure determines that “Req03” depends on the AF of “Req01”. In the STD, these dependencies are stated through the *Precondition* field. In the definition of “Req02”, the field would contain the value “Req01”. This would set the BF of “Req01” to have happened before “Req02” can be activated. For “Req03”, the value of the *Precondition* field is “Req01.AF”. So, regarding AFs, the id of the *Requirement* has to be set followed by a full stop and the id of the AF itself. Theoretically, a *Requirement* can have more than one *Precondition*. This is specified by an ordered comma-separated list of the values.

Participating Roles

In the next field within a *Requirement* definition, the *Participating Roles* are selected. The *Roles* field within the architectural perspective already specified all the participating entities that shall interact with the value-added service. The *Participating Roles* only contains selected *Roles* from the architectural perspective that are specifically playing a

role in the current *Requirement*. The following table shows an excerpt of a STD determining two different *Participating Roles*, “SIP phone” and “Web browser”.

Table 5.4: Excerpt of example STD containing two example Participating Roles

Participating Roles	<ul style="list-style-type: none"> • SIP phone: [s] • Web browser: [w]
----------------------------	--

Besides the mentioning of the concrete names of the *Participating Roles*, the test developer also has to define aliases for them (“[s]” for SIP phone and “[w]” for Web browser). They can be used within the complete STD as identifier for the relevant *Role*.

Communication Interfaces

The *Communication Interfaces* (CI) field contains the most relevant information regarding the aspect of reusability. In STD, the CIs are defined as part of the SUT. In fact, they represent the points of interaction between the currently specified value-added service, also referred to as SUT, and the participating entities defined as the *Participating Roles*. The following Figure 5.14 illustrates the relationship between *Roles* and CIs.

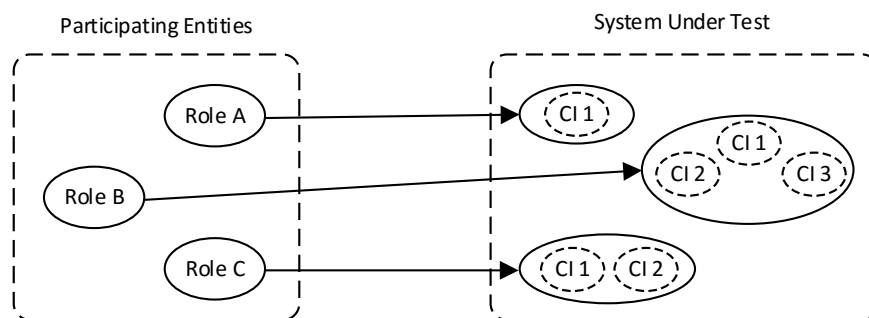


Figure 5.14: Relationship between Roles and CIs

One *Role* provides a potential functionality (or rather behaviour) that can be applied by the SUT when it communicates with the specific *Role*. The complete scope of potential functionality is represented by all CIs that are assigned to that *Role*. In Figure 5.14, there

are three different CIs (CI1, CI2, CI3) defined for *Role B* that can be applied by the SUT. By selecting one specific CI, for instance “CI2”, one aspect of the complete scope of functionality *Role B* provides is selected.

To show the relevance of CIs in the STD and how they are identified for *Roles*, “SIP phone” is used as an example *Role*. According to (ITU-T Q.3948, 2011) and (ITU-T Q.3949, 2012), a SIP entity can be described as a so-called SIP multimedia communication terminal that comprises all the functionality displayed in Figure 5.15. Permission to reproduce Figure 5.15 has been granted by ITU.

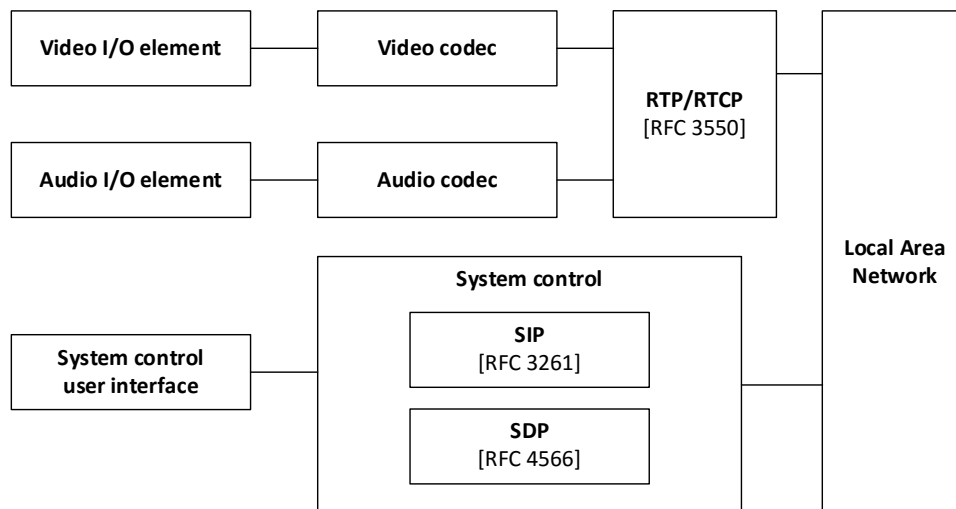


Figure 5.15: SIP multimedia communication terminal (ITU-T Q.3948, 2011)

Following the specification of a SIP multimedia communication terminal, a SIP phone as a SIP entity can be seen as an instance of the terminal. Correspondingly, a SIP phone, as well as any other SIP entity, has to be able to instantiate and terminate SIP sessions using the SIP protocol (IETF RFC 3261, 2002) and also the SDP protocol (IETF RFC 4566, 2006). Additionally, it has to be able to exchange multimedia data, either audio and/or video, via RTP (IETF RFC 3550, 2003). Based on these diverse aspects of functionality

provided by a SIP phone, the CIs can be derived. This is illustrated in the following Figure 5.16.

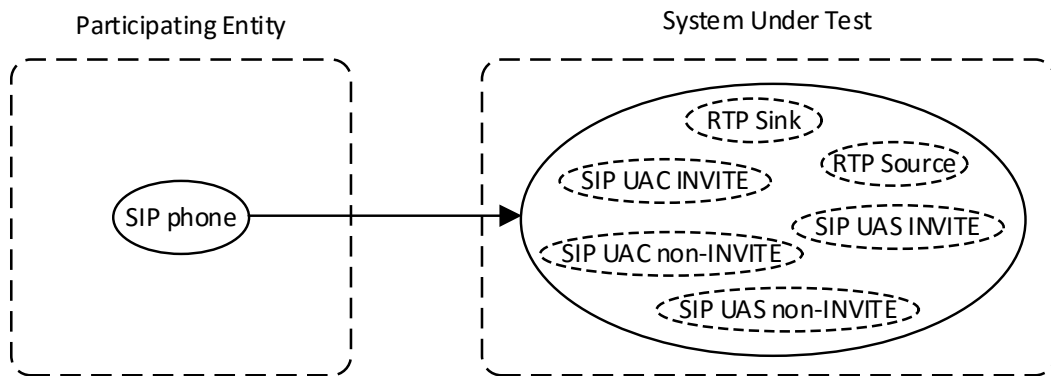


Figure 5.16: The Role SIP phone with its corresponding CIs

Here, six corresponding CIs have been identified for the *Role* SIP phone on the side of the SUT. The RTP CIs either represent the sending of RTP streams from the SUT to the SIP phone (*RTP Source*), or alternatively, from the SIP phone to the SUT (*RTP Sink*). The SIP CIs have been derived from the transaction state machines described in section 2.2.2. They define the handling of messages being initially sent from the SUT to the SIP phone (either *SIP UAC INVITE* for sending INVITE requests or *SIP UAC non-INVITE* for sending any type of SIP request different from INVITE requests) or from the SIP phone to the SUT (either *SIP UAS INVITE* for receiving INVITE requests or *SIP UAS non-INVITE* for receiving any type of SIP request different from INVITE requests). For the SDP protocol, no separate CI has been identified because SDP is usually embedded into SIP messages, for instance within INVITE or ACK requests or within 200 OK responses.

Overall, the determined CIs represent standard behaviour of the *Role* SIP phone. Of course, this aspect can be generalised. A set consisting at least of one CI is assigned to

each definable *Role* within the STD. Regarding the aspect of reusability in the novel approach, it is important to mention that the predefined test modules as part of the Test Modules Repository within the proposed Test Creation Framework will be automatically selected based on the determined CIs. So, for every determined CI, a corresponding test module has to exist in order to run the process.

The specification of CIs within a *Requirement* in the STD will be exemplified by means of the following Table 5.5.

Table 5.5: Example of specifying CIs in STD

Communication Interfaces	<ul style="list-style-type: none">• SIP UAS non-INVITE: [s1] → channel a• SIP UAC non-INVITE: [s2] → channel b
---------------------------------	---

The CIs *SIP UAS non-INVITE* and *SIP UAC non-INVITE* are selected for the example *Requirement*. Similar to the specification of *Participating Roles* in Table 5.4, aliases are assigned to the CIs. Here, a naming convention has to be followed to easily figure out which CI is assigned to which *Role*. Therefore, the CIs contain the same identifier as the corresponding *Role* followed by a number that increases with every further added CI. If the alias of SIP phone is “[s]”, the alias of the first mentioned CI will be “[s1]”. Besides the name of the CI and its alias, also a corresponding so-called *channel* is set. The significance of the *channel* will become apparent in the *Flow Definition*, but the following Figure 5.17 demonstrates the meaning of it.

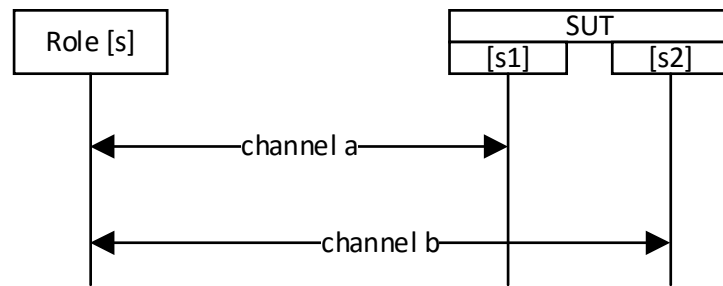


Figure 5.17: Significance of channel for Roles and corresponding CIs

As shown above, a *channel* represents the communication channel between a *Role* as participating entity and one of its corresponding CIs which is part of the SUT.

Parameters

The relevance of *Parameters* within a *Requirement* definition is very significant as they enable a great variability, especially regarding the CIs. As mentioned before, each CI in the STD can be assigned to a predefined test module within the framework. A detailed structure of a test module is explained in section 6.2, but it is worth mentioning that it describes behaviour that is common to the CI. The test modules also include variables that are instantiated from abstract data types which represent a communication protocol message (e.g. SIP request or SIP response). In the approach, each request-response protocol contains an abstract data type for its request and its response messages (see section 6.3). The variables within the test modules can be modified (or parameterised) by the STD through the *Parameters* field. Here, STD variables are instantiated and assigned the variables of a corresponding test module. The way how to do the assignment and the following modification is discussed in the following. In general, a communication protocol message as part of a CI (or rather test module) is a collection of data fields that build a compound domain. In STD, the compound domain concept has been derived from (Xiaoping and Maag, 2013) and can be defined as follows:

Definition: A *compound* value v of length $n > 0$ is defined by the set of pairs $\{(l_i, v_i) \mid l_i \in L \wedge v_i \in D_i \cup \{\varepsilon\}, i = 1 \dots n\}$, where $L = \{l_1, \dots, l_n\}$ is a predefined set of labels and D_i are data domains. Based on this, the *compound domain* is the set of all values with the identical set of labels and domains defined as $\langle L, D_1, \dots, D_k \rangle$.

(Xiaoping and Maag, 2013) discuss that for any given network protocol P it is possible to define a compound domain M_P by the set of labels and data domains that are defined in the underlying protocol specification. Accordingly, a *message* of a protocol P , independent of whether it is a request or a response type, is any element $m \in M_P$.

In the following Table 5.6, an example parameterisation of a SIP MESSAGE request is demonstrated.

Table 5.6: Parameterisation of an example SIP MESSAGE request

Parameters	<pre>var m = [s2]->s_Request; m = {(Method, "MESSAGE"), (FromURI, "service@sip.de"), (ToURI, "bob@sip.de"), (Text, "Hello Bob!")}</pre>
-------------------	--

First, a local STD variable m using a syntax derived from well-known scripting languages (such as JavaScript) is initialised. Now, m is assigned the variable “s_Request” from the CI “[s2]” which refers to the corresponding test module. In the syntax, this assignment is performed by the arrow symbol. The real parameterisation of the variable takes place subsequently and is based on the key-value pairs defined by the compound domain. Conveniently, only the labels *Method*, *FromURI*, *ToURI* and *Text* and the corresponding values were used to specify the SIP MESSAGE. Of course, a typical SIP request can be specified in more detail (see Figure 6.21). Regarding the value determination it is mentionable that quotation marks have to be used irrespective of whether the values are

alphanumeric or all kinds of numbers such as integers and floating point numbers. The example in Table 5.6 also shows how the *Parameters* field in STD allows differentiations within the CIs. In general, the *SIP UAC non-INVITE* CI describes the initiation of a request from the SUT to a SIP phone that is different from an INVITE request. In the example, the SIP MESSAGE (IETF RFC 3428, 2002) was used, but through parameterisation, also the following request types could be defined:

- ACK, BYE, CANCEL, OPTIONS and REGISTER (IETF RFC 3261, 2002)
- PRACK (IETF RFC 3262, 2002)
- SUBSCRIBE and NOTIFY (IETF RFC 6665, 2012)
- PUBLISH (IETF RFC 3903, 2004)
- INFO (IETF RFC 6086, 2011)
- REFER (IETF RFC 3515, 2003)
- UPDATE (IETF RFC 3311, 2002)

Besides the possibility to parameterise variables within the *Parameters* field, it is also possible to access the values that were set. Like in many programming languages, such as Java or C#, fields or rather attributes of variables instantiated from complex data types can be accessed by applying the dot operator (“.”). This concept is reused here. For the local variable *m* defined in Table 5.6, accessing for example the field *method* would be written as follows:

m.method

The accessing of fields can be done within the *Parameters* field on the one hand, or alternatively, within the *Basic Flow* and *Alternative Flow* definitions. The return value of

this operation will be the currently stored value (e.g. “MESSAGE” referring to the definitions made in Table 5.6).

The *Parameters* field, besides defining and parameterising variables, also enables the definition of timers. The following Table 5.7 shows how this is realised.

Table 5.7: Instantiation of timers in *Parameters* field

Parameters	timer t1 = [s2]→timerF;
-------------------	-------------------------

The test module referring to the CI *SIP UAC non-INVITE* contains a list of timers (e.g. “Timer E”, “Timer F”, “Timer G”). In this example, “Timer F” was chosen and bound to the name “t1”. Within the *Basic Flow* and the *Alternative Flow*, the state of this timer can be verified within specified constructs (*if-then-else*).

Basic Flow and Alternative Flows

The concept of the *Basic Flow* and the *Alternative Flows* within the *Requirement* of an instance of the STD is derived from the RUCM method (see section 5.1.2). Besides the determination of the CIs and the parameterisation, the *Basic Flow* is the most significant part of a *Requirement*. In principle, it contains the descriptions of steps that have to be taken to achieve the main target (or goal as it is described in the *Requirement Goal* field) of the *Requirement*. Within the steps of the *Basic Flow*, possible alternative behaviour can occur. The effects of the alternative behaviour can be specified by means of the *Alternative Flows*. Theoretically, a *Requirement* can contain an infinite number of *Alternative Flows*, but it will always contain only one *Basic Flow*.

In order to define the steps within the *Basic Flow*, many documented approaches have been considered, for instance the RUCM method. However, textual use case design might

also be error-prone even if restriction rules are established to reduce the major problems of natural language-based descriptions, namely imprecision and incompleteness. Based on the requirements on the STD drawn up at the beginning of this section, it should be machine-readable so that it can be parsed by the Automatic Composition Engine (ACE) within the TCF. Also, the description has to take the reusability aspect into consideration. To sum up, a new language is required which enables the precise description of behaviour flows on the one hand and realises the reference to the reusable test modules within the TMR on the other.

As appropriate foundation of a language being able to meet the mentioned requirements, a process algebra notation has been found, the pi-calculus (Milner, 1992), (Milner *et al.*, 1992). In principle, the pi-calculus is a model “of communication systems in which one can naturally express processes which have changing structure” (Milner *et al.*, 1992). It belongs to the family of process calculi, which are mathematical formalisms for describing and subsequently analysing properties of concurrent computation, and is an extension of the Calculus of Communication Systems (CCS) (Milner, 1989). One major benefit of pi-calculus is the simple language it is based on to specify interactive message-passing programs. The language is also very expressive. However, the original pi-calculus notation defined by (Milner, 1992) does not contain primitives such as numbers, booleans, variables, conditions or terms.

Through the syntax of pi-calculus, processes and channels can be represented. A *process* is an abstraction of an independent thread of control whereas a *channel* is an abstraction of the communication link between two processes. Interaction between processes is

enabled by sending and receiving messages over channels. The grammar for processes in pi-calculus is specified as follows.

Assume that there exists a countable infinite set of *names* N . Let x, y, \dots range over N and let P and Q denote processes. Then:

- $P \mid Q$ denotes a process composed of P and Q running in parallel (an example of this is illustrated in Figure 6.35).
- $a(x).P$ describes a process that receives an input over channel a , binds the result to x and then proceeds with P .
- $\bar{a}(y).P$ describes a process that sends out y over the channel a and then proceeds with P .
- $!P$ denotes that an infinite number of copies of P runs in parallel.
- 0 denotes that the current process is terminated.

Generally, the stated constructs for pi-calculus are sufficient to determine concurrent behaviour. Regarding the *Basic Flow* of the STD, however, some limitations of the pi-calculus concept have to be reconciled by means of minor enhancements. Firstly, the *names* being sent and received over the channels have to be substituted by terms. Such terms are placeholders for simple names, variables or even functions that expect input parameters (e.g. (x)) and of course return a value to be either sent or received. The concept of terms has been derived from (Abadi and Fournet, 2001) and (Abadi and Fournet, 2004). Another limitation of the standard pi-calculus is the syntax lacking the definition of conditional constructs such as *if-then-else*. In applied pi-calculus approaches such as in (Ryan and Smyth, 2011), the concept of including *if-then-else* constructs has

already been discussed. However, the checking of values of complex variables has not been considered in this approach. An example usage of the construct can be as follows:

$$\text{if } (x > 5) \text{ then } P \text{ else } Q$$

The expression states that if a number stored in a variable x is higher than the value of “5”, the current process proceeds with P , otherwise with Q .

Applying the pi-calculus with the proposed enhancements, the *Basic Flow* and also the *Alternative Flows* can now be determined. However, the way the pi-calculus is applied in the STD may differ minimally from its original application. The pi-calculus is used in the following way:

- *Basic Flows* and *Alternative Flows* within *Requirements* are generally described by means of enhanced pi-calculus descriptions.
- The *Basic Flow* contains possible transitions to existing *Alternative Flows* within the *Requirement*.
- Each *Alternative Flow* has to be specified with a unique identifier (e.g. “AF1”).
- A *Basic Flow* and each additional *Alternative Flow* within a defined *Requirement* are specified by their own processes $P, Q, R, \dots \in \rho$. *Basic Flow* processes and *Alternative Flow* processes are always running sequentially and not concurrently. So, if a step within a *Basic Flow* leads to an *Alternative Flow* (possibly because of an *if-else-then* construct), the process of the *Basic Flow* terminates automatically and is substituted with the process of the *Alternative Flow*.
- The pi-calculus description focuses on the potential behaviour of the considered value-added service (SUT), not on the external system (such as the participating

entities). So, the test is not in the focus of the pi-calculus description, but the service.

- Potential behaviour is described through *channels* and not through concurrent processes. In fact, the *Basic Flow* (and also the *Alternative Flows*) will always include the major process P and an implicit process Q representing the test environment.
- A process defined in one *Basic Flow* describes a period of time within the lifetime of the SUT (or rather value-added service). The sum of all *channels* represents the possible communication channels of the SUT to the test environment.
- The pi-calculus *channels* are directly mapped to the communication *channels* which describe the message exchange between the *CIs* (as part of the SUT) and the *Participating Roles* (see Figure 5.17).
- The variables that are about to be sent and received along the *channels* are defined within the *Parameters* field.
- Within the *if-else-then* constructs, fields of variables can be accessed through the dot operator (“.”) as well as states of defined timers can be verified (e.g. “timeout”).

In order to illustrate the approach with the pi-calculus-based *Basic Flow* and *Alternative Flow* definitions, a sample specification by means of the STD will be discussed in the following section.

5.2.4 Sample Specification with Service Test Description

The sample chat service introduced in section 5.1.1 will be applied. The service will be reused in a simplified form for the prototype validation in section 8.3, and a specification of the “Send Message” use case is given here for illustration. As discussed in the previous sections, first the architectural perspective has to be specified (see Table 5.8).

Table 5.8: STD architectural perspective of simplified sample chat service

Service ID	Chat Service
Prose Description	A chat communication should be provided. The service users are able to log in to the system and log out again. While being logged in, the service user can enter chat rooms and leave the chat rooms again. The service user can also send textual chat messages. The Administrator of the chat service can add new users to the system and is also capable of erasing existing users from the system. The Administrator can also create new chat rooms and erase old chat rooms.
Roles	<ul style="list-style-type: none"> • SIP phone: [admin] • SIP phone: [sender] • SIP phone: [recipient]
System Meta Information	ServiceURI: sip:chat-service@vas.de Protocol: UDP
Non-functional Properties	None

As described in the section 5.2.2, within the architectural perspective of the STD, the *Service ID* has to be set at first (“Chat Service”). The *Prose Description* describes the main functionality the sample chat service has to deliver as precise as possible. Three different *Roles* have been identified for the service and all are acting as SIP phones. The “[sender]” and the “[recipient]” are *Service Users* (see Figure 5.3) whereas the “[admin]” is, of course, the *Administrator*. A further information regarding the service addressability is given through the service URI. *Non-functional Properties* are not specified for the sample chat service.

After the architectural perspective is defined, the behaviour has to be specified. To specify the “Send Message” use case, a *Requirement* is defined within the sample chat service STD instance (see Table 5.9).

Table 5.9: STD Requirement definition for “Send Message” from sample chat service

Requirement ID	Req03
Requirement Goal	Service User [sender] sends a text message to another Service User [recipient] and gets informed whether the transmission was successful.
Precondition	Req02
Participating Roles	<ul style="list-style-type: none"> • SIP phone: [sender] • SIP phone: [recipient]
Communication Interfaces	<ul style="list-style-type: none"> • SIP UAS non-INVITE: [sender1] → channel a • SIP UAC non-INVITE: [sender2] → channel b • SIP UAC non-INVITE: [recipient1] → channel c
Parameters	<pre> var initMessage = [sender1] → r_Request; var forwMessage = [recipient1] → s_Request; var okMessage = [sender2] → s_Request; var errorMessage = [sender2] → s_Request; timer t1 = [recipient1] → timerF; initMessage = {(Method, "MESSAGE"), (Text, "Hello Bob!")} forwMessage = {(Method, "MESSAGE"), (Text, initMessage.Text)} okMessage = {(Method, "MESSAGE"), (Text, "ok")} errorMessage = {(Method, "MESSAGE"), (Text, "Message not received")}</pre>
Basic Flow	$P \stackrel{\text{def}}{=} a(\text{initMessage}). \bar{c}(\text{forwMessage}). \text{if } (t1.\text{timeout}) \text{ then } Q \text{ else. } \bar{b}(\text{okMessage}). 0$
Alternative Flow (AF1)	$Q \stackrel{\text{def}}{=} \bar{b}(\text{errorMessage}). 0$

Initially, the *Requirement ID* has to be set and a *Requirement Goal* is specified. The *Precondition* field contains the value “Req02”. Although this *Requirement* is not determined here, the specified behaviour within its respective *Basic Flow* has to happen before the *Basic Flow* of “Req03” begins. In this example, “Req02” indicates the entering

of both *Service Users* into a chat room. Notice that “Req02” itself includes a *Precondition*, namely “Req01”, which describes the login process of both *Service Users*. So, the *Service Users* have to be logged in to enter a chat room (“Req01” → “Req02”) and they have to have entered a chat room before sending messages (“Req02” → “Req03”). In the *Participating Roles* field, both *Service Users* “[sender]” and “[recipient]” are included. The *Administrator* does not participate within the “Send Message” *Requirement*. Three different *CI*s have been identified. The SUT requires two channels *a* and *b* to communicate with the initial sender of the text message. In channel *a*, the SUT is acting as SIP UAS whereas in channel *b*, it is acting as SIP UAC. Regarding the recipient of the text message, the SUT only requires one channel *c* where it is acting as SIP UAC. The *Parameter* field includes the definition of several variables all representing SIP MESSAGES, either being sent from the sender to the SUT (“initMessage”), from the SUT to the sender (“okMessage”, “errorMessage”) or from the SUT to the recipient (“forwMessage”). Additionally, the timer *F* of the *SIP UAC non-INVITE CI* is defined. Subsequently, the *Basic Flow* is defined. First, it denotes the SUT to receive the SIP MESSAGE “initMessage” over channel *a* and then consequently sends the SIP MESSAGE “forwMessage” over channel *c*. In the next step, the state of the timer “t1” is checked. If it has not timed out, the SUT sends out the SIP MESSAGE “okMessage” over channel *b* and the *Basic Flow* terminates afterwards. Otherwise, if the timer has timed out, the *Alternative Flow* “AF1” is activated. Here, a different SIP MESSAGE “errorMessage” is sent by the SUT over channel *b*. Then, also the *Alternative Flow* terminates.

To sum up, this section introduced the novel STD, a description language containing aspects of typical requirements specifications as well as relevant information of the test

environment. In contrast to the introduced specification languages in section 5.1, the STD fulfils all the relevant requirements stated in section 5.2. At first, the relevant data specified in both the architectural and behavioural perspective can be read and interpreted by a machine as it is existing either in a structured or formal manner. Although formality in languages usually means that the compilation of the language is difficult for the modeller or creator, the pi-calculus-based descriptions to specify the *Basic Flows* and *Alternative Flows* are straightforward and can be defined in a very compact and intuitive way. At the same time, the descriptions have a very precise meaning and do not allow any ambiguities. A further requirement mentioned was the possibility to trace the requirements within the language. This aspect is supported by the STD, as it is indeed possible to map the use cases defined in the “Structured Requirement” document to the *Requirements* within the STD. Moreover, the STD itself supports tracing within an instance through the *Precondition* field. *Requirements* that are based on each other can easily be specified. Another important aspect, the possibility to reuse certain aspects of behaviour, is also a very important part of the STD. Through *Roles* in combination with the *CIs* that are belonging to the SUT, sorts of reusable components can be derived. This concept is discussed in more detail in section 6.4. Of course, the concept will clarify as soon as the concept behind the reusable test modules is described in section 6.2. Further specified requirements are covered, such as the test data integration and parameterisation. This is a very relevant part of the STD and can be realised through the *Parameters* field within a specific *Requirement*. The SUT interface description, which is also specified as a major requirement, can be defined through the *System Meta Information* within the architectural perspective of an STD instance. If demanded, further fields can be added here. In general, the STD allows extensions without having to change the specification

language. For example, new *Roles* besides the already mentioned SIP phone and Web browser can be added. Of course, this also requires the identification of the corresponding *CIs* and regarding the Test Creation Framework, the definition of the reusable test modules. Finally, the compliance to value-added services is given as standard communication protocols are supported, such as SIP and RTP and also HTTP.

5.3 Comparison of Service and Test Specification Languages

As predicted in the introduction, this section compares the service and test specification languages from section 5.1 and the proposed novel STD from section 5.2 to demonstrate their relative assets and weaknesses. Considering that, diverse requirements have been stated. To evaluate the specification languages with regard to the requirements, a rating will be applied to them in the upcoming Table 5.10 by using a three-level scale. The scale contains the following ratings:

- (+): the specification language fulfils the requirement completely.
- (o): the specification language fulfils only basic aspects of the requirement.
- (-): the specification language does not fulfil the requirement.

The requirements on a novel specification language have been derived based on their potential application within the proposed TCF (see section 4.3). A major advantage of the TCF is that only an instance of the specification language has to be created manually. Based on this instance, a behaviour model is automatically built, test cases are automatically derived and generated and subsequently executed against the SUT.

Therefore, high demands are placed on the specification language as it presents the foundation of the quality of testing.

The requirements stated in section 5.2 are as follows:

- Machine-readable and parsable by a machine.
- Usability by test developers who specify the considered value-added services.
- Traceability of use cases to enable an easy transition from standard use case description.
- Preciseness to avoid ambiguity within the specification.
- Support for functional specification (e.g. flow descriptions or rather use case descriptions).
- Reusability aspects to simplify and fasten the definition process.
- Test Data integration and parameterisation (e.g. parameterise variables).
- SUT interface description to already specify the addressability of the value-added service in order to achieve a fully automated process.
- Extension support to allow further changes and enhancements.
- Compliance to value-added telecommunication services.

Table 5.10: Comparison of specification languages

Characteristics	Specification Language			
	Structured Use Case Models	RUCM	UTML	Proposed STD
Readability/Parsability	+	-	+	+
Usability	o	+	+	+
Traceability of use cases	+	+	-	+
Preciseness	o	o	+	+
Functional specification	+	+	+	+
Reusability aspects	-	-	-	+
Test Data	-	-	+	+
SUT interface description	-	-	+	+
Support for extensions	-	o	o	+
Compliance to services	-	-	+	+

Readability and parsability of the specification language is best supported by UTML, the Structured Use Case Models and the proposed STD as they rely on formal models, descriptions or metamodels. RUCM is natural-language-based and therefore not easily parsable. The aspect of *Usability* is fulfilled by RUCM, UTML and the proposed STD because the procedure within the compilation phase is well-defined. The *Traceability of use cases* is supported by all approaches that are actually based on use case design. This is the case in Structured Use Case Models, RUCM and the proposed STD. Regarding the *Preciseness*, the Structured Use Case Models and RUCM have weaknesses as they are either natural language-based or allow loose determinations. Every specification language is meant to provide a *Functional specification* and therefore this requirement is fulfilled by all four languages. A unique position feature of the proposed STD emerges regarding the aspect of *Reusability*, which is not supported by any other specification language. The *Test Data* integration is supported by UTML as it is a typical test specification language. Like in the proposed STD, abstract data types and variables exist that can be

parameterised within the approach. The same argument can be emphasised regarding the *SUT interface description* which is also not supported by the Structured Use Case Models approach and RUCM. The *Support for extensions* is not specified by RUCM and UTML, but it should be possible. The Structured Use Case Model is based on a defined metamodel and does not allow any further extensions. Finally, the *Compliance to services*, especially value-added telecommunication services, is fulfilled by UTML and the proposed STD. Both support the integration of SIP.

5.4 Conclusion

Different specification languages have been presented within this chapter in order to compare them against the proposed novel STD. The STD has been described with all its features, starting from the simple way it is compiled, its ability to be parsed and interpreted, its integration of reusable modules (*CI*s) and of course, its property to allow traceability of use cases. The language has been compared to other specification languages by means of stated requirements. It has been identified that all requirements are fulfilled by the novel STD.

Up to now, a solid basis has been defined to implement the TCF. However, new essential questions arise, for instance, regarding the structure and definition of the reusable test modules. Also the relationships between the *CI*s within an STD instance and the reusable test modules has to be clarified. Furthermore, an algorithm needs to be introduced which builds the formal behaviour models based on an STD instance. All of these questions will be discussed in the upcoming chapter.

6 Reusable Test Modules and Behaviour Model Generation

“In most engineering disciplines, systems are designed by composing already existing components that might have been used in other systems” (Sommerville, 2012). In this chapter, this statement will be taken up for testing, or to put it more precisely, a novel concept of reusable test modules will be introduced. These reusable test modules are part of the proposed TCF and are used together with the STD in order to build behaviour models from which test cases can be derived later on (see section 4.3). The following Figure 6.1 illustrates a simplified flow chart based on the proposed TCF (see Figure 4.5) showing the generation of behaviour models based on STD instances.

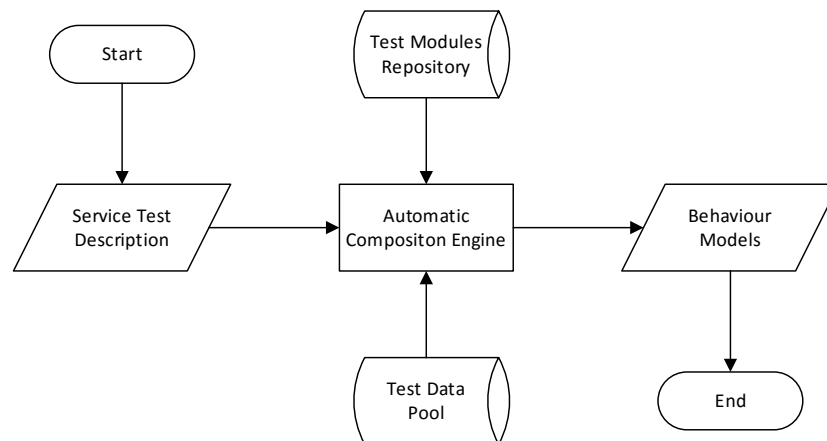


Figure 6.1: Generation of Behaviour Models based on STD and reusable test modules

The main concern of this chapter is the introduction of the composition algorithm in the Automatic Composition Engine (ACE) which combines instances of the reusable test modules and generates behaviour models. The application of such an algorithm, however,

requires a deep knowledge of its inputs (see Figure 6.1) such as the Service Test Description (STD) instance, a selection of reusable test modules from the Test Modules Repository (TMR) as well as test data from the Test Data Pool (TDP). Whereas the structure of the STD has already been discussed in chapter 5, the upcoming sections 6.1 and 6.2 deal with the reusable test modules. First, an appropriate modelling notation is selected in section 6.1 based on general requirements for model-based notations and specific requirements which take the concept behind the proposed TCF (see Figure 4.5) and the properties of value-added services into consideration. Furthermore, the section introduces the TU concept which allows a view on client- and server-based cores and in parallel enables the underlying semantics used in the selected modelling notation. Section 6.2 introduces the architecture component within the proposed TCF, the Test Modelling Environment (TME), which enables the modelling and definition of new reusable test modules. The test data integration is discussed in section 6.3 and illustrates how the abstract data types and concrete test templates for request and response messages of the given protocol SIP are stored. This concept can be reused for further application layer protocols (such as HTTP). The main process and the major outcome of this chapter will be explained in section 6.4. The main task of this ACE algorithm is to produce a well-defined output, the behaviour models, based on the previously defined input, the STD instance. The STD instance has to exist in a parsable form to be interpreted by the ACE algorithm. After the algorithm has read the STD instance, the next step is to identify the appropriate reusable test module instances by parsing the specified *CommunicationInterfaces* within the STD instance. Then, the parameterisation of the reusable test module instances is performed by reading the *Parameter* field. Finally, the ACE algorithm realises the composition of the reusable test modules according to the

content specified in the *Requirements* of the STD instance. The different steps that have to be taken by the algorithm during the composition phase can be derived from the different categories of steps existing in the pi-calculus-based behavioural description. The result at the end is a list of behaviour models. Each behaviour model within this list is related to a specified *Requirement* within the STD instance.

6.1 Notation for Behaviour Modelling

To generate appropriate functional test cases and execute them against a System Under Test (SIP AS with deployed value-added services), a formal modelling notation needs to be selected that enables a behavioural description of the service. However, regarding the proposed framework (TCF), a number of requirements has to be fulfilled by such a modelling notation. The upcoming section gives an overview of the essential requirements, presents possible modelling notations and gives reasons for the selection of one specific modelling notation.

6.1.1 Evaluation of Potential Modelling Notations

With regard to “Model-Based Testing” described in section 3.2.4, behavioural aspects of a system or service can be specified by means of modelling notations. According to (ETSI ES 202 951, 2011), such a modelling notation has to provide basic means for algorithmic design and data manipulation. The ETSI standard lists further general requirements that have to be fulfilled by potential modelling notations. The most relevant aspects are mentioned in the following (ETSI ES 202 951, 2011):

- The notation shall be based on unambiguous operational semantics.
- The notation shall support diverse simple data types such as boolean, integer and character strings.
- The notation shall support user-defined abstract data types.
- The notation shall support basic control structures like variables, assignment and conditional statements.
- The notation shall support advanced control constructs such as loops.

Considering these general requirements, the authors of (ETSI ES 202 951, 2011) point out that modelling notations for the specification of behaviour are limited to rule-based notations, process-oriented notations and Statecharts (Harel and Politi, 1998). Whereas Statecharts are clearly defined as a special presentation form for finite-state machines, rule-based notations and process-oriented notations each represent a group of more or less well-known modelling notations. Rule-based notations are “textual modelling notations where state transition rules describe the behaviour of the system” (ETSI ES 202 951, 2011). They can be referred to as extended finite state machines (EFSM) (Cheng and Krishakumar, 1993) or abstract state machines (ASM) (Börger and Stärk, 2003). In contrast, process-oriented notations focus on describing the activity of a system as a sequential process (or thread). During its lifetime, the process listens to inputs from its environment and also produces outputs. A well-known representative is the Business Process Execution Language (BPEL) (OASIS, 2007).

In order to find the appropriate modelling notations, further specific requirements have been determined. These requirements take the general requirements on the TCF (see section 3.4) as well as the properties of value-added services into consideration:

- The notation shall allow the definition of reusable test modules.
- The notation shall enable the composition of reusable test modules.
- The notation shall support the description of concurrent behaviour.
- The notation shall support temporal logic (e.g. timer integration).
- The notation shall deliver a standardised formal description.

First of all, the aspect of reusability is a major requirement a modelling notation has to fulfil. Reusability shall be provided by so-called reusable test modules. The characteristics of these reusable test modules and further information regarding their identification is discussed in section 6.2. As part of the modelling notation, a reusable test module shall exist in the form of a formal model which describes recurring behaviour. With regard to value-added services in the telecommunication domain, recurring behaviour can be for instance the sending or receiving of instant messages or the initiation and termination of audio or video calls. Such behaviour has to be specified in a generalised manner within a reusable test module. As soon as such a behaviour becomes relevant within a value-added service, the appropriate reusable test modules can be chosen and adapted to the given scenario, e.g. through parameterisation. All of the mentioned types of modelling notations enable the definition of reusable test modules although not all support the principle from scratch. In a rule-based approach with EFSMs, for instance, the concept of reusability has not been considered. However, behaviour of a test module can be defined within one state machine. This state machine can be stored and be reused as part of another state machine later on. As the EFSM-based state machines include variables, parameterisations at a given point can be performed. BPEL as representative for process-oriented notations also includes a concept of reusability (through so-called *Partner Links*). Within a BPEL process, the behaviour of a reusable test module can be

specified and be reused in any other BPEL process. The final notation, Statecharts, explicitly supports modularity through the defined concept of hierarchical states. Within such a hierarchical state, the behaviour of one reusable test module can be specified.

The next requirement is directly connected with the previous one, however, it refers to the composability of reusable test modules. It has to be clarified, if a modelling notation allows the integration of a test module at any given point within the overall model. It might also be relevant to modify the internal behaviour of a test module. In principle, EFSM-based approaches support the composability of test modules. Every state and every transition within a formal EFSM model describing the behaviour of a test module is visible and accessible. Therefore, any new transition can be included and a composition is supported. Although the BPEL process supports reusability in principle, a reused module in form of BPEL processes is treated like a black box. Only the input parameters of the test modules can be specified, no changes can be done within the behaviour definition of a reusable test module based on BPEL. As the syntax of Statecharts is very similar to EFSMs, the composability of test modules is also supported.

The next requirement is highly relevant for the implementation and test of value-added services, because especially message flows (e.g. SIP messages) are usually not exchanged in a fixed sequential order. For instance, if a value-added service sends two SIP INVITE requests directly one after the other to two different participants in order to instantiate a Third party call control (3PCC) call (IETF RFC 3725, 2004), the sequence of received messages such as “200 OK” response and ACK request cannot be determined. In fact, the sequence of messages can differ from one execution to the other. This aspect requires a modelling notation that supports the definition of concurrent behaviour. EFSM-based

approaches do not support concurrency. BPEL contains a special “Flow” element that enables the definition of parallel processes. Statecharts support concurrency through so-called concurrent hierarchical states. Within such a concurrent hierarchical state, it can be more than one state executing simultaneously.

The next requirement concerns the integration of timers. Within the specified behaviour, it shall be possible to determine that a timer has started or that a timeout occurred. Originally, EFSMs do not support timer integration. However, some EFSM-based approaches included the starting of timers within states and the timeouts as events on transitions (Wacht *et al.*, 2011a) or both as transition actions (Ernits *et al.*, 2006). BPEL supports timers through a special “onAlarm” element that corresponds to a timer-based alarm. Finally, Statecharts support timers the same way it has been described for the EFSM-based approach in (Wacht *et al.*, 2011a). As soon as a state is reached, a specific timer can be started. The timeout is then specified on a transition as an event.

Modelling notations such as EFSM, BPEL and Statecharts are mainly described graphically. For further processing of the underlying models, a formal description is required. This requirement on a modelling notation is essential for the proposed TCF and has to be fulfilled because of two reasons. Firstly, the reusable test modules have to be stored persistently in order to be selectable from the *Test Module Repository*. This can be done if a formal and textual representation of the modelling notation exists. Secondly, the generation of the behaviour models also requires a formal and parsable representation. Particularly the EFSM-based approaches lack standardised formal descriptions as there are many different notations. In contrast, BPEL processes can be serialised in a standardised XML-based language (OASIS, 2007). There is also a grammar-based

scheme defined which specifies the exact structure of the XML presentation. For the Statecharts approach, there is also a formal language called State Chart extensible Markup Language (SCXML) (W3C, 2015) exists which has been defined as World Wide Web Consortium (W3C) recommendation.

The analysis of the diverse modelling notations resulted in the following Table 6.1. It demonstrates an evaluation based on a rating scale that has been applied in Table 5.10.

Table 6.1: Comparison of potential modelling notations

Requirements	Modelling notations		
	EFSM (Rule-based notation)	BPEL (Process-oriented notation)	Statecharts
Definition of reusable test modules	o	+	+
Composition of reusable test modules	+	-	+
Support for concurrency	-	+	+
Support for timer integration	o	+	+
Existing standardised formal description	-	+	+

To sum up, Statecharts are the modelling notation of choice regarding the formal description of the reusable test modules and the behaviour models.

6.1.2 Relevant Portions of the Selected Modelling Notation

As elaborated in the previous section, Statecharts fulfil the requirements as modelling notation for the reusable test modules and behaviour models. However, not all aspects of the notation are required to create formal models in order to specify the behaviour of a

value-added service. The relevant components and aspects of Statecharts are described in the following.

Similar to other state machine-based notations, a Statechart is a finite set of states and transitions. According to (Harel, 1996) and (Harel and Kugler, 2004), there are two different types of states in a Statechart definition, *basic states* and *hierarchical states*.

Basic States

Basic states are not composed of other states and are therefore the lowest in the state hierarchy. Each state contains a set of transitions that define how the state reacts to events. In contrast to other state machine notations (such as EFSM-based approaches), a Statecharts basic state includes different action types, so-called *entry* and *exit* actions. They can appear associated with the entrance to or exit from a state. Figure 6.2 illustrates an initial state that is connected to a basic state (“State A”) by means of a default transition. The basic state itself is then connected to an end state, again through a default transition. Default transitions differ from standard transitions (which are connecting basic states and hierarchical states) in a way that they do not contain any information, such as events, actions or conditions (Chattopadhyay, 2013).

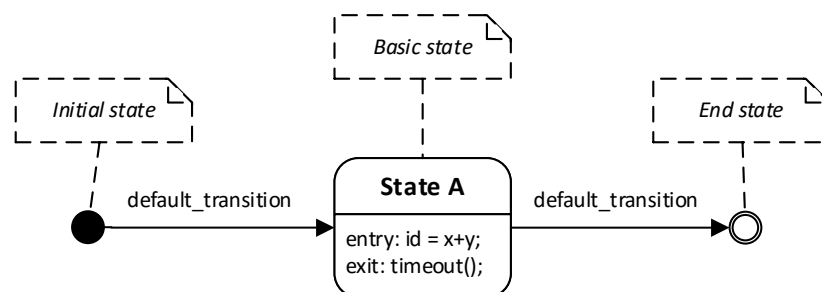


Figure 6.2: Statecharts basic state example

Figure 6.2 also shows what kinds of actions can be defined within a basic state. These can either be arithmetic operations of given variables known in the model (e.g. “ $id=x+y$ ”) or the invocation of known functions (e.g. “`timeout()`”).

Hierarchical States

Statecharts also allow the modelling of hierarchical states. In principle, hierarchical states are states that are able to contain other states. The Statecharts definition according to (Harel, 1996) makes a distinction between hierarchical OR-states and hierarchical AND-states. OR-states have substates related to each other by “exclusive or”. So, if an OR-state is active, only one of the internal substates will be active. The following Figure 6.3 illustrates the concept of OR-states.

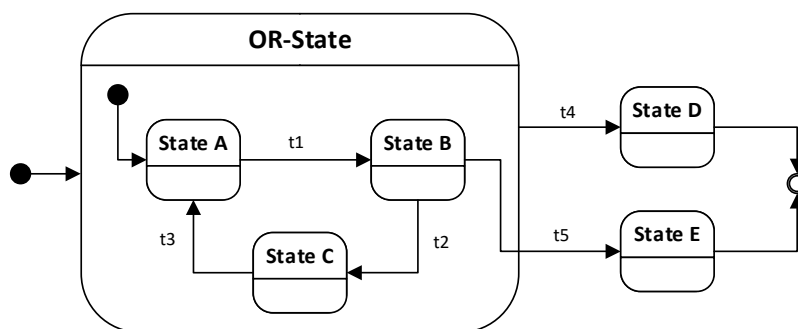


Figure 6.3: Hierarchical OR-state example

The example Statechart shows two initial states. The rule regarding initial states is that every Statechart model contains at least one initial state. Each hierarchical state within the Statechart has its own initial state to determine the initial entry point. The hierarchical OR-state contains a finite number of substates that are connected through transitions. In order to leave the OR-state, both standard transitions (e.g. “`t4`”) as well as inter-level transitions (e.g. “`t5`”) can be used. The standard outgoing transitions of a hierarchical state signify that the outer state can be reached from every substate within the hierarchical

state. In the example, “State D” is reachable from “State A”, “State B” and “State C” through transition “t4”. In contrast, the inter-level transitions to an outer state can only be reached from the originating substate within the hierarchical state. So, only if “State B” is active, the hierarchical state can be left through “t5” to “State E”.

The second type of hierarchical states, the AND-states, enable the specification of concurrent behaviour (Chattopadhyay, 2013). Figure 6.4 displays an example illustration.

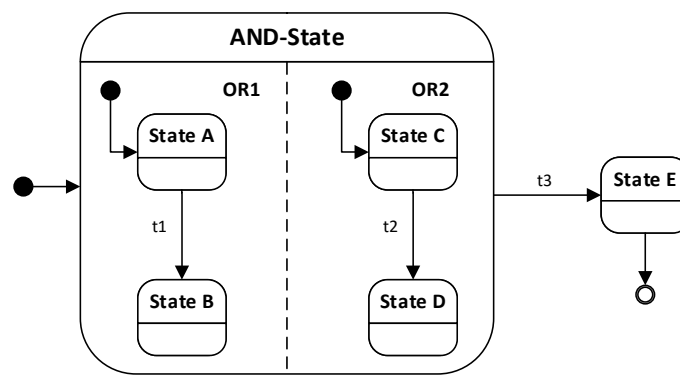


Figure 6.4: Hierarchical AND-state example

The hierarchical “AND-State” encompasses two substates, each of which is a hierarchical OR-state (“OR1” and “OR2”). Thus, the system can be simultaneously in one of the basic states $\{State A, State B\}$ for the first subsystem, and in one of $\{State C, State D\}$ for the second subsystem. The concurrent substates are left as soon as an event occurs that leads to an outer state of the hierarchical AND-state. In this example, the occurrence of an event specified in the transition “t3” leads to the outer basic state “State E”.

Transitions

The most important part of Statecharts besides basic states and hierarchical states are the connectors of states, the so-called transitions. In principle, transitions define the

conditions under which Statecharts can move between states. Figure 6.5 shows the labelling of transitions.

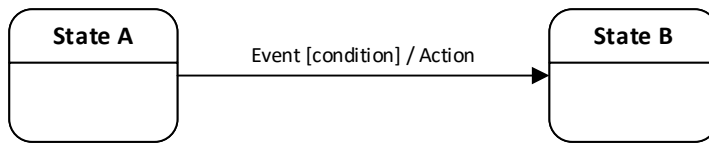


Figure 6.5: Labelling of transitions

The two states related by the transition are called source (“State A”) and destination (“State B”) states. The *Event* indicates the trigger that forces the transition to be activated. The *condition*, also known as *Guard*, is a boolean expression which decides whether the state transition actually occurs. Finally, the *Action* is executed if and when the transition is taken. A special form of transition is the so-called “self transition”. It implies that source and destination state of a transition is identical (Harel, 1996).

Timers

The integration of time within behaviour modelling is very relevant. In Statecharts, time constraints are expressed by using implicit timers and timeouts. The implicit timer generates the timeout event after a specified number of time units has elapsed. Timers are associated with states and transitions through events (Chattopadhyay, 2013). The corresponding Statecharts notation to define a timeout is illustrated in Figure 6.6.

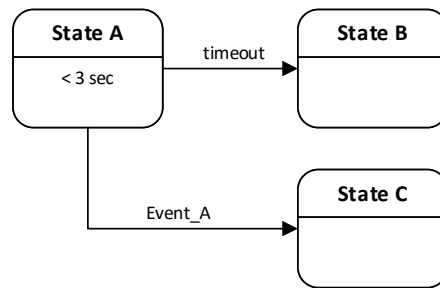


Figure 6.6: Specification of timeouts

The shown example states that if an event “Event_A” does not occur within the next three seconds, a timeout will take place and “State B” will be reached. This standard description is vague as there is no information given about the origin of the timer. Regarding the final notation, some enhancements will be done and presented in section 6.2.4.

Formal description (SCXML)

As mentioned in the previous section, SCXML can be applied to describe Statecharts in a formal structure. It is a “general-purpose event-based state machine language that combines concepts from Call Control eXtensible Markup Language (CCXML) and Harel State Tables.” (W3C, 2015) and its main goal is to “combine Harel semantics with an XML syntax” (W3C, 2015). In September 2015, SCXML became a W3C recommendation (W3C, 2015). All introduced features within this section are supported by SCXML. In the following, an example Statechart will be demonstrated in order to show how the components of a Statechart are described with SCXML language (see Figure 6.7).

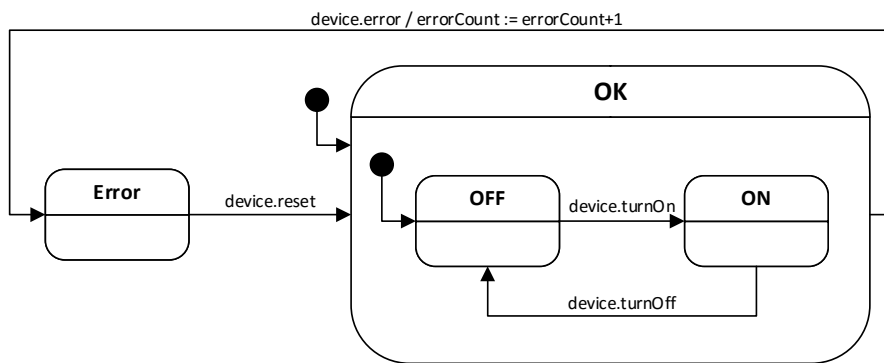


Figure 6.7: Light Switch Statechart example

The shown example contains a hierarchical OR-state (“OK”) which represents the possible states a light switch can have when it works properly (“OFF” and “ON”). If an error occurs, the light switch will be moved into the “Error” mode (“device.error”) and the number of error occurrences are counted (“errorCount”). After a reset (“device.reset”), the light switch should work properly again. The corresponding SCXML description for this example Statechart is illustrated in the following Figure 6.8.

```

<?xml version="1.0" encoding="UTF-8"?>
<scxml
  xmlns="http://www.w3.org/2005/07/scxml" version="1.0" name="LightSwitch"
  datamodel="ecmascript" initial="OK">
  <datamodel>
    <data id="errorCount" expr="0"/>
  </datamodel>
  <state id="OK">
    <initial>
      <transition target="OFF"/>
    </initial>
    <transition event="device.error" target="Error">
      <assign location="errorCount" expr="errorCount + 1"/>
    </transition>
    <state id="OFF">
      <transition event="device.turnOn" target="ON"/>
    </state>
    <state id="ON">
      <transition event="device.turnOff" target="OFF"/>
    </state>
  </state>
  <state id="Error">
    <transition event="device.reset" target="OK"/>
  </state>
</scxml>
  
```

Figure 6.8: SCXML representation of Light Switch Statechart

The main element in Figure 6.8 is the root element `<scxml>` which encompasses all elements of the description. Within the `<datamodel>` element, possible used variables within the Statechart descriptions are initialised (here: “errorCount”). The other elements are either `<state>` or `<transition>` elements. An important aspect is that every outgoing transition from a source state is represented as a state’s child within the XML-based structure. The destination state of the transition is then determined by the *target* attribute of the `<transition>` element. A hierarchical state, both OR-state and AND-state, is represented as a parent-child relationship in SCXML. The `<initial>` element refers to the initial state and its default transition. Figure 6.7 also determines the starting point of the overall Statechart example, the “OK” state. In the SCXML representation, this is set through the attribute *initial* of the `<scxml>` element.

To sum up, the Statecharts notation is a powerful modelling notation in order to specify behaviour and also includes features such as timer integration, concurrency and an underlying formal description. In the following section, it has to be specified how the Statecharts notation can be applied to describe the behaviour of value-added telecommunication services.

6.1.3 Principles of Modelling Service Behaviour with Statecharts

As described in (ETSI ES 202 951, 2011), a behavioural description or rather a formal model of a SUT has to be specified by means of the modelling notation. The primary use of such a formal model is to automatically create abstract specifications of the tests. Test cases can then directly be derived from the formal model by a specific test derivation algorithm.

A very relevant aspect regarding the definition of the formal model is the viewpoint of modelling the behaviour. According to (Malik *et al.*, 2010), this viewpoint can be either internal or external with regard to the interfaces of the SUT. In the case of internal modelling, the formal model is a kind of system model. In general, system models are in a passive role and describe how the SUT responds to given stimulus. They include the partial or the complete behaviour of the SUT. In contrast to system models, there are also test models which define behavioural aspects of the SUT from an external point of view. (Malik *et al.*, 2010) state that a test model determines what kinds of events the SUT should accept at a certain moment and which not. In summary, test models provide stimuli and examine the reactions of the SUT whereas system models expect the stimuli and perform reactions.

This research work suggests a different point of view regarding the modelling of behaviour as the formal model includes both system-specific and test-specific artefacts. This novel concept was derived from the transaction user (TU) which is the fourth and topmost layer of the SIP structure (see section 2.2.2). In the context of the SIP protocol specified in (IETF RFC 3261, 2002), the TU contains both UAC and UAS core. According to (IETF RFC 3261, 2002), a “core designates the functions specific to a particular type of SIP entity”. So, the TU is either able to send requests and receive responses through UAC or receive requests and send responses through UAS. In the context of this research work, the TU is part of the SUT and it is enhanced by further client-based and server-based cores. Although the TU concept has been taken from the standard of the SIP protocol, also cores of other protocols that are dedicated to the OSI application layer can be applied. Having access to a set of client-based and server-based cores, the TU can act as a mediator between available client and server cores. Although

the TU does not have any information about the internal implementation of a value-added service, it can control the service logic through the mediator role. It is notified as soon as a server core received a request message or a client core received a response message. The TU can also initiate request messages through the accessible client cores or response messages through the accessible server cores. A generalised example of the TU acting as mediator between a server core of a not specified “Protocol A” and a client core of a not specified “Protocol B” is illustrated in the following Figure 6.9.

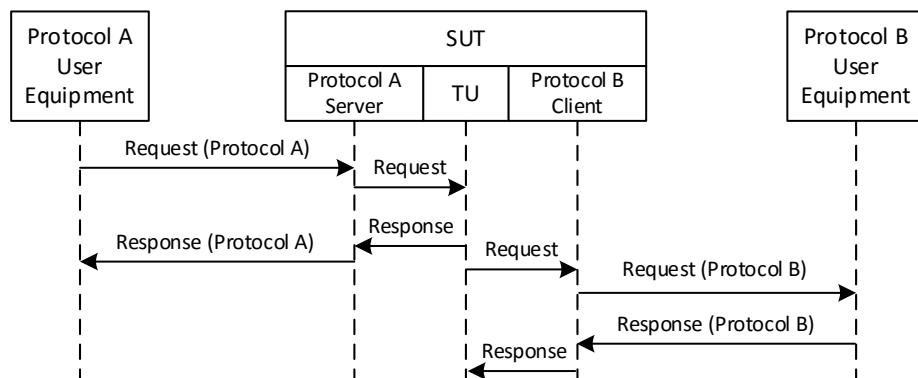


Figure 6.9: Transaction user as mediator between client and server cores

The shown scenario starts with a “Request” message that is received by the server core of “Protocol A”. The TU is informed about the receipt of the request and subsequently initiates the sending of a response message to the “Protocol A User Equipment” through the server core. Afterwards, the TU initiates a request message through the client core of “Protocol B” in order to send it to the “Protocol B User Equipment”. At the end of the scenario, the client core of “Protocol B” informs the TU about the receipt of a response message. Although the illustration is theoretical and based on generic protocols, real protocols can be used with this concept. To realise this, the displayed generic cores just have to be substituted with existing cores. If a “HTTP Server” core substitutes “Protocol A Server” and a “SIP UAC” core substitutes “Protocol B Client”, a real inter-protocol

communication can be described. A well-known value-added service representative of this compilation of cores (HTTP Server and SIP UAC) can be a “Click-to-Instant Message” service. By actuating a button on a web site, a SIP MESSAGE is sent to a specific SIP User Agent.

Now, the relevance of the TU concept for the Statecharts notation has to be examined. To clarify this aspect, the following Figure 6.10 is shown which describes the behaviour specified in Figure 6.9 by means of a Statechart.

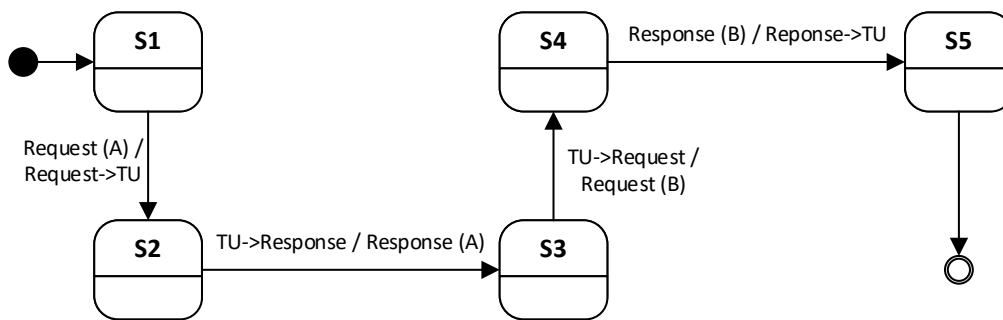


Figure 6.10: Statechart example with explicit TU involvement

The example Statechart includes an initial state, an end state as well as five basic states in between. The information regarding the behaviour is included in the transitions between the prevailing states, either through their specified *events* or *actions*. The sum of events and actions (eight) matches the number of messages (or message informing with regard to the TU) being exchanged between the different parties in Figure 6.9. This leads to the fact that events as well as actions in this novel Statecharts notation are represented by protocol messages (both requests and responses). The focus of interest regarding the notation are the participating cores and the transactions they manage. An event within the Statecharts notation means that a certain core, which is part of the SUT, receives a message. If it is a server-based core, the received message from the external equipment

is always a request type. Otherwise, if it is a client-based core, the externally received message is always a response type. So, an event in the Statecharts notations always refers to an input the SUT has to process. In contrast, the actions defined in the Statecharts notation refer to the reactions of the SUT through the corresponding cores. If the action within a transition is a request type it is always handled by a client-based core whereas response types are handled by server-based cores. The view on actions and events that involve the TU as initiator or receiver of messages differs from the externally specified messages. However, the TU does not really transmit real messages such as requests or responses to its cores. It just triggers the cores to initiate messages or to react on incoming messages by sending further messages. In fact, the messages with TU involvement and the corresponding messages that are handled by the cores contain redundancies. Therefore, the Statecharts notation can be simplified by erasing all the events and actions the TU is involved in. This does not mean that the concept of the TU is also erased, the meaning is only described implicitly through the cores. An advantage of the simplified illustration is the saving of states in the Statecharts models. The following Figure 6.11 shows the simplified Statechart example.

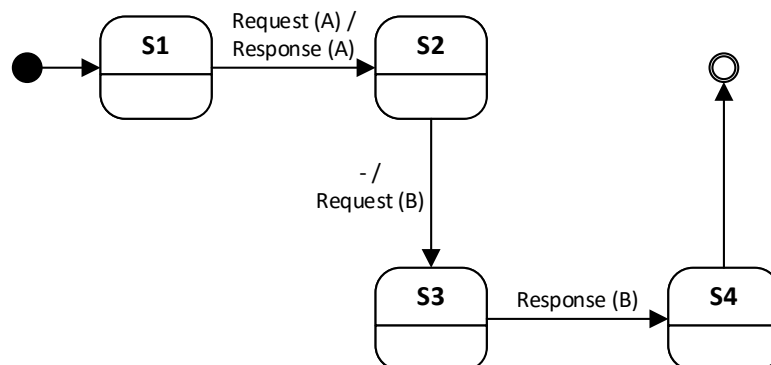


Figure 6.11: Simplified Statechart example without explicit TU involvement

Figure 6.11 shows that not every transition requires both events and actions to be determined. Now, only the messages between the cores and the external equipments are specified. The messages within the SUT between the TU and the cores are erased.

At the beginning of this section, two different types of models have been discussed, system models and test models. In fact, the applied Statecharts notation includes both system-specific as well as test-specific aspects. The system-specific aspect relates to the way a Statechart model is designed. Specified events on transitions can directly be mapped to events the SUT (or system) receives and specified actions can directly be mapped to the reactions the SUT performs. So, a Statechart model directly describes the behaviour on the part of the SUT. The test-specific aspect mainly refers to the definition of the test data. As mentioned before, events on transitions are events the SUT receives. These events, which can be either SIP requests and responses or HTTP requests or messages of any other kind of application layer protocol, have to be set with proper data so that they can be processed by the SUT. The same can be applied to the actions, where the SUT actually sends messages. Although the SUT sets the values of the actions, they have to be verified by the test. So, the definition of test data regardless of whether it was received by the SUT or sent by the SUT has to be specified in the model (see section 6.3). This is a typical test-specific aspect.

Besides the meaning of events and actions in the presented Statecharts notation, of course all other components of standard Statecharts (see section 6.1.2) are used (such as conditions on transitions, hierarchical AND- and OR-states, timers and timeouts and variables). There will be examples where these components are used in the upcoming

sections. The next section deals with the reusable test modules and how they can be designed within the Test Modules Environment.

6.2 Reusable Test Modules

This section deals with one major feature the proposed TCF provides, the reusable test modules. First, the following section introduces the concept and architecture of the Test Modules Environment (TME), a significant part of the TCF.

6.2.1 Test Modules Environment Architecture

It is the task of the test developer to create new reusable test modules for the proposed TCF as soon as the potential functionality of value-added services is extended, possibly through enhancements within the service provider infrastructure. The proposed TCF (see Figure 4.5) provides a special environment for the design and definition of new reusable test modules, the TME. The following Figure 6.12 illustrates a multi-layered software architecture of the TME.

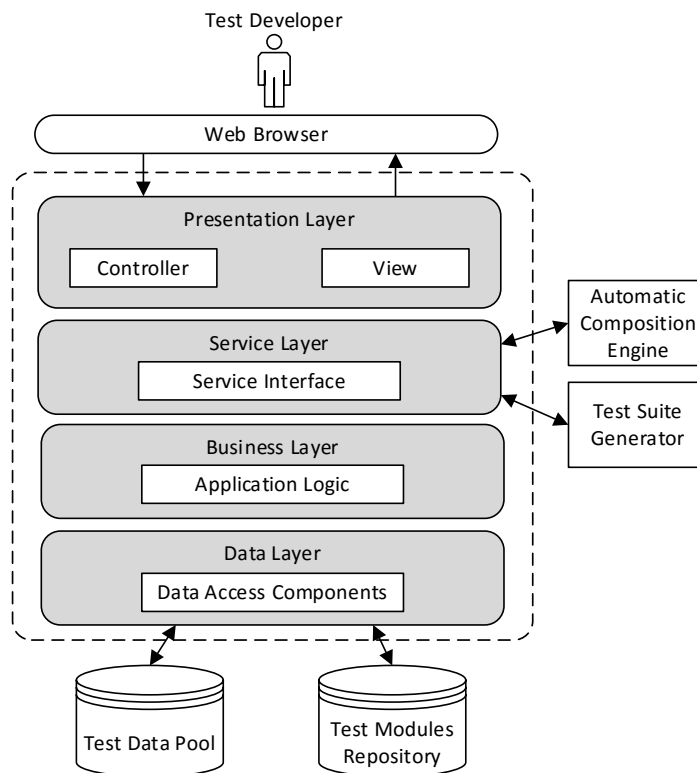


Figure 6.12: Test Modules Environment architecture

The lowest layer of the architecture, the *Data Layer*, provides access to the databases *Test Modules Repository* (TMR) and *Test Data Pool* (TDP). In the TMR, each defined reusable test module is stored by two XML-based documents. The first XML document contains the classification template for the reusable test module. Section 6.2.3 describes the structure of the classification template in detail. The second XML document contains the formalisation, the SCXML document (see section 6.2.4). The other database, the TDP, includes the potential data structures for all supported application layer protocols. In addition, this database contains all parameterised variables that have been instantiated during the behaviour models generation (see section 6.3). The Data Layer itself provides so-called *Data Access Components* that provide functionality for accessing the stored data.

The *Business Layer* of the TME architecture contains the *Application Logic*. Generally speaking, the main target of this component is to handle the data objects it receives and to modify them. Therefore, it has to move and process data between the *Data Layer* and the upper layers.

The *Service Layer* is integrated within the TME architecture, because the TMR and the Test Data Pool have to be accessible by other applications, such as the Automatic Composition Engine (ACE) and the Test Suite Generator (TSG). Through the *Service Interfaces*, the ACE can select, read and write from and to both databases. The TSG just requires access to the Test Data Pool in order to read the parameterised variables that have to be transformed into TTCN-3 templates.

The *Presentation Layer* provides a web-based graphical user interface (GUI) that can be accessed through a web browser by the test developer. The *Controller* and the *View* are typical elements of the well-known data/view/controller pattern for web-based applications. The website enables the test developer to create new reusable test modules and to add new abstract data types and variables. First, the test developer has to define the metadata for the specific reusable test module through the classification template. Then, he models the corresponding behavioural description by means of the Statecharts notation and saves the new reusable test modules to the TMR.

Before the steps for the definition of a reusable test module will be described in detail, the next section deals with the aspect of reusability and how it can especially be identified with respect to value-added telecommunication services.

6.2.2 Identification of Reusability

In the field of computer science and software engineering, the term reusability often refers to the “use of existing assets within the software product development process” (Lombard Hill Group, 2015). Assets are, for instance, software components, test suites, designs and documentation. In the case of this research, assets represent the description of potential recurring behaviour. The term “behaviour” in this context stands for a typical black box approach as it is described in section 3.1.2. The behaviour describes how a system (or value-added service) behaves (output) if it is stimulated by a specified input. No internal aspects regarding the implementation of the underlying system are known. Focussing on value-added telecommunication services, the behaviour can be described through potential protocol (such as SIP or HTTP) messages that are exchanged between the service (SUT) and the service consumers. If the potential behaviour of a consumed service can be categorised and classified, reusability can be derived. In fact, the reusability aspect regarding value-added services depends very much on the network element that provides services, the SIP Application Server (AS) (see section 2.2.4).

Considering SIP as an example, the SIP AS contains SIP-based components such as a SIP Proxy, a Redirect Server, a SIP User Agent and a B2BUA. The functionality of the basic components can be used by a service in order to provide an added value to consumers. So, SIP protocol messages (requests and responses) are the key inputs and outputs for a value-added service that is deployed on a SIP AS. Of course, the service can act in different roles, either as server or client. The IETF standard of the SIP protocol (IETF RFC 3261, 2002) specifies potential behaviour regarding SIP transactions by means of formal descriptions based on finite state machines. Four basic types of formal descriptions

exist, the “INVITE client transaction”, “non-INVITE client transaction”, “INVITE server transaction” as well as the “non-INVITE server transaction”. They distinguish between INVITE requests and all other possible SIP requests (such as “MESSAGE” or “BYE”), once focused on the server-side (UAS) and once on the client-side (UAC). Besides the basic protocol message flows (e.g. “MESSAGE → 200 OK” or “INVITE → 200 OK → ACK”), the formal descriptions also need to consider specific non-conventional message flows, for example server errors through “500” responses. The formal descriptions can be reused in this approach as their combination enables the modelling of behaviour for any kind of SIP communication.

It is important to mention that this research deals with SIP as an example protocol to demonstrate the principles of modelling recurring behaviour. In fact, a value-added service can provide far more functionality besides SIP communication. For example, this aspect relates to the data interface of the SIP AS. Through its data interface, the SIP AS can include other servers such as web servers, email servers, directory servers and media servers (Trick and Weber, 2015). The integration of these servers leads to a broader range of functionality of potential value-added services which again leads to a broader range of potential behaviour that needs to be specified. As mentioned before in section 6.1.3, the behaviour of any other OSI application layer protocol can be specified because of the integration of the TU concept into the applied Statecharts notation. Theoretically, only new cores for the protocols (both client-based and server-based) need to be included.

The following Table 6.2 illustrates a list of potential server types and the relevant protocols to use the functionality the servers provide.

Table 6.2: Potential server types and their corresponding application layer protocols

Server type	Relevant protocols
Web server	<ul style="list-style-type: none"> Hypertext Transfer Protocol (HTTP) (IETF RFC 2616, 1999) to transfer files on the WWW.
Email server	<ul style="list-style-type: none"> Simple Mail Transfer Protocol (SMTP) (IETF RFC 5321, 2008) to store and forward emails. Post Office Protocol (POP) (IETF RFC 1939, 1996) to download emails.
Directory server	<ul style="list-style-type: none"> Lightweight Directory Access Protocol (LDAP) (IETF RFC 4511, 2006) to locate resources such as files and devices in a network.
Media server	<ul style="list-style-type: none"> Real-Time Transport Protocol (IETF RFC 3550, 2003) to deliver audio and video over IP networks.

It should be mentioned that principally, a media server also uses the SIP protocol to be controlled by the SIP AS. However, because of the black box focus of the functional testing approach, this communication is not relevant. It does not directly involve the service consumers and therefore also not the test environment.

6.2.3 Classification of Reusable Test Modules

In this research work, a reusable test module is a formal description of recurring behaviour based on the applied Statecharts notation (see sections 6.1.2 and 6.1.3). The behaviour refers to a given application layer protocol and to a specific core, either server or client-based.

The ACE as part of the proposed TCF automatically selects appropriate test modules from a database of predefined reusable test modules (TMR) based on the parsing of a given STD. Additionally, the ACE realises the composition or rather combination of the selected test modules and adds data to them. These aspects make it necessary to add some further information, so-called metadata, to each reusable test module that is stored in the

TMR. This is particularly important as the reusable test modules are part of a completely automated process.

The following Figure 6.13 contains a classification template for reusable test modules that is described by means of an XSD document and is illustrated graphically.

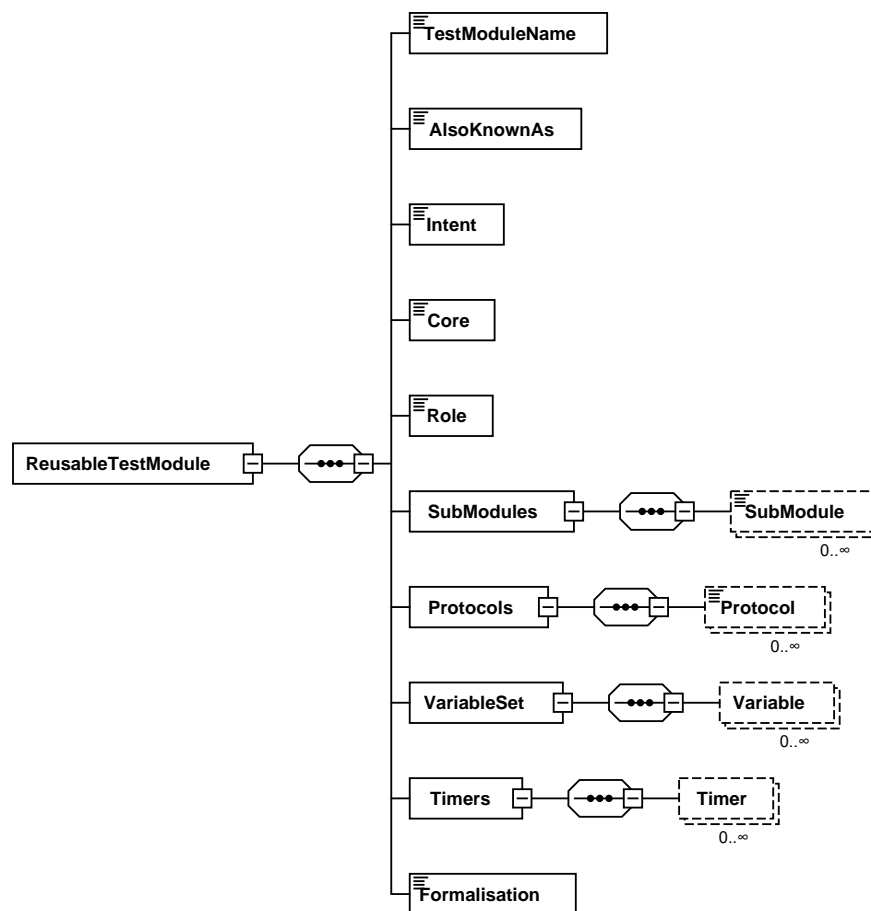


Figure 6.13: Classification template for reusable test modules

The classification template comprises the list of properties that have to be specified whenever a new reusable test module is defined. One of the most important properties is the *TestModuleName*, because it is the identifier of the reusable test module. While parsing an STD instance and especially the determined *CI*s mentioned within the

Requirements, the ACE will select the reusable test modules based on the identifiers of the *CI*s. As discussed before, the potential behaviour of a *CI* is described by the reusable test modules. The next property, *AlsoKnownAs*, contains possible aliases of the reusable test module. A prose description of the major test objective of the reusable test module is part of the *Intent* property. The *Core* property specifies to which core (either client or server core) the reusable test module refers to. Afterwards, the involved *Role* is specified. The *Role* as part of the STD is identical to the *Role* specified in the classification template of the reusable test module. As depicted in section 5.2.2, a *Role* (e.g. “SIP phone”, “Web browser”) is an external hardware that interacts in different ways with the SUT. These different ways are specified through all reusable test modules that determine the same *Role* in the classification template. As these reusable test modules relate to the same *Role*, they are also called “related test modules”. Reusable test modules can be composed of other reusable test modules that exist in the TMR. They can then be determined in the *SubModules* property. The next property *Protocols* contains all application layer protocols that are used in the behaviour described in the reusable test modules. The *VariableSet* includes all variables that can be set within a reusable test module. Although the attributes are not shown in Figure 6.13, each *Variable* contains a *name* and a *type* attribute. The *name* attribute refers to the name that is part of the Statechart description of the reusable test module. The *type* specifies the underlying abstract data type of the *Variable* which should be also present in the Test Data Pool (see section 6.3). Just as with the *Variables*, every specified *Timer* within a reusable test module has to be included in the Statechart description through its attribute *timerID* (not shown in Figure 6.13). There is another attribute defined, the *value* attribute, which determines the default time interval. Finally, the classification template contains the property *Formalisation*. It includes the

link to the behavioural Statechart description which is stored as SCXML file within the TMR.

To sum up, a classification template holds all the relevant metadata of a reusable test module. It is the task of the test developer to carry out the definition of the classification template as well as the modelling of the behaviour of the reusable test module.

6.2.4 Modelling of Reusable Test Modules

As mentioned in the sections 6.1.2 and 6.1.3, reusable test modules are modelled by means of the applied Statecharts notation. Of course, the test modules have to be defined in a generalised way so that they can be specified in detail through the parameterisations that are included within the STD.

When a test developer starts modelling the behavioural description for a new reusable test module, he has to observe the following rules:

1. The behavioural description of a reusable test module is defined within one hierarchical OR-state.
2. The hierarchical OR-state has to include one initial state with a default transition (transition without events and actions or conditions) to the first relevant state (“start” state) of the behavioural description.
3. The transition (so called “initial transition”) from the “start” state to the second state contains the input parameter (either event or action) of the whole reusable test module.

4. If the reusable test module refers to a server core (“SUT receives initial request”), the input parameter within the initial transition contains an event and optionally a further action. In contrast, a client core (“SUT sends initial request”) must only contain an action.
5. Every variable within the behavioural description has to be specified in the classification template. This is necessary, because the classification template also contains the abstract data type the variable is based on.
6. Every defined timer within the behavioural description has to be included in the classification template. There, the default timer value is set.
7. For every timer started within a state of the behavioural description, there has to be a corresponding “timeout” event.
8. Every transition apart from the default transition within the behavioural description has to include either event or action, or both.
9. The behavioural description does not contain a specific end state, but a final state that is always called “Terminated”.

The modelling process will be demonstrated in the following using a server core (“SIP UAS non-INVITE”) and a client core (“SIP UAC INVITE”) reusable test module.

SIP UAS non-INVITE reusable test module

First, the server core-based “SIP UAS non-INVITE” reusable test module is introduced. It describes the potential behaviour of a SUT (or rather service) that receives a SIP request from a participating external entity (such as a SIP phone). The request type is described as a generic type that can be further specified through paramterisation (by the STD). When developing the reusable test module, the test developer first has to define the

classification template. The example classification template for the SIP UAS non-INVITE module is shown in the following Figure 6.14.

```

<ReusableTestModule>
  <TestModuleName>SIP UAS non-INVITE</TestModuleName>
  <AlsoKnownAs>non-INVITE server transaction</AlsoKnownAs>
  <Intent>This test module specifies the potential behaviour of
    a SIP UAS core that receives a request of any SIP request
    type different from INVITE.
  </Intent>
  <Core>server</Core>
  <Role>SIP phone</Role>
  <SubModules />
  <Protocols>
    <Protocol>SIP</Protocol>
  </Protocols>
  <VariableSet>
    <Variable name="r_Request" type="SIP_Request" />
    <Variable name="s_Response1xx" type="SIP_Response" />
    <Variable name="s_ResponseBlxx" type="SIP_Response" />
    <Variable name="s_Response2xx_6xx" type="SIP_Response" />
  </VariableSet>
  <Timers>
    <Timer timerID="globalTimer" value="30000" />
    <Timer timerID="timerJ" value="0" />
  </Timers>
  <Formalisation>SIP_UAS_non-INVITE.scxml</Formalisation>
</ReusableTestModule>

```

Figure 6.14: Example classification template for SIP UAS non-INVITE reusable test module

Besides general information such as the naming, the *Core* (“server”), the participating *Role* (“SIP phone”), the application layer protocol (“SIP”) and the used variables including their names and types are defined. The relevance of the variables will be discussed in the upcoming section 6.3. The classification template also includes two timers, a “globalTimer” and a “timerJ”. The global timer is started as soon as the behaviour within the reusable test module is started, in other words, if the request is received by the SUT. The timeout of the global timer is not explicitly defined. It can take place within any state of the behavioural description. As a consequence of a timeout of the global timer, a derived test case will definitely fail. The “timerJ” refers to a protocol-specific transaction timer and is initialised with the value “0”. The setting of the value depends on the underlying transport protocol. According to (IETF RFC 3261, 2002), the timer should be set to a value of $T1*64$ (where $T1$ stands for a value of 500 milliseconds),

if an unreliable protocol such as UDP is used. If a reliable protocol such as TCP is used, the timer can be set to “0”. After the definition of the classification template, the test developer can model the behavioural description of the SIP UAS non-INVITE reusable test module. The result is displayed in Figure 6.15.

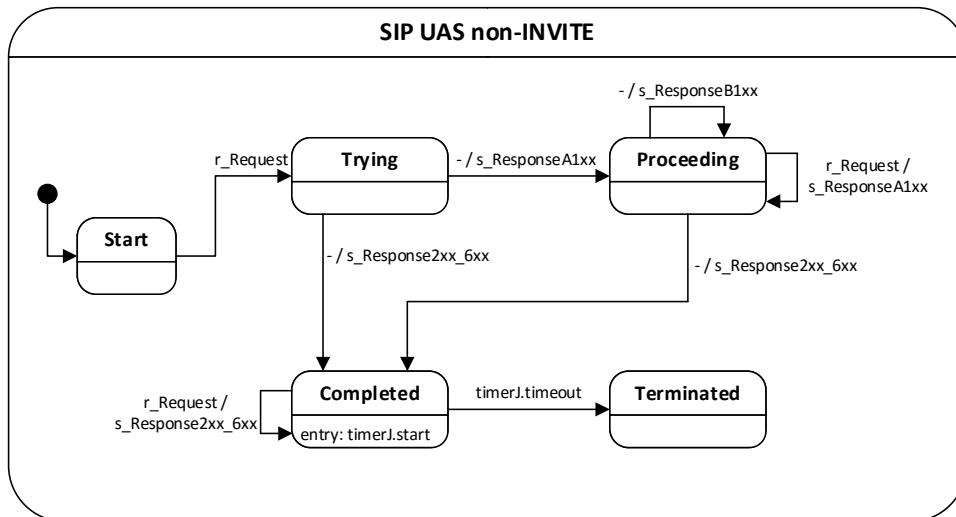


Figure 6.15: Behavioural description of SIP UAS non-INVITE reusable test module

The illustrated behavioural Statechart description is derived from (IETF RFC 3261, 2002), the protocol specification of SIP. It includes the initial transition as entry point into the reusable test module. There, the “r_Request” event is expected by the SUT. The “r” prefix is a help for the test developer to orientate himself within the reusable test module. It is an abbreviation for “received” and refers to the SUT that actually “receives” a message. As soon as the event “r_Request” takes place, the state “Trying” is reached. From this state, there are two valid optional paths that can be taken, either to the “Proceeding” state with the “s_ResponseA1xx” action or to the “Completed” state with the “s_Response2xx_6xx”. Both actions also have a prefix within the names, the “s” (abbreviation for “send”), which states that the SUT actually “sends” the message back to the initiator of the “r_Request”. The alternative paths that are determined here describe

the potential behaviour of the SUT (the value-added service). It could happen that based on the “r_Request” (e.g. a SIP MESSAGE), the SUT directly acknowledges with a “200 OK” response by performing the action “s_Response2xx_6xx”. Here, the range of status codes from 200 until 699 can be selected. Alternatively, the SUT first sends a provisional response “s_ResponseA1xx” (status codes from 100 until 199) and afterwards sends a “s_Response2xx_6xx”, which is also the action determined in the transition that has “Proceeding” as source and “Completed” as destination state. As soon as the “Completed” state is reached, the “timerJ” is started and its timeout is expected (either immediately when TCP is used or after $T1*64$ milliseconds when UDP is used). The reaching of the state “Terminated” after the timeout denotes the end of the transaction. Besides the straight paths within the behaviour description, there are also three self-transitions defined that describe specific recurring behaviour that could take place.

Based on this specified behaviour, test cases can be later on derived by means of a specific test case derivation algorithm (see section 7.1.2). Of course, this algorithm will be performed on the resulting behaviour models, which are compositions of several reusable test modules.

The formalisation of the reusable test module is based on SCXML and is illustrated in the following:


```
<scxml
  xmlns="http://www.w3.org/2005/07/scxml" version="1.0" name="SIP UAS non-INVITE"
  datamodel="ecmascript">

  <datamodel>
    <data id="r_Request"/>
    <data id="s_Response1xx"/>
    <data id="s_ResponseBlxx"/>
    <data id="s_Response2xx_6xx"/>
  </datamodel>

  <state id="SIP UAS non-INVITE">
    <initial>
      <transition target="Start"/>
    </initial>
    <state id="Start">
      <transition event="r_Request" target="Trying" />
    </state>
    <state id="Trying">
      <transition target="Proceeding">
        <send event="s_Response1xx" />
      </transition>
      <transition target="Completed">
        <send event="s_Response2xx_6xx" />
      </transition>
    </state>
    <state id="Proceeding">
      <transition target="Proceeding">
        <send event="s_ResponseBlxx" />
      </transition>
      <transition event="r_Request" target="Proceeding">
        <send event="s_Response1xx" />
      </transition>
      <transition target="Completed">
        <send event="s_Response2xx_6xx" />
      </transition>
    </state>
    <state id="Completed">
      <onentry>
        <send event="timerJ" delay="0"/>
      </onentry>
      <transition event="r_Request" target="Completed">
        <send event="s_Response2xx_6xx" />
      </transition>
      <transition event="timerJ.timeout" target="Terminated"/>
    </state>
    <state id="Terminated"/>
  </state>
</scxml>
```

Figure 6.16: SCXML document of SIP UAS non-INVITE reusable test module

The `<datamodel>` element in Figure 6.16 states the possible variables that are used in the behavioural description. The hierarchical OR-state “SIP UAS non-INVITE” comprises all sub states. As specified in Figure 6.15, each state has transitions to other states or self-transitions. An important aspect regarding the formal description is how events and actions are determined. An event is defined as direct attribute *event* within a `<transition>` element whereas an action is specified within the attribute *event* of the element `<send>`

which is a child element of `<transition>`. The syntax looks ambiguous because of the similar name *event* for both events and actions. However, it is explicitly expressed through the `<send>` element that the message is “sent”. A timer within the SCXML description can also be specified. Within the `<onentry>` element of the “Completed” state, the timer event of “timerJ” is specified and the value is set as attributes of the `<send>` element. Then, a `<transition>` element is defined within the “Completed” state which specifies the occurrence of the timer event. This is synonymous with a timeout of the timer.

SIP UAC INVITE reusable test module

The SIP UAC INVITE reusable test module differs from the SIP UAS non-INVITE in two major aspects. First, the SIP UAC INVITE is client core-based, so the SUT is the initiator or sender of the initial request. Second, it deals with a special SIP message, the INVITE request, which is generally sent to set-up a VoIP call. As it includes the Three-Way-Handshake, the behaviour definitely differs from the non-INVITE behaviour. As demonstrated before, initially the classification template for the SIP UAC INVITE reusable test module has to be defined. It is shown in Figure 6.17.

```

<ReusableTestModule>
  <TestModuleName>SIP UAC INVITE</TestModuleName>
  <AlsoKnownAs>INVITE client transaction</AlsoKnownAs>
  <Intent>This test module specifies the potential behaviour of
    a SIP UAC core that initiates a SIP INVITE to initiate a call.
  </Intent>
  <Core>client</Core>
  <Role>SIP phone</Role>
  <SubModules />
  <Protocols>
    <Protocol>SIP</Protocol>
  </Protocols>
  <VariableSet>
    <Variable name="s_Invite" type="SIP_Request" />
    <Variable name="r_Response1xx" type="SIP_Response" />
    <Variable name="r_Response1xx" type="SIP_Response" />
    <Variable name="r_Response2xx" type="SIP_Response" />
    <Variable name="r_Response3xx_6xx" type="SIP_Response" />
    <Variable name="s_Ack" type="SIP_Request" />
  </VariableSet>
  <Timers>
    <Timer timerID="globalTimer" value="30000" />
    <Timer timerID="timerA" value="500" />
    <Timer timerID="timerD" value="0" />
  </Timers>
  <Formalisation>SIP_UAC_INVITE.scxml</Formalisation>
</ReusableTestModule>

```

Figure 6.17: Example classification template for SIP UAC INVITE reusable test module

There is no significant difference to the classification template of the SIP UAS non-INVITE test module. Of course, a different *Core* is stated (“client”) and a different set of variables. Additionally, the further *Timers* “timerA” with a default value of “500” milliseconds and a “timerD” with a default value of “0” milliseconds. Just as “timerJ” in the “SIP UAS non-INVITE” behavioural description, the value of “timerD” depends on the reliability of the underlying transport protocol. The following Figure 6.18 shows the behavioural description of the SIP UAC INVITE.

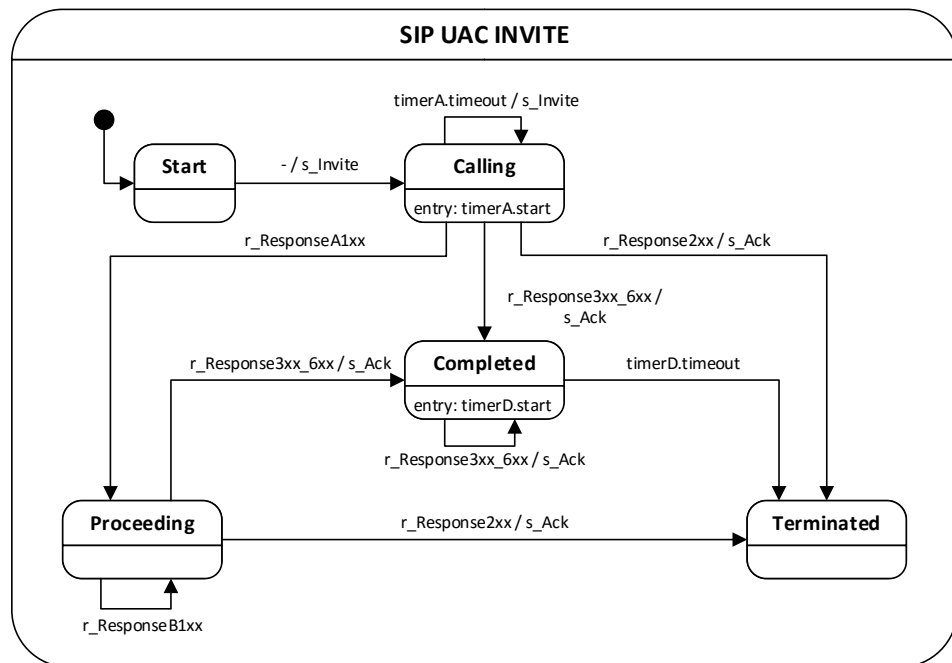


Figure 6.18: Behavioural description of SIP UAC INVITE reusable test module

One major difference to the SIP UAS non-INVITE behavioural description is directly visible regarding the events and actions. In the behavioural description of the server core-based SIP UAS non-INVITE module, every request message was determined as event and every response message as action. Figure 6.18 describing the behavioural description of SIP UAC INVITE illustrates the opposite. Now, every response message is determined as event and every request message as action. This opposite view is up to the different cores. The general specification of the behaviour starts with the SUT sending a “s_Invite” request. As soon as the state “Calling” is reached, “timerA” is started. Now, the participating entity has to respond to the initial INVITE request, for instance, by sending a provisional response “r_ResponseA1xx”. Then, the state “Proceeding” of the reusable test module will be reached. Alternatively, a successful response “r_Response2xx” can be sent by the participating entity, which is directly acknowledged by the SUT sending back an “s_Ack” request (state “Terminated” is reached). The Three-Way-Handshake

(see section 2.2.1) is then successfully established. Finally, the participating entity can also respond to the initial INVITE request with a redirection or failure response “r_Response3xx_6xx” which leads to reaching the “Completed” state after the “s_Ack” request is sent by the SUT. It can also happen that the participating entity does not send a response within 500 milliseconds. Accordingly, a timeout of “timerA” takes place and the “s_Invite” request will be sent once again by the SUT. The further behaviour is quite evident. It should be mentioned that the successful Three-Way-Handshake creates a SIP dialog through which further message processing can be performed.

In the following, the formalisation of the SIP UAC INVITE reusable test module is defined. Figure 6.19 shows the example SCXML document.

```

<scxml
  xmlns="http://www.w3.org/2005/07/scxml" version="1.0" name="SIP UAC INVITE"
  datamodel="ecmascript">

  <datamodel>
    <data id="s_Invite"/>
    <data id="r_Response1xx"/>
    <data id="r_Response1lxx"/>
    <data id="r_Response2xx"/>
    <data id="r_Response3xx_6xx"/>
    <data id="s_Ack"/>
  </datamodel>

  <state id="SIP UAC INVITE">
    <initial>
      <transition target="Start"/>
    </initial>
    <state id="Start">
      <transition target="Calling">
        <send event="s_Invite" />
      </transition>
    </state>
    <state id="Calling">
      <onentry>
        <send event="timerA" delay="500"/>
      </onentry>
      <transition event="timerA.timeout" target="Calling">
        <send event="s_Invite" />
      </transition>
      <transition event="r_Response1xx" target="Proceeding" />
      <transition event="r_Response2xx" target="Terminated">
        <send event="s_Ack" />
      </transition>
      <transition event="r_Response3xx_6xx" target="Completed">
        <send event="s_Ack" />
      </transition>
    </state>
    <state id="Proceeding">
      <transition event="r_Response1lxx" target="Proceeding" />
      <transition event="r_Response2xx" target="Terminated" />
        <send event="s_Ack" />
      </transition>
      <transition event="r_Response3xx_6xx" target="Completed" />
        <send event="s_Ack" />
      </transition>
    </state>
    <state id="Completed">
      <onentry>
        <send event="timerD" delay="0"/>
      </onentry>
      <transition event="timerD.timeout" target="Terminated" />
      <transition event="r_Response3xx_6xx" target="Completed">
        <send event="s_Ack" />
      </transition>
    </state>
    <state id="Terminated" />
  </state>
</scxml>

```

Figure 6.19: SCXML document of SIP UAC INVITE reusable test module

This section described the relevant steps to specify the behavioural part of the reusable test modules. Of course, one important aspect is still missing, the definition of the test data. Additionally, the relationships between test data templates within one reusable test

module have to be specified. This is the final task the test developer has to do before the reusable test modules can be processed within the TCF.

6.3 Test Data Integration

The main objective of the previous section was to show how reusable test modules are designed. Based on a classification template and the behavioural Statecharts-based description, an abstract definition of potential behaviour is introduced. However, as it is abstract, there is no real data defined. In principle, every determined event or action within a behavioural description, irrespective of which underlying protocol is specified, represents an identifier for a real protocol message. Depending on the protocol, one message may contain a lot of content and may also comprise a considerably high amount of headers. In the TCF approach, the content of protocol messages is the test data described below.

According to the U2TP approach (OMG, 2013a), test data is the data that is transmitted between the SUT and the test execution environment. In general, two different groups of test data exist, test data for stimuli and test data for observations. The test data for stimuli relates to data that is sent from the test execution environment to the SUT whereas the test data for observations describes the opposite, consequently the sending of data from the SUT to the test execution environment. The term “observations” states that something is observed. In fact, the test data the SUT sends to the test execution environment is observed. When protocol messages such as SIP requests or responses are received by the test execution environment, its observer component verifies that the incoming message matches the predefined conditions. Otherwise, if the test execution is the sender of a

protocol message (“stimuli”), every mandatory header has to be set with appropriate content. Within the reusable test modules, the names of the variables also indicate whether they are referring to stimuli messages or observing messages. Variables with the prefix “r” describe messages the SUT expects and simultaneously, they are describing test data for stimuli. Contrary to this, variables with the “s” prefix describe messages the SUT sends and they are also test data for observations.

Every variable specified within the reusable test modules are instances of abstract data types. For each request-response application layer protocol described within the reusable test modules, two different abstract data types are defined, one for the request messages and one for the response messages. For other protocols that do not distinguish between requests and responses, such as RTP, there is only one abstract data type defined. The following Figure 6.20 demonstrates an example class structure that includes messages for the protocols SIP, HTTP and RTP.

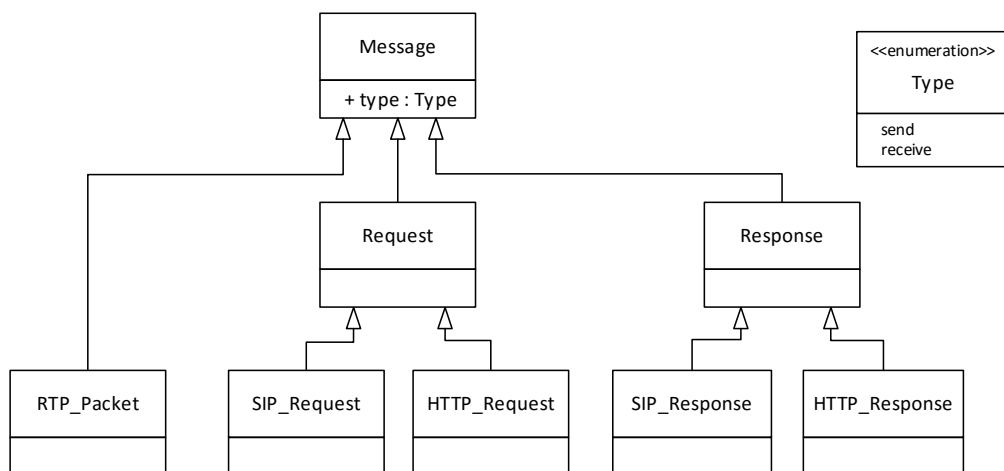


Figure 6.20: Structure of abstract data types for test data

The base class *Message* contains an attribute *Type* specifying whether the underlying message is a “send” (prefix “s”) or a “receive” (prefix “r”) message. Of course, the

displayed protocol messages are just examples. Further protocols could be integrated into the class structure. The specific classes for the displayed protocols, such as “SIP_Request” and “SIP_Response” for the SIP protocol, “HTTP_Request” and “HTTP_Response” for the HTTP protocol, and “RTP_Packet” for RTP, are far more complex than illustrated in Figure 6.20. Of course, the complexity depends on the principle structure of the protocol messages. The core specification of the SIP protocol (IETF RFC 3261, 2002), for instance, utilises actually almost 50 header fields, but there are even more defined within various extensions of the protocol. The following Figure 6.21 illustrates the supported header types within the “SIP_Request” abstract data type.

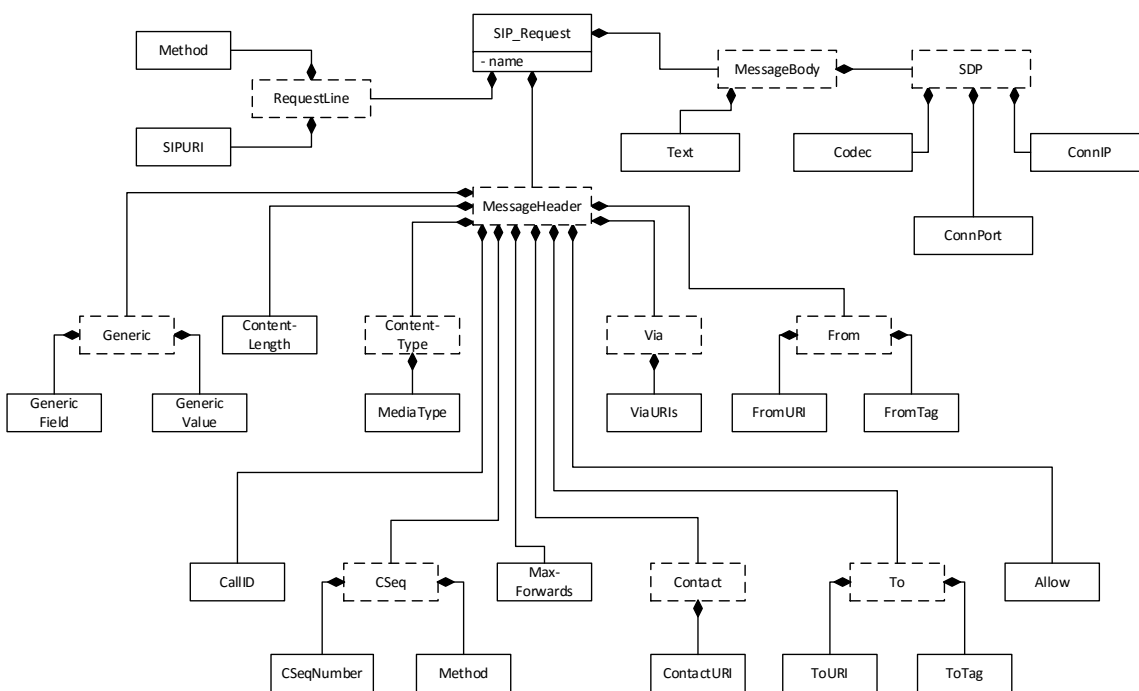


Figure 6.21: Conceptual structure of SIP_Request abstract data type

The shown structure of the abstract data type for “SIP_Request” is based on a XSD structure and illustrates header fields of a SIP request. The elements are either marked with solid lines or broken lines. The difference between these two element categories is that the values of the solid line elements can be modified through STD variables whereas

the broken line elements can not. An example of setting values of a SIP request (SIP MESSAGE) is shown in Table 5.6. It is important that exactly the identifiers of the elements are used, such as *Text*, which represents a text within a SIP MESSAGE. Besides the mandatory headers, it is possible to add further headers that are optional. Here, the *Generic Field* and the *Generic Value* elements can be used. Of course, the test developer has to know the exact syntax of such an optional header. Besides the “SIP_Request” structure, the “SIP_Response” structure has to be defined. In contrast to the “SIP_Request”, the “SIP_Response” does not include a *Text* element. Furthermore, the *RequestLine* as part of the “SIP_Request” is substituted by the *StatusLine* in the “SIP_Response” structure. Besides the mentioned SIP-specific requests and responses, it is of course possible to also define requests and responses of other protocols. As with the SIP messages, it is necessary to define XSD structures for the protocol messages.

The introduced conceptual structure (exemplified for the “SIP_Request”) of request and response messages defines which elements the corresponding message contains. However, until now, there is no real data stored. Therefore, it should be possible to create instances of the specified abstract data types. In fact, every reusable test module that is stored within the TMR contains a set of variables which are further specified in the corresponding classification templates. For the “SIP UAS non-INVITE” reusable test module, the “r_Request” is an instance of the abstract data type “SIP_Request” whereas the other specified variables “s_ResponseA1xx”, “s_ResponseB1xx” and “s_Response2xx_6xx” are instances of the abstract data type “SIP_Response”. When a test developer defines a new reusable test module, he can already predefine certain copies of header fields within the description. The following Figure 6.22 shows, what the test

developer can prepare in the “SIP UAS non-INVITE” reusable test module regarding the test data.

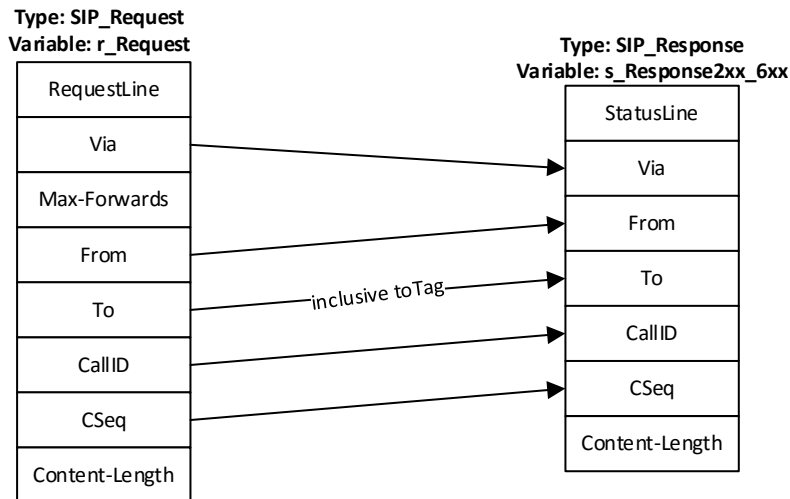


Figure 6.22: Predefined copying of message headers

The mandatory fields of a SIP response message (here: “s_Response2xx_6xx”), the headers *Via*, *From*, *CallID* and *CSeq*, can be directly copied from the originating SIP request (here: “r_Request”). The *To* header field of the originating SIP request does not contain a *toTag*, this needs to be added within the SIP response. The *Content-Length* header usually contains the value “0” as there is no data transmitted in the message body of the response message. The only aspect of the response that is variable is the *StatusLine*. In general, the *StatusLine* is expected to include the “200” for *StatusCode* and “OK” for *ReasonPhrase*. Of course, the predefinition illustrated in Figure 6.22 can be applied to every defined “SIP_Response” in the “SIP UAS non-INVITE” reusable test module. At this point, the principles of stimuli messages and observing messages have to be emphasised again. The “r_Request” is a SIP request message the SUT receives and therefore, it is a stimuli message. Contrary to this, the “s_Response2xx_6xx” is a message that the SUT sends which means that it is an observing message. As a matter of fact, some

data of an observing message cannot be predefined because it is simply unknown before the test execution. Of course, this does not apply to the whole message, but to certain aspects that are generated by the sender (SUT) of the message. Referring back to the observing “s_Response2xx_6xx” message, this aspect can be exemplified. The *StatusCode* as well as the *ReasonPhrase* contain data of a fixed set of possible values so they can also be specified exactly. The value of the *toTag*, however, cannot be foreseen. In this approach, special symbols, so-called *wildcards*, have been included from the TTCN-3 notation that can be used instead of exact values (ETSI ES 201 873-1, 2015):

- “?” is a wildcard for any value.
- “*” is a wildcard for any value or no value at all.

In the case of the *toTag*, the “?” has to be chosen, because it is a mandatory field that has to be set by the SUT. The following Figure 6.23 illustrates the “s_Response2xx_6xx” message in the form of an XML document.

```
<SIP_Response name="s_Response2xx_6xx">
  <StatusLine>
    <StatusCode>200</StatusCode>
    <ReasonPhrase>OK</ReasonPhrase>
  </StatusLine>
  <MessageHeader>
    <Via>
      <ViaURIs>r_Request.ViaURIs</ViaURIs>
    </Via>
    <From>
      <FromURI>r_Request.FromURI</FromURI>
      <FromTag>r_Request.FromTag</FromTag>
    </From>
    <To>
      <ToURI>r_Request.ToURI</ToURI>
      <ToTag>?</ToTag>
    </To>
    <CallID>r_Request.CallID</CallID>
    <CSeq>
      <CSeqNumber>r_Request.CSeqNumber</CSeqNumber>
      <Method>r_Request.Method</Method>
    </CSeq>
    <Content-Length>0</Content-Length>
  </MessageHeader>
  <MessageBody />
</SIP_Response>
```

Figure 6.23: Example XML document of SIP response message “s_Response2xx_6xx”

The simplified example document shows that most data is directly copied from the originating request, the “.” operator syntax is used to copy the values into the response message. The other values are explicitly defined (*Status Code* and *ReasonPhrase*) or a wildcard is used (*toTag*).

Just as the abstract data types for the protocol messages, every variable defined within a reusable test module is stored within a database, the so-called *Test Data Pool*. The abstract data types are stored as XSD structures whereas the variables are stored as XML documents (such as in Figure 6.23). However, there is another group, the variables of instances of reusable test modules that have been parameterised within the STD instance and that are integrated within the generated behaviour models. The following Figure 6.24 illustrates the connection between the three data groups that are stored within the *Test Data Pool*.

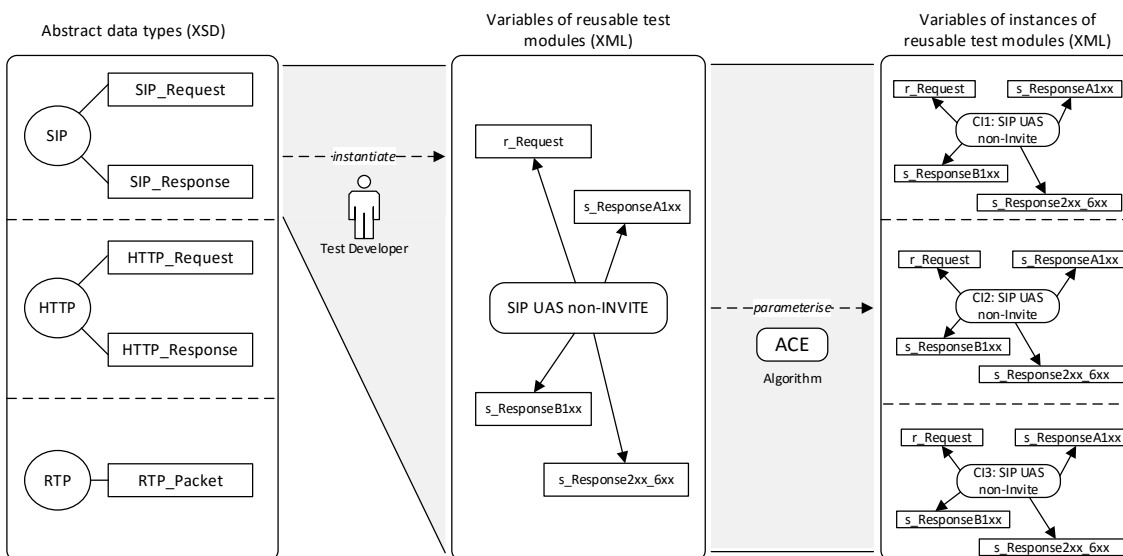


Figure 6.24: Stored data within Test Data Pool

The transition from abstract data types to variables of reusable test modules has been demonstrated by means of examples in this section. The modelling of the reusable test

modules and the definition of the corresponding variables is done by the test developer within the TME. Then, the behavioural description in form of a Statecharts notation is stored in the TMR, whereas the variables are stored in the *Test Data Pool*. As soon as a new STD instance is created, the behavioural models for the described value-added service will be automatically generated by the ACE through a specific algorithm. During the generation, the ACE selects reusable test modules and creates instances of them. Each instance is assigned a set of variables that are parameterised through the *Parameters* field of the STD. Theoretically, it is possible that one instance of a reusable test module contains diverse sets of variables. So, the test coverage at the end can be modified or even improved.

The next section will describe the behaviour models generation through the ACE algorithm.

6.4 Generation of Behaviour Models

The main concern of this section is the Automatic Composition Engine (ACE), a component within the proposed TCF. Its main task is to process a well-defined input and produce a specified output. In this case, the well-defined input are instances of the STD that have been established by test developers for given value-added telecommunication services. The output, in contrast, are so-called behaviour models that describe the potential behaviour of a service based on a formal Statecharts notation. The ACE requires further information to be able to generate the behaviour model. On the one hand, it has to be able to access the predefined reusable test modules that describe recurring behaviour (see section 6.2). So, it can actually reuse the test modules, instantiate them within the

behavioural models, and compose them according to the specifications within the STD. On the other hand, the ACE also requires access to the specified test data that is used within the reusable test modules (see section 6.3) in order to parameterise instantiated test modules. The following Figure 6.25 demonstrates the input and output as well as the relevant processes that take place within the ACE.

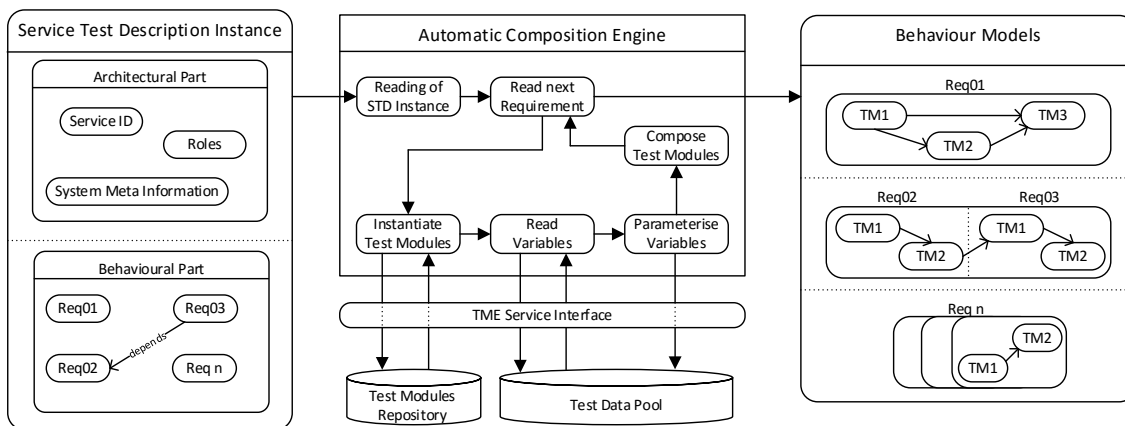


Figure 6.25: Behaviour models generation process with ACE

On the left side of Figure 6.25, an STD instance is shown as input of the ACE. As described before, it contains an *Architectural Perspective* as well as a *Behavioural Perspective*. Both perspectives contain information that are relevant for the ACE. The *ServiceID* within the *Architectural Perspective* determines the name of the value-added service and simultaneously, the name for the whole project. So, the name will be defined within the STD instance and will be within the namings of the behaviour models and within the tests that are generated on the basis of the behaviour models. The *System Meta Information* might contain information that are relevant for the test data parameterisation. A very important parameter is the service URI which can be resolved as soon as the value-added service running on a SIP AS is registered in the location database of a call server. This service URI is very relevant for participating entities (or rather test components)

when they are about to, for instance, send INVITE requests to the SUT. The request line of the INVITE request will contain this service URI. Consequently, the request line value of the SIP request variable “r_Invite” as part of the SIP UAS INVITE reusable test module instance will contain the service URI. Besides the service URI, there are, of course, other relevant parameters, such as the permanent SIP URIs of registered participating entities that are involved in the service consumption. The last parameter, the *Roles*, is not relevant for the ACE process itself, however, it delivers the *Roles* and the *System Meta Information* as well as the *ServiceID* directly to the Test Configuration Unit. This is not illustrated in Figure 6.25 but will be further discussed in section 7.2.

The *Behavioural Perspective* contains all the *Requirements* and of course, within the *Requirements*, dependencies are set through the *Precondition* field. This is exemplified in Figure 6.25 (“Req03” depends on “Req02”), because it has an effect on the resulting behaviour models. A *Requirement* that does not contain another *Requirement* in its *Precondition* field and that is not determined as *Precondition* within any other specified *Requirement* itself, is exactly specified through one behavioural model. If a dependency between two *Requirements* exists, there will also be two generated behaviour models. However, the generated behaviour model of the dependent *Requirement* will reuse the behaviour model of the *Requirement* it depends on. In the example illustrated in Figure 6.27, a behaviour model for “Req02” is generated which is also reused as part of the behaviour model that is generated for “Req03”. The relationship between *Requirements* defined in the STD and the behaviour models is very important regarding the aspect of traceability of requirements throughout the test generation, execution and evaluation process.

In the following, the processes taking place within the ACE will be further analysed.

Reading the STD Instance

First, the reading of the current STD instance is performed within the ACE. Therefore, a conceptual model for the STD has to be established. The following Figure 6.26 illustrates this conceptual model by means of a UML class diagram.

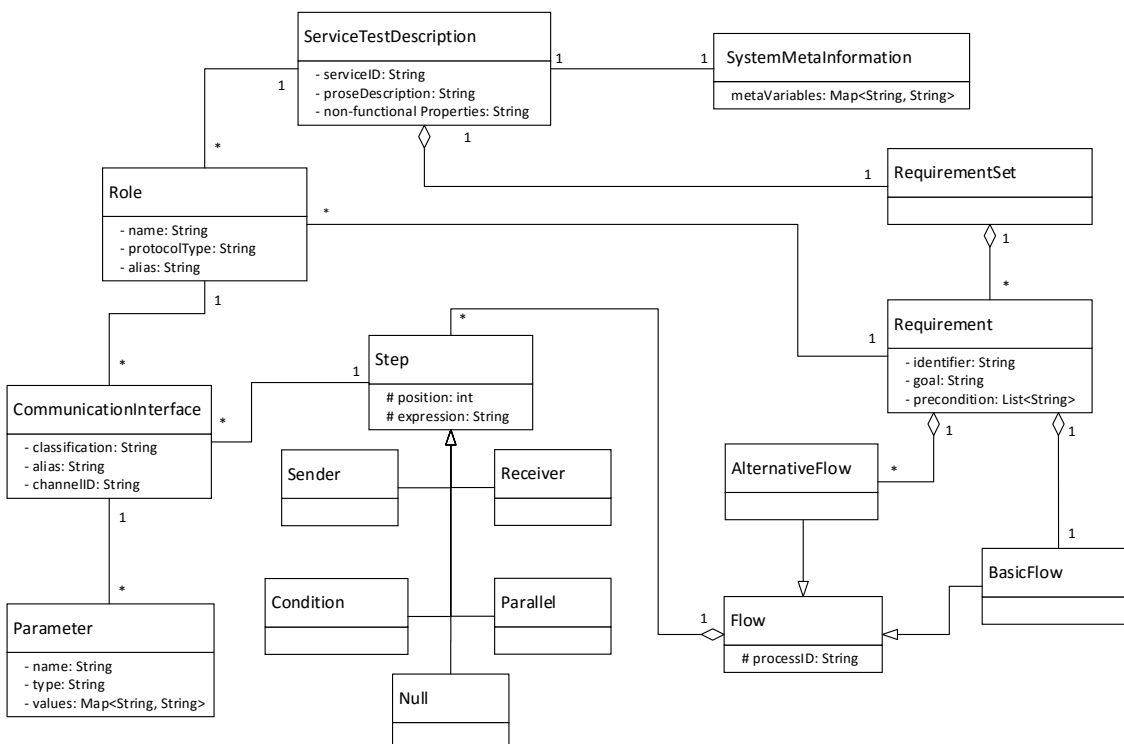


Figure 6.26: Conceptual model of Service Test Description

The UML class diagram shows all the relevant components (or classes) including their attributes that have to be readable for the ACE. The main class, of course, is the *ServiceTestDescription*, which is specified through its attributes *serviceID*, *proseDescription* and *non-functionalProperties*. Furthermore, the class has a reference to the *SystemMetaInformation* and to one or many *Roles* and contains one *RequirementSet*. The *SystemMetaInformation* class includes all possible key-value pairs (e.g.

“serviceURI” as key and “chatservice@sip.de” as value). The *Role* class is described through its attributes *name* (e.g. “SIP phone”), *protocolType* (e.g. “SIP”) and *alias* (e.g. “[s]”). It has a reference to one or many *CommunicationInterface* objects. Of course, this depends on the *Role* type (SIP phone for instance contains four different CIs). A *CommunicationInterface* class includes the attributes *classification*, *alias* (e.g. “[sender1]”) and *channelID* (e.g. “channel a”). The *classification* attribute refers to the type of CI and also directly to the reusable test module (e.g. “SIP UAS non-INVITE”). Each *CommunicationInterface* has a set of *Parameters* from which each can be specified through its *name* (here, any kind of name can be determined), *type* (e.g. “r_Request”) and *values*. The *RequirementSet* as part of the *ServiceTestDescription* contains an unspecified number of *Requirement* objects. A *Requirement* has an *identifier* (e.g. “Req02”), a *goal* as well as one or many *precondition* items (e.g. “Req01”). A *Requirement* has a one or many *Role* objects that are participating within the *Requirement* and it contains one *BasicFlow* and one or many *AlternativeFlow* objects. Each *Flow*, irrespective of whether it is a *BasicFlow* or *AlternativeFlow*, has a *processID* (e.g. “P”) and includes a list of *Steps*. There can be five different types of *Step* objects: *Sender*, *Receiver*, *Condition*, *Parallel* and *Null*. The *Sender* object refers to a *Step* where a message is sent through a channel (e.g. “ \bar{b} (okMessage)”) whereas *Receiver* refers to the opposite (e.g. “ a (initMessage)”). A *Condition* obviously specifies an *if-then-else* construct and the *Parallel* class the specification of concurrent behaviour. Finally, the *Null* class refers to the end of a process.

Based on the illustrated conceptual model of the STD, each instance can be completely specified. Another advantage is that instances can be persistently stored, e.g. in a relational database.

Read Requirement and instantiate Test Modules

Now that STD instances can be read, the parsing is processed. The key components of a behaviour model are the reusable test module instances. Based on the conceptual model of an STD instance, the ACE algorithm can parse the relevant information to create instances of the reusable test modules and integrate them into new behaviour models. The following flow chart describes the algorithm how the test module instantiation is performed (see Figure 6.27).

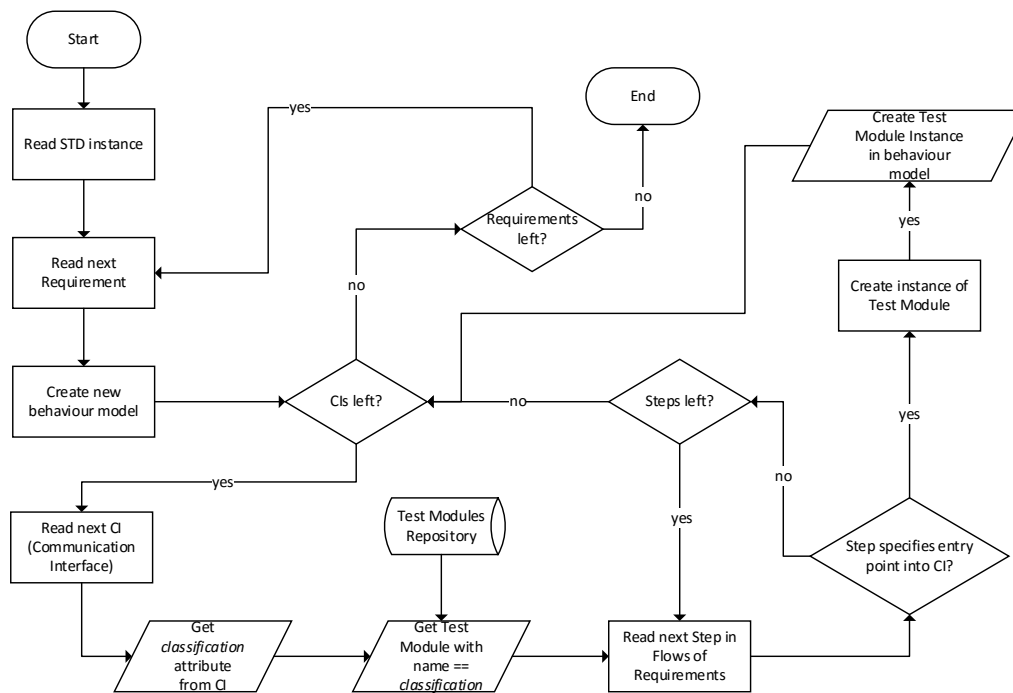


Figure 6.27: Test modules instantiation in behaviour model flow chart

First, the initial *Requirement* within the *ServiceTestDescription* is parsed by the algorithm and a new behaviour model is created. Then, for each *CommunicationInterface* specified within the *Requirement*, the algorithm compares the *classification* attribute with the entries in the TMR. If there is a match, the stored reusable test module is read in (both classification template as well as SCXML description). Afterwards, the algorithm parses

the flow descriptions (both *BasicFlow* and *AlternativeFlow*) and detects the *Step* objects where the corresponding *CommunicationInterface* is involved. For every *Step* object that describes an entry point into the channel of the *CommunicationInterface* (a new transaction, for instance an “r_Request” for the “SIP UAS non-INVITE” test module), a new instance of the reusable test module is created and added to the behaviour model. The following Figure 6.28 exemplifies the process.

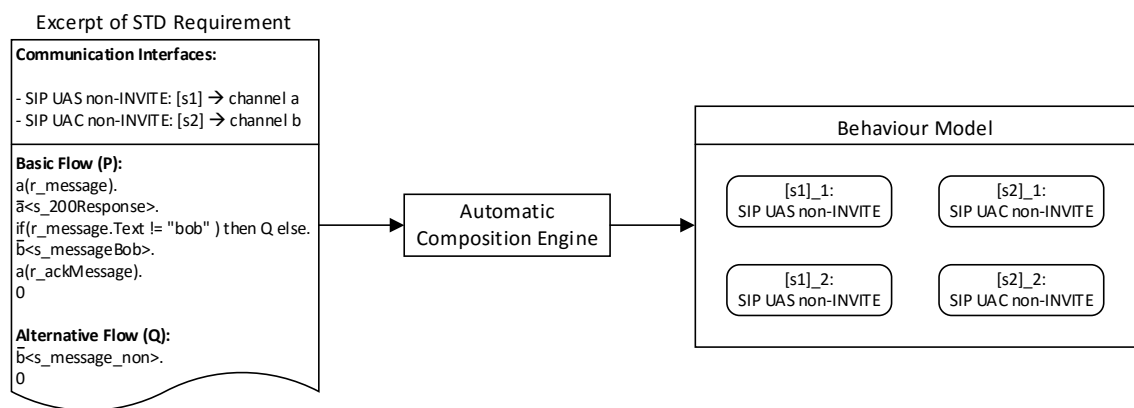


Figure 6.28: Test modules instantiation process example

The excerpt of the example STD instance defines two different *CIs* and the behavioural description contains one *BasicFlow* and one *AlternativeFlow*. Within the *Steps* defined in the *Flows*, there are two entry points for “[s1]” (“r_message” as well as “r_ackMessage”) and also two for “[s2]” (“s_messageBob” and “s_message_non”). The sending of the “s_200Response” does not describe an entry point because it specifies behaviour within the test module. So, the resulting behaviour model for the specific *Requirement* contains four reusable test module instances.

Read and parameterise variables

The next two processes within the ACE deal with the handling of test data (see Figure 6.25). As the reusable test module instances have already been identified and integrated in the resulting behaviour model, the variables can now be read. Figure 6.24 already illustrated how variables of reusable test modules are stored within the *Test Data Pool*. Now, they need to be integrated into the behaviour models. The following flow chart (see Figure 6.29) illustrates the process.

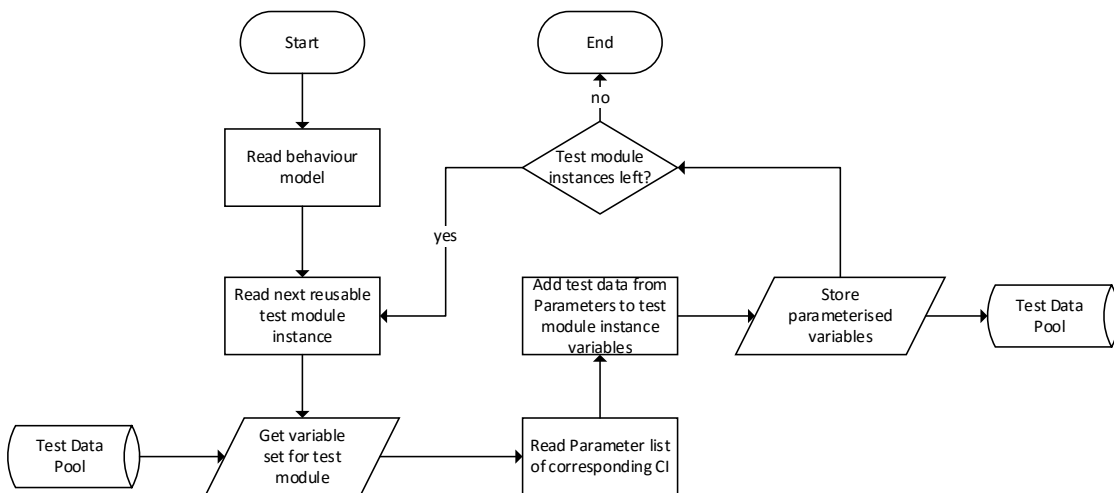


Figure 6.29: Variable reading and parameterisation flow chart

To each instance of a reusable test module within the behaviour model, the adequate sets of variables are assigned in the first step. Then the relevant information within the STD instance is parsed, namely the *Parameter* objects as part of the *CI* that relates to the reusable test module. Then, the test data specified in the *Parameter* objects is integrated into the variable instances of the reusable test module instance. Finally, the parameterised variables are stored within the *Test Data Pool*.

Compose test modules

The final task the ACE algorithm has to perform for a given *Requirement* within an STD instance is the composition of the reusable test module instances into a valid behaviour model. Regarding the previous steps, the behaviour model consists of a set of reusable test module instances that contain parameterised variables according to the specifications in the STD description. Now, the composition algorithm as part of the ACE has to parse the defined *Step* objects within the *BasicFlow* and *AlternativeFlow* sequentially. For every parsed *Step*, the composition algorithm has to decide what effect its definition has on the behaviour model. As illustrated in the conceptual model of the STD (see Figure 6.26), there are five different categories of *Steps*: *Sender*, *Receiver*, *Parallel*, *Condition* and *Null*. Each of the *Steps* have a different impact on the composition algorithm. Besides, both *Sender* and *Receiver* need to have knowledge about the prior *Step*, whereas the *Condition* and *Parallel* can involve both the direct prior *Step* as well as the next *Step*. The *Null Step* is the exception, because it is actually the final *Step* within any *Flow*.

In general, the sum of all *Steps* within the *Flows* specify the behaviour of the value-added service following the *Requirement*. The change from one *Step* A to a next *Step* B can cause different changes in the static behaviour model that only includes instances of reusable test modules so far. First, a *Step* change can lead to a change of the current active reusable test module instance if B refers to a different channel than A. Secondly, if both *Steps* refer to the same instance of a reusable test module, the behaviour is restrictively determined. An example can be, for instance, that A specifies the SUT to send an INVITE request and B specifies in the same channel that a “200 OK” response is expected. In this case, the specified “path” is determined as the success path. Only if the messaging is

performed in this sequence, the test cases that are derived later on will pass. On every transition within an instance of a reusable test module, a flag “pass” can be set, either to “false” (wrong path, which leads to an error) or to “true” (correct path). Theoretically, the “pass” flag can be determined for every transition within a reusable test module. This makes sense for SIP messaging when for instance only provisional or successful response messages are expected.

In the following, the different *Step* types will be discussed with regard to the composition algorithm. The *Sender* refers to a *Step* where any kind of message (either request or response message) is sent from the SUT. If the message is a request type (e.g. “ \bar{b} (*okMessage*)”), the *Step* definitely describes an entry point into a reusable test module. This means that a new transition is established that does not include an event but the request message as action. The target of the transition is the first state after the “start” state of an instance of the reusable test module that refers to the corresponding *CI*. It has to be specified which instance is taken as there can be a number of instances for one *CI* (e.g. “[s2]_1” and “[s2]_2” illustrated in Figure 6.28). Therefore, the algorithm counts how many transactions have been initiated and terminated before the current *Step* within the *Flows* regarding the specific channel. If the number is for instance “1”, then the second instance of the reusable test module will be selected as target. Of course, a transition also requires a source state. This is derived through the direct prior *Step* of the current one. Based on the channel and specified message, the source state can be detected within the corresponding reusable test module instance. The whole process is different if the message is a response type (e.g. “ \bar{b} (*okResponse*)”). If the prior *Step* determines a request type to be received over the same channel than the *Step* does not effect a change of reusable test module instance. In that case, a restricted path has been determined and the

“pass” flag will be set to “true”. Alternatively, if the prior *Step* determines a message sent or received over a different channel, a new transition will be created which also includes the message response as action.

The flow chart displayed in Figure 6.30 illustrates the composition algorithm focusing on the *Sender Step*.

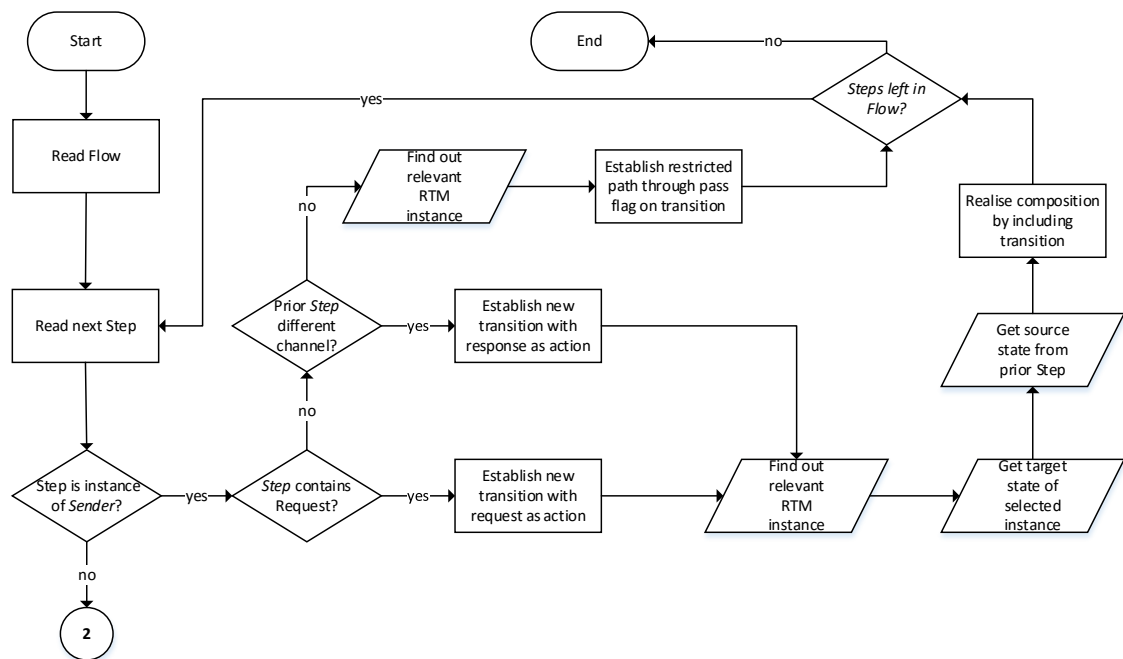


Figure 6.30: Composition algorithm flow chart for Sender Step

The main decisions that have to be made regarding the parsing of a *Sender Step* depend on the type of message that is sent. If it is a request message, it is obvious that a new reusable test module instance will be connected with a transition. Contrary to this, every response message leads either to a staying within the current reusable test module instance (if the prior *Step* contains the identical channel) or an establishing of a new transition (if the prior *Step* contains a different channel). The flow chart also contains a reference to another flow chart (“2”) which will describe the *Receiver Step*. Before focusing on the

next *Step*, an example composition using a *Sender Step* will be shown in the following Figure 6.31.

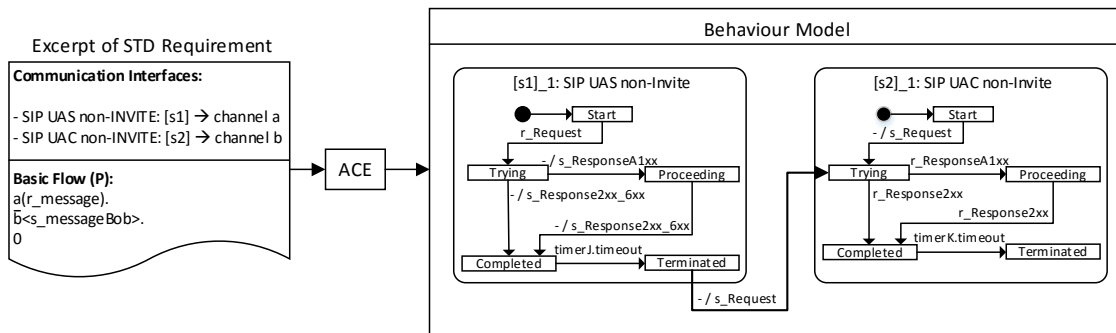


Figure 6.31: Example composition of reusable test module instances with focus on Sender Step

The displayed example composition includes simplified reusable test module instances of “SIP UAS non-INVITE” and “SIP UAC non-INVITE”. The second *Step* in the *BasicFlow* specifies the *Sender Step*. Here, the channel “b” of the *CI* “[s2]” is used to send a request message after a message request was received over channel “a”. The composition algorithm then generates a new transition from the “Terminated” state of the “[s1]_1” reusable test module instance to the “Trying” state of the “[s2]_1” reusable test module instance with the specified request message defined as action.

The *Receiver Step* is very similar to the *Sender Step*, because the consequence is the same. A message request leads to a new transition which targets a new instance of a reusable test module. The only difference is that the new transition in comparison to the *Sender Step* only contains an event but no action. The same aspect is valid if a response message is specified. The following flow chart (see Figure 6.32) enhances the previous flow chart displayed in Figure 6.30 but focusing on the *Receiver Step*.

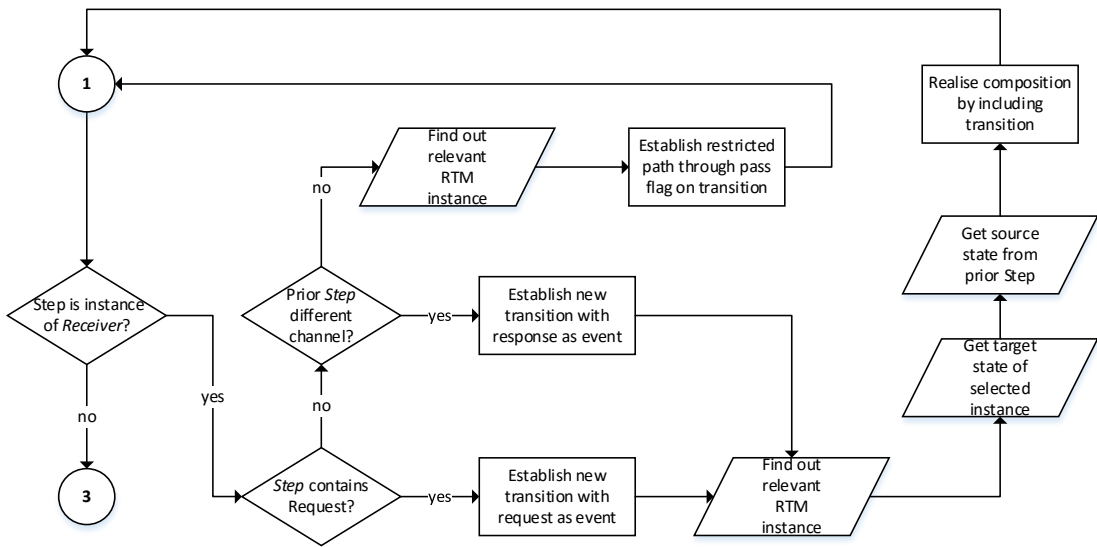


Figure 6.32: Composition algorithm flow chart for Receiver Step

The main difference between the two different flow charts for *Sender* and *Receiver Step* can be identified on the created transitions. Here, the transitions include events instead of actions. Besides, the flow chart also contains two references (“1”) and (“3”). The reference (“1”) targets back to the *Sender Step* flow chart, specifically to the decision module “Steps left in Flow?”. This is relevant because the flow charts describe the parsing process as a loop. The reference (“3”) targets to the next possible *Step* to be analysed.

An example illustration is also given for the *Receiver Step* in the following Figure 6.33.

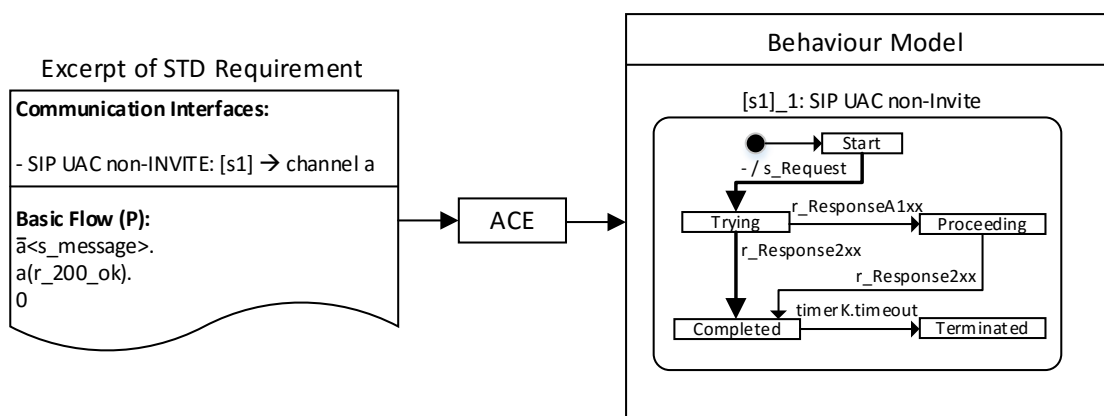


Figure 6.33: Example parsing with focus on Receiver Step

This example does not specify a composition between reusable test module instances, but the determination of a restricted path through a *Receiver Step*. The second *Step* within the *BasicFlow* of the *Requirement* states that a response message is expected on channel “a” after a request was sent over the identical channel. This leads to a restricted path within the “[s1]_1” reusable test module instance.

The next *Step* specifies concurrent behaviour, the *Parallel Step*. It describes a behaviour where transactions, either server-based or client-based, are opened in a short time interval. The problem with this is that the order of the potential incoming messages cannot be specifically determined. Even if a request within a transaction A is for instance sent before the request within a transaction B, still it is possible that response messages relating to B will be received earlier than the response messages that relate to A. The following flow chart (see Figure 6.34) describes what the composition algorithm has to do.

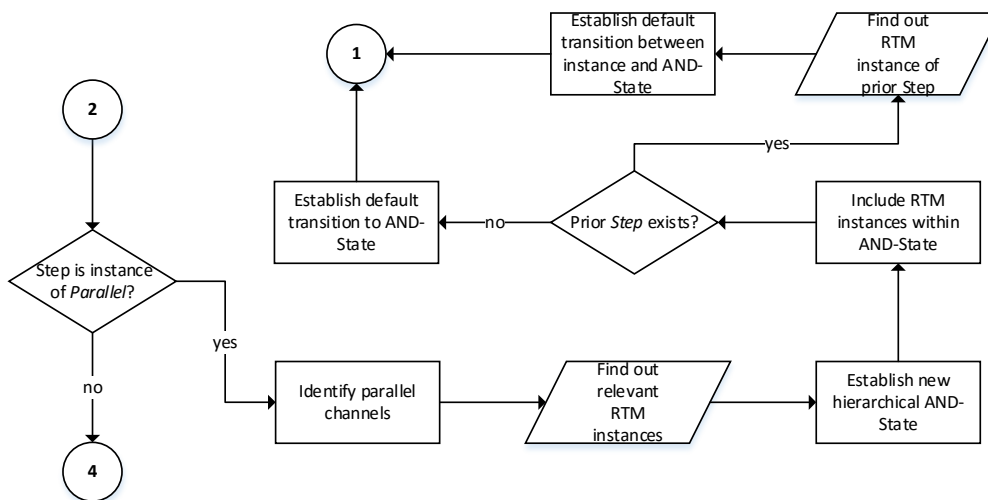


Figure 6.34: Composition algorithm flow chart for Parallel Step

First, all relevant parallel channels are identified and the corresponding reusable test module instances are detected. Then, a new hierarchical AND-state is established, which enables to describe concurrency in Statecharts notation (see section 6.1.2). The detected

reusable test module instances are then included into the hierarchical AND-state. It is important to mention that incoming and outgoing transitions on AND-states are always default transitions that do not contain any events or actions. However, it has to be analysed which reusable test module instance contains the originating state. If there is a prior *Step* before the *Parallel Step*, the corresponding reusable test module instances will be found and a transition can be established. Otherwise, a new default transition from the start state of the behaviour model to the AND-state will be generated.

The following Figure 6.35 demonstrates an example where the *Parallel Step* is used.

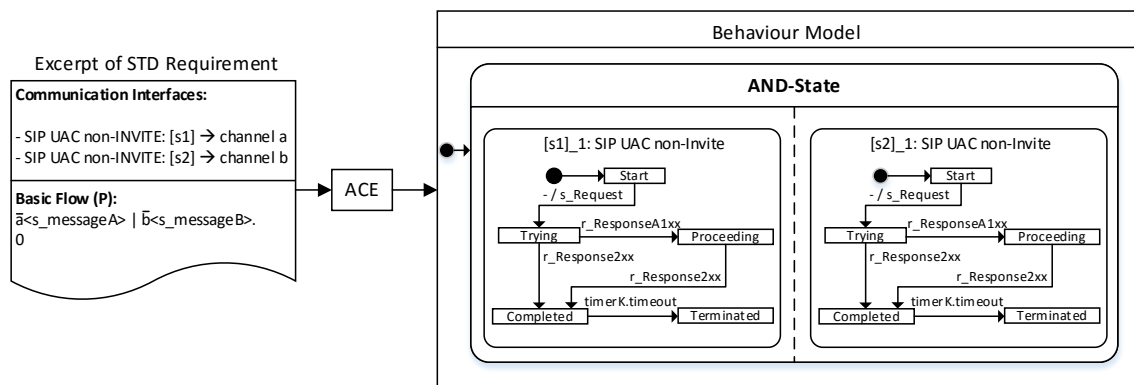


Figure 6.35: Example composition of reusable test module instances with focus on Parallel Step

The simple example STD shows a *Requirement* with two “SIP UAC non-INVITE” *CIs*. Within the *BasicFlow*, the parallel sending of request messages through the channels “a” and “b” is determined through the *Parallel Step* (by means of the “|” statement in pi-calculus). Based on the given notation, the ACE creates a new instance of a hierarchical AND-state. Within the AND-state, the two corresponding reusable test module instances are included. Then, a default transition from the start state is included that has the AND-state as target.

The *Condition Step* does not contain any message sending or receiving, but it can use content of messages from prior *Steps*. The *Condition Step* itself describes a distinction of cases and is comparable to a standard if-then-else structure. In the following, an example STD specification is illustrated that contains one *Condition Step*. The result of the composition algorithm is also shown in Figure 6.36.

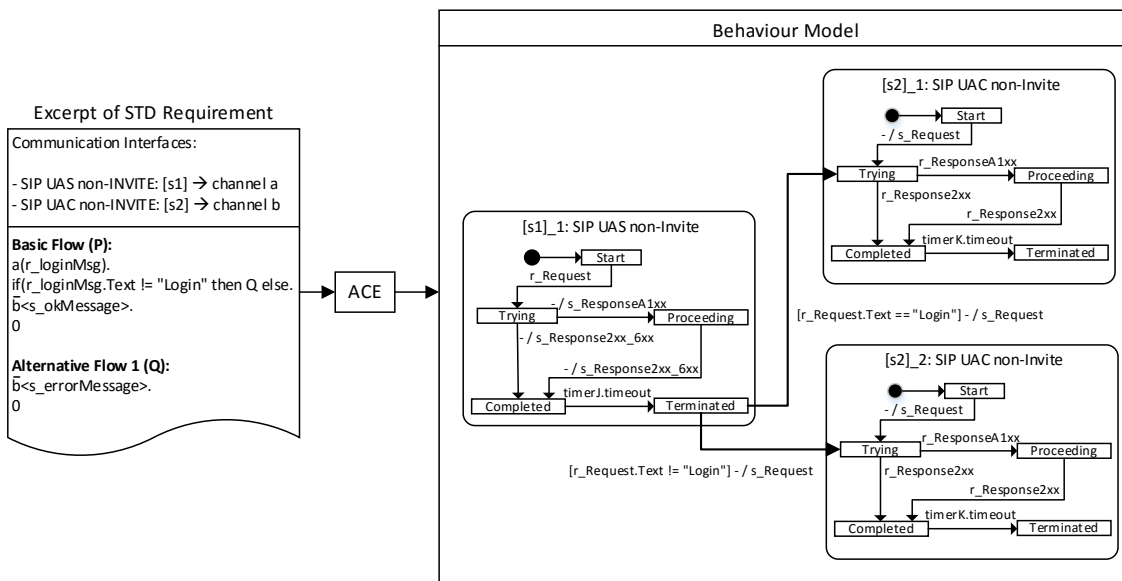


Figure 6.36: Example composition of reusable test module instances with focus on Condition Step

A request message is received over channel “a”, specifically a SIP MESSAGE. It contains a text that is checked in the following *Condition Step*. If the message does not contain the value “Login”, the *AlternativeFlow* is invoked and a new request message “s_errorMessage” is sent over the “b” channel. Alternatively, the “Login” is part of the incoming SIP MESSAGE and the request message “s_okMessage” is sent. Altogether, the *Flow* specification contains three entry points, so three reusable test module instances have to be established within the behaviour model. The “[s1]_1” reusable test module instance deals with the receiving of the initial message. As soon as it terminates, two new outgoing transitions are created because of the *Condition Step* definition. Both contain a

transition guard and determine the specified conditions and lead to different instances of the “SIP UAC non-INVITE” reusable test module.

The final *Null Step* refers to the end states of each defined *Flow* (both *BasicFlow* and *AlternativeFlow*). A behaviour model specifying the behaviour of a given *Requirement* contains as many ends as it contains *Flows*. The end states can be seen as connection points between *Requirements* that are depending on one another. If *Requirement* “Req02” depends on “Req01”, the end point of the *BasicFlow* in the behaviour model of “Req01” can be eliminated and the loose connection can be linked to the start state of “Req02”. This principle allows the connection of diverse *Requirements*.

This section demonstrated the role of the ACE, the automatic building of the behaviour models. The whole process can be summarised as follows:

1. The ACE reads the instance of the STD by means of the conceptual model (see Figure 6.26).
2. For each *Requirement* within the STD instance, the ACE creates a new behaviour model instance. Based on the participating *Roles* and *CI*s within the *Requirement*, the ACE also creates new instances of reusable test modules and assigns them to the behaviour model instance.
3. Based on the *Parameters* specified within each *Requirement*, the ACE creates new sets of variables for each reusable test module instance and stores them in the Test Data Pool.
4. Finally, the *BasicFlow* and the *Alternative Flows* of each *Requirement* is parsed. Depending on the category of the parsed steps within the flows (either *Sender*,

Receiver, Condition, Concurrency or *Null Step*), a different composition of the reusable test module instances is performed by the ACE.

5. The result at the end is a set of behaviour model instances.

6.5 Conclusion

Within this chapter, the concept of the reusable test modules has been introduced as well as the generation of the behaviour models based on the content of STD instances. First, a suitable modelling notation had to be found in order to specify the occurring behaviour within the reusable test modules. Taking into consideration relevant criteria such as the possibility to integrate concurrency, reusability, temporal logic as well as having an underlying formal specification, the Statecharts notation has been chosen. A new way of defining Statecharts has been introduced by means of the TU concept (see section 6.1).

Furthermore, the chapter has introduced in section 6.2 how the reusable test modules are created by the test developer by means of the Test Modelling Environment (TME). It has been discussed how reusability can be detected specifically for value-added telecommunication services and how the resolving reusable test modules can be classified (e.g. classification template) and modelled.

Then, the aspect of handling test data has been shown in section 6.3. As a result, each defined reusable test module contains a set of variables that can be parameterised by *Parameters* that are defined within the *Requirements* of STD instances.

Finally, the behaviour model generation has been described in section 6.4. The focus of the chapter is the ACE, an important component of the proposed TCF, which realises the

parsing of STD instances and simultaneously generates behaviour models for given value-added telecommunication services.

The behaviour model concept also has a significant meaning for the upcoming chapter, as all generated behaviour models that are assigned to a value-added service build the foundation for the generation of test cases. An algorithm has to be developed which realises the test case derivation and also the test case generation of TTCN-3 test cases. Furthermore, the upcoming chapter gives answers regarding the test case execution and evaluation.

7 Test Case Generation, Execution and Management

The chapter comprises three very relevant processes within the TCF architecture, the generation of test cases, their execution against the SUT as well as the subsequent analysis and management of the upcoming test results. Based on the output of the ACE algorithm discussed in the previous chapter, these processes can apply. Figure 7.1 illustrates the processes as well as their inputs and outputs.

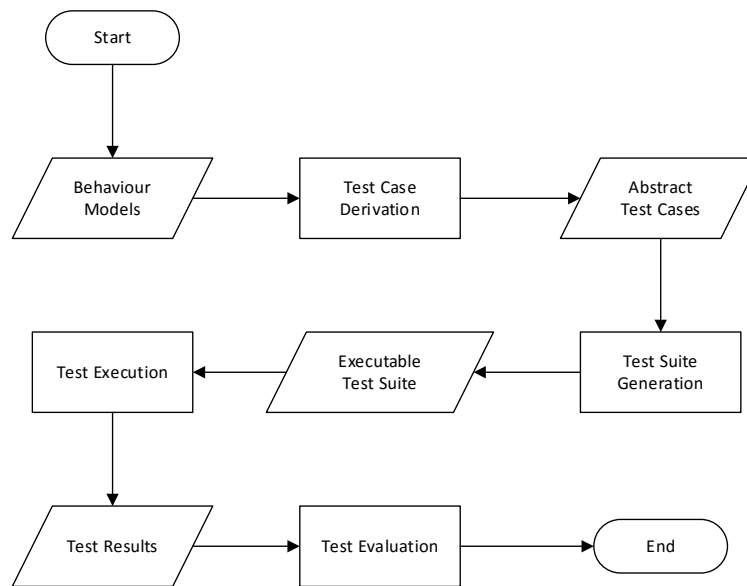


Figure 7.1: Generation, Execution and Evaluation of Test Cases

This chapter is structured based on the illustrated processes in Figure 7.1. Initially, section 7.1 (see Figure 7.1, *Test Case Derivation*) deals with the derivation of test cases from the behaviour models generated by the ACE algorithm. The relevant steps are performed by

the Test Case Derivation Unit (TCDU) which is part of the TCF architecture (see Figure 4.5). For each behaviour model, the TCDU derives a reasonable amount of abstract test cases by applying a specific structural coverage criterion. Advantages and disadvantages of existing coverage criteria will also be discussed in section 7.1. Section 7.2 (see Figure 7.1, *Test Suite Generation*) introduces the Test Suite Generator (TSG) as architecture component of the TCF (see Figure 4.5). It comprises a Test Code Generator (TCG) as well as a Test Suite Builder (TSB). The main task of the TCG is to read the abstract test cases derived from the TCDU and to subsequently generate the appropriate TTCN-3 code. The TTCN-3 code generation is separated into three different parts:

1. Generation of test code for test configuration.
2. Generation of required test data templates.
3. Generation of test behaviour by means of TTCN-3 test cases.

In the final step, the TCG generates collections of test cases which can be directly mapped to the *Requirements* specified within the corresponding STD instance. Now, the TSB performs a compilation of the generated TTCN-3 code and generates an Executable Test Suite (ETS). The final step includes a transmission of the ETS to the TTCN-3 test execution environment. The third process described in this chapter is the execution of the tests against the SUT (see Figure 7.1, *Test Execution*) in section 7.3. Here, the principles of executions within TTCN-3-based environments is discussed. An example test case invocation is shown as well as its impact on the components of a TTCN-3 system. The final process in section 7.4 (see Figure 7.1, *Test Evaluation*) introduced in this chapter refers to the management and evaluation of test results. It has to be specified how a valid product can be achieved which involves all stakeholders within the service development

process. In principle, the test developer analyses the tests. If test case errors occur, the test developer first has to figure out if he made a mistake in the STD definition. If this is not the case, the test management requires the involvement of the Service Quality Group (SQG) (see section 4.2) and the service customer.

7.1 Generation of Abstract Test Suite

Regarding the TCF architecture, this section deals with the *Test Case Derivation Unit* (TCDU), a component which derives abstract test cases and builds an abstract test suite from the generated behaviour models.

7.1.1 From Behaviour Models to Abstract Test Cases

As described in section 6.4, the behaviour models, just as the reusable test modules, are based on the applied Statecharts notation. If there are n *Requirements* defined for a value-added service within an STD instance, there will also be n different behaviour models from which test cases have to be derived. The following Figure 7.2 is based on the example behaviour models illustrated in Figure 6.25 and shows the test case derivation from behaviour models.

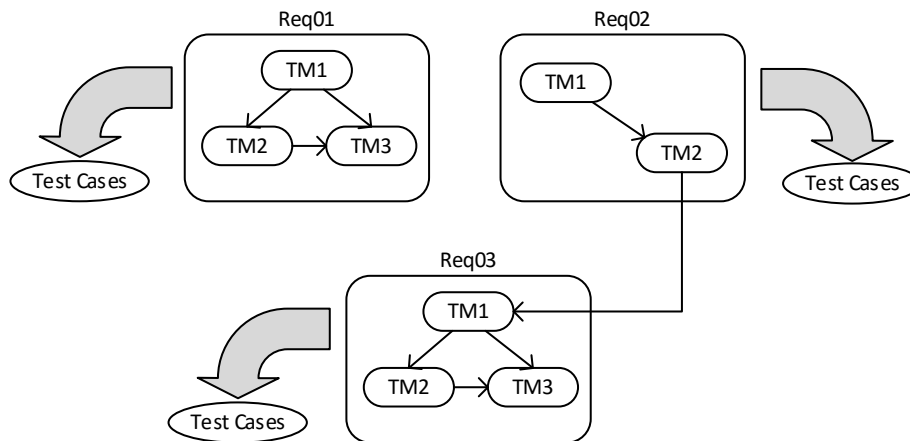


Figure 7.2: Test case derivation from behaviour models

For each behaviour model, irrespective of whether or not it includes a dependency to another behaviour model, test cases are derived. If there is a dependency included between two *Requirements* (such as between “Req02” and “Req03”), the test case derivation of “Req03” needs to consider the behaviour model of “Req02” while deriving the test cases. Each test case for “Req03” will then start at the beginning of the description of the behaviour model of “Req02” and will end within the behaviour model of “Req03”. Theoretically, it would then be sufficient to just generate test cases from independent behaviour models (such as “Req01”) and from composed behaviour models (such as “Req03”) because it also includes the test cases that are relevant for the *Requirement* it depends on (here, it is “Req02”). However, the proposed approach in this research enables a thorough traceability of requirements, especially to be able to do a “rapid prototyping”-alike procedure where both service developer and test developer can focus on implementing or rather testing the requirements step by step. Following this approach, “Req02” can be tested even if “Req03” is not yet specified.

Focusing on the task of the TCDU, the TCF component needs to analyse the diverse behaviour models that it gets as input and produce one abstract test suite as output. An abstract test suite contains collections of abstract test cases that are sorted according to the behaviour models (and therefore also according to the *Requirements*) that they have been derived from. In offline Model-based testing approaches, abstract test cases are quite commonly derived from models (see section 3.2.4). According to (Devroey *et al.*, 2014), an abstract test case is defined as a trace $atc = (\alpha_1, \dots, \alpha_n)$ within a model that specifies behaviour and can therefore be understood as a finite sequence of actions α that might occur according to the model description. In the case of this research, the concept of abstract test cases differs a little bit. The concept will be explained in the next section 7.1.2. The following Figure 7.3 presents the main task of the TCDU showing the necessary inputs and outputs.

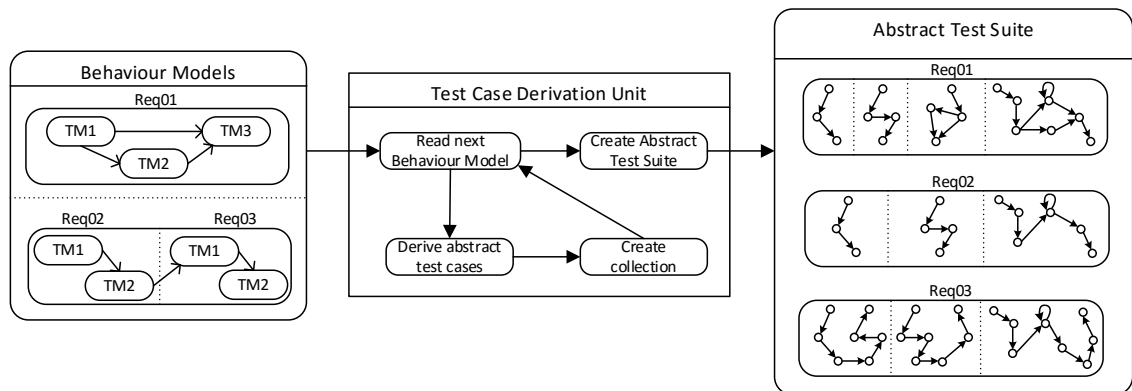


Figure 7.3: Abstract test case generation from behaviour models by Test Case Derivation Unit

As soon as the TCDU gets the behaviour models as input, it successively reads them and derives the abstract test cases. For every set of abstract test cases belonging to a specific behaviour model, a collection is created. After all behaviour models have been processed, the TCDU creates the abstract test suite exemplified on the right side of Figure 7.3. It has to be noticed that each of the three collections illustrated within the abstract test suite

includes abstract test cases as test paths. In principle, this is comparable to the definition of (Devroey *et al.*, 2014), however, Figure 7.3 also shows some test paths that contain loops or alternative paths. The reason for this will be described in the following section which includes the description of the underlying abstract test case derivation algorithm.

7.1.2 Test Case Derivation

For the derivation of test cases from formal models, the literature discusses several approaches and algorithms that can be applied, such as in (Ammann and Offut, 2008), (Utting and Legeard, 2006), (Binder, 1999) and (Tahat *et al.*, 2001). In general, the approaches are referred to as so-called structural coverage criteria. Especially for transition-based models such as Statecharts, there are many different structural coverage criteria that can be used to manage test case derivation. Depending on the selected structural coverage criteria, a test case generator automatically generates a set of test paths within the model from an initial state to the end state. A selection of possible structural coverage criteria is illustrated in the following Figure 7.4. Permission to reproduce Figure 7.4 has been granted by ACM.

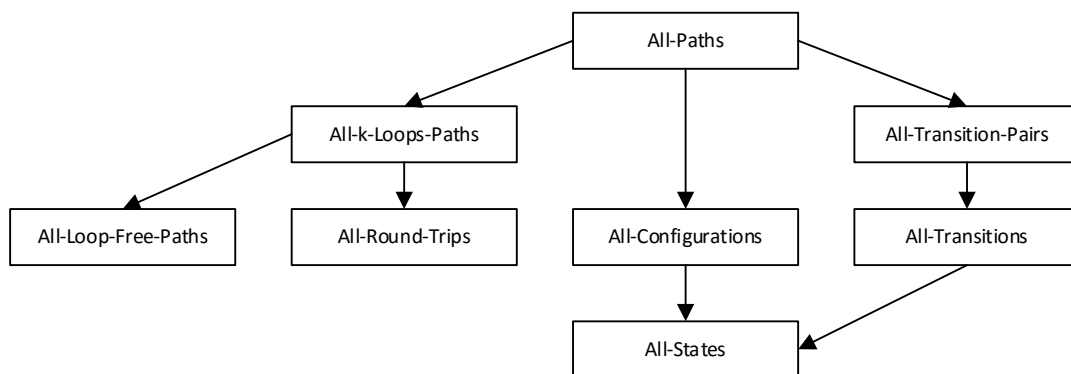


Figure 7.4: Hierarchy of structural coverage criteria (adapted from (Haschemi, 2009))

According to (Haschemi, 2009), the diagram shows the strongest structural coverage criterion at the top and weaker ones in a lower level. The arrow between the criteria illustrates that every test suite satisfying a criterion c_1 (arrow source) subsumes another criterion c_2 (arrow destination). The meaning of the diverse structural coverage criteria is as described in (Binder, 1999), (Ammann and Offut, 2008) and (Haschemi, 2009):

- *All-States* – Every defined state within a given model is visited at least once.
- *All-Transitions* – Every transition of the model must be traversed at least once.
- *All-Transition-Pairs* – Every pair of adjacent transitions in the model must be traversed at least once.
- *All-Configurations* – A configuration is a set of concurrently active states. This criterion requires that all configurations of the model's states are visited.
- *All-Round-Trips* – This criterion requires a test case for each loop in the model and that it only has to iterate once around the loop.
- *All-k-Loops-Paths* – Every path that contains at most two repetitions of one configuration has to be traversed at least once. This requires all the loop-free paths within the model to be visited at least once and additionally, all the paths that loop once.
- *All-Loop-Free-Paths* – Every path free of loops has to be traversed at least once. A path is loop-free if it does not contain any repetitions.
- *All-Paths* – This coverage is satisfied as soon as all paths of the model are traversed at least once. This criterion is usually not practical because models typically contain an infinite number of paths, especially if they contain loops.

For this research, the existing structural coverage criteria have been evaluated, however, none of them could be directly applied for the given behaviour models. Of course, it would be possible to apply each of the mentioned structural coverage criteria on the Statechart-based notation, but most of the derived abstract test cases will run result in an inconclusive verdict as soon as they have been made executable. This has to do with the fact that resulting from all these coverage criteria, linear test cases are derived consisting of a linear sequence of events and actions. In principle, this aspect is not well suited for testing of a value-added service that is supposed to operate within a reactive environment. It might be possible that a value-added service responds to a stimuli triggered by the test execution environment in a valid but unexpected way. To exemplify the issue, a standard Three-Way-Handshake for SIP (IETF RFC 3261, 2002) is considered. The test execution environment sends an INVITE request in order to establish a session to the value-added service. The linear test cases that this behaviour relies on, first expects a provisional message (e.g. “100 Trying”) from the SUT and afterwards a successful “200 OK” response. Now the SUT, after having sent the expected “100 Trying” message (incidentally, this message will always be sent by a Stateful Proxy Server that is included within the NGN environment), sends another provisional message (e.g. “180 Ringing”). Although this behaviour is allowed as an option, the test system compares the incoming “180 Ringing” with the expected “200 OK” message and will come to the conclusion that the response does not match. Accordingly, the test case will fail or will be evaluated as inconclusive. The problem of this test case derivation strategy is that the linear test cases do not describe multiple expected output states. However, the concept of the applied Statecharts notation (see section 6.1.3), having the messages that the SUT expects as events and the ones it potentially sends as actions, allows a different representation of test

cases than in the standard linear form. In fact, a test case derived from a behaviour model can also be presented as a directed graph $G = (V, E)$, where V is a set of vertices and E is a set of edges and where each edge is a pair of vertices. Especially in a directed graph, an edge is an ordered pair of two vertices (u, v) with the edge pointing from u to v . Contrary to linear representations of test cases, a graph is able to determine branches. So, any given vertex $v_i \in V$ can theoretically have an infinite number of outgoing edges. However, according to the test case representation, there is a restriction defined. A vertex $v_i \in V$ can only have more than one outgoing edge if it specifies an action and not an event.

In order to exemplify the novel principle of the test case representation with graphs, two example Statechart descriptions will be analysed. But before, an appropriate structural coverage criterion has to be selected. Even though the general output of the mentioned coverage criteria is a linear test sequence, still the concept behind the criteria can be applied for the graph-based test sequences. For this research, the structural coverage criterion *All-Round-Trips* has been selected. According to (Binder, 1999) and (Utting and Legiard, 2006), this structural coverage criterion can be satisfied with a linear number of test cases whereas the All-Paths-based criteria (such as *All-Paths* itself, *All-k-Loops-Paths* and *All-Loop-Free-Paths*) require an exponential number of test cases if the model contains many alternative branches. This is important, because all of the specified SIP-based example Statecharts contain a few branches and also loops. In comparison to standard structural coverage criteria such as *All-Transitions* and *All-States*, the *All-Round-Trips* is able to detect faults more thoroughly (Antoniol *et al.*, 2002), as the tests are more extensive. Additionally, (Binder, 1999) explicitly recommends this coverage criterion for model-based approaches.

In the following illustration (see Figure 7.5), the “SIP UAC non-INVITE” behavioural description is illustrated with a special identification of the transitions (e.g. “{a1}”).

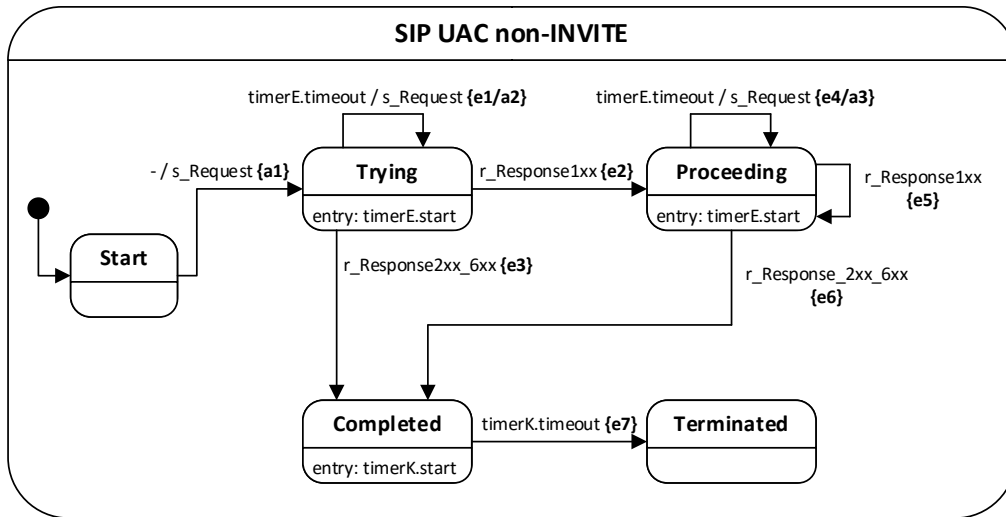


Figure 7.5: Behavioural description of SIP UAC non-INVITE (with transition marking)

The test derivation based on the behavioural description will be realised as follows. In principle, the *All-Round-Trips* algorithm includes the *All-Transitions* algorithm without loops and adds one further test case for each occurring loop within the model. Based on the behavioural description of “SIP UAC non-INVITE”, five test cases can be derived. They are illustrated in the following Figure 7.6.

TC1	S → a1 → Tr → e3 → C → e7 → Te
TC2	S → a1 → Tr → e2 → P → e6 → C → e7 → Te
TC3	S → a1 → Tr → e1/a2 → Tr → e3 → C → e7 → Te
TC4	S → a1 → Tr → e2 → P → e4/a3 → P → e6 → C → e7 → Te
TC5	S → a1 → Tr → e2 → P → e5 → P → e6 → C → e7 → Te

Figure 7.6: Test case derivation from SIP UAC non-INVITE

The state names within Figure 7.5 have been abbreviated, “Start” to “S”, “Trying” to “Tr”, “Proceeding” to “P”, “Completed” to “C” and finally, “Terminated” to “Te”. The first two test cases “TC1” and “TC2” shown in Figure 7.6 are based on the *All Transitions* without loops. Both describe a standard behaviour of a SIP request being sent from the SUT to the participating entities (or rather the test execution environment). The difference is that “TC2” includes a further provisional message that is sent before the terminating response is sent. The other three test cases “TC3”, “TC4” and “TC5” refer back to the three loops or rather self-transitions that are part of the behavioural description of the “SIP UAC non-INVITE” reusable test module. “TC3” specifies the first timeout of “timerE” that could happen in the “Trying” state, “TC4” correspondingly describes the next timeout of “timerE” in the “Proceeding” state and finally, “TC5” specifies that a further provisional response is sent before the final terminating response. Of course, the loops could be visited more than once and it could also be possible that multiple loops occur within one test case. However, this is not relevant in the “SIP UAC non-INVITE” reusable test module because of the perspective. As it is a client core-based reusable test module, the SUT acts as a trigger by sending the initial request. The test execution environment will react based on the request and send the appropriate responses the SUT has to deal with. The perspective changes if a server core-based reusable test module is applied. Then, the graph-based test case descriptions with branches become relevant. In the “SIP UAC non-INVITE”, there have not been any branches.

The following Figure 7.7 illustrates the “SIP UAS non-INVITE” behavioural description. It specifies the SUT to receive a SIP request from a participating entity or rather test execution environment. Also within this description, the transitions have identifiers included in order to represent the test case graphs.

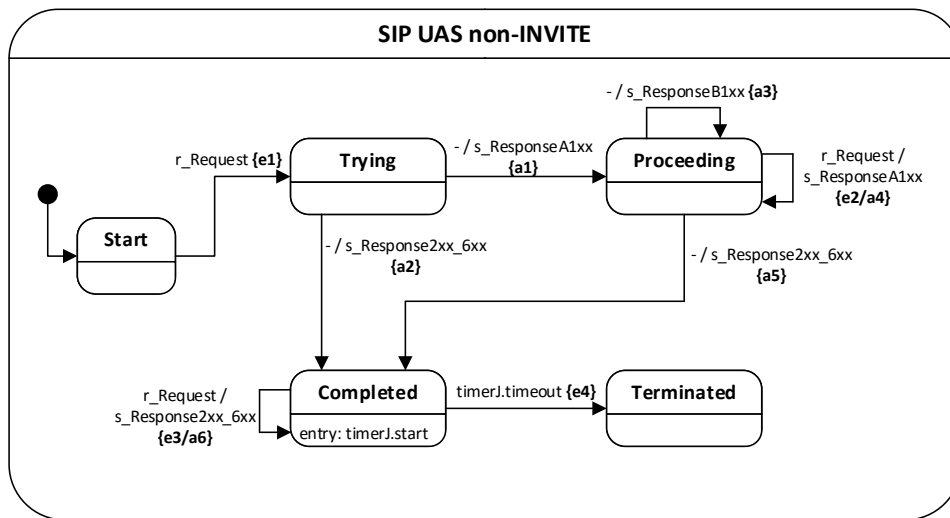


Figure 7.7: Behavioural description of SIP UAS non-INVITE (with transition marking)

From the “SIP UAS non-INVITE” reusable test module (see Figure 7.7), three test cases can be derived by the TCDU. In the following Figure 7.8, they are represented as directed graphs.

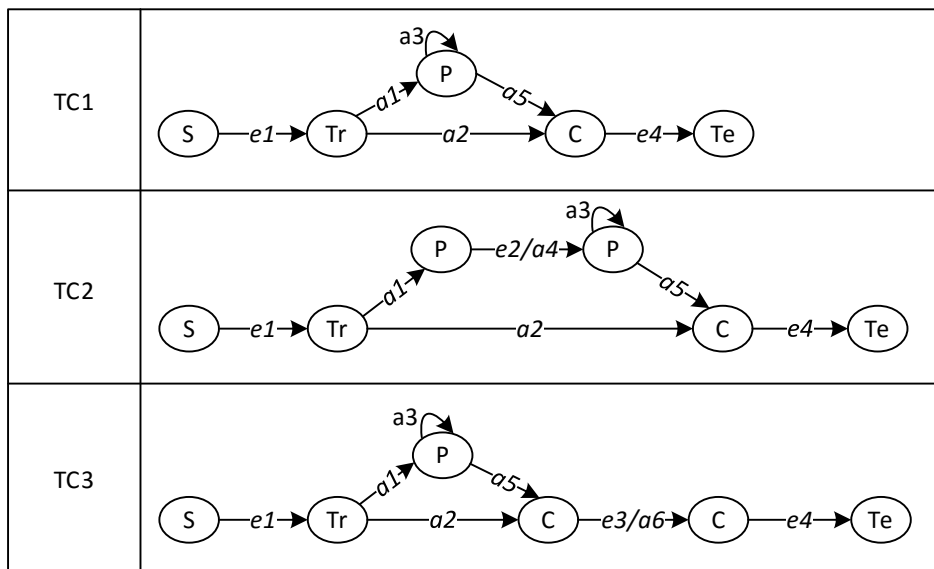


Figure 7.8: Test case derivation from SIP UAS non-INVITE

All three test cases start the same way describing an event “e1” received by the SUT. Afterwards, the SUT can act in two different ways either by first sending a provisional

response (action “a1”) or a terminating response (action “a2”). This branch illustrates why a graph-based test case description is required. It cannot be predicted whether the SUT responds with “a1” or “a2”, but it is obvious that both responses represent valid behaviour. If “a1” is sent by the SUT, the state “Proceeding” is reached. In the graph-based description of “TC1”, the vertex “P” contains a self-loop “a3”. As mentioned before, loops in the *All-Round-Trips* algorithm lead to a new test case and should there only be iterated once. This was a valid approach in the “SIP UAC non-INVITE” reusable test module, because the iteration can be controlled. This cannot be done in the “SIP UAS non-INVITE” case because the SUT is actually allowed to send provisional messages as long as the global timer times out. So, every self-transition within a behavioural description leads to a self-loop within a resulting test case if it only contains an action. Contrary to this, an event specified in a self-transition leads to a new test case that will iterate once in that specific self-transition. The test case will also not contain a self-loop, but a new edge to the corresponding vertex. It symbolises that the state of the SUT actually changed. For instance, “TC2” evolves from the self-transition containing an event in the “Proceeding” state. As soon as the state is reached, the SUT will receive a retransmitted request event (“e2”) and has to respond to this correspondingly (“a4”). This example shows that event-based self-transitions and action-based self-transitions are treated differently. “TC3” evolved from another self-transition containing an event “e3” in the “Completed” state of the behavioural description of “SIP UAS non-INVITE”.

Besides the evident information shown in the test case graphs, there is further information that needs to be included in the edges and vertices. For the later generation of real executable test cases based on the abstract test cases, it is necessary to know to which reusable test module instance the events or actions belong to. Therefore, identifiers of the

instances (e.g. “[s1]_1”) are stored within the edges. Furthermore, the pass flags that the test developer might have set can be included, too. In the vertices, it is relevant to store the starting of timers if it has been determined within the corresponding states of the Statechart description.

The two examples illustrated the principle test case derivation by the TCDU and depicted that there is a difference between test case derivation of reusable test module instances which are server core-based or client-core-based. There are a few questions left regarding the test case derivation. The first one focusses on the composition of two reusable test modules. It is quite obvious what would happen if, for example, two instances of the reusable test modules “SIP UAS non-INVITE” and the “SIP UAC non-INVITE” are composed. The amount of test cases will be the product of the derived test case for each reusable test module instance (in this case 15, because 3 are derived from “SIP UAS non-INVITE” and 5 from “SIP UAC non-INVITE”). Of course, depending on the number of reusable test module instances, the number of test cases can increase quite fast. A possibility to decrease this amount can be the use of a different structural coverage criteria, such as the *All-Transitions* strategy. For both reusable test module instances, applying *All-Transitions* will lead to 2 test cases each. This is an enormous reduction, especially if many instances of reusable test modules are used within one *Requirement* or between depending *Requirements*. Besides changing the coverage criteria, the test developer can also make use of the characteristics and the resultant flexibility of reusable test modules. In fact, each reusable test module within the TMR can be modified according to the present circumstances. For instance, the test developer could load the “SIP UAS non-INVITE” reusable test module, erase the state “Proceeding” within the behavioural description as well as the variables from the classification template (here:

“s_ResponseA1xx” and “s_ResponseB1xx”) and store the whole reusable test module under a new name, such as “SIP UAS non-INVITE without Proceeding”. This would minimise the behavioural description, but of course, maybe relevant test cases to verify the value-added service’s functionality will also be erased. However, for certain behaviour, such as for instance instant messaging with SIP MESSAGEs, this would make sense. In fact, although it is allowed according to (IETF RFC 3261, 2002) to send provisional responses on receipt of a SIP MESSAGE request, it is not very common and does not have to be specifically required for a value-added service.

Regarding the test case derivation, there are still some peculiarities that need to be mentioned. The first concerns possible conditions that are defined in the STD instance and are therefore part of the behavioural description. A condition will always compare some variable with a given value. If for example there is a condition that compares a text to a given value (for instance through *if(message.Text == “Login”)*) then within the STD instance, exactly this value will be specified in order to test that it works properly. However, also the “else” part of a condition needs to be verified. This is performed through the establishment of a new data set for the reusable test module instance. All variables that are belonging to the current test module instance will be copied and stored as another set of test data. However, the field *message.Text* will be automatically modified through some generated value. The establishment of a new data set does not affect the description within the STD instance. There is the rule that test cases for a given reusable test module instance need to be invoked for every defined data set that is stored in the *Test Data Pool* and that exactly belongs to the reusable test module instance.

A further peculiarity regarding the test case derivations concerns the concurrent behaviour. Here, it has been determined that the test cases for the reusable test module instances within an AND-state are fixed and will be invoked step by step for each concurrent behaviour specified through the reusable test module instances. A test case graph contains a special vertex to describe that the upcoming behaviour is concurrent.

Now that the abstract test cases have been derived, the TCDU can generate the abstract test suite that contains the collections sorted by the *Requirements*. In the following section, the *Test Suite Generator* as part of the TCF architecture transforms the abstract test cases into executable TTCN-3 test cases and creates a complete TTCN-3 test suite for the value-added service that is about to be tested.

7.2 Test Suite Generation

Before the automatic generation of executable TTCN-3 test cases is presented, a short introduction of the TTCN-3 technology is given in the following section together with the reasons why it has been selected in this research.

7.2.1 Motivation for a TTCN-3-based Approach

According to (Willcock *et al.*, 2011), the Testing and Test Control Notation Version 3 (TTCN-3) is an “internationally standardised language for defining test specifications for a wide range of computer and telecommunication systems. It allows the concise description of test behaviour by unambiguously defining the meaning of a test case pass or fail”. There are a lot of ETSI standards specifying TTCN-3, such as (ETSI ES 201 873-

3, 2015) describing the core language. Furthermore, there are ETSI standards for the existing presentational formats, either tabular-based (ETSI ES 201 873-2, 2007) or graphical-based (ETSI ES 201 873-3, 2007). Further important interfaces that are usually part of a TTCN-3 test system are the TTCN-3 Control Interface (TCI) (ETSI ES 201 873-6, 2015) and the TTCN-3 Runtime Interface (TRI) (ETSI ES 201 873-5, 2015). Based on the following Figure 7.9 illustrating the conceptual model of a TTCN-3 test system, the interfaces of such a system as well as the components are explained. Permission to reproduce Figure 7.9 has been granted by the publisher John Wiley and Sons.

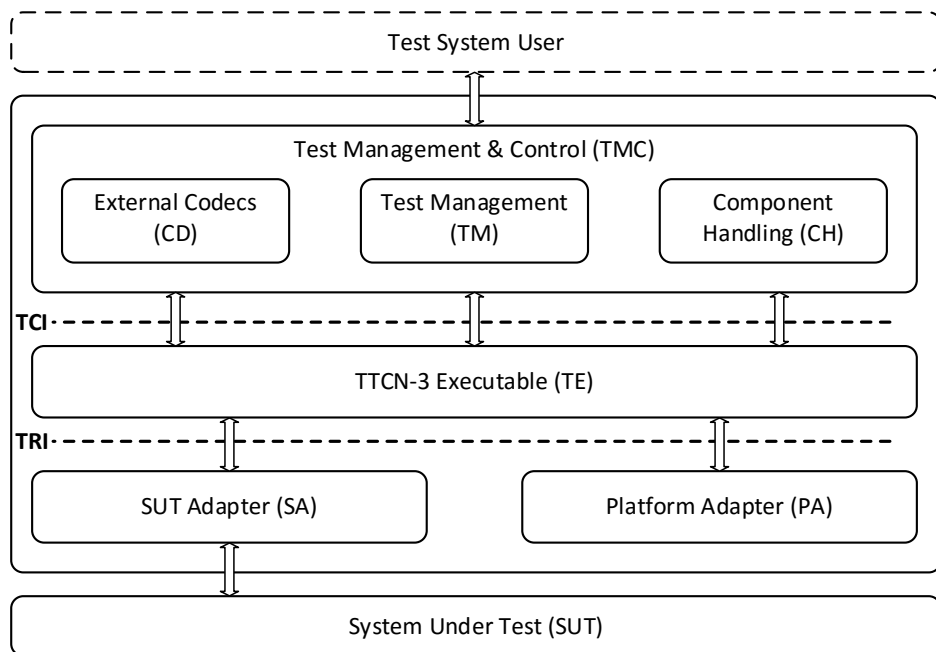


Figure 7.9: Conceptual model of a TTCN-3 test system (Willcock *et al.*, 2011)

(Willcock *et al.*, 2011) describes a TTCN-3 test system as a collection of entities that interact with each other while the test suite execution is performed. The central layer of the TTCN-3 test system, the TTCN-3 Executable (TE), deals with the execution of TTCN-3 statements. The TE itself depends on several services provided by the Test Management (TM), External Codecs (CD) and Component Handling (CH) entities that

are accessible via the standardised TCI. The entities are part of the Test Management & Control (TMC) layer. The TM is responsible for the overall management of the test system by providing a test system user interface to analyse the executed tests and to set relevant test parameters. The CD enables the encoding and decoding of data that is associated with message-based communication within the TE. Finally, the CH provides means in order to realise a communication between parallel test components (TTCN-3, 2015). Through the TRI, the TE is able to use services provided by the two adapters SUT Adapter (SA) and Platform Adapter (PA). The SA adapts message-based communication (or procedure-based alternatively) to and from the SUT whereas the PA is responsible for the TTCN-3 external functions and timers. Finally, on the top of Figure 7.9, the test system user is able to coordinate the testing through the TMC.

Based on the explanation of a TTCN-3 test system, the main arguments for using a TTCN-3-based specification of test cases are as follows:

- *TTCN-3: ETSI standard* – First of all, the language is a respected standard for the specification of tests by both academia and industry.
- *Hiding of the underlying complexity* – The TTCN-3 language allows to quite easily implement test sequences without complex steps. This has to do with the fact that the underlying complexity, for instance the memory allocation, network communication or the representation of data is hidden behind so-called abstract artefacts (e.g. test components, test behaviours, test templates). This aspect also allows a quite straightforward generation of TTCN-3 test code.
- *Programming language* – TTCN-3 itself is an abstract language that can only be executed by means of the SA and PA within a special TTCN-3 test system.

However, it contains constructs that are known from programming languages, such as variables and control structures (if-else and loops). These aspects are relevant in order to compare values and act accordingly.

- *Parallel Test Components* – This aspect of TTCN-3 is one of the most relevant factors for its usage in this research. A TTCN-3-based test system can create multiple test components to perform behaviours in parallel. Within an STD instance, several *Roles* can participate in a value-added service consumption through their *CI*s. These *Roles* or participating entities can be mapped to the parallel test components of the TTCN-3 test system. So, each *Role* defined in the STD is represented by one test component in the TTCN-3 test system and execution environment.
- *Concurrent behaviour* – Behaviours can be specified for the parallel test components. In TTCN-3, so-called behaviour functions are defined which can be bound to the test components. If the behaviour is then explicitly started for several test components, concurrent behaviour is possible. This aspect is a solution for the hierarchical AND-state and its representation of concurrency. For each *Role* that is addressed to a specific behaviour within the AND-state, the behaviour can be started.
- *Codecs* – The reusable test modules describe recurring behaviour that is specifying protocol messaging. The codec concept of TTCN-3 allows to enhance the test systems if for instance new reusable test modules are defined. Of course, the codecs have to be implemented once by a developer.

To sum up, TTCN-3 provides a lot of features that are required within this research and has therefore been chosen as notation of the executable test cases. The following section

describes how TTCN-3 test cases can be generated based on the derived abstract test cases.

7.2.2 Test Code Generation and Test Suite Building

The TCF architecture integrates the component Test Suite Generator (TSG) in order to build executable test suites for specified value-added services. The TSG itself comprises two components, a Test Code Generator (TCG) and a Test Suite Builder (TSB). The main tasks of these two elements are presented in the following Figure 7.10.

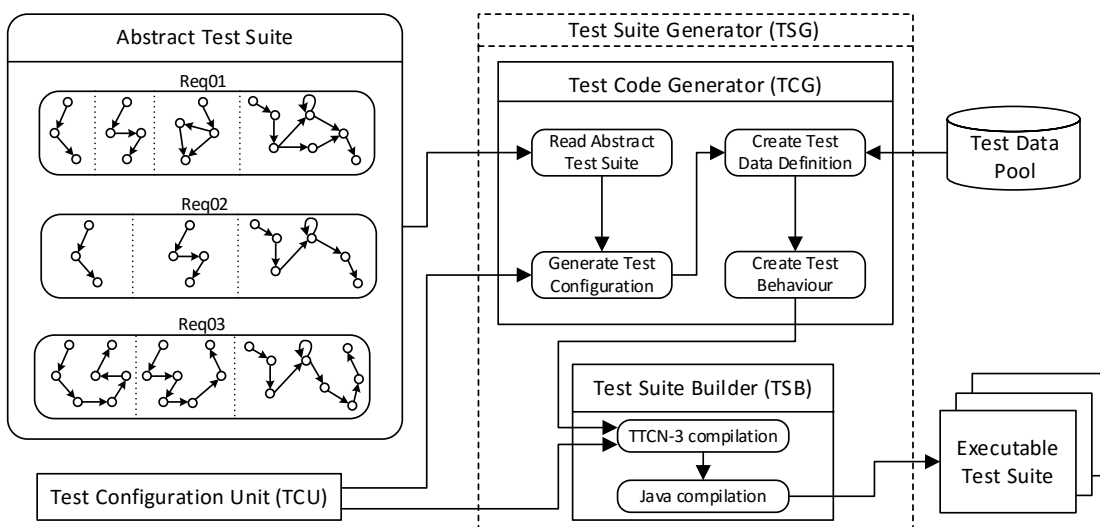


Figure 7.10: Generation of executable TTCN-3 test suite based on abstract test cases

The input of the executable test suite generation process is the output of the test derivation process of the last section (the abstract test suite containing graph-based test cases). First, the TCG will read the abstract test suite. Afterwards, it will generate a test configuration based on the parameters it retrieves from the Test Configuration Unit (TCU) and will then continue with the generation of test data definitions. This requires a connection to the Test Data Pool database, because all variables of instances of the reusable test modules

being consumed have to be generated as so-called TTCN-3 data templates. In the following, the test behaviour has to be created. For every requirement-based collection within the abstract test suite, a new TTCN-3 module is created which will contain the set of test cases that are generated from the graph-based test cases. Of course, this is an iterative process, because the test cases will be analysed sequentially and according to the inputs, the relevant TTCN-3 code is generated. The test behaviour creation process includes all particularities that are integrated within the graph-based tests, such as the sending of messages initiated by the test system, the subsequent receiving and evaluation of messages from the SUT, the handling of conditions and timers as well as the description of concurrency between the defined test components. The final step of the TCG is to deliver the generated TTCN-3 code to the TSB component within the Test Suite Generator. The TSB then compiles the code by means of a special TTCN-3 compiler which generates Java code. The subsequent Java compilation process generates executable Java bytecode which can be run within a Java VM (Java Virtual Machine). The Java bytecode is represented as the “Executable Test Suite” which now can automatically be executed within a TTCN-3 test execution environment. The described steps will now be analysed in detail.

Generate test configuration

The test configuration is the responsible part for the communication between the SUT on the one side and the test system (or test execution environment) on the other. However, the real physical connection is not directly supported via TTCN-3 but through appropriate SUT Adapters (see Figure 7.9). Instead, TTCN-3 provides well-defined abstract definitions of test system interfaces that shall be associated with the generated test cases.

Generally speaking, a complex test configuration can contain several test components that are able to communicate with each other and with the SUT.

As illustrated in Figure 7.10, the TCU provides the TCG with parameters from the STD instance, especially the *ServiceID* and the *Roles* are relevant. Whereas the *ServiceID* is just used for the further naming of test cases and files, the *Roles* have a very significant meaning. In fact, each *Role* specified in an STD instance defined for a given value-added service is represented by one so-called parallel test component (PTC) in TTCN-3. According to (Willcock *et al.*, 2011), a PTC is not a test component on which statements are executed, it is just a set of ports. A message-based TTCN-3 port defines which messages or message types are allowed to transfer through a specific port. In the following Figure 7.11, an example illustration of the test configuration is shown.

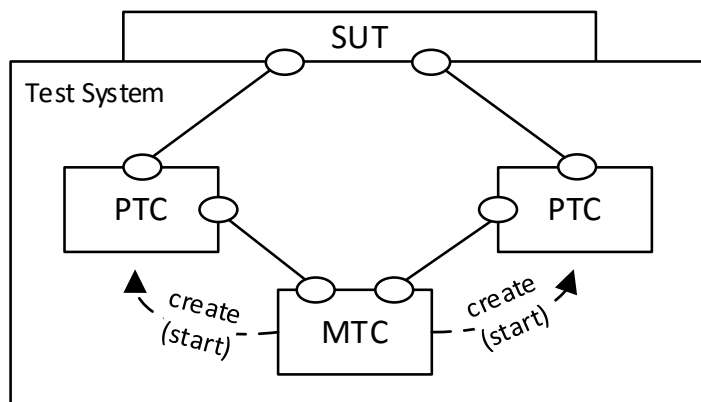


Figure 7.11: Dynamic test configuration with TTCN-3 test system

The test configuration shows two example PTCs that are directly connected to the SUT interfaces and are also connected to another element, the so-called MTC (Main Test Component). The major role of the MTC is to coordinate the creation and execution of the PTCs and usually, it does not interact with the SUT. It is also responsible for the logging of the verdict for every test step. The question is now how the illustrated

configuration can be generated by the TCG. However, before the MTC and the PTCs can be initialised, the kinds of interfaces of the test components and the SUT have to be determined. The following TTCN-3 statements (see Figure 7.12) specify the abstract SUT interfaces as well as the interfaces for the test components (PTCs).

```
group Interfaces {
    type component SUTInterface {
        port UdpPort UDP1, UDP2, UDP3;
        port TcpPort TCP1, TCP2, TCP3;
        port RTPPort RTP1, RTP2, RTP3;
        port HttpPort http;
    }
    type component SipComponent {
        timer globalTimer := 30.0;
        port SipPort SIPP;
        port RTPPort RTP1;
        port Coordination cpA;
        port Coordination cpB;
        port Coordination cbC;
    }
}
```

Figure 7.12: Abstract interface definition in TTCN-3 for SUT and test components

The component “SUTInterface” represents the SUT component containing abstract interfaces for the protocols UDP, TCP, RTP and HTTP. There is no specified SIP port defined, because SIP can be transported via diverse transport protocols (such as UDP and TCP). The “SUTInterface” as it is defined represents the interfaces provided by the SIP AS on which the value-added service is deployed. Additionally, the “SipComponent” refers to a test component that represents the *Role* “SIP phone”. In contrast to the “SUTInterface”, the test component has a timer “globalTimer” which is also defined within the classification templates of reusable test modules. Furthermore, the

SipComponent has a SIP port defined as well as an RTP port. Besides, the defined coordination ports are used for the coordination of the MTC and the synchronisation of the PTC.

In the following, the TTCN-3 syntax of the test configuration illustrated in Figure 7.11 is shown. An example test scenario could require the involvement of two *Roles*, both of the type “SIP phone”.

```
function createTestConfiguration(SipComponent mtcComp, SipComponent comp1,  
    SipComponent comp2, SUTInterface sut) {  
    map(comp1: SIPP, sut: UDP1);  
    map(comp2: SIPP, sut: UDP2);  
    connect(mtcComp:cpA, comp1: cpA);  
    connect(mtcComp:cpB, comp2: cpB);  
}
```

Figure 7.13: Example test configuration with two example PTCs

The illustrated TTCN-3 function could be generated by the TCG. The “map” function actually maps the ports between the PTCs and the SUTInterface whereas the “connect” function is only possible between MTC/PTCs in order to coordinate and synchronise. For every test case that is generated later on in the process, the test configuration function needs to be invoked.

Create Test Data Definition

One of the major benefits of TTCN-3 in comparison to other test specification methods is the ability to send and receive complex messages over the communication ports that have been defined by the test configuration. Besides predefined basic data types such as *charstrings* and *integers*, the syntax also provides a special language element called

template. According to (Willock *et al.*, 2011), TTCN-3 templates are used to either transmit a set of specific values (so-called send template) or to test whether received values are contained in a set of expected messages, which are again represented by a specification template (so-called receive template).

The task of the TCG in the test data definition process is to take all the relevant data sets (instantiated variables) from the Test Data Pool and to automatically generate either send or receive templates. Both send and receive templates are based on the same abstract data type. In TTCN-3, these abstract data types are called *records*, so there is no significant difference between the two. However, in contrast to send templates which have to contain explicit values, receive templates can either include explicit values or alternatively, wildcards (see section 6.3). In the Test Data Pool, send and receive templates can be distinguished through their names. A variable within a reusable test module instance containing a prefix “s” signifies that the SUT sends this message to the test system. So, as the test system actually receives this message, every “s” prefix message is a TTCN-3 receive template. Contrary to this, every “r” prefix message is a message that the SUT receives and accordingly a TTCN-3 send template.

In the following Figure 7.14, an excerpt of an example mapping is demonstrated between the XML-based structure of a variable within the Test Data Pool and the resultant TTCN-3 code. As an example, a SIP MESSAGE is specified that is sent by the SUT to the test system (prefix “s”). In the example, the request line as well as the text body of the SIP message are specified.

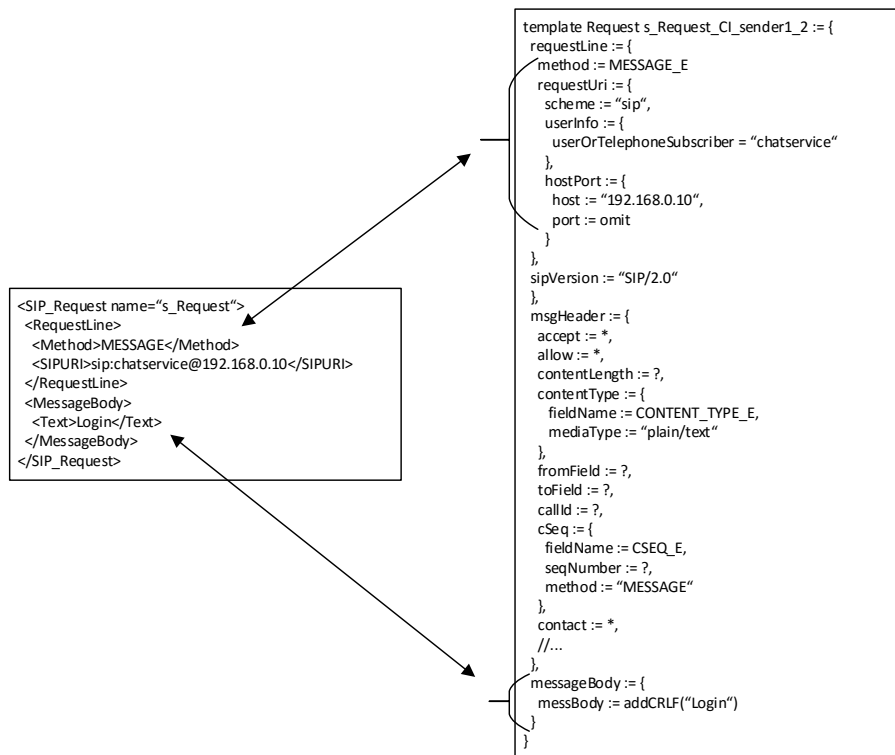


Figure 7.14: Mapping between XML representation of test data and resultant TTCN-3 template

The example shows how specific the definition of an example SIP request template is. On the one hand side, this has to do with the complexity of the protocol itself with all the possible headers that can be used. On the other side, the complexity in TTCN-3 templates is caused by the underlying TTCN-3 codec implementation. For this approach, the TTsuite-SIP is applied (TTsuite-SIP, 2015). Figure 7.14 also demonstrates the steps the TCG has to perform within the test data definition process. Every element of the XML-based structure has to be parsed and depending on the element name (e.g. “<RequestLine>”), the generation of a complete block of TTCN-3 code will be generated and integrated within a predefined TTCN-3 template. If a SIP URI is specified within the XML-based definition, the pieces of it will be splitted by “scheme”, “user info” and “host port”. This enables a precise analysis when a test case fails because of a wrong SIP URI. It is noted that the resultant TTCN-3 template includes message header definitions besides

the relevant fields request line and message body. This is required, because every template has to include all the fields of the *record* type it is based on. This does not mean that the fields have to be part of the real SIP protocol message. If they are not required at all, TTCN-3 provides the wildcard “*” for receive templates and the “omit” statement for send templates. As described in section 6.3, the wildcard “?” for receive templates is a little bit different from the “*” wildcard as it requires that at least a value is provided. In the example shown in Figure 7.14, this is valid for the mandatory headers of a SIP request (such as the “FROM” header). Of course, the example does not show all the headers included as it is just an excerpt.

As soon as the TCG has generated all the relevant TTCN-3 templates, they will be included in a separate TTCN-3 module. This module will then be integrated in the requirement-based modules which are generated in the following process, the creation of the test behaviour.

Create Test Behaviour

The test behaviour is a specification of what has to be tested by means of given inputs, results and conditions. The TTCN-3 syntax provides diverse constructs for describing the functionality of a test system and it also allows an efficient description of behaviour by means of sequences, alternatives and loops.

The TCG performs the test behaviour generation sequentially considering the requirements-based collections of abstract test cases. Then, each graph-based representation of an abstract test case is analysed. In the first step, all the edges are parsed

in order to find out which *Roles* are participating in the test case. Based on the result, the following initialisation of the test case can be done (see Figure 7.15):

```
testcase req01_tc1() runs on SipComponent system SUTInterface {
    var SipComponent v_sender := SipComponent.create alive;
    var SipComponent v_recipient := SipComponent.create alive;
    createTestConfiguration(mtc, v_sender, v_recipient, system);
}
```

Figure 7.15: Instantiation of test components in TTCN-3 test case

First, the TTCN-3 test case “req01_tc1” for the currently analysed abstract test case is established. The “runs on” clause signifies on which component type the described behaviour is to be executed and through the “system” clause, the SUT abstract interface specification is determined. In the following two lines, two test components of the type “SipComponent” are created and are then accessible through the variables “v_sender” and “v_recipient”. The “alive” statement used within the creation process signifies that the components can execute so-called behaviour functions more than once before they terminate. The concept behind the behaviour functions will be introduced later in this section. Regarding the defined test components, everything indicates that the parsing of edges of the current graph-based test case resulted in two *Roles* that are now represented as test components. In the final step of the initialisation process, the test configuration is established by invoking the function illustrated in Figure 7.13. The first parameter refers to the MTC where the behaviour is currently executed in.

In the next step, the graph-based abstract test cases are converted to test behaviour. The tree-like structure of a graph is advantageous because the concept of the “alt” statement in TTCN-3 is also a tree-based representation. So, the TCG algorithm has to traverse the tree and has to act according to the information stored on the edges (events, actions,

reusable test module instances and the “pass” flag) and the vertices (starting of timers). To exemplify the following steps, the code generation is shown by means of a graph-based test case in the following Figure 7.16.

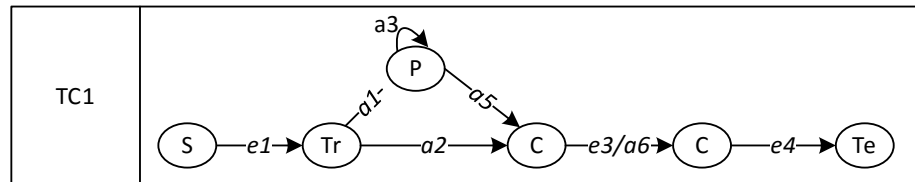


Figure 7.16: Example graph-based test case

As described in section 6.2.4, the behaviour models are focusing on the SUT. An event is referred to an actual event message that the SUT receives. For the test system, an event is a message that has to be sent and correspondingly, each defined action has to be received by the test system. This aspect has to be considered by the test code generation algorithm.

The illustrative graph-based test case example (see Figure 7.16) has been taken from Figure 7.8 and describes a SIP request being received by the SUT (standard “SIP UAS non-INVITE” behaviour). Then, the SUT either sends provisional messages back to the test system or an immediate terminating response. The test case also includes a possible retransmission of the initial SIP request after the terminating response has already been received by the test system.

The TTCN-3 code to specify this test case will be split into two parts. The first part specifies the test case from vertex “S” until the first vertex of “C”, the second part from the first vertex “C” to “Te”. The separation demonstrates how behaviour descriptions can

be modularised through so-called behaviour functions. Then, the algorithm also enables an efficient code generation.

```
1  function behaviour_tc1_1() runs on SipComponent {
2      globalTimer.start;
3      SIPP.send(r_Request_sender1_1);
4      alt {
5          [] SIPP.receive(s_ResponseAlxx_sender1_1) {
6              alt {
7                  [] SIPP.receive(s_ResponseBlxx_sender1_1) {
8                      repeat;
9                  }
10                 [] SIPP.receive(s_Response2xx_6xx_sender1_1) {
11                     globalTimer.stop;
12                     setverdict( pass );
13                 }
14                 [] SIPP.receive { setverdict ( inconc ); }
15                 [] globalTimer.timeout { setverdict ( fail ); }
16             }
17         }
18         [] SIPP.receive(s_Response2xx_6xx_sender1_1) {
19             globalTimer.stop;
20             setverdict( pass );
21         }
22         [] SIPP.receive { setverdict ( inconc ); }
23         [] globalTimer.timeout { setverdict ( fail ); }
24     }
25 }
```

Figure 7.17: First generated TTCN-3 behaviour function based on abstract test case

The behaviour function “behaviour_tc1_1” begins with the starting of the timer “globalTimer” which is part of the “SipComponent” test component. The timer is accessible because the behaviour code can only run on a “SipComponent”. Upon the timer has been started, the template “r_Request_sender1_1” representing the SIP request is sent

via the “SIPP” port of the “SipComponent”. As a consequence, alternative behaviour is specified through a so-called “alt” statement which enables to specify several different alternative behaviour that can take place at a given point. Here (see Figure 7.17, line 4), four different kinds of alternatives are defined. The first option is a valid one (see Figure 7.16, action “a1”, and correspondingly, see Figure 7.17, line 5), the receipt of the provisional SIP response “s_ResponseA1xx_sender1_1”. If there is a match, further SIP responses are received (see Figure 7.17, line 7 and line 10). If there are further provisional responses, the “repeat” statement (see Figure 7.17, line 8) determines that the current “alt” construct is still active. If a terminating response (for instance a successful “200 OK” response) is received (see Figure 7.17, line 10), the “globalTimer” will be stopped immediately, the test case will be considered as “pass” and the behaviour function is done. However, there are also two further alternative steps defined that always have to be integrated besides the explicit ones defined in the graph-based test case. Firstly, it is possible that the test system receives a message on the port “SIPP” that is not recognised as one of the specified messages (see Figure 7.17, line 14). In this case, the test case will be “inconclusive”. This verdict describes a situation where neither a pass nor a fail can be assigned. Secondly, the “globalTimer” can time out (see Figure 7.17, line 15), because the SUT does not respond to the initial SIP request at all. Accordingly, the test case fails. It is also possible that the SUT does not respond with a provisional message in the first place, but directly sends back a terminating response (see Figure 7.17, line 18). Of course, the test case passes, too. The other two alternatives (see Figure 7.17, line 22 and 23) again specify timeouts or wrong messaging.

As mentioned before, there is a second behaviour function required to process the complete test case shown in Figure 7.16. The trigger or reason for establishing a new

behaviour function is the junction of two or more paths into one identical vertex. Furthermore, the following outgoing edge of the vertex has to be an event and not an action. The first “C” vertex in the graph-based illustration is an example as it has two incoming edges “a5” and “a2” and an outgoing edge with an event “e3”.

```
1 function behaviour_tc1_2() runs on SipComponent {
2     globalTimer.start;
3     SIPP.send(r_Request_sender1_1);
4     alt {
5         [] SIPP.receive(s_Response2xx_6xx_sender1_1) {
6             timer timerJ := 0.0;
7             timerJ.start;
8             alt {
9                 [] timerJ.timeout (
10                    globalTimer.stop;
11                    setverdict ( pass );
12                }
13                [] SIPP.receive { setverdict ( inconc ); }
14                [] globalTimer.timeout { setverdict ( fail ); }
15            }
16        }
17        [] SIPP.receive { setverdict ( inconc ); }
18        [] globalTimer.timeout { setverdict ( fail ); }
19    }
20 }
```

Figure 7.18: Second generated TTCN-3 behaviour function based on abstract test case

The second behaviour function “behaviour_tc1_2” specifies exactly the behaviour that takes place if the verdict of the behaviour specified in “behaviour_tc1_1” passed. Consequently, the “globalTimer” was stopped which now has to be restarted again (see Figure 7.18, line 2). Then, the retransmission of the initial request is initialised (see Figure 7.18, line 3) and the retransmission of the terminating response is expected (see Figure

7.18, line 5). If that does not take place, then the usual two alternatives might occur (see Figure 7.18, lines 17 and 18). Otherwise, the timer “timerJ” is started and the test waits for the timer to time out. If that occurs, the complete test case passes.

Now, the specified behaviour functions have to be explicitly invoked by the test component. In the following Figure 7.19, the TTCN-3 test case “req01_tc1” (see Figure 7.15) is enriched with the test behaviour.

```
1  testcase req01_tc1() runs on SipComponent system SUTInterface {
2      var SipComponent v_sender := SipComponent.create alive;
3      var SipComponent v_recipient := SipComponent.create alive;
4      createTestConfiguration(mtc, v_sender, v_recipient, system);
5
6      v_sender.start(behaviour_tc1_1());
7      v_sender.done;
8      v_sender.start(behaviour_tc1_2());
9      v_sender.done;
10     all component.kill;
11 }
```

Figure 7.19: Starting of behaviour functions on test components

After the test case configuration, the first behaviour function (“behaviour_tc1_1”) is invoked by the test component “v_sender” with the statement “start” (see Figure 7.19, line 6). In the next line, the statement “done” is used on the same test component. This signifies that the test system waits until the behaviour invoked by the test component is terminated. As soon as this takes place, the second behaviour function (“behaviour_tc1_2”) can be invoked by the test component. After the behaviour for all test components has terminated, the statement “kill” has to be executed on the existing test components. This is relevant, because the test components have been created with the additional “alive” property.

The previously described example already illustrates how concurrent behaviours can be defined in TTCN-3. An example value-added service where this is relevant includes the functionality to setup a call between two participating entities. This would mean that the SUT (the value-added service) has to send two INVITE requests to the participating entities and has to handle the upcoming concurrent behaviour (“Three-Way-Handshake”). In TTCN-3, the code would look very similar to the example illustrated in Figure 7.19. However, the test case would not wait until the termination of a specified behaviour but directly invoke the two behaviours on the existing test components. A short excerpt of the TTCN-3 code is illustrated in the following Figure 7.20.

```
1 testcase req02_tc2() runs on SipComponent system SUTInterface {
2     //Test Configuration
3     v_recipient1.start(behaviour_tc2_1());
4     v_recipient2.start(behaviour_tc2_2());
5     all component.done;
6 }
```

Figure 7.20: Concurrency example with two test components

The example does not show the test configuration and the creation of the test component, but this does not differ much from the previous example (see Figure 7.19). The starting of the concurrent behaviour functions is performed by the two test components “v_recipient1” and “v_recipient2”. Then, the test waits until the concurrent behaviour is terminated. Therefore, the “done” statement is used for all existing test components.

A further aspect the TCG has to take into consideration is the occurrence of conditions. The following Figure 7.21 is a simplified illustration of a test case derived from the composed behaviour model in Figure 6.36.

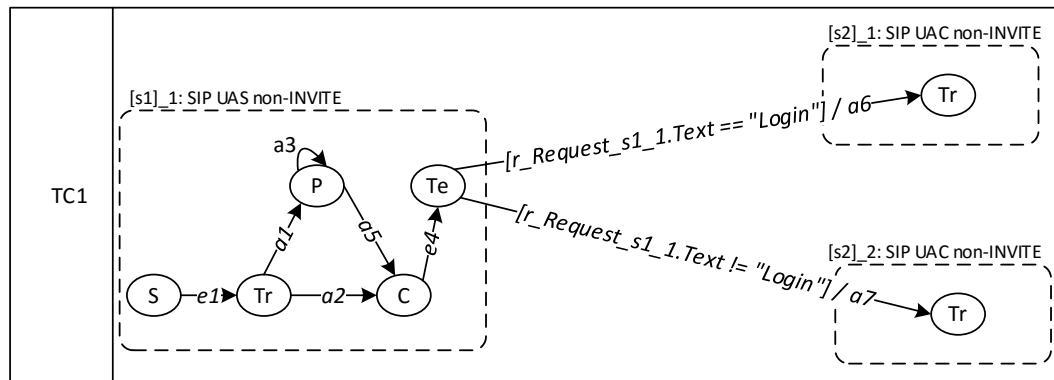


Figure 7.21: Example test case with conditions

The displayed test case excerpt involves one instance of the “SIP UAS non-INVITE” reusable test module and two instances of the “SIP UAC non-INVITE” reusable test module. The test case specifies a typical behaviour that might take place when some action should occur based on a specific test data value. Here, the value is the content of an instant message that has been sent from the test system to the SUT in order to “Login” into a specific service and consume further functions (such as the chat room service). In order to login successfully, the content of the message should be “Login”. All the message flows that take place within the behavioural part of the “SIP UAS non-INVITE” describe the receipt of the initial instant message on the part of the SUT until the transaction is terminated in the “Te” vertex. From here, there are two edges which lead to vertices of different “SIP UAC non-INVITE” behavioural descriptions. Both edges contain conditions which check whether the value of the “Text” attribute (see Figure 6.21) of the initial instant message “r_Request” contains the text “Login” or some different value. Therefore, two alternative edges are included here. This branching is valid, because the edges only contain actions and no events. Independent of the content of the “Text” attribute of “r_Request”, a response from the SUT is expected based on the initial instant

message. In fact, it will send an instant message which informs the user (or the test system in this case) whether the “Login” process was successful or not.

Based on the graph-based test case excerpt illustrated in Figure 7.21, the corresponding TTCN-3 code has to be generated by the TCG. Basically, the TTCN-3 code generated from “SIP UAS non-INVITE” has already been shown in Figure 7.17 and Figure 7.18. So, the following Figure 7.22 includes a behaviour function that it started as soon as the “SIP UAS non-INVITE” behaviour is terminated in the vertex “Te” (see Figure 7.21).

```
1 function behaviour_tc1_3() runs on SipComponent {
2     globalTimer.start;
3     var Request v_r_Request_s1_1 := valueof(r_Request_s1_1);
4     alt {
5         [v_r_Request_s1_1.messageBody.messBody == "Login"]
6             SIPP.receive(s_Request_s2_1) {
7                 //...
8             }
9         [v_r_Request_s1_1.messageBody.messBody != "Login"]
10            SIPP.receive(s_Request_s2_2) {
11                //...
12            }
13        [] SIPP.receive { setverdict ( inconc ) }
14        [] globalTimer.timeout { setverdict ( fail ) }
15    }
16 }
```

Figure 7.22: Example of conditions within generated TTCN-3 code

The behaviour function “behaviour_tc1_3” initially restarts the “globalTimer” in order to verify that the test case fails if no event occurs after a given amount of time. Then, a temporary variable “v_r_Request_s1_1” is initialised based on the initial instant message which is contained in the TTCN-3 template “r_Request_s1_1” (see Figure 7.22, line 3).

The “valueof” operation used here allows the value specified within a template to be assigned to the fields of a variable. This step is always included in the generated TTCN-3 code as soon as values within templates have to be accessed. The name of the variable is always identical to the corresponding template’s name including a “v_” prefix. In the test case illustrated in Figure 7.22, the variable is needed, because it is relevant for the conditions. After the variable is created, alternative behaviour is specified within the test case through the “alt” statement (see Figure 7.22, line 4). The first two alternative steps within the “alt” statement refer to the two possible conditions (see Figure 7.22, lines 5 and 9), whereas the other two are the common alternative steps that are always included in “alt” statements (see Figure 7.22, lines 13 and 14). Both conditions are included within the brackets “[]” symbolising alternative steps and within each of them, the field “messBody” of the Request attribute “messageBody” is accessed and compared to either the value “Login” or not “Login”. This field refers to the “Text” attribute that is used in the graph-based test case description. The difference in the syntax has to do with the mapping concept between the XML representation of test data within the *Test Data Pool* and the resultant TTCN-3 code (see Figure 7.14). The main reason behind this is to simplify the definition of test data for the test developer without losing the possibility to check any given field within a SIP message. The XML representation is a more abstract representation of the underlying TTCN-3 templates that specify SIP requests and responses. To take up the issue of conditions again, the process is as follows. For each alternative step that includes a condition, the behaviour specified within it can only take place if the condition is true and if the test system verifies that the incoming message and the defined template match. In the case of the first conditions (“==”), the template

matching will be done with the SIP request template “s_Request_s2_1” (see Figure 7.22, line 6).

Test Suite Builder

After the abstract test cases have been translated into TTCN-3 test cases by the TCG, the Test Suite Builder (TSB) as part of the Test Suite Generator (TSG) builds the “Executable Test Suite” (ETS). This process is illustrated in the following Figure 7.23.

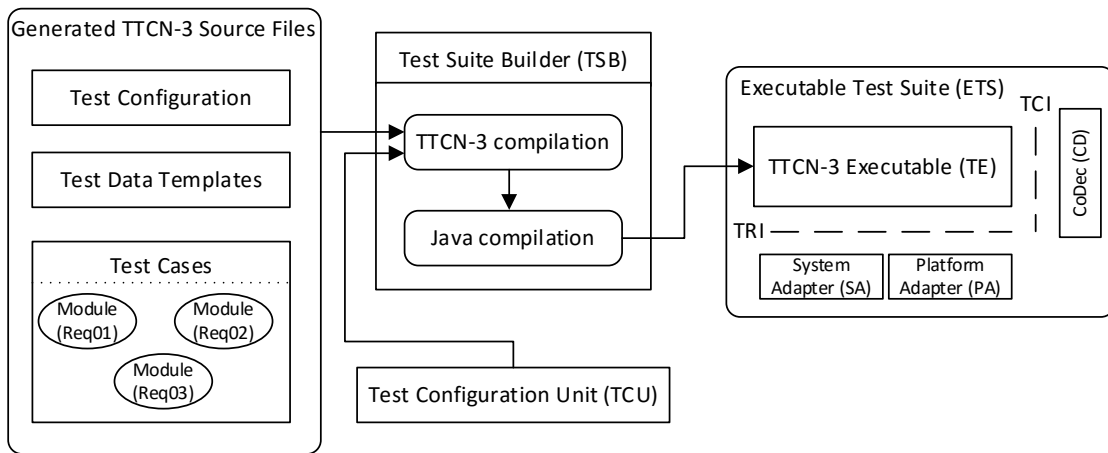


Figure 7.23: Generation of Executable Test Suite by Test Suite Builder

The input of the TSB is the collection of TTCN-3 files, such as the “Test Configuration” (see Figure 7.12), the generated TTCN-3 test data templates as well as the generated TTCN-3 test cases. As shown in Figure 7.23, the test cases are separately included within TTCN-3 test modules. For every specified *Requirement* in the STD, a separate TTCN-3 module exists which includes all TTCN-3 test cases that are required to verify that the *Requirement* is fulfilled by the value-added service.

As soon as the TSB receives the collection of TTCN-3 files, it invokes the TTCN-3 compilation process. Here, a specific TTCN-3 compiler reads the module definitions of the TTCN-3 files and compiles them into Java-based sources. Most commercial TTCN-

3 execution environments also include TTCN-3 compilers which enable the compilation into other programming languages (e.g. C, C++ and C#), such as Elvior (Elvior, 2015) and OpenTTCN (OpenTTCN, 2015). In this research, the TTworkbench (TTworkbench, 2015) has been applied which also includes a TTCN-3 compiler called “TTthree”. Figure 7.23 also shows a second input into the TTCN-3 compilation process from the Test Configuration Unit (TCU). In principle, this input is a so-called test adapter configuration file (“taconfig” file), an XML-based document the TTCN-3 compiler has to know during the compilation process. The “taconfig” file is generated by the TCU based on the information it holds from the STD instance, such as the SUT addressability and the information about the participating test components. Within the file, the TCU specifies the required TTCN-3 Codecs (CD) as well as the real ports that are used to communicate with the SUT. The following Figure 7.24 shows a simplified excerpt of an example “taconfig” file.


```
<testadapter>
  <codec encode="SipNist">
    <plugin id="com.testingtech.ttworkbench.tt3rt.sip.codec.SipCodecPlugin">
      <parameter id="class"
        value="com.testingtech.ttworkbench.tt3rt.sip.codec.SipCodecProvider"/>
    </plugin>
  </codec>
  <port>
    <plugin id="com.testingtech.ttcn.tri.udp.UDPPortPlugin">
      <parameter id="class" value="com.testingtech.ttcn.tri.UDPPortProvider"/>
    </plugin>
    <parameter id="UDP1">
      <parameter id="UDP_LOCAL_PORT" value="{PX_ETS_PORT}"/>
      <parameter id="UDP_LOCAL_ADDRESS" value="{PX_ETS_IPADDR}"/>
      <parameter id="UDP_REMOTE_PORT" value="{PX_IUT_PORT}"/>
      <parameter id="UDP_REMOTE_ADDRESS" value="{PX_IUT_IPADDR}"/>
    </parameter>
  </port>
</testadapter>
```

Figure 7.24: Excerpt of test adapter configuration file for compilation process

The `<testadapter>` element is the root element of the “taconfig” file and specifies all the required CDs and ports that are required within the execution process of the ETS. The `<codec>` element comprises the relevant information of an existing CD that can be applied within the test execution environment (here, a CD has been chosen which is part of the TTworkbench). For each determined CD, a so-called provider has to be determined, a Java class which handles the CD processing. In Figure 7.24, the selected CD is a codec for the SIP protocol. Furthermore, the “taconfig” file includes the specification of the communication endpoints of the ETS. Here, a `<parameter>` with the *id* “UDP1” is specified, which contains four specific variables, the IP address and port number of the test component running within the test execution environment as well as the IP address

and the port number of the SUT. The example in Figure 7.24 just includes placeholders (such as “\${PX_IUT_PORT}”). A valid “taconfig” file contains real values, such as 5060 as the SIP standard port number. With the help of the port specification within the “taconfig” file, the loading of the appropriate System Adapters (SAs) and CDs are realised. Now, the communication between the test components of the test system and the SUT can be established.

As discussed before, the TTCN-3 compilation process generates Java-based sources. Additionally, a so-called “Campaign Loader File” (CLF) is generated. The CLF file contains the test adapter configuration (“taconfig” file) as well as a list of all test cases within the ETS. The CLF file is also based on XML and is therefore machine-readable.

In the second step, the TSB (see Figure 7.23) compiles the Java sources into byte code class files. Furthermore, they are combined into a Java Archive (JAR). The JAR actually represents the TTCN-3 Executable (TE) within the ETS. Through the CLF file as part of the ETS, the services of the relevant SA and the Platform Adapter (PA) are activated and can be used by the TE through the TRI (TTCN-3 Runtime Interface). Furthermore, the CD is accessible by the TCI (TTCN-3 Control Interface).

Now, the ETS including all the relevant test cases has been generated in order to verify the functionality of a value-added service. In the following section, the principles of the test case execution within TTCN-3 based test execution environments is discussed.

7.3 Test Case Execution

As introduced in the previous section, the part of the TTCN-3 conceptual model or rather system architecture (see Figure 7.9) which executes the TTCN-3 test cases is the TE entity. However, according to (Willcock *et al.*, 2011), not all relevant functions for the test case execution are integrated within the executable Java bytecode produced by the TTCN-3 and Java compilers. Some functions deal with aspects that cannot be extracted from information included within the TTCN-3-based tests. An example for such a function is the mapping of the “send” statement. Of course, the TE does not include any information on how to send data to the SUT. To achieve this, the TE needs to call an operation which is provided by the SA through the TRI. In general, the following functionality is not included in the TE but is supported through the entities running within a TTCN-3 test system:

- The communication with the SUT is provided by the System Adapter (SA) through the TRI.
- The timer functionality is provided by the Platform Adapter (PA) through the TRI.
- The data encoding functionality is provided by the External Codecs (CD) through the TCI.

The following Figure 7.25 is a modified illustration taken from (Willcock *et al.*, 2011) and exemplifies the execution of a test case which involves all the relevant entities of a TTCN-3 test system and the SUT.

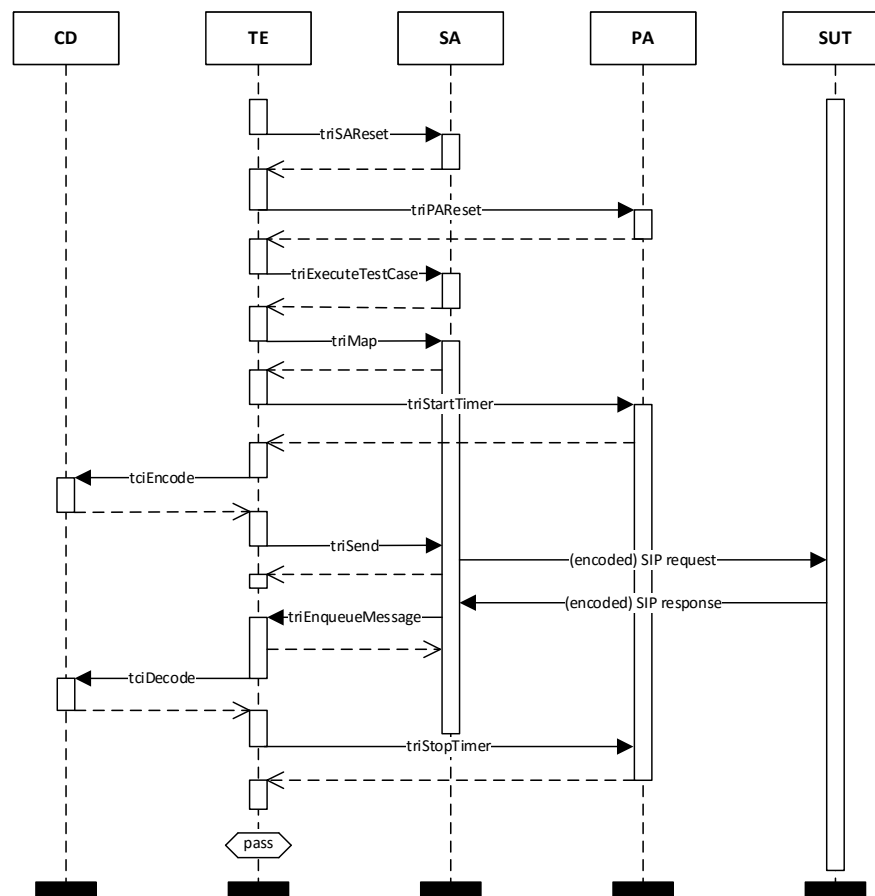


Figure 7.25: Interaction of test system entities during test case execution

The displayed test case execution performs the sending of a SIP request (for instance a SIP MESSAGE) to the SUT and subsequently expects a SIP response from the SUT. The process starts with the TE invoking the “triSAReset” and the “triPAReset” operations which are provided by the SA and the PA. According to (ETSI ES 201 873-5, 2015) and (Willcock *et al.*, 2011), the “triSAReset” operation resets all communications means the SA is currently maintaining, such as static connections to the SUT. Dynamic connections to the SUT are closed and pending messages are discarded. If the operation has been performed successfully, it returns a status which indicates the local success (e.g. “TRI_OK”) or failure (“TRI_Error”) of the operation. This status is sent by all the upcoming operations that are related to the TRI. The second operation, “triPAReset”,

concerns the PA. Here, all timing activities the PA is currently performing shall be resetted. A typical example is the stopping of all running timers.

As soon as both SA and PA are resetted, the TE calls the “triExecuteTestCase” operation on the SA immediately before the execution of a test case. The operation includes two further parameters, the test case name and a list of ports that have been declared in the definition of the system component for the test case (see the system component “SUTInterface” in Figure 7.12).

In the next step, the “triMap” operation (see Figure 7.25) is called by the TE upon executing a “map” statement in a TTCN-3 test suite (an example is illustrated in Figure 7.13). According to the TRI standard (ETSI ES 201 873-5, 2015), the operation is used to prepare a SUT communication interface (also defined as test system interface port) for the interaction with the SUT. A successful completion of the “triMap” operation enables a test component within a test case to communicate with the SUT. For a test case that includes SIP communication, the invocation of the “triMap” operation could trigger the allocation of a UDP socket (or alternatively, a TCP socket) and port through which SIP messages can be sent and received. Although an unmapping is not included in Figure 7.25, there is also an operation “triUnmap” defined in the TRI standard which can be invoked immediately after the termination of a test case. The main task of the operation is to close a dynamic connection to the SUT for a specific test system interface port.

The next operation invoked by the TE, “triStartTimer”, is implemented by the PA (see Figure 7.25). Of course, its invocation depends on the definition of the current test case. If a timer is started by means of a TTCN-3 statement, the operation is invoked. The call itself specifies the duration of the timer and includes an identifier for the timer in future

communication between the TE and PA. Although it is not included within Figure 7.25, it is possible that the timer expires before it is explicitly stopped again. The PA indicates a timeout by calling the “triTimeout” operation (Willcock *et al.*, 2011). As soon as the “triStartTimer” operation has been invoked and the timer has been started successfully, the execution of the test case continues with the sending of a SIP request message.

The sending of a message on the part of the test system requires to first encode it into a message the SUT accepts. Encoding as well as decoding services are provided by the CD entity which can be accessed by the TE via the TCI. The operation “tciEncode” (see Figure 7.25) encodes a requested TTCN-3 message value and subsequently passes it back to the TE as a binary string. According to (ETSI ES 201 873-5, 2015), the binary string is then one of the input parameters of the following “triSend” (see Figure 7.25) operation. Another parameter “componentId” identifies the test component that is actually sending the message and a further parameter “tsiPortId” specifies the test system interface port via which the message is sent. As soon as the operation has been invoked, it is the task of the SA to transmit the message to the SUT.

If the SUT accepts the SIP request, it answers back with a SIP response to the UDP port (or TCP port) from where the corresponding message originated. The SIP response is received by the SA which forwards it to the responsible test component within the TE by invoking the “triEnqueueMsg” (see Figure 7.25) operation. The message is passed in an encoded form. According to (Willcock *et al.*, 2011), the arrival of any incoming message (here, the SIP response) triggers a new evaluation of the “alt” statement which contains different alternatives to deal with the different possible reactions from the SUT. The “alt” statement blocks until one of its determined alternatives matches. However, the matching

process requires the encoded message to be decoded into a structured TTCN-3 value. The CD entity provides the decoding service which is implemented in the “tciDecode” operation. Besides the encoded message, the “tciDecode” operation needs to know the assumed type of the message, the so-called decoding hypothesis. The CD can then select the appropriate decoding mechanism. In the case of the SIP response, the decoder would check if the received message is a correctly encoded SIP response message. A successful check automatically creates a TTCN-3 “SIP_Response” value of the message and returns it to the TE.

If the received SIP response matches, the execution of the test case proceeds with the explicit stopping of the timer. Here, the TE invokes the “triStopTimer” operation implemented by the PA. This operation succeeds, even if the timer has already stopped or timed-out. It allows the PA to discard the timer.

As soon as the timer has stopped, finally, a test case verdict has to be determined. In the example test case execution illustrated in Figure 7.25, the test case is judged as “pass”, because the message flow (including the checking of the test data) has taken place as specified in the TTCN-3 test case definition.

For each generated test case within an ETS which has been built by the TSB, the described interaction of test system entities within the test execution environment, the TTworkbench, is performed. The execution results of the test cases are documented in an incident report which includes any event that occurs during the testing process that requires further investigation. Then, the stakeholders have to figure out where the issue originates. The following section discusses these issues and introduces methods supported by the TCF to simplify the test evaluation process.

7.4 Test Evaluation and Management

In the chapters 5 and 6 as well as in the previous sections of chapter 7, the concept of the TCF and its components have been introduced thoroughly. The integration of the proposed TCF within a service provider environment would change the tasks of the roles participating in the service development and testing process. Especially test developers benefit from the proposed TCF as they are also involved in the requirements elicitation process due to their participation in the so-called Service Quality Group (SQG). The SQG is a novel concept developed in this research and has been introduced in section 4.2. Besides test developers, also service analysts and service developers participate in the SQG. The SQG has been established to build a foundation for successful functional test integration and to deliver products (value-added services) to service customers that have been verified and validated. Most of the related work (such as related frameworks for functional testing, see section 3.3) that has been done in the field of automated functional testing of services focus only on developing efficient methods to build test models from which test cases can be derived. The emerging agile concepts are not considered in these approaches. In order to achieve a valid product, this research considers the agile concept through the establishment of the SQG, which enables a “Whole Team” approach in the methodology. In agile principles, the “Whole Team” approach (also called team-based approach) describes a style in project management in which all project members are equally responsible for the quality and success of a project (Gregory and Crispin, 2015). The benefits of the SQG supporting the “Whole Team” approach are as follows:

- It helps the team in building a strong working relationship through effective cooperation, communication and teamwork.

- It enables the team members to learn from each other and share knowledge.
- Every member of the SQG is responsible for the outcome.

Especially the enforcing of service developers and test developers to intensify their collaboration is a very important aspect of the SQG. In this section, the statements regarding the SQG will be clarified. As discussed in section 4.2, both development (for instance by means of a Service Creation Environment (SCE)) and testing (by means of the proposed TCF) can start as soon as the service analyst hands out the “Structured Requirements” document which contains the informal textual use cases specifying the functionality of the value-added service. Both developers can rely on the described use cases within the “Structured Requirements”. The service developer can implement the service logic, possibly by means of reusable building blocks that are integrated within the applied SCE, whereas the test developer can define an instance of the Service Test Description by applying the TCF accordingly. It is important to emphasise once again that every use case specified in the “Structured Requirements” document can be mapped to a *Requirement* defined within an STD instance. This aspect allows an iterative testing approach because a generated Executable Test Suite (ETS) does not have to contain all test cases for a given value-added service. It is also possible to successively enhance the STD instance and allow test iterations. The first test iteration might include just the initial *Requirement* (e.g. “Req01”), whereas the last test iteration includes all the *Requirements* specified within the final STD instance. The aspect of establishing test iterations has many advantages. First, the test development process and the service development process can be synchronised. If a service developer implemented the service partly so that it fulfils the initial use case specified in the “Structured Requirements” document, it can be automatically tested by means of the first test iteration. Further iterations can be

established so that throughout the duration of the value-added service development, tested so-called prototypes shall be demonstrably even if they just fulfil a range of use cases. In principle, this approach can be called “rapid prototyping”. A second advantage of the approach refers to the collaboration with the service customer. As soon as a prototype exists as well as the corresponding ETS for this stage of development (test iteration), the test cases will be executed against the prototype. If all test cases within the ETS pass, the prototype can be declared as a verified prototype. In order to validate the prototype, the service customer can be involved. So, each successfully tested (all test cases passed) prototype can be demonstrated to the service customer. If the prototype meets the requirements of the service customer, the prototype can be declared as a validated prototype. If it does not meet the requirements of the service customer, possibly the “Structured Requirements” document is not complete or includes misunderstandings or errors. It is possible that the service analyst made a mistake while creating the document or the service customer did not have a clear vision of the value-added service at the beginning. In both cases, the “Structured Requirements” document needs to be updated as well as the corresponding steps within the test and service development. The whole process continues until the prototype fulfils all use cases specified in the “Structured Requirements” document. This prototype is then the final value-added service.

The previously described process assumed that all test cases within an ETS passed during their execution against a prototype. However, test cases can also fail. Independent of the underlying category of occurred error, such as “timer expired” or “other message received than expected”, the following reasons can be stated:

1. While defining the *Requirements* within the STD instance, the test developer made a syntactical error (e.g. error in the parameterisation of variables).
2. The test developer did not understand the description of a use case properly (maybe because it has not been described precisely enough) and defined a different corresponding *Requirement* in the STD instance.
3. The service developer did not understand the description of a use case properly (maybe because it has not been described precisely enough) and implemented a different service logic.
4. The service developer made mistakes during the implementation of the service logic.

Besides the mentioned reasons, there are of course alternative flaws that might occur. The handling of errors in the test execution is always the same. First, the test developer has to analyse the test cases that failed. If he made mistakes on his own (see reason 1), an experienced test developer will find them quite fast and will be able to fix them. Otherwise, for the reasons 2, 3 and 4, the test developer needs to first get in contact with the service developer to discuss the issues. If both cannot fix the issue or have different understandings of the matter, the service analyst is consulted and the SQG meets officially. Generally, the service analyst should be able to solve the issues by clarifying the possible misunderstandings or ambiguities in the “Structured Requirements” document. However, it might also be necessary to contact the service customer for further clarification.

In principle, the SQG requires a separate framework to control and monitor the testing and development process and to simplify the communication with one another. Each role

within the group and also the service customer could get access to a personalised graphical user interface (GUI). The test developer and the service developer get an overview of their current projects. For each project, they get information regarding the included use cases and whether there are already existing prototypes for the use cases. Furthermore, the developers can see the status of the prototype (“verified”, “validated” or “final”). Over the GUI, the developers can get in contact with each other (e.g. via instant chat message, audio/video calls or even via audio/video conferences) and are able to request for a meeting of the SQG. The service analyst can also retrieve information regarding the projects he is currently participating in. He sees the status of the projects and is able to arrange SQG meetings with the service and test developers. If a prototype is declared as verified, the service analyst can personally contact the service customer. Finally, the service customer gets informed as soon as a prototype has been verified. The next step for the service customer would be to test the verified prototype in order to validate it.

The proposal regarding a separate framework for the SQG and the collaboration with the service customer is not within the scope of this research, but it can be analysed for further research.

7.5 Conclusion

This chapter has introduced the test case generation based on behaviour models. Furthermore, it has dealt with the execution of the generated test cases and the subsequent evaluation of test results.

First, an appropriate algorithm had to be found to derive test cases from the generated behaviour models. Although the finding of traces within the Statechart-based behaviour models seemed to be the only choice, linear sequences of events and actions have not been identified to suite well for testing of value-added services. Therefore, a graph-based representation of derived test cases has been chosen which fits best to the structure and properties of the bahaviour models. Furthermore, several structural coverage criteria have been discussed and analysed. In order to reduce the number of generated test cases and taking literature into consideration, the structural coverage criteria *All-Round-Trips* has been selected. Finally, the properties of the graph-based test cases have been introduced and examples have been discussed.

The generation of TTCN-3 test cases from the graph-based test cases has been described in section 7.2. First, the reason for selecting the TTCN-3 language have been discussed. The result has shown that the technology is a respected ETSI standard, supports concurrency, is similar to programming languages and provides a lot of further features that are required in this research. In the following, the test code generation of TTCN-3 test cases has been introduced, mainly the generation of the test configuration, the test data definitions as well as the test behaviour. Finally, the building process of an Executable Test Suite (ETS) has been described by means of the Test Suite Builder (TSB).

The next section 7.3 has illustrated the execution of test cases within a TTCN-3 test system and has emphasised the relevance of the several entities.

The evaluation and handling of test results has been discussed in section 7.4. Here, the relevance of the Service Quality Group (SQG) in order to enable the validation of a value-

added service has been identified. Finally, the section gives guidelines for the test developer how to deal with the test results. If all test cases of an ETS pass, the prototype can be declared as verified whereas a fail requires further analysis.

The concept of the TCF has been introduced completely in the chapters 5, 6 and 7. The upcoming chapter 8 investigates whether the requirements that have been established in section 3.4 can be met by the described solution. Furthermore, the prototypical implementation of the TCF is described as well as the evaluation of the prototype by means of an example value-added service.

8 Framework and Prototype Evaluation

This chapter deals with the prototype implementation as proof of concept evaluation to demonstrate that the proposed framework for automated functional testing of value-added services meets the requirements that have been derived from the deficits and assets of related projects. In section 8.1, each of the defined requirements is analysed and it is explained how it is fulfilled by the proposed framework. The upcoming section 8.2 depicts the architecture of the prototype implementation of the framework, its utilised components and their functionality. Section 8.3 discusses the use of the prototype implementation by means of an example value-added service in order to evaluate the application of the prototype and framework.

8.1 Evaluation of the Defined Framework Requirements

This section evaluates the proposed TCF with regard to the derived requirements listed in section 3.4. Each requirements is evaluated in the following regarding its fulfilment within the proposed framework (TCF).

- *Test Execution and Test Report* – The ability to execute tests is provided by the framework through the connection to an external TTCN-3-based test execution environment (in the case of the provided prototype, a connection to the TTworkbench was established, an example TTCN-3 test execution environment). After a test suite for a given value-added service is generated by the Test Code

Generator and built by the Test Suite Builder, the suite can be executed against the SUT (see section 7.3). Fortunately, the TTCN-3-based test execution environment already provides a test report for test executions that have been performed.

- *Collaboration and support for agile principles* – This agile aspect of the proposed framework and methodology is supported through the initiated Service Quality Group (SQG), which involves the service customer, the service developer and test developer as well as a new role, the service analyst. The service analyst realises the coordination and acts as a mediator between the developers on the one side and the service customer on the other side. A thorough methodology of the tasks that can be initiated by the diverse roles within the SQG is defined in chapter 4 and some further information is given in section 7.4. Here, some agile principles are highlighted such as the support for rapid prototyping.
- *Comprehension* – This aspect depends on the collaboration requirement. The major concern of the comprehension requirement is the strict involvement of the service customer within the process and that he is always able to see the current progress of the project. The collaboration web site concept that has been mentioned in section 7.4 is an example solution to let the service customer participate in the development and test process.
- *Manageability and time exposure* – This requirement refers directly to the test developer who is actually applying the functionality of the proposed TCF. Manageability and time exposure refers to the difficulty level of the application on the one hand and to the time that is required to achieve the goals. Regarding

the proposed framework, the test developer gets a straightforward service and test specification language (STD) in order to create individual instances for the value-added services that have to be tested (see section 5.2). In comparison to related projects that involve the manual modelling of formal behaviour models based on EFSMs, a lot of time can be saved. Besides the straightforward foundation of the STD, also the concept of the reusable test modules and their instances fastens the process, because the behaviour only has to be specified once (see section 6.2).

- *Tool support* – A prototype implementation of the major components of the proposed TCF is described in section 8.2. The test developer is able to create instances of the STD on a web page and can then trigger the whole automated process which will start from the automated building of the behaviour models, will then derive and subsequently generate the test cases (an ETS) and finally, the test execution against the SUT is performed. As a feedback, of course, the test developer will get a test report.
- *Traceability of requirements* – Requirements are playing a major part within the proposed TCF and within the whole process, of course. The initial and informal requirements are initially specified by the service customer in collaboration with the service analyst. The result will be the “Structured Requirements” document, which can be, for instance, a standardised UML use case specification. The next trace of the requirements is performed by the test developer who creates an STD instance containing *Requirements*. The approach intends to have a direct mapping between the informal requirements specified in the “Structure Requirements” document and the *Requirements* specified in the STD instance. The next trace of requirements occurs in the building of behaviour models. For each *Requirement*

specified in the STD instance, a behaviour model is built. The requirements-based behaviour models are the input of the TCDU which generates an abstract test suite that contains a sorted list of abstract test cases. Of course, every generated abstract test case is assigned to a specific requirement. The Test Code Generator considers the abstract test cases belonging to a requirement and includes all generated TTCN-3 test cases based on the abstract test cases within one TTCN-3 module. If the “Structured Requirements” document contains five different requirements (in UML notation, the requirements are called use cases), the resulting TTCN-3 test suite will also contain five TTCN-3 modules for each specified requirement. The test execution can then be differentiated by means of the TTCN-3 control part.

- *Reusability* – This requirement is obviously fulfilled by the proposed TCF through the reusable test modules (see section 6.2).
- *NGN-compliance or support for general SIP-based IP networks* – The proposed TCF has been initially developed for the purpose of testing NGN-based value-added services. In fact, the framework is intended to be integrated into a service provider test environment and can be seen as a counterpart to the Service Creation Environments for service development. The prototype validation described in section 8.3 illustrates an example value-added service that requires the existence of typical components from a SIP-based NGN, such as a SIP AS.
- *Verification and Validation* – Verification is supported, because the test cases are directly derived from the requirements specification and are traceable throughout the whole process. In contrast, validation requires especially the involvement of the service customer. He needs to confirm that the value-added service meets his requirements. Through the establishment of the SQG, validation is supported.

- *Effectivity and efficiency of test generation* – The amount of test cases to be generated depends either on the selected coverage criteria (for instance *All-Transitions* or *All-Round-Trips*) or on the construction of a reusable test module itself. Regarding both aspects, the test developer is given flexibility by the test framework to achieve efficiency in the test case generation (see section 7.1.2). As the test cases are based on standard protocol behaviour that can be applied by any given value-added service, they have to be highly effective.
- *Expandability* – The proposed TCF supports further enhancements, for instance by defining new reusable test modules. New protocols can be specified and added to the TMR, however, this also requires an enhancement of the test execution environment. In order to exchange messages of a given protocol, the TTCN-3-based test execution environment needs implemented codecs.

Besides the mentioned requirements, the framework also supports a rapid prototyping-alike approach. Theoretically, the service developer and test developer could manage to implement a value-added service iteratively based on the specified requirements.

For the proof of concept of the framework, major parts of it have been implemented. The architecture of the prototype implementation is introduced in the following section.

8.2 Prototype Architecture and Implementation

To demonstrate the essential functionalities of the Test Creation Framework (TCF), a research prototype was developed. Most components of the TCF architecture were implemented, but not all functionality of each component has been implemented for the

proof of concept (see section 8.3). Besides the simplified version of the graphical user interface Test Framework User Terminal (TFUT), further trimmed versions of the Automatic Composition Engine (ACE), the Test Case Derivation Unit (TCDU), the Test Code Generator (TCG) and the Test Suite Builder (TSB) as well as the service interface of the Test Modules Environment (TME) to access the two databases Test Modules Repository (TMR) and Test Data Pool (TDP) have been implemented. The Test Suite Generator (TSG) has not been considered as a separate component as its only task is to comprise the TCG and the TSB. Furthermore, the graphical user interface of the TME has not been implemented because it is also not required for the proof of concept. Actually, it is not needed because a modelling environment is not required to add new reusable test modules to the TCF. To define a new reusable test module, three files have to be added: an SCXML description specifying the behaviour, a classification template containing the metadata of the reusable test module (both stored within the TMR) and, finally, a set of variables described by means of XML (stored within the TDP). The final component considered in the implementation is the Test Execution Environment (TEE). As mentioned before in chapter 7, the selected TEE is the TTworkbench (TTworkbench, 2015), a commercial TTCN-3 test system which enables the execution of tests and the generation of test reports. For the proof of concept of the research prototype and proposed TCF, an example value-added service has been selected and described by means of the STD (see section 8.3). Based on the STD instance, the whole process will be demonstrated for proof of concept until the test execution against the SUT has terminated.

The research prototype was implemented using the Java programming language because it is known to be platform independent. Furthermore, Java is required for the usage of OSGi (OSGi Alliance R5, 2012), formerly known as the Open Services Gateway

initiative. OSGi is a framework for Java which enables to install units of resources which are called bundles. These bundles can export services or run processes, and have their dependencies to other bundles or libraries managed by an OSGi container. It is also possible that each bundle has its own internal classpath so that it serves as an independent unit. In general, bundles are loosely coupled and interact either by service interfaces or by OSGi events. All of these features are standardised in order to verify that any valid OSGi bundle can theoretically be installed in any valid OSGi container. The OSGi platform has been chosen as development framework for the research prototype because of the architecture of the application. The TCF architecture (see Figure 4.5) contains many components that are loosely coupled. Within the research prototype, each of the loosely coupled components are implemented as OSGi bundle. The following Figure 8.1 illustrates the architecture of the research prototype.

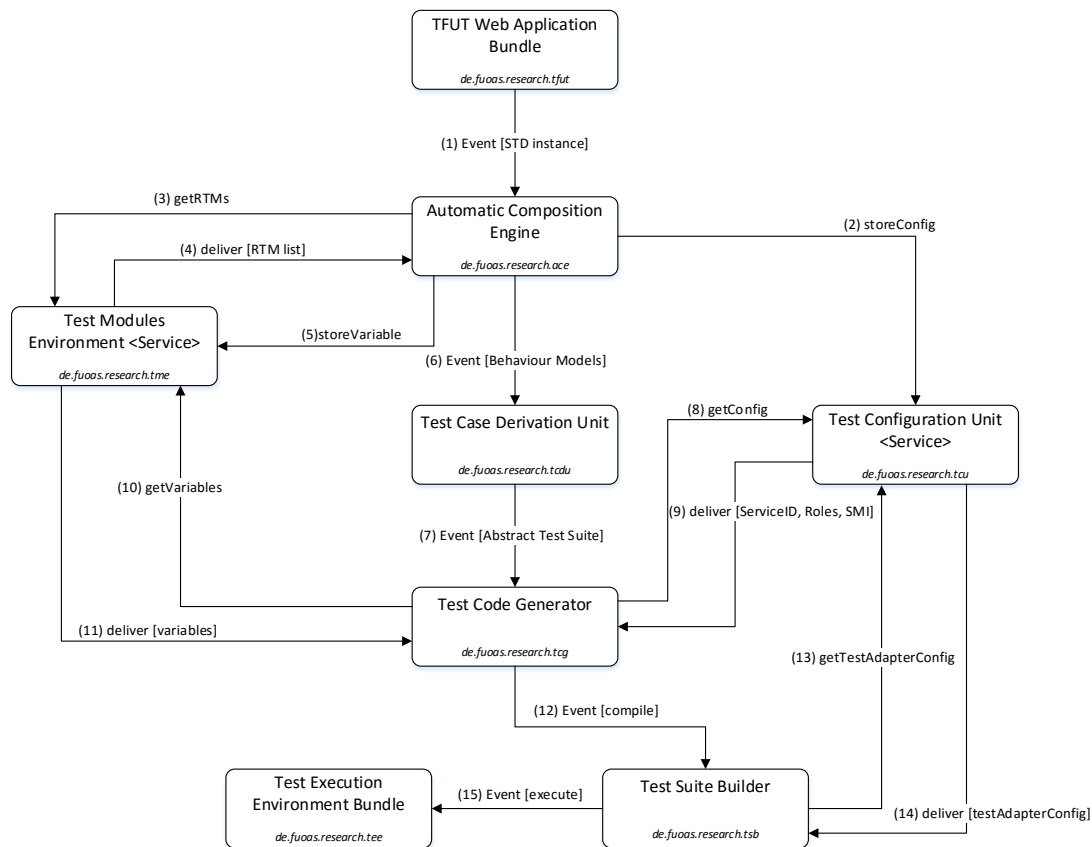


Figure 8.1: Prototype architecture components illustrated as OSGi bundles

The architecture illustration (see Figure 8.1) shows how the OSGi bundles communicate with one another. Each bundle is classified by its name and by a specific Java package name. The name of the “Automatic Composition Engine” OSGi bundle, for instance, also contains the Java package name “de.fuoas.research.ace”. Correspondingly, the source code for the implementation of the ACE is also included in this Java package. Furthermore, Figure 8.1 includes six processing bundles (such as “TFUT Web Application Bundle”, “Automatic Composition Engine”, “Test Case Derivation Unit”, “Test Code Generator”, “Test Suite Builder” and “Test Execution Environment Bundle”) as well as two service bundles (such as “Test Modules Environment” and “Test Configuration Bundle”). In the prototype implementation, each of these bundles were implemented and installed within an OSGi framework implementation. There are many

OSGi framework implementations available such as Apache Felix (Apache Felix, 2015) or Apache Karaf (Apache Karaf, 2015) and many others. The main difference between these available OSGi framework implementations is the set of features they support. Apache Felix just provides a basic set of features whereas Apache Karaf, although declared as lightweight, is still a powerful and enterprise ready OSGi framework implementation. Because of its useful features and easy handling, the Apache Karaf implementation has been chosen for the prototype implementation. The Karaf architecture is demonstrated in the following Figure 8.2.

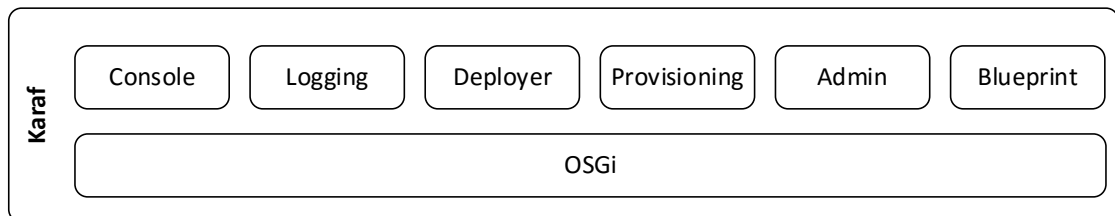


Figure 8.2: Apache Karaf architecture (adapted from (Apache Karaf, 2015))

The provided features of Apache Karaf enable a thorough monitoring and configuration of the platform (“Console”, “Logging” and “Admin”). Furthermore, the handling of bundles such as the deployment (“Deployer”) and the integration of external libraries (“Provisioning”) is supported. The “Blueprint” feature is required in order to classify developed bundles as services and to include their implemented functionality in other bundles.

Coming back to Figure 8.1, the Apache Karaf requires a further feature, an integrated Java web server. In fact, Apache Karaf can act as complete WebContainer powered by Jetty (Jetty, 2015) with fully support of the JavaServer Pages (JSP) and Java servlets. This is required to publish the web application that is included in the “TFUT Web Application Bundle” (see Figure 8.3).

The screenshot shows a web form for defining an STD instance, organized into two main sections: Architecture and Requirement.

Architecture Section:

- Service ID:** SendMessage
- Prose:** A message is sent to a service user.
- Roles:** SIP phone: [sender]
- System Meta Info:** ServiceURI: sip.messenger@192.168.50.27
- Non-functional properties:** none

Requirement Section:

- Requirement ID:** Req01
- Requirement Goal:** Service send message to user.
- Precondition:** Precondition
- Communication Interface:** SIP UAC non-INVITE (dropdown menu)
- Parameter Definition:**

```
var initMessage = [sender] -> s_Request;
initMessage = { (Method, "Message"), (Text, "Hello World!") }
```
- Basic Flow:**

```
p:=
a<initMessage>
0
```
- Alternative Flow:** A blue plus sign button (+).

Figure 8.3: Screenshot of TFUT web application showing the definition of an STD instance

The web page shows a form which enables a test developer to create an STD instance. All the required fields of the *architectural* and *behavioural perspective* (see section 5.2) are included on the web page. As soon as the STD instance has been completely defined, the form can be sent to the web server. Here, a Java servlet “STDWebServlet” is implemented which reads all the delivered parameters and creates a Java object of the “ServiceTestDescription” class, the STD instance. In the prototype implementation, the class structure of the conceptual model was realised (see Figure 6.26). As soon as the STD instance exists, it is automatically sent to the “Automatic Composition Engine” bundle via an OSGi event. An OSGi event can include any kind of data and can be transferred between bundles that acquire the “EventAdmin” service provided by the OSGi framework implementation (Apache Karaf).

When the “Automatic Composition Engine” bundle receives the STD instance it first acquires the service of the “Test Configuration Unit”. If a service within OSGi is established a Java interface is required in order to determine the offered functions. The Java interface for the “Test Configuration Unit” is as follows (see Figure 8.4):

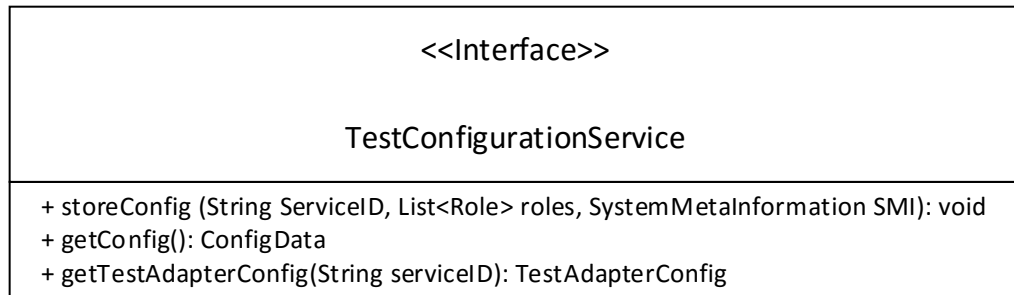


Figure 8.4: OSGi service interface provided by the “Test Configuration Unit” bundle

The “Automatic Composition Engine” bundle invokes the method “storeConfig” to store the metadata of the SUT (such as “ServiceID”, “Roles” and “SystemMetaInformation”) for further processing during the test case generation and execution. This is the illustrated step 2 of Figure 8.1. While parsing the *Requirements* of the STD instance, the “Automatic Composition Engine” bundle identifies the reusable test modules that are involved. Based on the information, the bundle consumes another service which is now provided by the “Test Modules Environment” bundle. Again, a Java interface is required in order to specify the OSGi service functionality. This is illustrated in the following Figure 8.5.

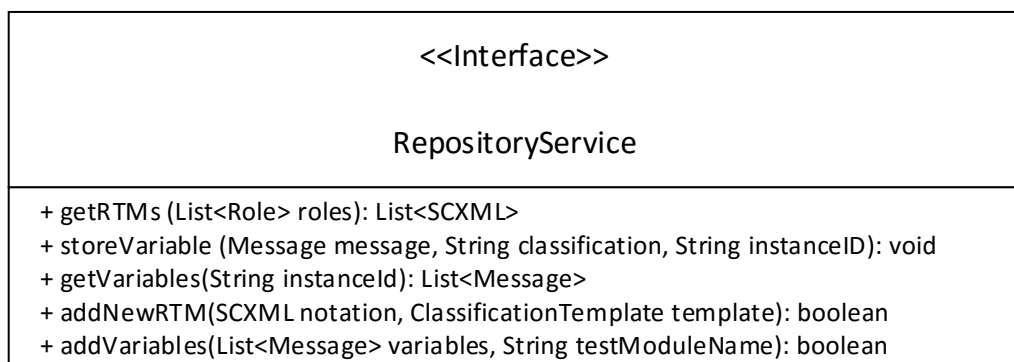


Figure 8.5: OSGi service interface provided by the “Test Modules Environment” bundle

First, the “Automatic Composition Engine” bundle invokes the “getRTMs” method (see Figure 8.5). The *Roles* (list of roles) need to be added as parameter because they include the selected *CIs* within the STD instance and accordingly, the information about the reusable test modules to be selected. The method returns a list of objects of the class “SCXML” which has not been further specified yet. In fact, this class refers to a Java library being applied to handle the SCXML-based descriptions of the reusable test module instances and the behaviour models. This Java library is called “Apache Commons SCXML” (Commons SCXML, 2015) and enables a complete representation of the XML-based Statecharts notation by means of Java classes. After the invocation of “getRTMs”, the reusable test modules are initialised within the behaviour models. As soon as the *Parameters* within the STD instance have been parsed, the “Automatic Composition Engine” invokes the “storeVariable” method (see Figure 8.5). Here, the variables of the reusable test module instances need to be stored in the Test Modules Repository (TMR) database via the service interface of the “Test Modules Environment” bundle. To store data, a NoSQL database has been applied for both TMR and Test Data Pool (TDP) which is called MongoDB (MongoDB, 2015). Coming back to the “storeVariable” method, three input parameters are required: the message itself, the type of reusable test module the variable refers to and the id of the reusable test module instance the variable is assigned to. For the class “Message” any kind of request or response type can be added due to the specific class structure (see Figure 6.20). The “storeVariable” method has to be invoked as often as a variable within the *Parameters* of the STD instance has been initialised and parameterised.

The following step 6 in Figure 8.1 describes the sending of a further OSGi event which already includes the behaviour models. So, the “Automatic Composition Engine” bundle

has already performed the formal processing and composition of reusable test module instances (see section 6.4). The bundle which receives the behaviour model is the “Test Case Derivation Unit” bundle which includes algorithms to derive the graph-based test cases applying the *All-Round-Trips* coverage criteria (see section 7.1). As a result, the abstract test suite is generated which is also sent within an OSGi event from the “Test Case Derivation Unit” bundle to the “Test Code Generator” bundle (see Figure 8.1, step 7).

In order to generate the TTCN-3 test configuration, the “Test Code Generator” bundle needs the meta information of the SUT (see Figure 8.1, steps 8 and 9). Therefore, the “getConfig” method (see Figure 8.4) provided by the “Test Configuration Unit” bundle is invoked. The return type “ConfigData” comprises the “ServiceID”, “Roles” and “SystemMetaInformation”. Besides the generation of the test configuration, the “Test Code Generator” creates TTCN-3 templates for the test data. To get the relevant data, the “Test Code Generator” needs to invoke the “getVariables” method (see Figure 8.1, steps 10 and 11). As input, the “getVariables” method requires the reusable test module instance id. The return type is a list of “Message” objects which can be processed and generated into TTCN-3 templates. Finally, the “Test Code Generator” needs to internally process the generation of the TTCN-3 test cases based on the graph-based test cases (see section 7.2.2). The generation of actual TTCN-3 text-based files is performed by means of a special test generating utility “Texen” which is part of the Apache Velocity Project (Apache Velocity, 2015). After all TTCN-3 source files have been generated, the “Test Code Generator” bundle sends a command OSGi event (see Figure 8.1, step 12) to the “Test Suite Builder” bundle.

Before the “Test Suite Builder” initiates the TTCN-3 compilation process, it requires the test adapter configuration file. As the “Test Configuration Unit” bundle holds the necessary meta information about the SUT, it can also generate the appropriate XML-based test adapter configuration. By invoking the “getTestAdapterConfig” method provided by the “Test Configuration Unit” bundle service (see Figure 8.4), the file can be fetched (see Figure 8.1, steps 13 and 14). Of course, it is important to add the correct “ServiceID” as input parameter of the method. As soon as the test adapter file is added to the TTCN-3 source files, the Executable Test Suite (ETS) can be generated. Therefore, the execution of the command line tool (or rather script) “Ttthree” is required which is provided by the TTworkbench in order to compile the sources. The “Test Suite Builder” bundle includes a specific Java class “ProcessBuilder” which invokes the “Ttthree” script. For a proper execution, the following options have to be added to the “Ttthree” script (TTworkbench UserGuide, 2015):

- *--clf-name*: Through this option, the name of the test campaign can be specified. It is advisable to include the “ServiceID” here so that the existing ETS for specific SUTs can be differentiated.
- *--clf-taconfig-file*: The presence of the test adapter configuration has already been discussed and an example of it is illustrated in Figure 7.24. Here, the project relative path to the file including its filename has to be specified.
- *--destination-path*: This option specifies the path where the compiled TTCN-3 modules will be placed in the file system. As the “Test Execution Environment Bundle” performs the execution of the test cases the complete ETS is copied to a location of the bundle scope.

- *moduleId*: Here, all the modules that have to be generated are specified. As discussed in 7.2.2, every module is assigned to a *Requirement* specified in the STD instance.

After the compilation has been performed, the generated Java classes that represent the ETS are automatically copied to the specified destination path. Furthermore, a campaign loader file (*.clf) is generated which includes the order of the test cases to be executed.

The final step 15 in Figure 8.1 is initiated by the “Test Suite Builder”. As soon as the compilation process has terminated the bundle sends an OSGi event including a command (“execute”) to the “Test Execution Environment Bundle”. Subsequently, this bundle uses a further command line tool or rather script, the “TTman”. This script also includes specific options that can be configured (TTworkbench User Guide, 2015):

- *--error*: If this option is set, the test execution stops in the case of an error. Otherwise, the execution continues.
- *--log*: This option defines a destination folder where the log file shall be stored after the test case execution. If this option is not used, the file is stored in the same directory where the ETS is stored.
- *--loop*: This option defines how many times all test cases within the ETS shall be executed.
- *--report*: Based on the test case execution, the output format of the results can be specified, either as HTML, PDF, Excel or Word document.
- *--wait*: This option allows to define a delay (in milliseconds) between the execution of two test cases. This might be useful for services where certain data has to be reset.

- *loader_file*: The loader file (*.clf) has been generated during the compilation phase. It is required to specify this file for a proper test execution.

As soon as the script is started, all test cases included in the ETS are executed against the SUT. The results are presented in the specified format and additionally included in a generated log file (*.tlz).

Before continuing with the prototype-based framework evaluation in the next section, two further methods the TME service interface provides are described which have not been used in the process (see Figure 8.5). First, the method “addRTM” enables the adding of new reusable test modules to the TMR database. There are two input parameters required, the formalised underlying Statecharts notation as SCXML type (used from the Apache Commons SCXML library) as well as the XML-based classification template which includes all the required metadata of the reusable test module. The second method “addVariables” also refers to the definition of new reusable test modules. Here, a new set of variables can be added to a stored reusable test module. Therefore, the variables have to be specified as input parameter (“variables”) as well as the unique identifier of the reusable test module (“testModuleName”). Both of the specified methods are not used in the process, but they are required for the extensibility of the prototype implementation. To support the inclusion of further reusable test modules, a test bundle has been implemented besides the specified ones in Figure 8.1. This test bundle consumes the service provided by the “Test Modules Environment” bundle and allows to add new reusable test modules to the TCF implementation.

8.3 Proof of Proposed Framework Concept

The implementation and architecture of the research prototype based on the proposed novel TCF concept has been briefly introduced in the previous section. This section deals with the proof of concept and evaluation of the underlying concept this research proposes.

Therefore, the following steps have to be performed:

1. An example service has to be specified for proof of concept and has to be described shortly.
2. A System Under Test (SUT) environment (SIP Application Server) has to be set up. The example service has to be developed and deployed on the SIP AS.
3. The example service has to be defined for proof of concept and has to be specified by means of the Service Test Description (STD).
4. The automatic TCF process needs to be started until the test case execution against the deployed service has terminated. The test case results can then be evaluated.

In general, it has to be shown that new value-added services can be tested by applying the novel concept of the TCF.

8.3.1 Description of Example Service Scenario

As mentioned before in section 5.2.4, a simplified form of the sample chat service introduced in section 5.1.1 will be applied as proof of concept for the proposed framework and prototype implementation. The following UML use case diagram shows the reduced functionality (see Figure 8.6).

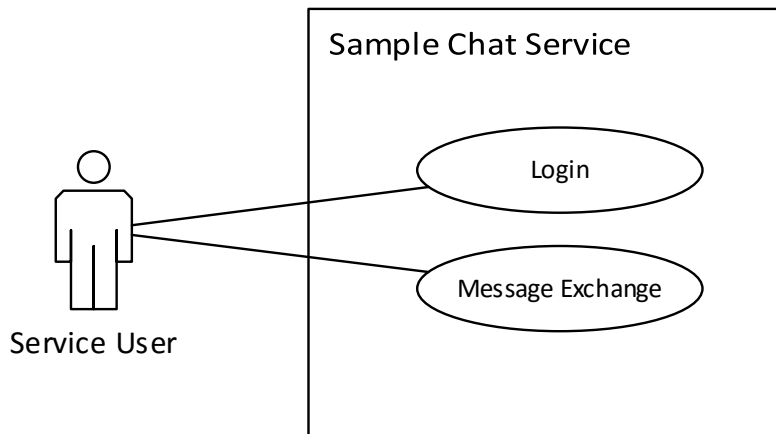


Figure 8.6: Simplified UML use case diagram of sample chat service

The sample chat service includes two major functionalities, the login of two service users as well as the exchange of instant chat messages between both service users.

The functionality of the “Login” process is illustrated by means of the following message sequence chart (see Figure 8.7).

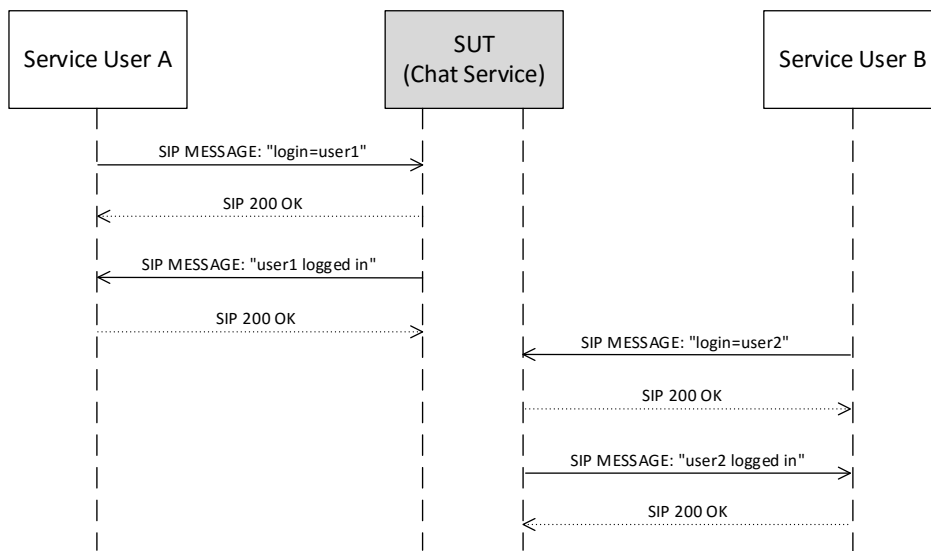


Figure 8.7: Basic functionality of login process in sample chat service

Both service users involved in the example login process send SIP MESSAGE requests to the SUT (SIP AS with deployed chat service) which contain a character string “login=”

followed by a specific user name (either “user1” or “user2”). If the login was successful, this is acknowledged by the SUT through a SIP MESSAGE request with the text “user1 logged in” or “user2 logged in”. Otherwise, if the user to be logged in by a service user is unknown (see Figure 8.8), the SUT responds with SIP MESSAGE containing the text “Unknown user! Login failed!”.

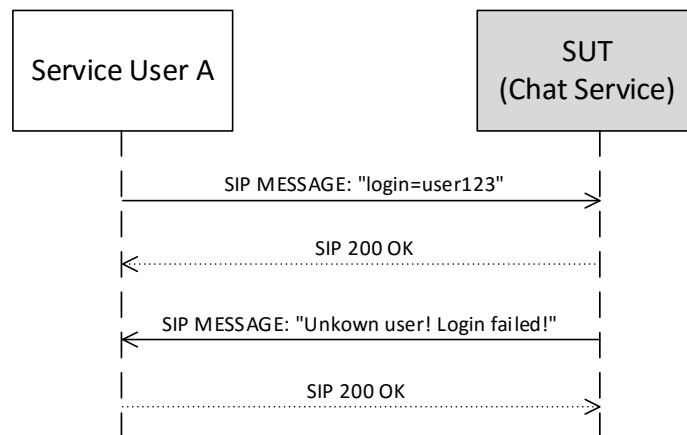


Figure 8.8: Alternative functionality of login process in sample chat service

Regarding the “Message Exchange” use case, the following message sequence chart illustrates the basic functionality.

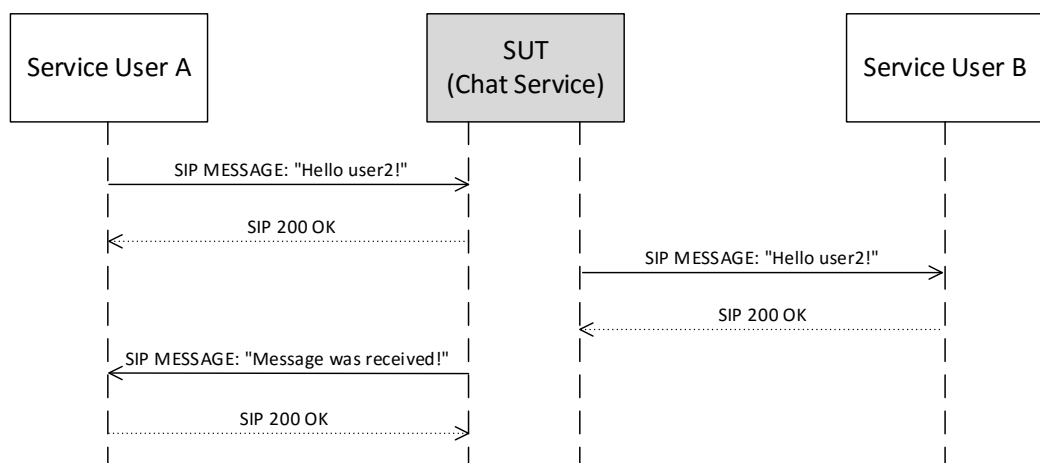


Figure 8.9: Basic functionality of message exchange in sample chat service

Based on a SIP MESSAGE sent from service user A, the service forwards the message to the user who is also currently logged in, service user B. During the login process, the service assigns the permanent SIP URIs of the service users to the user names that they selected. This enables the service to forward the messages to all users who are logged in except for the originator of the message. Finally, the service also informs the originator that the transmission was successful.

The main functionality has been specified. In the following section, the characteristics of the SUT environment will be introduced.

8.3.2 SUT Environment and Service Implementation

As described in section 2.2.4, value-added services running within SIP-based NGN environments are generally deployed on SIP Application Servers. The SIP AS enable a fast and cost-efficient provision of these services. For the proof of concept, a SIP AS implementation has been chosen that is based on SIP servlets. According to (Oracle, 2010), a SIP servlet “is a Java programming language server-side component that performs SIP signalling. SIP servlets are managed by a SIP servlet container, which typically are part of a SIP-enabled application server”. The specific SIP-enabled application server is called “Mobicents SIP Servlets” which “delivers a consistent, open platform on which to develop and deploy portable and distributable SIP and Converged JEE services.” (Mobicents, 2015). It implements the SIP Servlet v.1.1 (JSR 289 Spec, 2008) on top of Tomcat (Tomcat, 2015) and JBoss (JBoss, 2015) containers. In the following Figure 8.10, the components of the Mobicents SIP Servlets application server (AS) are illustrated.

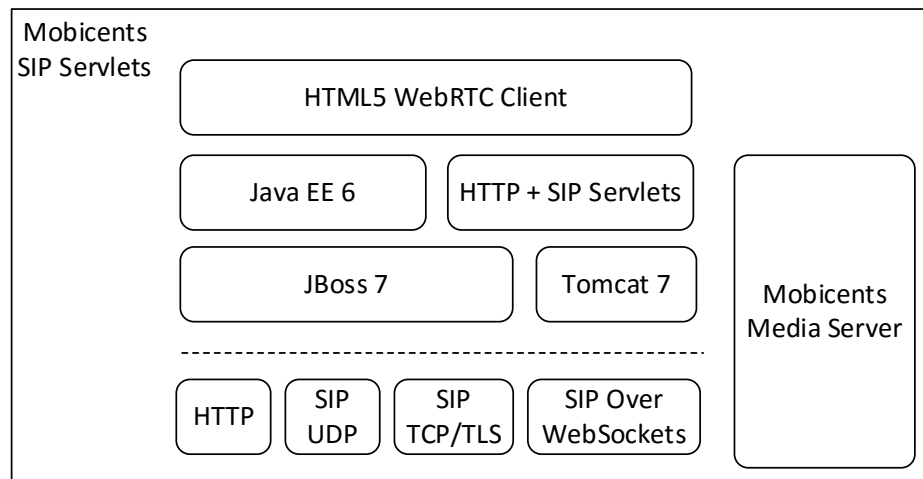


Figure 8.10: Components within Mobicents SIP Servlets application server (adapted from (Mobicents, 2015))

Besides the mentioned SIP servlets, the Mobicents SIP Servlet AS also enables the provision of HTTP servlets. Furthermore, a “Mobicents Media Server” (MMS) can be installed which provides functions a standard media server (see sections 2.3.2 and 6.2.2) provides, such as Interactive Voice Response (IVR) as well as generation and detection of tone including DTMF (Dual-tone multi-frequency signaling). The MMS can also act as a conference access point or an announcement access point (Mobicents, 2015).

For the proof of concept, the Mobicents SIP Servlets AS was installed on a Linux-based virtual machine and was integrated into the local IP network with the TTCN-3-based test execution environment (TTworkbench) and the prototype (Apache Karaf with deployed bundles). In the following, the implementation of the Java-based sample chat service was performed. The class diagram is shown in the following Figure 8.11.

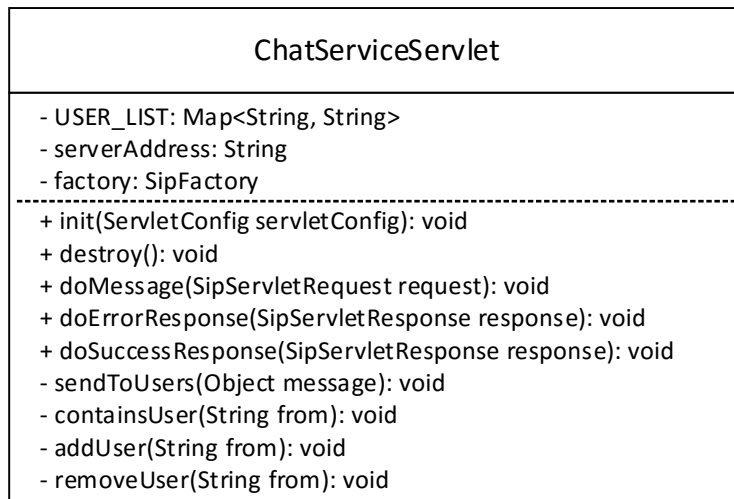


Figure 8.11: ChatServiceServlet class of proof of concept sample chat service

The “ChatServiceServlet” class contains as attribute a *USER_LIST* which holds all the users that are allowed to log in and that already have logged in. The *serverAddress* attribute holds the current IP address of the application server (here: “192.168.110.10”) whereas the *factory* enables the establishment of new SIP requests and responses within the “ChatServiceServlet”. The “ChatServiceServlet” inherits from a base class called “SipServlet” from which it takes over the methods “init” and “destroy”. Both methods are used to either set the relevant parameters at the beginning (“init”) or to be prepared as soon as the server shuts down (“destroy”). The further public (“+”) methods are referring to the message handling. The method “doMessage”, for instance, deals with incoming SIP MESSAGE requests and processes the content in the following. This method is invoked as soon as login messages are sent to the SUT or messages that have to be forwarded to other users. The methods “doErrorResponse” and “doSuccessResponse” refer to provisional messages the servlet receives. Furthermore, the private methods (“-“) perform internal processing, such as the sending of the messages to be forwarded to the users that are logged in (“sendToUsers”). The method “containsUser” checks if a “Login”

SIP MESSAGE contains a user that is allowed to be logged in. Finally, the methods “addUser” and “removeUser” manage the *USER_LIST* attribute.

In this section, the SUT environment has been set up and the example service (SUT) itself has been developed. The next section deals with the first task the test developer has to do, the definition of an STD instance for the sample chat service.

8.3.3 Specification of Chat Service with Service Test Description

The compilation of an STD instance with all the required components has been described thoroughly in this thesis (see section 5.2). Also for the selected proof of concept example service, first the *architectural perspective* has to be defined. The definitions are shown in the following Table 8.1.

Table 8.1: Architectural perspective of sample chat service

Service ID	ChatService
Prose Description	A chat communication should be provided. The service users are able to log into the system by sending a text message that contains predefined user names. If the login process was successful the service responds accordingly. A message exchange can be performed between two users if they are both logged in. If the message exchange was successful the service responds accordingly.
Roles	<ul style="list-style-type: none"> • SIP phone: [sender] • SIP phone: [recipient]
System Meta Information	ServiceURI: sip:chatservice@192.168.110.10:5060 Protocol: UDP
Non-functional Properties	None

The specification of the *architectural perspective* in Table 8.1 is very similar to the one defined in Table 5.8, however, a few aspects changed. The ID of the service, the *Service ID*, is very relevant as it will be reused throughout the process. Furthermore, the most important information is included in the *Roles* field and in the *System Meta Information*

field. For the “ChatService”, two *Roles* have been specified and both are acting as SIP phones. The “[sender]” and the “[recipient]” both can be mapped to the service users specified in section 8.3.1. The *System Meta Information* includes the service URI, which includes the addressability of the implemented “ChatServiceServlet”. The specified IP address “192.168.110.10” is the IP address of the Mobicents SIP Servlet AS. As transport protocol, UDP has been selected.

In the following the *behavioural perspective* of the STD instance has to be determined. The UML use case illustration (see Figure 8.6) of the sample chat service includes two use cases which have to be defined as *Requirements* within the STD instance. In the following Table 8.2, the “Login” process (see Figure 8.7 and Figure 8.8) is specified as “Req01”.

Table 8.2: Behavioural perspective for "Login" use case ("Req01")

Requirement ID	Req01
Requirement Goal	Service User A [sender] sends a login message to the service and receives a confirmation message. Service User B [recipient] sends a login message to the service and receives a confirmation message.
Precondition	None
Participating Roles	<ul style="list-style-type: none"> • SIP phone: [sender] • SIP phone: [recipient]
Communication Interfaces	<ul style="list-style-type: none"> • SIP UAS non-INVITE: [sender1] → channel a • SIP UAC non-INVITE: [sender2] → channel b • SIP UAS non-INVITE: [recipient1] → channel c • SIP UAC non-INVITE: [recipient2] → channel d
Parameters	<pre> var loginA = [sender1] → r_Request; var loginB = [recipient1] → r_Request; var okLoginA = [sender2] → s_Request; var okLoginB = [recipient2] → s_Request; var errorLoginA = [sender2] → s_Request; var errorLoginB = [recipient2] → s_Request; loginA = {(Method, "MESSAGE"), (Text, "login=user1")} loginB = {(Method, "MESSAGE"), (Text, "login=user2")} okLoginA = {(Method, "MESSAGE"), (Text, "user1 logged in")} okLoginB = {(Method, "MESSAGE"), (Text, "user2 logged in")} errorLoginA = {(Method, "MESSAGE"), {Text, "Unknown User! Login failed!"}} errorLoginB = {(Method, "MESSAGE"), {Text, "Unknown User! Login failed!"}} </pre>
Basic Flow	$P \stackrel{\text{def}}{=} a(\text{loginA}).$ $\text{if}(\text{loginA.Text} \neq \text{"login=user1"}) \text{ then } Q \text{ else.}$ $\bar{b}\langle \text{okLoginA} \rangle.$ $c(\text{loginB}).$ $\text{if}(\text{loginB.Text} \neq \text{"login=user2"}) \text{ then } R \text{ else.}$ $\bar{d}\langle \text{okLoginB} \rangle.$ 0
Alternative Flow (AF1)	$Q \stackrel{\text{def}}{=} \bar{b}\langle \text{errorLoginA} \rangle.$ 0
Alternative Flow (AF2)	$R \stackrel{\text{def}}{=} \bar{d}\langle \text{errorLoginB} \rangle.$ 0

Both Roles “[sender]” and “[recipient]” are participating within Requirement “Req01” through both of their CIs “SIP UAS non-INVITE” and “SIP UAC non-INVITE”. Within

the *Parameters* field, six potentially used SIP MESSAGE requests are parameterised. The *Basic Flow* describes the straightforward case by including the appropriate SIP MESSAGEs in order to log in both participating *Roles*. Two *if-then-else* structures are included in order to handle wrong login messages. The further actions are specified through both *Alternative Flows*. In the following Table 8.3, the second *Requirement* specifying the “Message Exchange” use case is described.

Table 8.3: Behavioural perspective for “Message Exchange” use case (“Req02”)

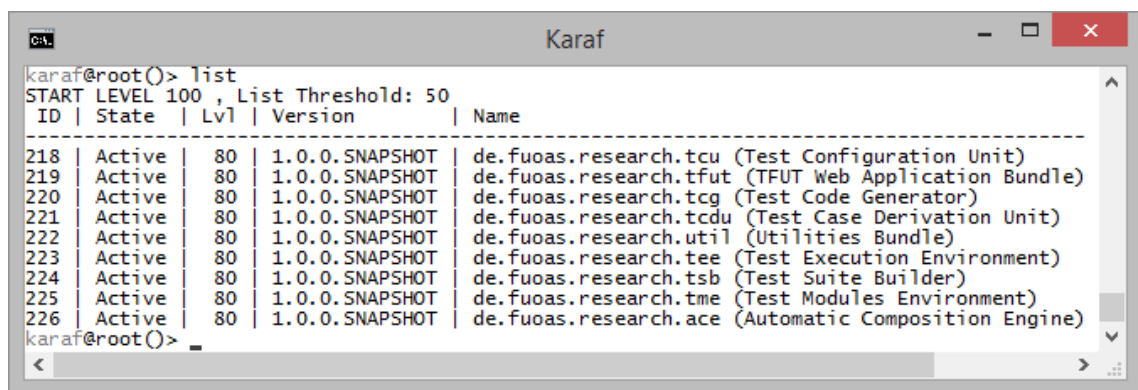
Requirement ID	Req02
Requirement Goal	Service User A [sender] sends a message to the service which is then forwarded to the service users that are currently logged in (except for the originator of the message). Service User A [sender] receives a confirmation message.
Precondition	Req01
Participating Roles	<ul style="list-style-type: none"> • SIP phone: [sender] • SIP phone: [recipient]
Communication Interfaces	<ul style="list-style-type: none"> • SIP UAS non-INVITE: [sender1] → channel a • SIP UAC non-INVITE: [sender2] → channel b • SIP UAC non-INVITE: [recipient2] → channel d
Parameters	<pre>var msgA = [sender1] → r_Request; var forwMsgB = [recipient2] → s_Request; var okMsgA = [sender2] → s_Request; msgA = {(Method, "MESSAGE"), (Text, "Hello user2!")} forwMsgB = {(Method, "MESSAGE"), (Text, "Hello user2!")} okMsgA = {(Method, "MESSAGE"), (Text, "Message was received!")}</pre>
Basic Flow	$P \stackrel{def}{=} a(msgA).$ $\bar{d}(forwMsgB).$ $\bar{b}(okMsgA).$ 0

Requirement “Req02” includes “Req01” in the *Precondition* field, in particular the *Basic Flow* of “Req01”. Furthermore, “Req02” also specifies the two participating *Roles* from the *architectural perspective* and selects both *CIs* for the “[sender]” *Role* and the “SIP UAC non-INVITE” *CI* for the “[recipient]” *Role*. The *Parameters* field includes the

parameterisation of both the initiated and the forwarded SIP MESSAGE as well as the confirmation SIP MESSAGE the service sends to the originator, the “[sender]”. The *Basic Flow* illustrates the steps to be taken. “Req02” does not contain any *Alternative Flows*.

8.3.4 Test Building and Test Execution

The STD instance from the previous section can be processed as soon as the test developer has defined it in the graphical user interface (see Figure 8.3) provided by the “TFUT Web Application Bundle”. First, it has to be verified that the prototype implementation is running within the Apache Karaf. The OSGi implementation provides a console which can be used for specific commands and also for logging. The following Figure 8.12 shows the list of the currently active OSGi bundles within Apache Karaf.



```

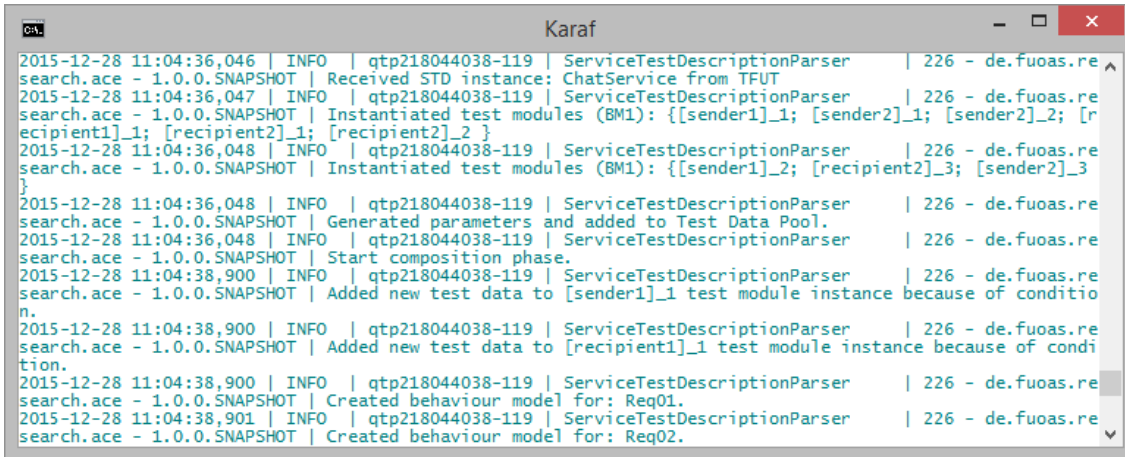
karaf@root()> list
START LEVEL 100 , List Threshold: 50
ID | State | Lvl | Version | Name
-----
218 | Active | 80 | 1.0.0.SNAPSHOT | de.fuoas.research.tcu (Test Configuration Unit)
219 | Active | 80 | 1.0.0.SNAPSHOT | de.fuoas.research.tfut (TFUT Web Application Bundle)
220 | Active | 80 | 1.0.0.SNAPSHOT | de.fuoas.research.tcg (Test Code Generator)
221 | Active | 80 | 1.0.0.SNAPSHOT | de.fuoas.research.tcd (Test Case Derivation Unit)
222 | Active | 80 | 1.0.0.SNAPSHOT | de.fuoas.research.util (Utilities Bundle)
223 | Active | 80 | 1.0.0.SNAPSHOT | de.fuoas.research.tee (Test Execution Environment)
224 | Active | 80 | 1.0.0.SNAPSHOT | de.fuoas.research.tsb (Test Suite Builder)
225 | Active | 80 | 1.0.0.SNAPSHOT | de.fuoas.research.tme (Test Modules Environment)
226 | Active | 80 | 1.0.0.SNAPSHOT | de.fuoas.research.ace (Automatic Composition Engine)
karaf@root()>

```

Figure 8.12: Active OSGi bundles in Apache Karaf environment

All the specified OSGi bundles of the prototype implementation are listed and can be identified by their name which includes both the Java package name and the standard name already known from Figure 8.1. Besides the specified bundles there is also a further “Utilities Bundle”. Here, so-called helper classes are included to support other bundles for recurring processing, such as string operations. All bundles are stated as “active”, so

the processing can start. As soon as the test developer has submitted the STD instance on the web site, the “Automatic Composition Engine” bundle receives the instance and starts processing (see Figure 8.13).



```
2015-12-28 11:04:36,046 | INFO | qtp218044038-119 | ServiceTestDescriptionParser | 226 - de.fuoas.re
search.ace - 1.0.0.SNAPSHOT | Received STD instance: ChatService from TFUT
2015-12-28 11:04:36,047 | INFO | qtp218044038-119 | ServiceTestDescriptionParser | 226 - de.fuoas.re
search.ace - 1.0.0.SNAPSHOT | Instantiated test modules (BM1): {[sender1]_1; [sender2]_1; [sender2]_2; [r
ecipient1]_1; [recipient2]_1; [recipient2]_2 }
2015-12-28 11:04:36,048 | INFO | qtp218044038-119 | ServiceTestDescriptionParser | 226 - de.fuoas.re
search.ace - 1.0.0.SNAPSHOT | Instantiated test modules (BM1): {[sender1]_2; [recipient2]_3; [sender2]_3
}
2015-12-28 11:04:36,048 | INFO | qtp218044038-119 | ServiceTestDescriptionParser | 226 - de.fuoas.re
search.ace - 1.0.0.SNAPSHOT | Generated parameters and added to Test Data Pool.
2015-12-28 11:04:36,048 | INFO | qtp218044038-119 | ServiceTestDescriptionParser | 226 - de.fuoas.re
search.ace - 1.0.0.SNAPSHOT | Start composition phase.
2015-12-28 11:04:38,900 | INFO | qtp218044038-119 | ServiceTestDescriptionParser | 226 - de.fuoas.re
search.ace - 1.0.0.SNAPSHOT | Added new test data to [sender1]_1 test module instance because of conditio
n.
2015-12-28 11:04:38,900 | INFO | qtp218044038-119 | ServiceTestDescriptionParser | 226 - de.fuoas.re
search.ace - 1.0.0.SNAPSHOT | Added new test data to [recipient1]_1 test module instance because of conditio
tion.
2015-12-28 11:04:38,900 | INFO | qtp218044038-119 | ServiceTestDescriptionParser | 226 - de.fuoas.re
search.ace - 1.0.0.SNAPSHOT | Created behaviour model for: Req01.
2015-12-28 11:04:38,901 | INFO | qtp218044038-119 | ServiceTestDescriptionParser | 226 - de.fuoas.re
search.ace - 1.0.0.SNAPSHOT | Created behaviour model for: Req02.
```

Figure 8.13: Logging from "Automatic Composition Engine" bundle

The logging displayed in Figure 8.13 shows that as soon as the ACE receives the STD instance, it instantiates the reusable test modules, six for the behaviour model of “Req01” and three for the behaviour model of “Req02”. Then, the processing continues with the instantiation of the test data and the composition of the reusable test module instances. A logging message also considers that a new test data set is included for a specific reusable test module instance as soon as a condition has been detected which includes test data from the specific reusable test module instance. Finally, the behaviour models for both “Req01” and “Req02” have been generated in the process. In the following Figure 8.14, the representation of the composed reusable test module instances within the ACE is illustrated.

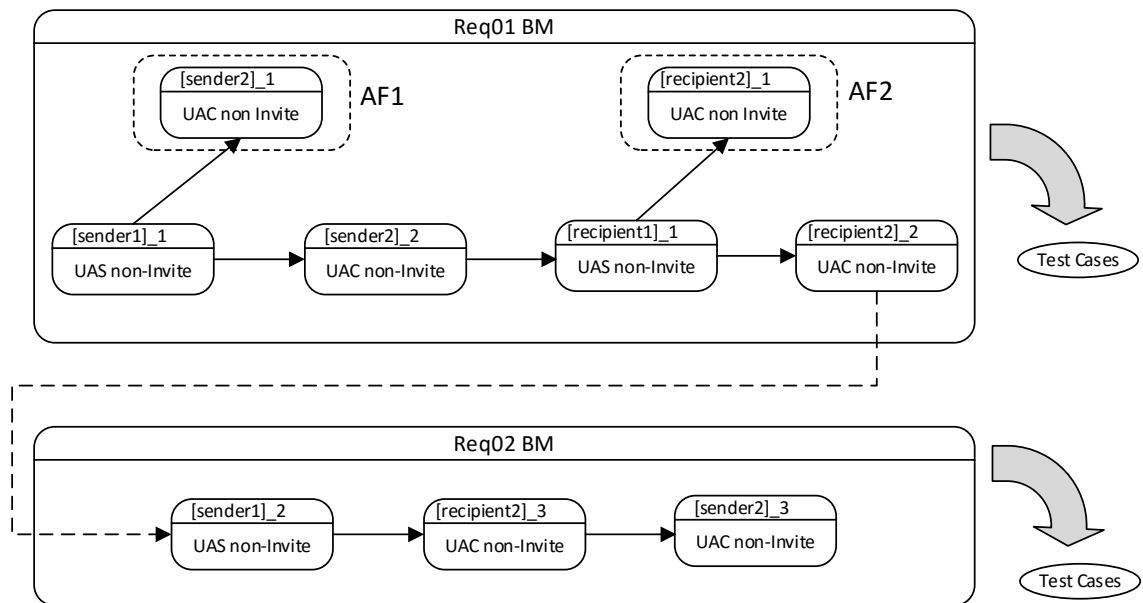


Figure 8.14: Created behaviour models by "Automatic Composition Engine" bundle

The behaviour model above refers to “Req01” with six instances of reusable test modules whereas the lower refers to “Req02” with three instances. In “Req01”, the *Alternative Flows* are labelled as “AF1” and “AF2”.

As soon as the behaviour models exist, they are automatically delivered to the “Test Case Derivation Unit” bundle which derives the test cases. As mentioned in section 7.1.2, the selection of an appropriate structure of the reusable test modules as well as the chosen coverage criterion is important with regard to the test coverage and the number of test cases. Before the test developer created the STD instance, a modified version of both standard “SIP UAC non-INVITE” and “SIP UAS non-INVITE” reusable test modules has been chosen. In fact, the “Proceeding” state which includes the handling of provisional SIP responses has been removed for both reusable test modules because it is not relevant for the handling of SIP MESSAGE requests. This possibility has already been discussed in section 7.1.2 (e.g. “SIP UAS non-INVITE without Proceeding”). As coverage criterion, the recommended *All-Round-Trip* algorithm has been applied. From

every reusable test module instance within Figure 8.14, two test cases will be derived. Regarding “Req01”, the amount of test cases for the *Basic Flow* is 16 ($2*2*2*2$) as there are four reusable test module instances visited one after the other. For each *Alternative Flow* path, two further test cases are added with the generated and modified test data sets. So, the sum of all test cases for “Req01” is 20. As “Req02” is based on “Req01”, the number of test cases now increases significantly. For each further reusable test module instance within “Req02”, the current number of test cases is multiplied by 2. In total, this leads to a number of 128 test cases for “Req02”. In total, the ETS generated for the sample chat service includes 148 test cases. Although they have not been implemented within the prototype implementation, there are a possibilities to reduce this seemingly high number of test cases. The concepts are as follows:

1. As soon as all test cases for a *CI* have been tested by traversing through one reusable test module instance, the following reusable test modules using the identical *CI* can derive a reduced set of tests (e.g. *All-Transitions* algorithm).
2. If *Requirements* are based on each other through the *Precondition* field (such as “Req02” depends on “Req01”), it is not necessarily required to execute all test cases for “Req01” once again before the test cases derived from “Req02” are executed. In fact, only one test case for the *Basic Flow* of “Req01” would be sufficient as basis to verify the functionality of “Req02”. This concept has a significant impact on the number of test cases. For the sample chat service, the total number of test cases for both “Req01” and “Req02” would be 28 (20 for “Req01” and 8 for “Req02”).

As soon as the test cases have been derived and afterwards generated by the “Test Code Generator” and “Test Suite Builder” bundles, the test execution can be fulfilled. The following test execution sample in Figure 8.15 (generated by the TTworkbench after test case execution) illustrates the “Login” process of “[sender]”.

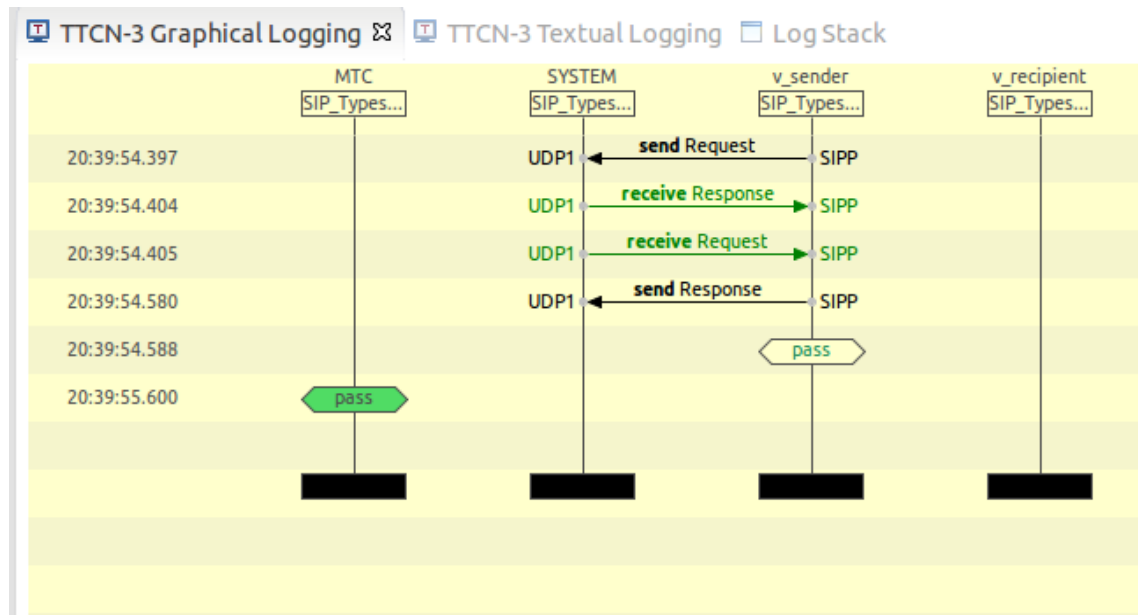


Figure 8.15: Test execution of "Login" process for "[sender]" Role

As discussed in section 7.2, every *Role* specified in the STD is represented as a test component in TTCN-3 and therefore also in the test execution process. In Figure 8.15, besides the SUT (“SYSTEM”), the *Roles* “[sender]” (“v_sender”) and “[recipient]” (“v_recipient”) are included. Between the test components and the SUT, the messaging is illustrated. Every message that is received by a test components will be highlighted either by green color (“match”) or red color (“fail”). The example test case (see Figure 8.15) is judged as “pass” at the end of the test component “v_sender”, however, the final judgement is done by the main test component (MTC).

After all 148 test cases have been executed against the SUT, a test report is generated (see Figure 8.16).

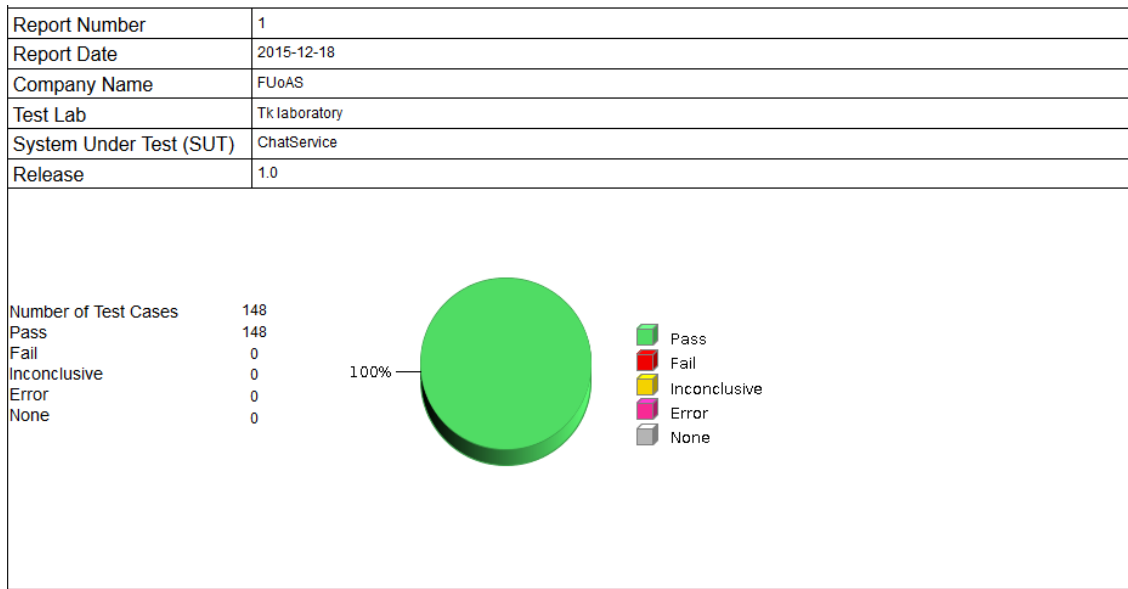


Figure 8.16: Test report for test execution against sample chat service

Besides the verdicts with regard to all 148 test cases, the test report contains documentation of the complete test case execution for each test case (like illustrated in Figure 8.15). This makes it easy for the test developer to figure out which test case failed and why it failed. The steps he has to take afterwards are described in section 7.4.

To sum up, the framework and prototype implementation could be evaluated by testing a sample value-added service. There is potential for improvement regarding the derivation of test cases within the prototype implementation. With respect to this matter, recommendations have been stated in this section.

8.4 Conclusion

This chapter has started with the evaluation of the proposed TCF (see section 8.1) with regard to the requirements stated in section 3.4. It has been analysed whether or not the requirements have been fulfilled by the proposed TCF. In fact, the framework meets all the given requirements successfully.

Furthermore, the research prototype (see section 8.2) within this project has been introduced. The prototype's architecture with the relevant developed TCF components and their interactions has been outlined as well as the used underlying Java-based modular system and service platform OSGi. The research prototype has been successfully adopted for a proof of concept evaluation of the proposed framework which demonstrates its major functionalities as well as its general applicability.

To demonstrate the applicability of the novel concept and to evaluate the framework in general, a typical value-added service has been considered as SUT, a simplified chat service. Of course, the chat service had to be specified in detail and an implementation had to be provided. Finally, the TCF process could be exemplified starting from the specification of the STD instance until the generation of the test report. The test report includes the results of the test case executions against the SUT.

9 Conclusions

This chapter concludes the thesis by summarising the main achievements of the research work (section 9.1). Additionally, limitations of the research are discussed (section 9.2) and scopes and ideas for further research are suggested (section 9.3).

9.1 Achievements of the Research

This research was dedicated to the development of a novel approach for functional testing of value-added services within NGN-based environments or SIP-based IP networks. A novel framework has been defined, which supports the test developer by means of a straightforward new service and test specification language. It includes all phases of testing starting from the reading of the STD instance compiled by the test developer and followed by the automated building of Statecharts-based behaviour models considering the information retrieved from the STD instance. Furthermore, the process includes an automated derivation of abstract test cases and the subsequent transformation into executable TTCN-3 test cases. Finally, it also performs automated testing of the test cases against the value-added service, the SUT. Because of its support for the whole testing life-cycle, the framework can be applied as a complete solution for functional value-added service testing.

The analysis of existing current methodologies in the field of agile testing and Model-based testing has been illustrated as well as related projects for functional testing (see section 3.2 and 3.3). Based on the deficits of the related projects, but also on a few assets they provide, a requirements catalogue on a novel framework for functional testing of value-added services has been established. It has been analysed whether one of the related projects could fulfil these requirements, however, an ideal solution could not be found. Besides, only one related project specifically was referred to the testing of NGN-based value-added services.

Applying the requirements that evolved from the deficits and assets of related projects, a novel framework has been developed (see chapter 4). The underlying framework architecture contains several components such as databases (Test Modules Repository and Test Data Pool) for data which are used within the tests, graphical user interfaces for the test developer to either create so-called reusable test modules which specify recurring behaviour or, alternatively, to define service and test specifications of value-added services by applying the proposed and novel STD. Furthermore, the framework includes process components for model constructions, test case derivation and test case generation. The framework can be instantiated within traditional service development life-cycles and by applying it, a path of testing is established besides the development path. As it is based on the informal requirements from which a service developer also retrieves his ideas for developing a value-added service, it enables requirements-based testing that is similar to rapid prototyping. This ability of the framework led to a novel concept that has also been established in this research, the Service Quality Group with a new role introduced in the process, the service analyst. The idea for this new integration within the process came up,

because in current solutions, the service customer is not as involved during the development and testing phase as he could be.

The basis for establishing a requirements-based testing approach lies in one of the key novelties of this research, the service and test specification language STD (see chapter 5). Here, the test developer can describe the potential behaviour of a value-added service by means of reusable behaviour which is determined through *Roles* in combination with their *CommunicationInterfaces*. In further related work, traditional test specifications are applied for this step, but the focus of the STD is different, as it puts the accent on the SUT. In fact, a *CommunicationInterface* is always part of the SUT and not part of the test system. A further novel aspect of the STD is its underlying behavioural notation that is based on the pi-calculus, a simple but very expressive process calculus in order to specify communication channels. There is no existing related work where a pi-calculus-based notation has been applied to functional testing.

A further novelty within the research has been discussed with the introduction of the reusable test modules (see chapter 6). The Statechart-based notation enables a novel view on specifying behaviour through the differentiation of server cores and client cores. This TU concept which has been taken from the SIP specification, can be applied to any application layer protocol. Based on standard SIP-related behaviour, example reusable test modules have been introduced and it has been demonstrated how they can be classified through so-called classification templates and formally stored through an XML-based notation called SCXML. Furthermore, a composition algorithm of reusable test modules based on STD instances is introduced which leads to the generation of behaviour

models. Another important aspect is the support of concurrency through so-called hierarchical AND-states which are part of Statechart-based notations.

The derivation of abstract test cases from the behaviour models introduces a new graph-based illustration of them (see chapter 7). In related works, abstract test cases which are generally derived from formal models, are represented as sequences within the model. Here, diverse coverage criteria have been taken into consideration to derive the abstract test cases. For thorough testing, the *All-Round-Trip* algorithm has been applied whereas *All-Transitions* does not lead to such a high amount of test cases if many reusable test module instances are involved. An important characteristic of the reusable test modules has been mentioned, the possibility to easily modify the behavioural description by removing states that lead to provisional behaviour. Furthermore, the generation of TTCN-3 test cases based on the abstract test cases is shown by means of a mapping.

In the final chapter 8, the proposed framework has been evaluated regarding the defined requirements (see section 3.4). Besides, for the verification of the overall framework functionalities, a research prototype has been developed. This research prototype has been successfully adopted for a proof of concept of the proposed framework by demonstrating the process by means of an example value-added service.

Several papers referring to diverse aspects of the results achieved during this research have been presented at refereed conferences and have received positive comments from delegates and reviewers.

9.2 Limitations of the Research

Even though the overall objectives of the research have been met, still some decisions had to be taken that resulted in limitations imposed on the work. In principle, those decisions were caused by practical reasons, or to limit the effort spent in areas where no new insights could be expected. The limitations are summarised below.

1. The research prototype was restricted to only implement as much functionality as required to prove that the approach taken for functional testing of value-added services was viable and that the methods developed were actually manageable. Therefore, the prototype only supports the specification of SIP-based value-added services. For instance, also the protocols HTTP and RTP could have been taken into consideration in order to check whether multimedia or web-based data could be received by a test component, but the value of knowledge would be limited.
2. Although a lot of research has been done in the field of test case derivation, the selection of an appropriate coverage criteria cannot be finally evaluated. Generally speaking, this field of research can be expanded.
3. Although it is a component of the TCF architecture, the TME has not been developed by the research prototype. However, the relevance would have been low, because new reusable test modules can of course be installed by defining a classification template and a SCXML description of the corresponding reusable test module.
4. There is no specific methodology defined within the proposed TCF to reset the state of the SUT so that a test case execution can be performed properly. It is the task of the test developer to take this into consideration.

5. The approach only supports specified functional tests or rather positive tests.

Negative tests such as ruggedness tests are not supported.

Despite these limitations, the research has made valid contributions to knowledge and provided sufficient proof of concept for the proposed approaches.

9.3 Suggestions and Scope for Future Work

This research has advanced the field of automated functional testing of value-added services in the field of NGN and SIP-based IP networks. However, there are numbers of areas for future work that can be identified upon the results of this project. Some of these have already been mentioned, however they are summarised in the following.

1. Further research may address how easily further protocols can be included into the approach and if every protocol can be described by the reusable test module concept.
2. The aspect of reusability can be further investigated. Maybe recurring behaviour can also be detected within the combination of protocols, such as SIP and HTTP.
3. The framework can be applied to different technologies and environments. For instance, functional testing of diverse software can be performed as soon as the underlying software models exist. Further different types of applications can be analysed (such as Machine-to-Machine applications).
4. Possibly, the TCF can be used for the analysis of protocols.

5. The ideas regarding the collaboration of service customers, service developers, service analysts as well as test developers can be further developed, e.g. by means of an interactive web interface for graphical monitoring and managing.

References

1. 3GPP TR 21.905 V7.0.0 (2005), Technical Report, “Vocabulary for 3GPP Specifications (Release 7)”, 3GPP
2. 3GPP TR 29.962 V6.1.1 (2005), Technical Report, “Signalling interworking between the 3GPP profile of the Session Initiation Protocol (SIP) and non-3GPP SIP usage”, 3GPP
3. Abadi, M.; Fournet, C. (2001), “Mobile values, new names, and secure communication”, *Proceedings of the 28th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 104-115, ACM
4. Abadi, M.; Fournet, C. (2004), “Private authentication”, *Theoretical Computer Science*, Vol. 322, Issue 3, pp. 427-476, Elsevier
5. Abrahamsson, P.; Hanhineva, A.; Jääliñoja, J. (2005), “Improving Business Agility Through Technical Solutions: A Case Study on Test-Driven Development in Mobile Software Development”, *Proceedings of the IFIP TC8 WG 8.6 International Working Conference*, pp. 227-243, Springer
6. Adzic, G. (2011), “Specification by Example: How successful Teams deliver the right Software”, Manning Publications, Shelter Island, USA, ISBN: 978-1-617-29008-4
7. Ammann, P. and Offutt, J. (2008), “Introduction to Software Testing”, Cambridge University Press, Cambridge, UK, ISBN: 978-0-521-88038-1
8. Antoniol, G.; Briand, L.C.; Di Penta, M.; Labiche, Y. (2002), “A case study using the round-trip strategy for state-based class testing”, *Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE 2002)*, pp. 269-279, IEEE
9. Apache Felix (2015), “Apache Felix Website”, Available at: <http://felix.apache.org>, [accessed 2nd November 2015]
10. Apache Karaf (2015), “Apache Karaf”, Available at: <http://karaf.apache.org/index.html>, [accessed 2nd November 2015]
11. Apache Tomcat (2015), “Apache Tomcat”, Available at: <http://tomcat.apache.org/>, [accessed 30th November 2015]
12. Apache Velocity (2015), “The Apache Velocity Project”, Available at: <http://velocity.apache.org/texten/devel/>, [accessed 3rd November 2015]
13. Baker, P.; Ru Dai, Z.; Grabowski, J.; Schieferdecker, I.; Williams, C. (2007), “Model-Driven Testing: Using the UML Testing Profile”, Springer, Berlin, Germany, ISBN: 978-3-540-72562-6

14. Binder, R. (1999), “Testing Object-Oriented Systems: Models, Patterns, and Tools”, Addison-Wesley, Boston USA, ISBN: 0-201-80938-9
15. Bittner, K. and Spence, I. (2002), “Use Case Modeling”, Addison Wesley, Boston, USA, ISBN: 978-0-201-70913-1
16. Börger, E. and Stärk, R. (2003), “Abstract State Machines”, Springer, Heidelberg, Germany, ISBN: 978-3-642-62116-1
17. Bozga, M.; Fernandez, J.Cl.; Ghirvu, L.; Graf, S.; Krimm, J.P. and Mounier, L. (1999), “IF: An Intermediate Representation and Validation Environment for Timed Asynchronous Systems”, *Proceedings of the World Congress on Formal Methods in the Development of Computing Systems*, pp. 307-327, Springer
18. Calisti, M. (2003), “An Agent-Based Approach for Coordinated Multi-Provider Service Provisioning”, Birkhäuser Verlag, Basel, Switzerland, ISBN: 3-7643-6922-1
19. Chattopadhyay, S. (2013), “Embedded System Design”, PHI Learning Private Limited, Delhi, India, ISBN: 978-8-120-34730-4
20. Cheng, K.-T. and Krishnakumar, A.S. (1993), “Automatic Functional Test Generation Using The Extended Finite State Machine Model”, *Proceedings of the 30th Conference on Design Automation*, pp. 86-91, IEEE
21. Cochenec, J.-Y. (2002), “Activities on next-generation networks under Global Information Infrastructure in ITU-T”, *Communication Magazine*, Vol. 40, Issue 7, pp. 98-101, IEEE
22. Cockburn, A. (2000), “Writing Effective Use Cases (Crystal Series for Software Development)”, Addison Wesley, Boston, USA, ISBN: 978-0-201-70225-5
23. Cohn, M. (2004), “User Stories Applied: For Agile Software Development”, Addison-Wesley, Boston USA, ISBN: 0-321-20568-5
24. Commons SCXML (2015), “Apache Commons SCXML”, Available at: <http://commons.apache.org/proper/commons-scxml/>, [accessed 3rd November 2015]
25. Craggs, I.; Sardis, M.; Heuillard, T. (2003), “AGEDIS Case Studies: Model-based testing in industry”, *Proceedings of the 1st European Conference on Model Driven Software Engineering*, pp. 106-117
26. Dai, Z.R.; Grabowski, J.; Neukirchen, H.; Pals, H. (2004), “From Design to Test with UML – Applied to a Roaming Algorithm for Bluetooth Devices”, *Proceedings of the 16th International Conference on Testing of Communication Systems (TestCom 2004)*, pp. 33-49, Springer
27. Devroey, X.; Perrouin, G.; Schobbens, P.-Y. (2014), “Abstract Test Case Generation for Behavioural Testing of Software Product Lines”, *Proceedings of the 18th International Software Product Line Conference (Companion Volume for Workshops, Demonstrations and Tools)*, pp. 86-93, ACM
28. Ding, J. (2010), “Advances in Network Management”, CRC Press, Boca Raton, USA, ISBN: 978-1-4200-6455-1

29. Ding, L. and Liu, L. (2008), “Modelling and Analysis of the INVITE Transaction of the Session Initiation Protocol Using Coloured Petri Nets”, *Proceedings of the 29th International Conference on Application and Theory of Petri Nets and Other Models of Concurrency (PETRI NETS 2008)*, pp. 132-151, Springer
30. Eberlein, A. (1997), “Requirements Acquisition and Specification for Telecommunication Services”, PhD thesis, Department of Electrical & Electronic Engineering, University of Wales, Swansea, United Kingdom
31. Eberlein, A.; Crowther, M.; Halsall, F. (1997), “Development Of New Telecommunications Services Using An Expert System”, *BT Technology Journal*, Vol. 15, Issue 1, pp. 217-222, ACM
32. Eberlein, A. and Halsall, F. (1997), “Telecommunications service development: A design methodology and its intelligent support”, *Engineering Applications of Artificial Intelligence*, Vol. 10, Issue 6, pp. 647-663, Elsevier
33. Eclipse TPTP (2015), “Eclipse Test & Performance Tools Platform Project”, Available at: <https://eclipse.org/tptp>, [accessed at 24th April 2015]
34. Eichelmann, T.; Fuhrmann, W.; Trick, U.; Ghita, B. (2010), “Enhanced concept of the TeamCom SCE for automated generated services based on JSLEE“, *Proceedings of the 8th International Network Conference (INC 2010)*, pp. 75-84, Heidelberg, Germany, ISBN: 978-1-84102-259-8
35. El-Far, I.K. and Whittaker, J.A. (2001), “Model-Based Software Testing”, *Encyclopedia on Software Engineering*, Vol. 2, Wiley
36. Elvior (2015), “TestCast TTCN-3 Professional”, Available at: <http://www.elvior.com>, [accessed 30th October 2015]
37. Ernits, J.; Kull, A.; Raiend, K.; Vain, J. (2006), “Generating TTCN-3 Test Cases from EFSM Models of Reactive Software Using Model Checking”, *GI Jahrestagung (2)*, Vol. 94, pp. 241-248, GI
38. ETSI ES 201 873-1 V4.7.1 (2015), ETSI Standard, “Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language”, ETSI
39. ETSI ES 201 873-2 V3.2.1 (2007), ETSI Standard, “Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 2: TTCN-3 Tabular presentation Format (TFT)”, ETSI
40. ETSI ES 201 873-3 V3.2.1 (2007), ETSI Standard, “Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 3: TTCN-3 Graphical presentation Format (GFT)”, ETSI
41. ETSI ES 201 873-5 V4.7.1 (2015), ETSI Standard, “Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3, Part 5: TTCN-3 Runtime Interface (TRI)”, ETSI
42. ETSI ES 201 873-6 V4.7.1 (2015), ETSI Standard, “Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3, Part 6: TTCN-3 Control Interface (TCI)”, ETSI

43. ETSI ES 202 951 V1.1.1 (2011), ETSI Standard, “Methods for Testing and Specification (MTS); Model-Based Testing (MBT); Requirements for Modelling Notations”, ETSI
44. ETSI Tdoc RP 030375 V0.10 (2003), Technical Document, “Overview of 3GPP Release 5”, ETSI Mobile Competence Centre
45. ETSI TR 180 000 V1.1.1 (2006), Technical Report, “NGN Terminology”, ETSI TISPAN
46. ETSI TS 122 001 V10.0.0 (2011), Technical Specification, “Principles of circuit telecommunication services supported by a Public Land Mobile Network (PLMN)”, ETSI
47. ETSI TS 122 101 V10.7.0 (2011), Technical Specification, “Service aspects; Service principles”, ETSI
48. ETSI TS 122 105 V10.0.0 (2011), Technical Specification, “Services and service capabilities”, ETSI
49. ETSI TS 122 228 V10.4.1 (2011), Technical Specification, “Service requirements for the Internet Protocol (IP) multimedia core network subsystem (IMS)”, ETSI
50. Evans, E. J. (2003), “Domain-Driven Design: Tackling Complexity in the Heart of Software”, Addison-Wesley, Boston, USA, ISBN: 0-321-12521-5
51. Felderer, M.; Zech, P.; Fiedler, F.; Breu, R. (2010), “A Tool-Based Methodology for System Testing of Service-Oriented Systems“, *Proceedings of the 2nd International Conference on Advances in System Testing and Validation Lifecycle (VALID 2010)*, pp. 108-113, IEEE
52. Felderer, M.; Chimiak-Opoka, J.; Zech, P.; Haisjackl, C.; Fiedler, F.; Breu R. (2011), “Model Validation in a Tool-Based Methodology for System Testing of Service-Oriented Systems”, *Advances in Software*, Vol. 4 no 1 & 2, pp. 129-143, IARIA
53. Fischer, M; Toenjes, R.; Lasch, R. (2011), “A New Approach For Automatic Generation of Tests for Next Generation Network Communication Services“, *Processings of the 16th Conference on Emerging Technologies & Factory Automation (ETFA 2011)*, pp. 1-6, IEEE
54. Feudjio, A.-G.V. (2009), “Model-Driven Functional Test Engineering for Service Centric Systems”, *Proceedings of the 5th International Conference on Testbeds and Research Infrastructures for the Development of Networks & Communities and Workshops (TridentCom 2009)*, pp. 1-7, IEEE
55. Feudjio, A.-G.V. (2011), “A Methodology For Pattern-Oriented Model-Driven Testing of Reactive Software Systems”, Doctor of Engineering Dissertation, Faculty of Electrical Engineering and Computer Science, Technical University of Berlin, Germany
56. Fokus!MBT (2015), “Fokus!MBT”, Available at: http://www.fokusmbt.com/key_features/index.html, [accessed 24th April 2015]

-
57. Gärtner, M. (2012), “ATDD by Example: A Practical Guide to Acceptance Test-Driven Development”, Addison-Wesley, Upper Saddle River, USA, ISBN: 978-0-321-78415-5
 58. George, E. and Williams, L. (2004), “A structured experiment of test-driven development”, *Information and Software Technology*, Vol. 46, Issue 5, pp. 337-342, ACM
 59. Glitho, R. H.; Khendek, F.; De Marco, A. (2003), “Creating Value Added Services in Internet Telephony: An Overview and a Case Study on a High-Level Service Creation Environment”, *IEEE Transactions on Systems, Man, and Cybernetics – Part C: Applications and Reviews*, Vol. 33, No. 4, pp. 446-457, IEEE
 60. Glover, A. (2009), “Agile testing: a whole team approach”, Available at: <http://www.javaworld.com/article/2072867/agile-testing--a-whole-team-approach.html>, [accessed 30th October 2015]
 61. Gregory, J. and Crispin, L. (2015), “More Agile Testing”, Addison-Wesley, Upper Saddle River, USA, ISBN: 978-0-321-96705-3
 62. Guo, J; Chen, J.; Cheng, B.; Liu, D. (2009), “A Choreography Approach for Value-Added Services Creation”, *Proceedings of the 5th International Conference on Wireless Communications, Networks and Mobile Computing (WiCom 2009)*, pp. 1-4, IEEE
 63. Gutjahr, W.J. (1999), “Partition Testing vs. Random Testing: The Influence of Uncertainty”, *IEEE Transactions on Software Engineering*, Vol. 25, Issue 5, pp. 661-674, IEEE
 64. Hartmann, A. and Nagin, K. (2004), “The AGEDIS tools for model based testing”, *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software testing and analysis (ISSTA 2004)*, pp. 129-132, ACM
 65. Harel, D. (1996), “The STATEMATE Semantics of Statecharts”, *Transactions on Software Engineering and Methodology (TOSEM)*, Vol. 5, Issue 4, pp. 293-333, ACM
 66. Harel, D. and Kugler, H. (2004), “The RHAPSODY Semantics of Statecharts (or, On the Executable Core of the UML)”, *Integration of Software Specification Techniques for Applications in Engineering*, Vol. 3147, pp. 325-354, Springer
 67. Harel, D. and Politi, M. (1998), “Modeling Reactive Systems with Statecharts: The Statemate Approach (Software Development)”, McGraw-Hill Inc., New York City, USA, ISBN: 978-0-070-26205-8
 68. Harte, L.; Hoenig, M.; McLaughlin, D.; Kikta, R. (1999), “CDMA IS-95 for Cellular and PCS: Technology, Applications, and Resource Guide (1st edition)”, McGraw-Hill Professional, New York City, USA, ISBN: 978-0-0702-7070-1
 69. Haschemi, S. (2009), “Model Transformations to Satisfy All-Configurations-Transitions on Statecharts”, *Proceedings of the 6th International Workshop on Model-Driven Engineering (MoDeVVa 2009)*, ACM
-

70. IEEE Std 610.12 (1990), IEEE Standard, “IEEE Standard Glossary of Software Engineering Terminology”, IEEE
71. IEEE Std 829 (2008), IEEE Standard, “IEEE Standard for Software and System Test Documentation”, IEEE
72. IEEE Std 1490 (2011), IEEE Standard, “IEEE Guide – A Guide to the Project Management Body of Knowledge (PMBOK® Guide) – Fourth Edition”, IEEE
73. IETF RFC 1034 (1987), Request For Comments, “Domain Names – Concepts and Facilities”, IETF
74. IETF RFC 1035 (1987), Request For Comments, “Domain Names – Implementation and Specification”, IETF
75. IETF RFC 1939 (1996), Request For Comments, “Post Office Protocol – Version 3”, IETF
76. IETF RFC 2234 (1997), Request For Comments, “Augmented BNF for Syntax Specifications: ABNF”, IETF
77. IETF RFC 2543 (1999), Request For Comments, “SIP: Session Initiation Protocol (Version 1.0)”, IETF
78. IETF RFC 2616 (1999), Request For Comments, “Hypertext Transfer Protocol – HTTP/1.1”, IETF
79. IETF RFC 3261 (2002), Request For Comments, “SIP: Session Initiation Protocol”, IETF
80. IETF RFC 3262 (2002), Request For Comments, “Reliability of Provisional Responses in the Session Initiation Protocol (SIP)”, IETF
81. IETF RFC 3311 (2002), Request For Comments, “The Session Initiation Protocol (SIP) UPDATE Method”, IETF
82. IETF RFC 3428 (2002), Request For Comments, “Session Initiation Protocol (SIP) Extension for Instant Messaging”, IETF
83. IETF RFC 3515 (2003), Request For Comments, “The Session Initiation Protocol (SIP) Refer Method”, IETF
84. IETF RFC 3550 (2003), Request For Comments, “RTP: A Transport Protocol for Real-Time Applications”, IETF
85. IETF RFC 3725 (2004), Request For Comments, “Best Current Practices for Third Party Call Control (3pcc) in the Session Initiation Protocol (SIP)”, IETF
86. IETF RFC 3903 (2004), Request For Comments, “Session Initiation Protocol (SIP) Extension for Event State Publication”, IETF
87. IETF RFC 4353 (2006), Request For Comments, “A Framework for Conferencing with the Session Initiation Protocol (SIP)”, IETF
88. IETF RFC 4511 (2006), Request For Comments, “Lightweight Directory Access Protocol (LDAP): The Protocol”, IETF

-
89. IETF RFC 4566 (2006), Request For Comments, “SDP: Session Description Protocol”, IETF
 90. IETF RFC 5321 (2008), Request For Comments, “Simple Mail Transfer Protocol”, IETF
 91. IETF RFC 6086 (2011), Request For Comments, “Session Initiation Protocol (SIP) INFO Method and Package Framework”, IETF
 92. IETF RFC 6665 (2012), Request For Comments, “SIP-Specific Event Notification”, IETF
 93. ISO/IEC/IEEE 29119-2 (2013), International Standard, “Software and systems engineering – Software testing – Part 2: Test processes”, ISO/IEC/IEEE
 94. ITU (2011), Constitution and Convention, “Collection of the basic texts of the International Telecommunication Union adopted by the Plenipotentiary Conference”, ITU
 95. ITU-T I.210 (1993), Recommendation, “Principles of Telecommunication Services supported by an ISDN and the means to describe them”, ITU-T
 96. ITU-T I.211 (1993), Recommendation, “B-ISDN Service aspects”, ITU-T
 97. ITU-T M.3340 (2009), Recommendation, “Framework for NGN service fulfilment and assurance management across the business to business and customer to business interfaces”, ITU-T
 98. ITU-T M.3050.1 (2007), Recommendation, “Enhanced Telecom Operations Map (eTOM) – The business process framework”, ITU-T
 99. ITU-T Q.3946.1 (2013), Recommendation, “Conformance tests specification for the session initiation protocol – Part 2: Test suite structure and test purposes”, ITU-T
 100. ITU-T Q.3948 (2011), Recommendation, “Service testing framework for VoIP at the user-to-network interface of next generation networks”, ITU-T
 101. ITU-T Q.3949 (2012), Recommendation, “Real-time multimedia service testing framework at the user-to-network interface of next generation networks”, ITU-T
 102. ITU-T T.174 (1996), Recommendation, “Application Programming Interface (API) for MHEG-1”, ITU-T
 103. ITU-T X.680 (2015), Recommendation, “Abstract Syntax Notation One (ASN.1): Specification of basic notation”, ITU-T
 104. ITU-T Y.2001 (2004), Recommendation, “General overview of NGN”, ITU-T
 105. ITU-T Y.2012 (2010), Recommendation, “Functional requirements and architecture of next generation networks”, ITU-T
 106. ITU-T Z.100 (2007), Recommendation, “Specification and Description Language (SDL)”, ITU-T

107. Jacobson, I.; Christerson, M.; Jonsson, P. (1992), "Object-Oriented Software Engineering: A Use CASE Approach (ACM Press)", Addison-Wesley, ISBN: 978-0-201-54435-0
108. Janevski, T. (2014), "NGN Architectures, Protocols and Services", John Wiley & Sons Inc, Hoboken, USA, ISBN: 978-1-118-60720-6
109. JBoss (2015), "JBoss Developer", Available at: <http://www.jboss.org/>, [accessed 30th November 2015]
110. Jetty (2015), "Jetty – Servlet Engine and Http Server", Available at: <http://www.eclipse.org/jetty/>, [accessed 5th November 2015]
111. JSR 289 Spec (2008), "JSR 289: SIP Servlet v1.1", Java Specification Request
112. JUnit (2015), "JUnit", Available at: <http://junit.org>, [accessed 10th May 2015]
113. Karlesky, M.J.; Bereza, W.I.; Erickson, C.B. (2006), "Effective Test Driven Development for Embedded Software", *Proceedings of the IEEE International Conference on Electro/information Technology*, pp. 382-287, IEEE
114. Kühn, P. (1991), "Vorlesungsskript Nachrichtenvermittlung I und II" (translated title: "Lecture notes message switching I and II"), University Stuttgart, Institut für Nachrichtenvermittlung und Datenverarbeitung
115. Lehmann, A.; Eichelmann, T.; Trick, U.; Lasch, R.; Ricks, B.; Tönjes, R. (2009), "TeamCom: A Service Creation Platform for Next Generation Networks", *Proceedings of the 4th International Conference on Internet and Web Applications and Services (ICIW 2009)*, pp. 12-17, IEEE
116. Lehmann, A. (2014), "Service composition based on SIP peer-to-peer networks", PhD thesis, School of Computing and Mathematics, Plymouth University, United Kingdom
117. Ling, J.; Ping, P.; Chun, Y.; Jinhua, L.; Qiming, T. (2009), "Rapid Service Creation Environment for Service Delivery Platform based on Service Templates", *Proceedings of the International Symposium on Integrated Network Management (IM 2009)*, pp. 117-120, IEEE
118. Lombard Hill Group (2015), "Software Reuse 101: What Is Software Reuse?", Available at: <http://lombardhill.com/articles/software-reuse-101-what-is-software-reuse/>, [accessed 20th July 2015]
119. Magedanz, T. and de Gouveia, F.C. (2006), "IMS – the IP Multimedia System as NGN Service Delivery Platform", *Elektrotechnik & Informationstechnik*, Vol. 123, Issue 7-8, pp. 271-276, Springer
120. Malik, Q.A.; Jääskeläinen, A.; Virtanen, H.; Katara, M. (2010), "Model-Based Testing using System vs. Test Models – What is the Difference?", *Proceedings of the 17th IEEE International Conference and Workshops on Engineering of Computer-Based Systems (ECBS 2010)*, pp. 291-299, IEEE
121. Mathur, A.P. (2008), "Foundations of Software Testing", Pearson Education India, New Delhi, India, ISBN: 978-8-1317-0795-1

-
122. Menkens, C. (2010), "From service delivery to application delivery in the telecommunication industry", *Proceedings of the IEEE GLOBECOM Workshops (GC Wkshps 2010)*, pp. 1339-1344, IEEE
 123. Milner, R. (1989), "Communication and Concurrency", Prentice Hall, Upper Saddle River, New Jersey, USA, ISBN: 0-13-115007-3
 124. Milner, R. (1992), "Functions as processes", *Mathematical Structures in Computer Science*, Vol. 2, Issue 2, pp. 119-141, Cambridge University Press
 125. Milner, R.; Parrow, J.; Walker, D. (1992), "A calculus for mobile processes", *Information and Computation*, Vol. 100, Issue 1, pp. 1-40, Elsevier
 126. Mobicents (2015), "Mobicents – The Open Source Cloud Communications Platform", Available at: <http://www.mobicents.org/>, [accessed 30th November 2015]
 127. MongoDB (2015), "MongoDB", Available at: <https://www.mongodb.org/>, [accessed 3rd November 2015]
 128. OASIS (2007), OASIS Standard, "Web Services Business Process Execution Language Version 2.0", OASIS
 129. Obermann, K. and Horneffer, M. (2013), "Datennetztechnologien für Next Generation Networks (2nd edition)" (translated title: "Data network technologies for Next Generation Networks (2nd edition)"), Springer Vieweg, Wiesbaden, Germany, ISBN: 978-3-8348-2098-3
 130. OMA OSPE (2005), "OMA Service Provider Environment Requirements (Candidate Version 1.0)", OMA
 131. OMA ORG (2007), "Dictionary for OMA Specifications (Approved Version 2.6)", OMA
 132. OMG (2002), "CORBA 3.0 – OMG IDL Syntax and Semantics", Version 3.0
 133. OMG (2011a), "Unified Modeling Language (UML)", Version 2.4.1
 134. OMG (2012a), "OMG System Modeling Language (SysML)", Version 1.3
 135. OMG (2012b), "Service oriented architecture Modeling Language (SoaML)", Version 1.0.1
 136. OMG (2012c), "Common Object Request Broker Architecture (CORBA/IIOP)", Version 3.3
 137. OMG (2013a), "UML Testing Profile (UTP)", Version 1.2
 138. OMG (2013b), "UML Profile for Advanced and Integrated Telecommunication Services (TelcoML)", Version 1.0
 139. OMG (2014), "Object Constraint Language (OCL)", Version 2.4
 140. OpenTTCN (2015), "OpenTTCN Tester 2012", Available at: <http://www.openttcn.com/>, [accessed 30th October 2015]
-

141. Oracle (2010), “The SIP Servlet Tutorial”, Available at: <https://docs.oracle.com/cd/E19355-01/820-3007/gfmpq/index.html>, [accessed 30th of November 2015]
142. OSGi Alliance R5 (2012), “OSGi Core Release 5”, OSGi specification, Version 5.0.0
143. Pancur, M.; Ciglaric, M.; Trampus, M.; Vidmar, T. (2003), “Towards Empirical Evaluation of Test-Driven Development in a University Environment”, *EUROCON 2003 Computer as a Tool*, Vol. 2, pp. 83-86, IEEE
144. Pezzè, M. and Young, M. (2009), “Software testen und analysieren” (translated title: “Testing and analysing software”), Oldenbourg, Munich, Germany, ISBN: 3-486-58521-6
145. Poikselkä, M. and Mayer, G. (2009), “The IMS: IP Multimedia Concepts and Services (3rd edition)”, John Wiley & Sons Inc, Hoboken, USA, ISBN: 978-0-4707-2196-4
146. Pretschner, A.; Prenninger, W.; Wagner, S.; Kühnel, C.; Baumgartner, M.; Sostawa, B.; Zölch, R.; Stauner, T. (2005), “One Evaluation of Model-Based Testing and its Automation”, *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, pp. 392-401, ACM
147. RMI (2015), “Remote Method Invocation”, Available at: <http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html>, [accessed 12th of May 2015]
148. Ryan, M.D. and Smyth, B. (2011), “Applied pi calculus”, *Formel Models and Techniques for Analyzing Security Protocols*, V. Cortier and S. Kremer
149. Ryndina, O.; Kritzinger, P. (2004), “Improving Requirements Specification for Communication Services with Formalised Use Case Models”, *Proceedings of the Southern African Telecommunication Networks and Applications Conference (SATNAC)*, Spier Wine Estate, South Africa
150. Ryndina, O.; Kritzinger, P. (2005), “Analysis of Structured Use Case Models through Model Checking”, *South African Computer Journal*, Vol. 35, pp. 84-96, South African Computer Society
151. Salina, J.L. and Salina, P. (2007), “Next Generation Networks – Perspectives and Potentials”, John Wiley & Sons Inc, Hoboken, USA, ISBN: 978-0-470-51649-2
152. Solís, C. and Wang, X. (2011), “A Study of the Characteristics of Behaviour Driven Development”, *Proceedings of the 37th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA 2011)*, pp. 383-387, IEEE
153. Sommerville, Ian (2012), “Software Engineering (9th edition)”, Pearson Deutschland GmbH, Munich, Germany, ISBN: 978-3-8689-4099-2
154. Tahat, L.H.; Vaysbury, B.; Korel, B.; Bader, A.J. (2001), “Requirement-based automated black-box test generation”, *Proceedings of the 25th Annual*

-
- International Computer Software and Applications Conference (COMPSAC 2001)*, pp. 489-495, IEEE
155. Telling TestStories (2015), “Telling TestStories”, Available at: <http://teststories.info>, [accessed 25th April 2015]
 156. Toral-Cruz, H.; Argaez-Xool, J.; Estrada-Vargas, L.; Torres-Roman, D. (2011), “An Introduction to VoIP: End-to-End Elements and QoS Parameters”, *VoIP Technologies*, Vol. 1, Issue 4, pp. 79-94, InTech
 157. Trick, U. and Weber, F. (2004), “SIP, TCP/IP und Telekommunikationsnetze (1st edition)” (translated title: “SIP, TCP/IP and Telecommunication Networks (1st edition)”), Oldenbourg, Munich, Germany, ISBN: 3-486-27529-1
 158. Trick, U. and Weber, F. (2009), “SIP, TCP/IP und Telekommunikationsnetze (4th edition)” (translated title: “SIP, TCP/IP and Telecommunication Networks (4th edition)”), Oldenbourg, Munich, Germany, ISBN: 3-486-59000-5
 159. Trick, U. and Weber, F. (2015), “SIP und Telekommunikationsnetze (5th edition)” (translated title: “SIP and Telecommunication Networks (5th edition)”), De Gruyter Oldenbourg, Berlin, Germany, ISBN: 3-486-77853-3
 160. TT-Medal Consortium (2005), “A Vision for Automated Testing [White Paper]”, Available at: <http://www.testingtech.com/download/publications/TTmedalWhitePaper.pdf>, [accessed 14th June 2014]
 161. TT-Medal ITEA (2005), “Advanced Test Processes using TTCN-3”, Available at: <https://itea3.org/project/result/download/5565/TT-Medal%20Innovation%20Report.pdf>, [accessed 14th June 2014]
 162. TTCN-3 (2015), “TTCN-3 Test System Reference Architecture”, Available at: <http://www.ttcn-3.org/index.php/about/reference-architecture>, [accessed 10th October 2015]
 163. TTsuite-SIP (2015), “TTsuite-SIP – Analyzing Internet System Components, Voice-over-IP, and SIP, the 3G Signaling Protocol”, Available at: <http://www.testingtech.com/solutions/ttsuite-sip.php>, [accessed 12th October 2015]
 164. TTworkbench (2015), “TTworkbench – The Reliable Test Automation Platform”, Available at: <http://www.testingtech.com/products/ttworkbench.php>, [accessed 10th May 2015]
 165. TTworkbench User Guide (2015), “TTworkbench 20 User’s Guide”, Available at: http://www.testingtech.com/download/users_guides/TTworkbench_UserGuide.pdf, [accessed 5th November 2015]
 166. UTML (2015), “Die UTML Notation für Musterbasierte Modelgetriebene Entwicklung von Tests” (translated title: “The UML notation for pattern-based model-driven development of tests”), Technology Article, Available at: http://www.wold.fokus.fraunhofer.de/de/sqc/wir_bieten/technologien/utml/index.html, [accessed 13th July, 2015], Fraunhofer FOKUS
-

167. Utting, M. and Legeard, B. (2006), “Practical Model-Based Testing: A Tools Approach”, Morgan Kaufmann Publishers Inc., San Francisco, USA, ISBN: 978-0-1237-2501-1
168. Van Deursen, A. (2001), “Program Comprehension Risks and Opportunities in Extreme Programming”, *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE 2001)*, pp. 176-185, IEEE
169. W3C (2012), Recommendation, “W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures”, W3C
170. W3C (2015), Recommendation, “State Chart XML (SCXML): State Machine Notation for Control Abstraction”, W3C
171. Wacht, P.; Lehmann, A.; Eichelmann, T.; Trick, U.; Fischer, F.; Lasch, R.; Toenjes, R. (2010), “Ein neues Verfahren zum automatisierten Testen von Mehrwertdiensten“ (translated title: “A new method to automated testing of value-added services”), *Proceedings of the Fifteenth VDE/ITG Mobilfunktagung*, pp. 73-79, VDE
172. Wacht, P.; Eichelmann, T.; Lehmann, A.; Trick, U. (2011a), “A New Approach to Design Graphically Functional Tests for Communication Services“, *Proceedings of the 4th IFIP International Conference on New Technologies, Mobility and Security (NTMS 2011)*, pp. 1-5, IEEE
173. Wacht, P.; Eichelmann, T.; Lehmann, A.; Fuhrmann, W.; Trick, U.; Ghita, B. (2011b), “A New Approach to model a formalised Description of a Communication Service for the Purpose of Functional Testing“, *Proceedings of the 4th International Conference on Internet Technologies & Applications (ITA 2011)*, Wrexham, UK, ISBN: 978-0-946881-68-0
174. Wacht, P.; Lehmann, A.; Eichelmann, T.; Trick, U. (2011c), “ComGeneration: die Dienstbeschreibung als Basis für automatisierte Tests“ (translated title: “ComGeneration: a service description as basis for automated tests”), *Proceedings of the Fifteenth VDE/ITG Mobilfunktagung*, pp. 118-123, VDE
175. Weber, F. (2012), “Quality of Service optimization framework for Next Generation Networks”, PhD thesis, School of Computing and Mathematics, Plymouth University, UK
176. Wendland, M.-F.; Hoffmann, A.; Schieferdecker, I. (2013), “Fokus!MBT – A Multi-Paradigmatic Test Modeling Environment“, *Proceedings of the Workshop on ACadeMics Tooling with Eclipse (ACME 2013)*, ACM
177. Willcock, C.; Deiß, T.; Tobies, S.; Keil, S.; Engler, F.; Schulz, S. (2011), “An Introduction to TTCN (2nd edition)”, Wiley Publishing, Chichester, West Sussex PO19 8SQ, England
178. Yenduri, S. and Perkins, L. (2006), “Impact of Using Test-Driven Development: A Case Study”, *Proceedings of the International Conference on Software Engineering Research and Practice (SERP 2006)*, pp.126-129, CSREA Press
179. Yue, T.; Briand, L.C.; Labiche, Y. (2009), “A Use Case Modeling Approach to Facilitate the Transition Towards Analysis Models: Concepts and Empirical

- Evaluation”, *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MODELS 2009)*, pp. 484-498, ACM/IEEE
180. Yue, T.; Ali, S.; Briand, L.C. (2011), “Automated Transition from Use Cases to UML State Machines to Support State-based Testing”, Technical Report 2011-05, University of Oslo, Norway
181. Zander, J.; Dai, Z.R.; Schieferdecker, I.; Din, G. (2005), “From U2TP Models to Executable Tests with TTCN-3 – An Approach to Model Driven Testing”, *Proceedings of the 17th International Conference on Testing of Communication Systems (TestCom 2005)*, pp. 289-303, Springer
182. Zhang, G.; Yue, T.; Ali, S. (2013), “Modeling Crisis Management Systems with the Restricted Use Case Modeling Approach”, *Proceedings of the 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013)*, ACM/IEEE
183. Xiaoping, C. and Maag, S. (2013), “Passive Testing on Performance Requirements of Network Protocols”, *Proceedings of the 27th International Conference on Advanced Information Networking and Applications Workshop (WAINA 2013)*, pp. 1439-1444, IEEE

Appendix A – Abbreviations

3GPP Third Generation Partnership Project

A

ACE Automatic Composition Engine
ACM Association for Computing Machinery, Inc.
ALG Application Layer Gateway
AML AGEDIS Modeling Language
API Application Programming Interface
AS Application Server
ASN.1 Abstract Syntax Notation One
ASM Abstract State Machines
ATDD Acceptance Test-Driven Development
ATS Abstract Test Suite

B

B2BUA Back-to-Back User Agent
BDD Behaviour-Driven Development
BPEL Business Process Execution Language

C

CCS Calculus of Communication Systems
CCXML Call Control eXtensible Markup Language
CD Codec
CH Component Handling
CLF Campaign Loader File
CS Call Server

D

DNS	Domain Name System
DTMF	Dual-tone multi-frequency signaling

E

ETSI	European Telecommunications Standards Institute
EFSM	Extended Finite State Machine
EMF	Eclipse Modeling Framework
ETS	Executable Test Suite

F

FSM	Finite State Machine
-----	----------------------

G

GUI	Graphical user interface
-----	--------------------------

H

HTTP	Hypertext Transfer Protocol
------	-----------------------------

I

IDL	Interactive Data Language
IEEE	Institute of Electrical and Electronics Engineering
IEC	International Electrotechnical Commission
ICT	Information and Communications Technology
IETF	Internet Engineering Task Force
IMS	IP Multimedia Subsystem
IP	Internet Protocol
ISO	International Organization for Standardization
ITEA	Information Technology for European Advancement
ITU	International Telecommunication Union
IVR	Interactive Voice Response

J

JAR	Java Archive
JSR	Java Specification Request

K**L****M**

MBT	Model-Based Testing
MDTE	Model-Driven Test Engineering
MGW	Media Gateway
MMS	Mobicents Media Server
MOF	Meta Object Facility
MTC	Main Test Component

N

NGN	Next Generation Networks
-----	--------------------------

O

OASIS	Organization for the Advancement of Structured Information Standards
OMA	Open Mobile Alliance
OMG	Object Management Group
OSI	Open Systems Interconnection

P

PA	Platform Adapter
PSTN	Public Switched Telephone Network
PTC	Parallel Test Component

Q

QoS	Quality of Service
-----	--------------------

R

RATS	Requirements Acquisition and specification of Telecommunication Services
RFC	Request for Comments
RTP	Real-Time Transport Protocol
RUCM	Restricted Use Case Modeling

S

SA	SUT Adapter
SAP	Service Access Point
SBC	Session Border Controller
SCXML	State Chart extensible Markup Language
SDK	Software Development Kit
SDL	Specification and Description Language
SDP	Service Delivery Platform
SEE	Service Execution Environment
SGW	Signalling Gateway
SIP	Session Initiation Protocol
SCTP	Stream Control Transmission Protocol
SOA	Service-Oriented Architecture
SoAML	Service oriented architecture Modeling Language
SQG	Service Quality Group
STD	Service Test Description
SUT	System/Service under Test
SysML	Systems Modeling Language

T

TelcoML	Telecommunication Modeling Library
TCF	Test Creation Framework
TCI	TTCN-3 Control Interface
TRI	TTCN-3 Runtime Interface
TCP	Transmission Control Protocol

TCU	Test Configuration Unit
TCDU	Test Case Derivation Unit
TE	TTCN-3 Executable
TEE	Test Execution Environment
TFUT	Test Framework User Terminal
TDD	Test-Driven Development
TLS	Transport Layer Security
TM	Test Management
TMC	Test Management & Control
TME	Test Modules Environment
TMR	Test Modules Repository
TPTP	Test and Performance Tools Platform
TSB	Test Suite Builder
TTCN	Testing and Test Control Notation
U	
U2TP	UML 2.0 Testing Profile
UA	User Agent
UAC	User Agent Client
UAS	User Agent Server
UDP	User Datagram Protocol
UMTS	Universal Mobile Telecommunication System
URI	Uniform Resource Identifier
UTML	Unified Test Modeling Language
V	
W	
W3C	World Wide Web Consortium

X

XML eXtensible Markup Language

XSD XML Schema Descriptor

Appendix B – Own Publications

Published in ITG-Fachbericht Mobilfunk (Mobilfunktagung 2010), pp. 73-80, University of Applied Sciences Osnabrück, Germany, ISBN: 978-3-8007-3269-3

Ein neues Verfahren zum automatisierten Testen von Mehrwertdiensten

Patrick Wacht¹, Armin Lehmann¹, Thomas Eichelmann¹, Ulrich Trick¹, Marten Fischer², Rolf Lasch², Ralf Toenjes²

¹Forschungsgruppe und Labor für Telekommunikationsnetze, Fachhochschule Frankfurt/Main

²Labor für Hochfrequenztechnik und Mobilkommunikation, Fachhochschule Osnabrück

Kurzfassung

Um eine gute Qualität entwickelter Mehrwertdienste zu erreichen, ist sorgfältiges Testen erforderlich. Dies kann je nach Vorgehensweise sehr hohe Kosten verursachen. In diesem Aufsatz wird daher ein Ansatz für ein Testverfahren beschrieben, das ein systematisches und effektives Testen von Diensten mit relativ geringem Aufwand ermöglicht. Dabei beruht das Verfahren darauf, dass ausgehend von einem Zustandsautomaten, welcher den zu testenden Dienst beschreibt, automatisch eine Testfallmenge generiert wird und mit Hilfe dieser Testfallmenge der Dienst automatisch getestet und das Ergebnis des Tests analysiert werden kann.

1 Einleitung

von Multimedia-Kommunikationsdiensten vereinfacht und damit Zeit- und Kostenaufwand reduziert.

1.1 Motivation und Überblick

In naher Zukunft besteht bei den Netzbetreibern und Diensteanbietern ein großer Bedarf, schnell, einfach und kostenoptimiert neue Dienste, sogenannte Mehrwertdienste, anbieten zu können. Hauptgrund hierfür ist, dass mit normalen Telefongesprächen kaum noch Einnahmen erzielt werden können und daher auf Basis der durch NGN (Next Generation Networks) gegebenen neuen Dienstmöglichkeiten neue Einnahmequellen erschlossen werden müssen. Zudem entstehen bei den Kunden durch die zukünftig in diesem Bereich prinzipiell unbegrenzten technischen Möglichkeiten auch neue Kommunikationsbedürfnisse. Daher ist es für Netzbetreiber und Dienste-Provider außerordentlich wichtig, leistungsfähige Dienstplattformen, sogenannte Service Creation and Delivery-Plattformen, zur Verfügung zu haben, mit denen in kürzester Zeit mit geringstem Aufwand neue Anwendungen entwickelt und im Markt eingeführt werden können.

Ein großes Problem bei der Entwicklung von Diensten bleibt aber das Testen der neu entwickelten Dienste, zumal es aufgrund wachsender Komplexität und geringerer Entwicklungszeit bis zur Einführung eines Dienstes zunehmend an Bedeutung gewinnt. Es ist jedoch noch nicht geklärt, wie Testverfahren in die Dienstentwicklung, und insbesondere in eine Dienstentwicklungsumgebung, systematisch integriert werden können. Im BMBF-Projekt ComGeneration (Testgesteuerte Evolution und automatisierte Bereitstellung von Kommunikationsdiensten) [1] soll daher eine durchgängige Lösung erarbeitet werden, die den ganzen Lebenszyklus eines Dienstes unterstützt, indem sie die Entwicklung, den Test und die Bereitstellung

1.2 Stand der Technik

Von diesen Anforderungen ausgehend wurde im Rahmen des BMBF-Forschungsprojekts TeamCom (IMS- oder P2P-basierte Dienstebereitstellung und –entwicklung für kundenspezifische Kommunikationsprozesse) [2] ein neues Verfahren zur automatisierten Entwicklung von Mehrwertdiensten entwickelt und praktisch verifiziert. Hier wurden in einem ersten Schritt aus realistischen Kommunikationsabläufen einzelne, häufig wiederkehrende Kommunikationsbausteine abgeleitet. Diese Kommunikationsbausteine werden nun bei der Erstellung von neuen und komplexen Mehrwertdiensten immer wieder verwendet und können maßgeblich zu einer schnellen und kostengünstigen Dienstentwicklung beitragen. In der Folge war es das Ziel, ausgehend von einer einfachen textuellen Beschreibung teilautomatisiert den gewünschten Mehrwertdienst auf einem JAIN SLEE AS (Java API for Integrated Networks Service Logic Execution Environment Application Server) im Netz bereitzustellen. Dabei wird – wie im unteren Teil von Bild 1 dargestellt – die genannte Dienstskeizze mittels BPEL (Business Process Execution Language) [3] formal beschrieben, wobei u.a. auf die oben genannten Kommunikationsbausteine zurückgegriffen wird.

Ein Compiler erzeugt aus der BPEL-basierten Dienstbeschreibung den auf dem JSLEE Application Server ausführbaren Java-Code. Hierzu werden die aus dem BPEL-Prozess resultierenden XML-Dateien mit Hilfe eines Codegenerators in Java-Klassen und XML-Beschreibungsdateien (Deskriptoren) transformiert. Der Dienst besteht aus den Java-Klassen, die den zugehörigen JSLEE Service Building Block (SBB) rep-

räsentieren, und aus dazugehörigen Deskriptoren. In der Folge werden die Java-Klassen mittels Codegenerator kompiliert, die resultierende Software ist auf einem JAIN SLEE Application Server lauffähig und realisiert den neuen Mehrwertdienst.

In einem weiteren Schritt geht es nun darum, durch Qualitätssicherung in Form von automatisierten Softwaretests zu gewährleisten, dass die realisierten Dienste den Anforderungen genügen. Um diese Automatisierung zu gewährleisten, wird dieser Ansatz mit TTCN-3 (Testing and Test Control Notation Version 3) [4] realisiert. Hierbei handelt es sich um eine abstrakte Testbeschreibungssprache, welche von der ETSI [5] und der ITU-T [6] normiert wurde. In der aktuellen Version unterstützt sie die modularisierte Erstellung von Testscenarien für nachrichtenbasierte oder prozedurale Systeme. Der Standard gibt dabei nicht nur die Notation vor, in der Tests geschrieben werden, sondern beschreibt auch Schnittstellen, mit denen die Tests nach außen kommunizieren.

1.3 Gliederung

Das folgende Kapitel 2 behandelt das Konzept des Projektes und geht näher darauf ein, wie automatisierte Testfälle aus Zustandsautomaten gewonnen werden können. Kapitel 3 geht näher auf die Struktur der entwickelten modularen Teilzustandsautomaten ein und in Kapitel 4 wird der komplette Ansatz beispielhaft an einem Dienst veranschaulicht.

2 Modellbasiertes Testen

2.1 Konzept

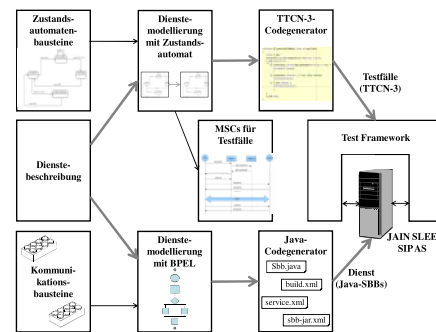


Bild 1 Verfahren zum automatisierten Entwickeln und Testen von Kommunikationsservicemehrwertdiensten

Das Ziel ist es, ein neues Verfahren zum automatisierten Testen der auf einem SIP AS (Session Initiation

Protocol) bereitgestellten Mehrwertdienste zu entwickeln. Hierbei wird Wert darauf gelegt, dass auch das BPEL-Design verifiziert wird, daher wird gemäß Bild 1 ebenfalls von der ursprünglichen, vom zukünftigen Dienstnutzer gelieferten Dienstbeschreibung ausgegangen. In der Folge erstellt der Testentwickler, wie im oberen Teil des Bildes 1 dargestellt, aus verfügbaren, zuvor erarbeiteten wiederverwendbaren Zustandsautomatenbausteinen (z.B. für bestimmte Kategorien und Protokolle wie SIP, TCP, HTTP oder Datenbanken) einen den Dienst aus Testsicht beschreibenden Zustandsautomaten. Alle möglichen zu durchlaufenden Zustände und Zustandsübergänge inkl. der diese hervorruhenden Events (Eingangssignale) und resultierenden Ausgangssignale vom Starten bis zum Beenden des Zustandsautomaten beschreiben die zu testenden Nachrichtenabläufe, die z.B. in Form von MSCs (Message Sequence Charts) beschrieben werden können. Da jedoch die resultierenden Testfälle innerhalb eines realen Test Frameworks auf den Dienst und damit den SIP AS angewandt werden sollen, wird auf Basis der parametrisierten Zustandsautomaten mittels eines entsprechenden Codegenerators TTCN-3-Code generiert.

Der grundlegende Ablauf beim modellbasierten Testen, der auch beim ComGeneration-Ansatz zum Tragen kommt, wird anhand des Bildes 2 verdeutlicht und wird in den folgenden Abschnitten näher beschrieben.

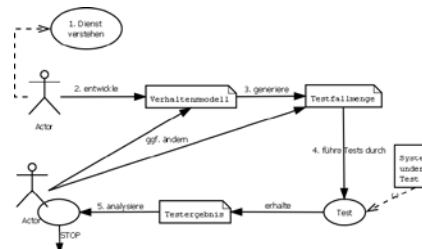


Bild 2 Vorgehensweise beim modellbasierten Testen

2.2 Entwicklung des Verhaltensmodells

Um einen Dienst testen zu können, muss der Tester wissen, wie der Dienst sich den Anforderungen nach verhalten sollte. Um das Verhalten des Dienstes zu verstehen, sollte der Tester auf alle verfügbaren Hilfsmittel zurückgreifen. Dazu zählen beispielsweise Dokumente, wie die Anforderungsbeschreibung, Use Cases, die Spezifikation der Software und weitere Entwurfsdokumente. Aus den gegebenen Informationen über den Dienst erfährt der Tester, welche Eingaben unter welchen Bedingungen möglich sind, welche Protokolle bei der Implementierung verwendet wer-

den und wie die entsprechende Ausgabe bzw. Reaktion des Dienstes aussehen sollte. Auf der Grundlage seines Wissens über den Dienst kann der Tester dann ein entsprechendes Verhaltensmodell erzeugen.

Zunächst besteht die Aufgabe des Testers darin, sich zu entscheiden, welches Verhaltensmodell er für seine Zwecke nutzen möchte. Für das ComGeneration-Projekt wurden mehrere Modelle evaluiert, z.B. zahlreiche UML-Diagramme sowie Zustandsautomaten. Die Kriterien für den Einsatz von Zustandsautomaten haben überwogen, da aus diesen beispielsweise bereits Algorithmen zur Verfügung stehen, um automatisiert Testfälle ableiten zu können. Außerdem existieren für zahlreiche standardisierte Protokollabläufe bereits Zustandsautomaten.

Im Allgemeinen handelt es sich bei Zustandsautomaten um ein verbreitetes Beschreibungsmittel für gedächtnisbehaftetes Dienstverhalten. Ein typischer Vertreter der Zustandsautomaten sind die deterministischen, endlichen Zustandsautomaten. Sie haben eine endliche Anzahl von Zuständen. Determinismus bedeutet, dass in jedem Zustand für jede gültige Eingabe aus dem Eingabealphabet eindeutig eine Transition bestimmt wird, die diesen Zustand verlässt. Im ComGeneration-Projekt wurden ferner erweiterte Zustandsautomaten verwendet, welche die Möglichkeit bieten, Bedingungen bezüglich Variablen zum Schalten von Transitionen zu definieren. Diese liegen meist in Form von Boole'schen Ausdrücken vor.

Wie bereits erwähnt, ist ein weiterer positiver Aspekt von Zustandsautomaten, dass für zahlreiche Protokolle, die von einem Dienst genutzt werden, bereits standardisierte Abläufe in Form von Zustandsautomaten existieren. Diese Abläufe dienen im ComGeneration-Projekt als Grundlage für die Definition von modularen wiederverwendbaren Teilzustandsautomaten, welche dem Tester in einem Repository zur Verfügung gestellt werden. Der Tester kann nun das Verhaltensmodell erzeugen, indem er die relevanten Teilzustandsautomaten auf Basis des von ihm angenommenen Dienstverhaltens komponiert. Das entstandene Verhaltensmodell ist dann je nach Komplexität des Dienstes ein mehr oder weniger umfangreicher Zustandsautomat.

2.3 Generierung der Testfallmenge

Da nun das Verhaltensmodell in Form eines Zustandsautomaten vorliegt, können die relevanten Testfälle abgeleitet werden. Ein Testfall ist spezifiziert durch den Anfangszustand des Systems und der Umgebung, die Werte aller Eingabedaten, die notwendigen Bedingungen und die erwarteten Ausgaben [7]. Im Kontext von Zustandsautomaten entsprechen Testfälle den Pfaden durch einen Automaten. Eine Testfallmenge ist

eine endliche Menge solcher Testfälle. Diese Testfallmenge sollte anhand von festgelegten Kriterien generiert werden, um einerseits anhand einer festgelegten Systematik testen zu können und andererseits eine Abschätzung machen zu können, was getestet wurde und wo noch Fehlerpotential vorhanden ist. Im Folgenden werden Kriterien beschrieben, die bei der Ermittlung der Testfallmenge verwendet werden könnten [8] [9].

Das einfachste Abdeckungskriterium für endliche Zustandsautomaten ist die *Zustandsüberdeckung*. Die Zustandsüberdeckung hat das Ziel, dass jeder Zustand, in dem sich ein zu testendes System (der Dienst) befinden kann, mindestens einmal besucht wird.

Um ein etwas umfangreicheres Abdeckungskriterium handelt es sich bei der *Ereignisüberdeckung*. Hierbei sollte für jeden Zustand, in dem sich das System befindet, jedes in diesem Zustand ausführbare Ereignis mindestens einmal durchgeführt werden. Für einen endlichen Zustandsautomaten entspricht dieses Kriterium einer Kantenüberdeckung, d.h., da jede Kante des Automaten einem Ereignis entspricht, sollte jedes Ereignis mindestens einmal eintreten.

Ein noch mächtigeres Kriterium als die Ereignisüberdeckung ist die *Überdeckung der Kombination von Ereignissen* [8]. Dieses Kriterium ist erfüllt, wenn jede eingehende Kante eines Zustands in Kombination mit jeder ausgehenden Kante ausgeführt wird. Dies bedeutet für einen Zustand z mit x eingehenden Kanten und y ausgehenden Kanten, dass jede mögliche Sequenz (eingehende Sequenz, ausgehende Kante) mindestens einmal ausgeführt wird, also der Zustand z genau $x*y$ Mal innerhalb eines Tests entsprechend einer solchen Testfallmenge durchlaufen wird.

Das stärkste Überdeckungskriterium ist die *Pfadüberdeckung*, bei der jede mögliche Abfolge von Ereignissen mindestens einmal durchgeführt wird. Eine derartige Überdeckung ist in der Regel bei etwas komplexeren Systemen nicht mehr durchführbar. Ein System ist für das Kriterium der Pfadüberdeckung schon dann zu komplex, wenn es dynamische Schleifen enthält. Weitere komplexe Systeme sind nebenläufige und nicht-deterministische Systeme. Ebenso können zu viele Zustände und Kanten eine zu große Testfallmenge erzeugen und so den zeitlichen Rahmen, in dem die Tests durchgeführt werden sollen, sprengen.

Unabhängig davon, welches der hier aufgeführten Überdeckungskriterien verwendet wird, entsteht daraus eine Menge an Testfällen. Aus jedem ermittelten Testfall kann anhand der Nachrichtenfolge auch ein Message Sequence Chart abgeleitet werden.

Wenn die Testfallmenge vollständig ist, wird der eigentliche Test durchgeführt. Dieser Schritt beinhaltet beim ComGeneration-Ansatz zunächst eine Umsetzung der identifizierten Testfälle in TTCN-3-Quellcode über einen Codegenerator. Wenn die Umsetzung erfolgt ist, wird der erzeugte TTCN-3-Code in eine Testumgebung eingebunden, das "System unter Test" wird überprüft, und ein Testergebnis wird ausgegeben. Bei der Analyse der Testergebnisse wird das tatsächliche mit dem erwarteten Systemverhalten verglichen. Weicht das tatsächliche Verhalten vom erwarteten ab, muss überprüft werden, ob das Verhaltensmodell korrekt ist. Gegebenfalls muss dies verändert werden. Andernfalls ist das abweichende tatsächliche Verhalten als Fehler des "Systems unter Test" zu werten.

3 Modulare Teilzustandsautomaten

Im vorigen Kapitel wurde bereits erwähnt, dass das vom Tester zu entwickelnde Verhaltensmodell des Dienstes ein Zustandsautomat ist, der durch die Komposition von modularen Teilzustandsautomaten entsteht. Diese Teilzustandsautomaten entstammen bestimmten Kategorien bzw. Protokollen (z.B. SIP, TCP, Datenbanken etc.).

Die Verknüpfung der Teilzustandsautomaten stellt bei der Modellierung eines Verhaltensmodells durch einen Tester die einzige veränderliche Komponente dar, die internen Abläufe der Teilzustandsautomaten sind unveränderlich. Um zu gewährleisten, dass aus der Verknüpfung der Teilzustandsautomaten ein Gesamtzustandsautomat entsteht, wurde der so genannte Transaction User (TU) eingeführt. Der TU versteht sich als die Vermittlungseinheit zwischen den Rollen eines Application Servers als User Agent Server und User Agent Client (bzw. als Client und Server beispielsweise bei TCP-Verbindungen) und gleichzeitig als Bindeglied zwischen den kleinteiligen Teilzustandsautomaten. Nur mittels des TUs kann ein Testentwickler die Logik eines bestimmten Dienstes abbilden.

Nachfolgend sind in Bild 3 beispielhaft drei Teilzustandsautomaten dargestellt, welche aus dem SIP-Repository entnommen und vom SIP-Standard abgeleitet sind (RFC3261 [10]). Bei diesen Teilzustandsautomaten handelt es sich im engeren Sinn um Templates, bei denen abstrakt die möglichen Nachrichtenverläufe dargestellt sind. Konkretisiert werden die Templates erst, wenn sie von Testern eingesetzt werden, um aus den Automaten Vorgehensmodelle für Dienste zu erzeugen.

Insgesamt sind zum aktuellen Zeitpunkt insgesamt 13 Teilzustandsautomaten für das SIP-Protokoll definiert

worden. Die in Bild 3 dargestellten Teilzustandsautomaten beschreiben die Nachrichtenfolgen beim User Agent Server für die SIP-Methode INVITE.

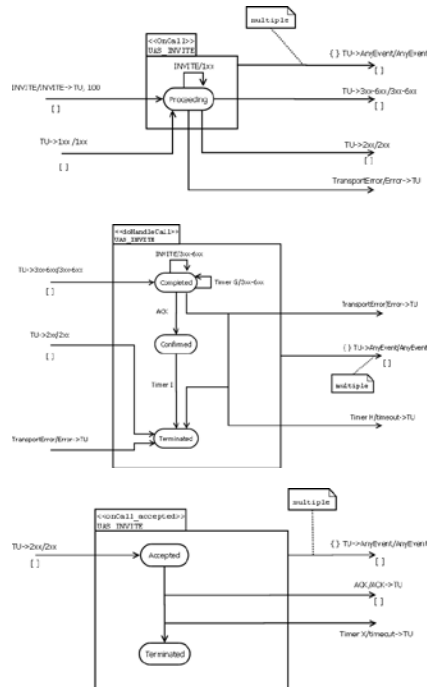


Bild 3 Modulare Teilzustandsautomaten zu SIP-INVITE für den User Agent Server

Die jeweiligen internen Zustandsübergänge sind fest und werden bei der Nutzung des jeweiligen Teilzustandsautomaten immer verwendet. Die Schnittstellen der Teilzustandsautomaten nach außen müssen vom Tester bearbeitet werden. Protokollinterne Übergänge (z.B. die Statusnachrichten 2xx oder 3xx-6xx) können verwendet werden, um Teilzustandsautomaten aus einem Repository miteinander zu verknüpfen. Eine solche Statusnachricht (z.B. TU->3xx-6xx) ist z.B. ein Ausgang des *onCall*-Teilzustandsautomaten und gleichzeitig Eingang des *doHandleCall*-Teilzustandsautomaten. Über diese identische Schnittstelle können die beiden Teilzustandsautomaten verbunden werden, wenn ein solches Verhalten vom Dienst erwartet wird. Genauso verhält es sich mit weiteren Nachrichten.

Bei der Testentwicklung kann es zwischenzeitlich von entscheidender Bedeutung sein, dass für einen Dienst konkrete Statusmeldungen angegeben werden können, um den korrekten Dienstablauf in dem Zustandsautomaten darstellen zu können. Dies wird in den Teilzu-

nun vom Testentwickler die drei folgenden möglichen Ausgänge definiert werden:

1. Es kommt direkt zu einem Transportfehler, weil keine Nachricht empfangen wurde. Der TU wird benachrichtigt, der Übergang erfolgt in den terminate-Zustand des Teilzustandsautomaten UAS_INVITE (doHandleCall).
2. Möglicherweise ist die URI des Konsumenten nicht registriert, sodass er den Dienst nicht nutzen darf. Dies führt dazu, dass durch den TU eine 400er-Statusmeldung initiiert wird. Der Zustandsübergang erfolgt in den complete-Zustand des Teilzustandsautomaten doHandleCall.
3. Der letzte Fall spiegelt den Erfolgsfall wieder. Hier ist die URI registriert und ein Transportfehler bleibt aus. Als Folge initiiert der TU einen TCP-Verbindungsaufbau zur Datenbank, es kommt zu einem Wechsel in den Teilzustandsautomaten TCP_Connect.

Sobald das SYN-Ereignis erfolgt ist, beginnt der Ablauf innerhalb von TCP_Connect. Nun wird eigentlich nur unterschieden, ob der Verbindungsaufbau erfolgreich ist oder nicht. Falls er nicht erfolgreich ist, wird der TU entsprechend informiert, dass es zu einem timeout gekommen ist. In diesem Fall initiiert der TU eine 500er-Statusmeldung, der Übergang erfolgt in den complete-Zustand des UAS_Invite (doHandleCall). Der Erfolgsfall wird erreicht, sobald der TU das ACK vom DB-Client erhalten hat, welcher für den Aufbau der Verbindung zur Datenbank zuständig ist. Daraufhin erzeugt der TU ein Query-Statement und schickt dieses an den DB-Client, dann erfolgt ein Zustandswechsel zum Teilzustandsautomaten DB_Request.

Im Teilzustandsautomat DB_Request führt ein Transportfehler führt dazu, dass der TU wiederum eine 500er-Statusmeldung initiiert, welche dann über den UAS an den User Agent des Konsumenten herausgeschickt wird. Sowohl ein Timeout bei der Datenbankabfrage als auch das korrekte Erhalten einer Antwort auf das Query Statement führen dazu, dass durch den TU der TCP-Verbindungsabbau zur Datenbank durch ein FIN-Ereignis eingeleitet wird. Dieser Übergang führt zu dem entsprechenden Teilzustandsautomaten TCP_Disconnect.

Auch bei diesem Teilzustandsautomaten können die bekannten Fälle des TransportError und timeout auftreten. Allerdings hat ein Verbindungsabbau grundsätzlich keinen Einfluss darauf, ob der Dienst korrekt funktioniert, weil der Abbau nicht relevant ist. So findet sich beim letzten Übergang vom TCP_Disconnect zu doHandleCall nur eine Verbindung. Bei diesem Übergang, welcher in den complete-Zustand von do-

HandleCall mündet, initiiert der TU entweder eine 500er-Statusmeldung, wenn kein Eintrag in der Datenbank gefunden wurde oder im anderen Fall eine 302er-Statusmeldung. Durch diese Statusmeldung wird die Rufumleitung eingeleitet, dies repräsentiert den Erfolgsfall.

Die Verknüpfung der Teilzustandsautomaten ist beendet und das Verhaltensmodell in Form eines Gesamtzustandsautomaten für den Redirect Service liegt vor. Das folgende Bild 5 zeigt nun, wie der Erfolgsfall als Testfall in einem Message Sequence Chart dargestellt werden kann.

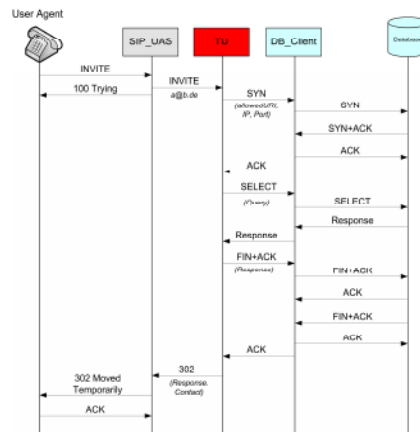


Bild 5 Message Sequence Chart zum Erfolgsfall des Redirect Service

Je nachdem, welche Teilzustandsautomaten im Verhaltensmodell verwendet wurden, werden die zugehörigen Rollen des Application Servers in dem MSC integriert. Da zwei UAS_INVITE-Teilzustandsautomaten verwendet wurden, wird die Rolle SIP_UAS benötigt. Außerdem wird die Rolle DB Client benötigt, welche für die datenbankspezifischen Nachrichtenabläufe gebraucht wird (TCP-Verbindungsauf- und abbau sowie Datenbankabfrage).

Sobald alle Testfälle aus einem Verhaltensmodell abgeleitet sind, müssen diese mittels eines Codegenerators automatisiert in TTCN-3-Code umgesetzt werden. Der folgende Quellcode zeigt einen kleinen Auszug aus dem Erfolgsfall des Redirect Services, welcher hier prototypisch als Testfall in TTCN-3-Code spezifiziert wurde.

```

module Redirect {
    type record SIPInvite {
        charstring sipAddress,
        ...
    }
}

```

```

template SIPInvite t_invite(charstring address)
:= { address };
// define data types, ports, test data, and
components
testcase tc_Erfolg() runs on UA_A system
RedirectService {
var Database DBComp;
DBComp := Database.create;
map(mtc:sipPort, system:sipPort);
map(DBComp:dbPort, system:dbPort);
sipPort.send(t_invite("a@b.de"));
alt {
[] sipPort.receive(t_SIPMessage(100)) {
DBComp.start( tc_ConnectDBBehaviour());
}
[] sipPort.receive {
setverdict(fail);
}
}
}
function tc_ConnectDBBehaviour() runs on
Database {
// receive SYN, send SYN+ACK ect.
....
}
}

```

5 Fazit und Ausblick

Die Betrachtung der Teilzustandsautomaten aus Kapitel 3 und dem Beispieldienst aus Kapitel 4 mag den Eindruck erwecken, dass die Testentwicklung aufgrund der komplexen Gebilde sehr schwierig werden könnte, je umfangreicher ein Dienst wird. Allerdings hat der Testentwickler eine andere Sicht auf die Teilzustandsautomaten als in Bild 3 dargestellt. Er kennt die Teilzustandsautomaten lediglich als Black Boxes, die er per Drag-and-Drop in einem Tool für sein Vorgehensmodell aktiviert. Außerdem unterstützt das Tool den Tester dabei, die Schnittstellen der Black Boxes nach außen zu behandeln.

Bei der Modellierung mehrerer Dienste hat sich ergeben, dass bestimmte Black-Boxen bei Standard-Protokollabläufen immer identisch miteinander verknüpft werden. Somit lassen sich vereinzelt Black-Boxen zu Gruppen zusammenfassen, die dem Tester wiederum als komplexere Black-Boxen zur Verfügung gestellt werden können. Dies vereinfacht ebenfalls die Modellierung des Verhaltensmodells durch den Tester.

In naher Zukunft werden die bestehenden Teilzustandsautomaten weiter optimiert und um weitere ergänzt. Außerdem muss ein effizienter Algorithmus gefunden werden, der die MSCs und damit die Testfälle aus den Gesamtzustandsautomaten bzw. Verhaltensmodellen ableitet. Zuvor muss allerdings analysiert werden, welches Überdeckungskriterium für die Testumgebung verwendet wird.

Ein weiterer Aspekt ist die Implementierung eines Codegenerators, welcher die aus den Zustandsautomaten abgeleiteten Testfälle in passenden TTCN-3-Code umwandelt. Eine korrekte und lauffähige Umsetzung in TTCN-3-Code setzt voraus, dass die Verhaltensmodelle und die Testfall-Beschreibungen in einer Form vorliegen (z.B. XML), die durch einen Parser interpretiert werden können.

6 Literatur

- [1] <http://www.ecs.fh-osnabrueck.de/27619.html>
- [2] <http://www.ecs.fh-osnabrueck.de/teamcom.html>
- [3] <http://www.ibm.com/developerworks/library/specification/ws-bpel>
- [4] <http://www.ttcn-3.org>
- [5] <http://www.etsi.org/WebSite/technologies/ttcn3.aspx>
- [6] <http://www.itu.int/ITU-T/studygroups/com07/ttcn.html>
- [7] El-Far, I.K.; Whittaker, A.: Model-Based Software Testing . Encyclopedia of Software Engineering (edited by J.J. Marciniak) . Wiley, 2001
- [8] Liggesmeyer, P.: Software Qualität – Testen, Analysieren und Verifizieren von Software . Spektrum Akademischer Verlag Heidelberg, 2009
- [9] Hartmann, A.: AGEDIS – Final Project Report . <http://www.agedis.de/downloads.shtml>, Feb. 2004
- [10] Rosenberg, J.; Schulzrinne, H.; Camarillo, G.; Johnston, A.; Peterson, J.; Sparks, R.; Handley, M.; Schooler, E.: RFC 3261 – SIP: Session Initiation Protocol . IETF, June 2002

Published in *Proceedings for the Sixth Collaborative Research Symposium on Security, E-learning, Internet and Networking (SEIN 2010)*, pp. 61-74, University of Plymouth, Plymouth, UK, ISBN: 978-1-84102-269-7

Integration of Model-Based Functional Testing Procedures within a Creation Environment for Value Added Services

P. Wacht^{1,2}, A. Lehmann^{1,2}, T. Eichelmann^{1,2}, W. Fuhrmann³, U. Trick¹ and B. Ghita²

¹Research Group for Telecommunication Networks, University of Applied Sciences Frankfurt/M., Frankfurt/M., Germany

²Centre for Security, Communications and Network Research, University of Plymouth, Plymouth, United Kingdom

³University of Applied Sciences Darmstadt, Darmstadt, Germany
email: wacht@e-technik.org

Abstract

Actual Service Creation Environments (SCE) do not support functional testing of automatically created value added services. This leads to a problem as there is no verification that the service is created properly according to the requirement's specification. This paper presents an approach to integrate a testing framework into an existing SCE, which enables systematic and effective functional testing of value added services. The procedure is based on the idea that a behaviour model is created from which the amount of test cases for a specific service can be derived. The identified test cases are transferred to TTCN-3 (Testing and Test Control Notation 3) code and executed on the created service, which is the SUT (System under Test).

Keywords

SCE (Service Creation Environment); finite state machines; functional test automation; TTCN-3 (Testing and Test Control Notation 3)

1. Introduction

In the near future, network operators and service providers aim for Service Creation Environments (SCE) that enable fast, easy and cost efficient provisioning of value added services. Currently, the building of such SCEs has been done in several research projects, as in the TeamCom project (TeamCom, 2009; Lehmann et al., 2009). The TeamCom SCE offers a possibility for developers to design value added services with the help of a graphical user interface and the executable language BPEL (Business Process Execution Language). After the design is fulfilled it is analysed by a code generator and translated into the specific service code. Subsequently, the service can be deployed on an Application Server.

The TeamCom approach proved to work properly for several services. However, a very important aspect is not yet supported by the SCE: the integration of automated

functional tests to validate and verify the created value added services. This enhancement of the SCE will have to be done, because functional tests are derived from the service's specification, which contains the customer's requirements and wishes for the service. So the integration of testing procedures enables a service provider to check if the built service meets the demands of a customer.

The aim of this paper is to show how testing procedures can be integrated within SCEs systematically. For this purpose, the ComGeneration (ComGeneration, 2010) project has been established that should provide a consistent solution to support the life cycle of a service by simplifying development, testing and provisioning of multimedia communication services. This approach reduces the expenditure of time and cost.

A similar approach to ComGeneration was accomplished in the project TT-Medal (TT-Medal, 2010), which has proven the advantages of using UML to generate TTCN-3 tests. Also, the firm Conformiq (Conformiq, 2010) implemented a test suite (Conformiq Tool Suite) which automates the design of functional tests for software and systems. However, the handling of the suite requires deep knowledge in UML and in several programming languages.

The content of this paper is structured as follows: In section 2 the consistent concept of the service and test platform is described. The 3rd section is concerned with the actual approach to describe customer's requirements within a "Service Description". Section 4 describes the relevant parts of the "Test Development" process and in section 5 the execution of test cases is introduced. Finally, section 6 offers a conclusion.

2. Service Creation and Testing Environment

Before looking at the detailed issues about how functional testing of value added services looks like, it is worthwhile having a look at the concept of the ComGeneration approach. Figure 1 gives an abstract overview.

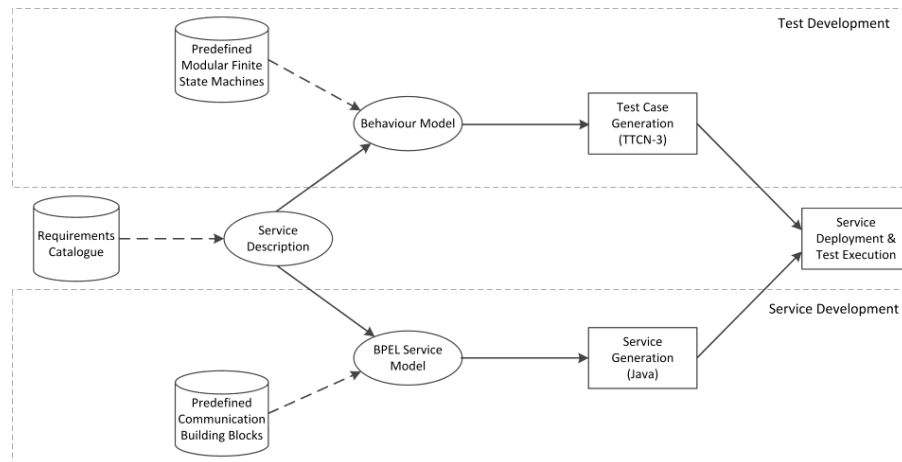


Figure 1: Service and test development for value added services

The shown architecture can be divided into two main layers: The Service Development and the Test Development. In between there are tasks that are relevant to both layers. The kinds of shapes illustrated in the picture have a special meaning, for instance, the container-like shapes generally represent predefined data which a person being involved in the process can choose from. This is shown by the dashed arrow in Figure 1. The circle shapes define actions where a human has to be involved. In contrast, the rectangle shapes only represent tasks that are fulfilled automatically without human interaction.

The initial task that concerns both Service and Test Development is the definition of a “Service Description”. This is a document that can be understood as a requirements specification and is created by the service provider in consultation with a customer. It contains all possible demands a customer might have for a specific value added service. To simplify the creation process of the “Service Description”, the service provider provides the customer with a so-called “Requirements Catalogue”. This catalogue contains predefined standards, restrictions and requirements. The selection of these predefined aspects for a specific value added service results in a form of service description. Furthermore, the relevant roles for the usage of a service are identified within this document.

After the “Service Description” is defined, both the “Service Development” and the “Test Development” are triggered in parallel. The “Service Development” part already exists in the TeamCom Service Creation Environment (Eichelmann et al., 2008). The service creation within TeamCom works as follows: a service designer describes the business process of the corresponding value added service through a formal control logic based on BPEL. So that the modelling of the business process can be done correctly, it requires the usage of predefined communication building blocks which cover the functionality of typical service aspects. This concept of using elementary communication service components is a key advantage of the approach

because it hides the underlying heterogeneous communication networks. Thus, the service designer does not need any detailed knowledge of certain communication protocols and is able to focus on the application logic instead. As BPEL has not been developed for control of real time communication services in heterogeneous networks, a code generator respectively “Service Generator” has been implemented to translate the business process description into Java code. The generated code is based on the JAIN SLEE (Sun and Open Cloud, 2008) architecture, as this technology fulfils the necessity of communication services. The final step of the approach is the deployment of the code on a specific JAIN SLEE Application Server such as Mobicents (Mobicents, 2010).

In parallel with the “Service Development” process, the “Test Development” process is initiated by a test developer. First of all, the test developer has to interpret the “Service Description” properly and also has to extract the relevant service information for the test purpose. Afterwards, he has to choose the service related characteristics out of a repository of predefined modular finite state machines. These state machines cover typical service characteristics like protocol sequences for TCP (Transmission Control Protocol), SIP (Session Initiation Protocol) or HTTP (Hypertext Transfer Protocol). By composing the chosen predefined modular finite state machines, the test developer creates a behaviour model, which describes the possible behaviour of a value added service. Depending on the service’s complexity, the behaviour model itself is also a more or less complex finite state machine. If the behaviour model is complete, an algorithm generates the service specific test cases by identifying every possible path through the finite state machine. A behaviour model can be seen as complete, if all the requirements specified in the “Service Description” are covered within the model.

After the generation is done, every identified test case is converted to TTCN-3 (Testing and Test Control Notation Version 3) (TTCN-3, 2010) within the “Test Case Generation” process. TTCN-3 is an abstract test scripting language which was standardized by ETSI (ETSI, 2010) and ITU-T (ITU-T, 2010) and supports the modularized creation of test scenarios for message and procedure based systems. In the ComGeneration approach, the execution of the generated TTCN-3 test cases on the deployed service is done within a TTCN-3 test framework.

3. Service Description

Defining the “Service Description” for a specific value added service is maybe the most important aspect within the process of creating a service, because it can be seen as a kind of contract between a customer and a service provider. It enables the customer to communicate his requirements for a service to the service developer so that the service can be realized properly. For the ComGeneration project, a specific way of defining a “Service Description” has been developed. It has been derived from a standardized object oriented method and includes the following steps:

1. Short description
2. Identification of the roles (without the system)

3. Requirements specification (with customer)
4. Enhanced requirements specification (without customer)
5. Identification of the communication interfaces

The initial step is to write a very short description about the service’s functionality. Exemplarily, this is shown for the Web2IM (Web-to-Instant-Message) service:

A website should deliver two input masks. The first input mask should contain the address or telephone number (SIP URI) of any participant and the second one should carry any kind of text. A button should be integrated on the web site. When submitting it, the text included in the second input mask should be transferred to the address that was filled in the first input mask. If the SIP URI is not reachable or the text couldn’t be transferred an error should occur on the web site. If the transfer worked, a success message should occur.

This short description of the service is followed by the second step, the identification of the roles respectively participants. For the Web2IM example, this would be on the one hand a web browser ([B]) and on the other hand a text display unit. As SIP is used to transfer the text, the display unit could be a SIP softphone ([S]).

The third step to define a “Service Description” requires the cooperation of the customer and the service developer. Both define significant cases that may occur when using the service. The table illustrates a possible case for the Web2IM service.

		Role
Preconditions	Website available	[B]
	SIP URI entered	[B]
	Text entered	[B]
	Entry approved	[B]
Target	Softphone reachable	[S]
Postcondition	Softphone gets text	[S]
	Approval is displayed	[B]
Description	After accessing the website, SIP URI and text are entered. Entries are approved and text is delivered to the softphone with the SIP URI. The receipt is approved on the website.	

Table 1: Standard case for requirements specification of Web2IM service

Depending on the kind of service, a few of such cases may have to be identified. Afterwards, some enhanced requirements are defined without the customer in step 4 of the “Service Description” process. Here, some specific information is defined such as the maximal length of the SIP URI or the input text.

In the last step, the communication interfaces for the service are identified. This is very helpful information for the test developer, because he will then be able to choose the relevant modular finite state machines from the repository to build a behaviour model. For the Web2IM service, the communication interfaces are the following two: HTTP Client and SIP UAC nonInvite.

4. Test Development

The most significant aspect of this paper is the generation and execution process of test cases for specific value added services to verify that they meet the demands of the customer's requirements. For this purpose, Figure 2 shows the relevant steps for "Test Development" in detail.

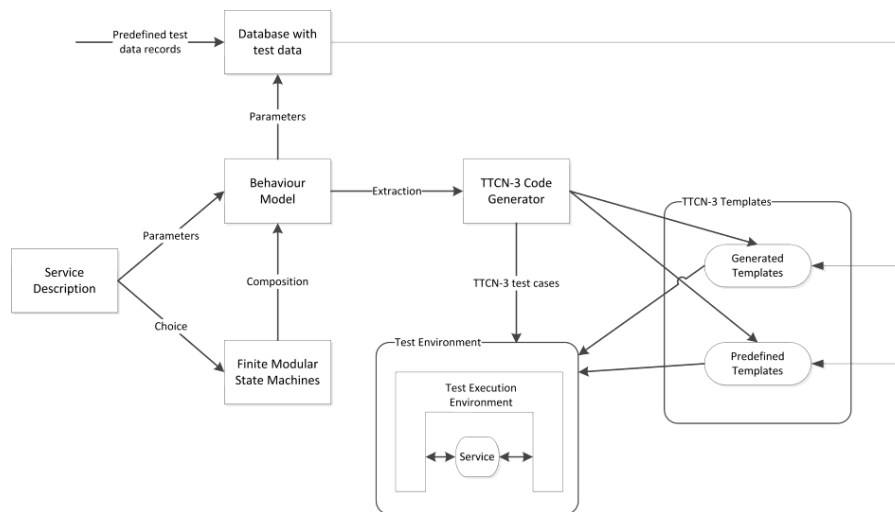


Figure 2: Test Development process

As already shown in Figure 1 and now also in Figure 2, the first condition to start the process is an existing "Service Description". On the basis of the description, the relevant finite modular state machines are chosen and composed to a behaviour model. It was already mentioned in the previous section that some important service related parameters can be specified within the "Service Description" that also have to be integrated into the behaviour model. In TTCN-3, parameters and their values are defined as TTCN-3 templates. This leads to the fact that every parameter within the behaviour model has to be transformed to a TTCN-3 template. An example for a relevant parameter within the behaviour model could be the name of a SIP instant message (e.g. "MESSAGE"). This could mean that during the service flow such a message is expected to be sent, e.g. to a specific SIP User Agent. The information of possible message structures used within the behaviour model has to be available during the "Test Development" process. So, a database with test data is required. In this database, many possible test data records are predefined as TTCN-3 templates.

These templates can be enhanced by the data from the behaviour model. One has to distinguish between predefined and generated templates. Predefined templates already exist in the database, even before the behaviour model was created. Depending on which finite modular state machines are used within the model, the predefined templates are activated and integrated within the test framework. The generated templates are completely new. They are associated to the parameter inputs made by the test developer.

The last step of the “Test Development” process is the testing of the service itself within the test framework. This can only be done if all the extracted test cases exist as TTCN-3 test cases and all the relevant TTCN-3 templates were activated respectively generated and integrated into the test environment.

4.1. Modular finite state machines

Before the structure of the behaviour model is introduced, first the components, the modular finite state machines, are described. The finite state machines are predefined and reusable components which are usually based on specific protocols (SIP, TCP, HTTP) or categories (databases). The structure of the finite state machines for the protocols is derived from the particular protocol specification. Depending on the specification, each finite state machine can have several inputs and outputs. These interfaces are used to compose more finite state machines with each other and to enable the building of the behaviour model.

Figure 3 shows exemplarily the structure of the finite state machines with the help of the two components SIP UAS_INVITE and TCP Client_SYN.

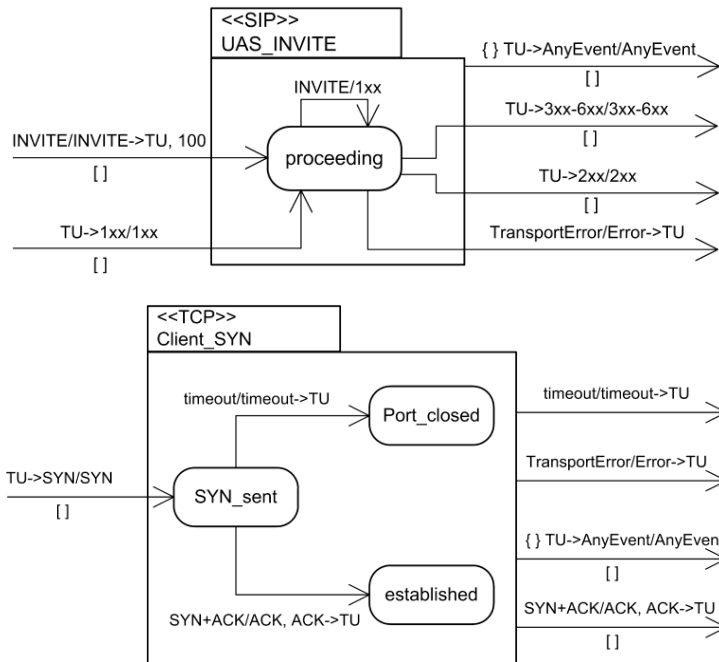


Figure 3: Structure of the finite state machines SIP UAS_Invite and TCP Client_SYN

The finite state machine SIP UAS_INVITE describes the handling of an incoming SIP INVITE message for a User Agent Server (UAS). Every incoming and outgoing transition represents a message that either is received or sent by the UAS. The possible responses, which can be initiated by the User Agent Server, are defined as outputs. Besides the relevant protocol specific outputs like the SIP status codes (2xx, 3xx-6xx) and occurring transport errors, there is also a so-called “AnyEvent” defined. This output can be understood as a placeholder for any kind of message from any protocol. This technique enables the composing of all available finite state machines.

The structure of the finite state machine TCP Client_SYN is a little bit different from the SIP UAS_INVITE, as there are three existent states within the finite state machine. The internal transitions between these states are fixed and always used in the same manner. The TCP Client_SYN represents a TCP connection establishment. The meaning of the “TU” statement within the transitions is discussed in the following section 4.2.

A test developer only knows about the available finite state machines from specific protocols. His main task to build a behaviour model is to handle the interfaces of the finite state machines.

4.2. Behaviour model

In order to do functional testing of a value added service, a test developer has to know, how the service should behave according to the specification, if, for instance, certain messages occur. This knowledge can be retrieved from the “Service Description”. If the understanding of the service is fulfilled, the test developer chooses the relevant modular finite state machines and composes them to get the behaviour model. The composition of finite state machines is the only changing component, the internal transitions, however, are unchangeable. In order to assure, that a behaviour model can be established, a new concept, the Transaction User (TU), was installed. The TU perceives itself as a switching unit between the possible roles of an Application Server (AS) as User Agent Server and User Agent Client. Concurrently, the TU is a connector between modular finite state machines. It enables the test developer to reproduce the service logic for test purposes.

An example of composing the finite state machines from different protocols (Figure 3) is shown in Figure 4. This service logic could be interpreted as follows: the service expects calls from a selection of User Agents (alice, bob). Only if these User Agents call the service, a TCP connection to a specific socket (IP, Port), e.g. to an external database, is established. Here, the SIP INVITE is a sort of trigger.

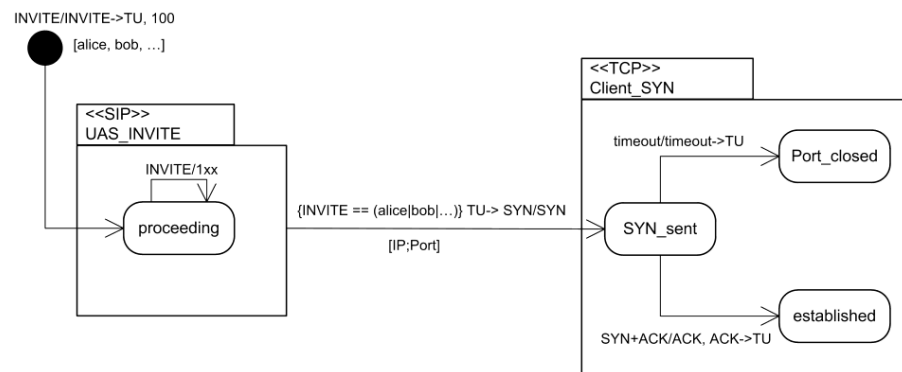


Figure 4: Exemplary role of the TU as a connector of finite state machines

The composition of the two finite state machines is done by using the input message from the TCP Client_SYN as the AnyEvent output message of the SIP UAS_INVITE. Figure 5 clarifies the whole concept of the TU.

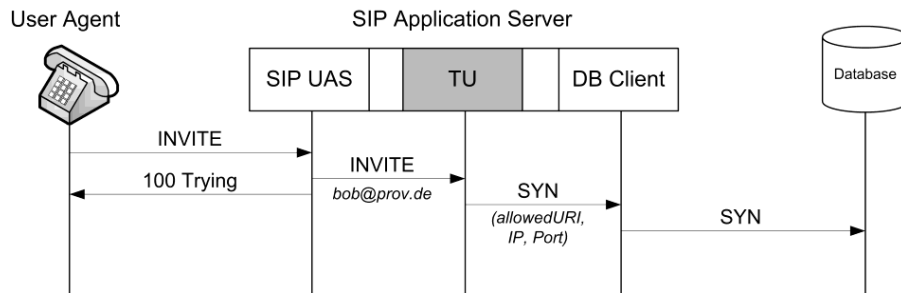


Figure 5: Equivalent message flow by traversing the behaviour model

The demonstrated message flow in Figure 5 reflects the transition path within the finite state machine shown in Figure 4.

When the test developer creates a behaviour model for a value added service he does not need to have any information about the insides of a finite state machine, because he only has to handle the interfaces and has to set relevant parameters. Figure 6 demonstrates a simplified but complete behaviour model of the Web2IM service which was introduced in section 3. The two HTTP modular finite state machines, Server_Req and Server_Resp, represent the initiation of the POST request and the expected responses from the server. In contrast, the three SIP modular finite state machines describe the behaviour of the SIP Message being sent by the service. As the service is the sender of the SIP Message, only the UAC finite state machines are considered.

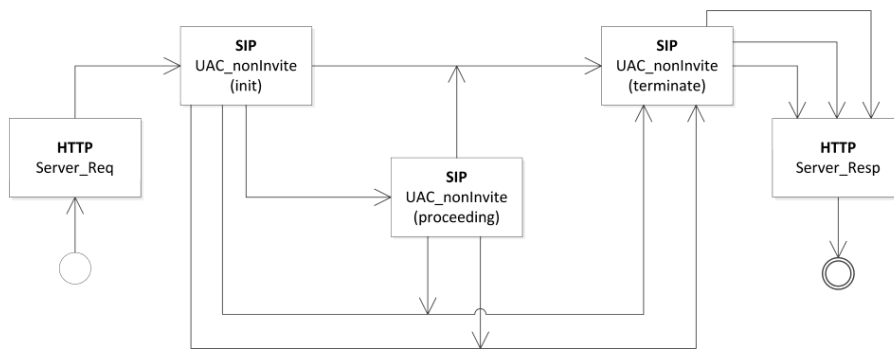


Figure 6: Behaviour model for the value added service Web2IM

4.3. Test Case Generation

The main argument for using finite state machines as behaviour models is that the transition paths within a state machine represent test cases for a value added service.

Therefore, an algorithm has to be defined that identifies all the possible paths. Such an algorithm has not yet been realized as the implementation phase of the ComGeneration project has started recently.

The path finding within the behaviour model can be associated to the following criteria:

- state coverage: every state has to be visited once
- transition coverage: every transition has to be passed once
- event coverage: every possible event has to occur once

For every above-mentioned criterion, the identification of paths respectively the generation of test cases is fulfilled. Afterwards, when the test cases are available, they are transferred to real TTCN-3 test cases by a TTCN-3 code generator.

5. Test Execution

After the generation of the TTCN-3 test cases for a specific value added service has been done, the test cases have to be executed on the service respectively the System under Test (SUT). For this purpose, a TTCN-3 test execution environment is required. Within the ComGeneration project, the integrated test development and execution environment TTworkbench is used, which was developed by Testing Technologies (Testing Technologies, 2010). In order to connect the test execution environment to the SUT, a system adapter is required. Such a system adapter contains adapters that are relevant to enable the communication with the SUT.

Using the example of the Web2IM service, which has been introduced in the previous section, the system adapter would possibly contain a UDP adapter and a HTTP adapter. The UDP adapter is responsible for transferring SIP messages and it is configured to map the TTCN-3 ports to the UDP ports. As in TTCN-3 messages are defined as data structures, the test case executives will use the SIP codec for encoding the data structures to real SIP text messages and vice versa. For the usage of HTTP requests and responses that are used to trigger the service, there is also an adapter required.

Before the test cases can be executed, PTCs (Parallel Test Component) have to be configured that represent the relevant endpoints. For the Web2IM service, two different PTCs have to be defined. The first PTC sends the initial HTTP request to the SUT which contains the text message and the SIP URI and receives the HTTP response with the failure or success message. In contrast, the second PTC receives the SIP MESSAGE from the SUT and answers with a specific SIP status code.

Figure 7 illustrates the structure of the TTCN-3 testing environment for the Web2IM service.

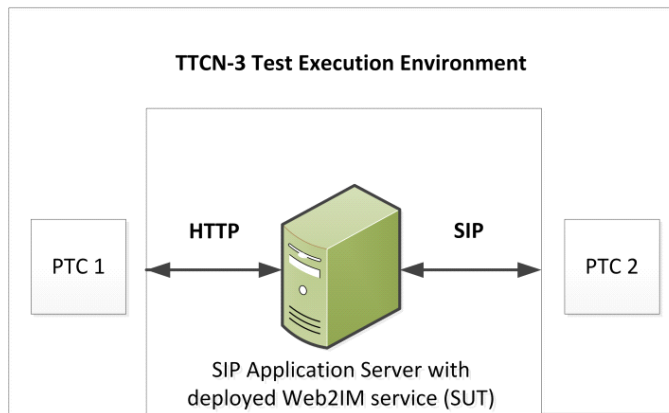


Figure 7: Test Execution Environment for Web2IM service

6. Conclusion

In this paper, we have introduced an approach to integrate functional testing within an existing SCE to validate that a created value added service meets the requirements of the customer who ordered the service. For the testing purpose, a test developer has to get a deep knowledge about the service requirements from the “Service Description” and then has to build an abstract model in the form of a finite state machine, the behaviour model. Although the creation of such a behaviour model tends to be complicated, the advantages dominate, because the extraction of test cases from the model can be done easily with an adequate algorithm.

Once the behaviour model has been developed, it takes very little time until the execution of the generated test cases on the SUT can be done. With the support of quite a lot of communication protocols like SIP, TCP or HTTP, many sorts of value added services can be tested. Besides these positive effects of such an implementation, the test developer who uses the tool has to have a deep knowledge of every used protocol.

As soon as the ComGeneration development environment is implemented, which enables the creation of a behaviour model, an evaluation of the approach is required. To prove the reduction of time and costs in comparison with manual testing, both procedures have to be accomplished for a couple of exemplary value added services.

7. Acknowledgment

The research project ComGeneration providing the basis for this publication was partially funded by the Federal Ministry of Education and Research (BMBF) of the Federal Republic of Germany under grand number 1724B09. The authors of this publication are in charge of its content.

8. References

ComGeneration Project Web Site (2010): <http://www.ecs.fh-osnabrueck.de/27619.html>. (Accessed 15 August 2010)

Conformiq (2010): <http://www.conformiq.com> . (Accessed 20 August 2010)

Eichelmann, T., Fuhrmann, W., Trick, U. and Ghita, B. (2008), “Creation of value added services in NGN with BPEL”, *SEIN*, Wrexham, 2008

ETSI Testing and Test Control Notation (TTCN-3) (2010): <http://www.etsi.org/WebSite/technologies/ttcn3.aspx>. (Accessed 14 August 2010)

ITU-T – The Evolution of TTCN (2010): <http://www.itu.int/ITU-T/studygroups/com07/ttcn.html> (Accessed 14 August 2010)

Lehmann, A., Eichelmann, T., Trick, U., Lasch, R., Ricks, B. and Tönjes, R. (2009), “TeamCom: A Service Creation Environment for Next Generation Networks“, ICIW, Venice, 2009

Mobicents Open Source JAIN SLEE Server Project Web Site (2010): <http://www.mobicents.org>. (Accessed 14 August 2010)

Sun Microsystems, Open Cloud (2008), JSR-000240 Specification, Final Release, “JAIN SLEE (JSLEE) 1.1”, Sun.

TeamCom Project Web Site (2009): <http://www.ecs.fh-osnabrueck.de/teamcom.html>. (Accessed 14 August 2010)

TTMedal (2010): <http://www.tt-medal.org>, (Accessed 20 August 2010)

Testing Technologies (2010): <http://www.testingtech.com>. (Accessed 15 August 2010)

TTCN-3 application areas (2010): <http://www.ttcn-3.org/ApplicationAreas.htm>. (Accessed 14 August 2010)

Published in *Proceedings for the Fourth IFIP International Conference on New Technologies, Mobility and Security (NTMS 2011)*, pp. 1-5, Paris, France, IEEE, ISBN: 978-1-4244-8704-2

A New Approach to Design Graphically Functional Tests for Communication Services

Patrick Wacht, Thomas Eichelmann, Armin Lehmann, Ulrich Trick

Research Group for Telecommunication Networks, University of Applied Sciences
Frankfurt/Main, Germany

{wacht, eichelmann, lehmann, trick}@e-technik.org

Abstract—This paper presents a new concept of how a service provider using any kind of Service Creation Environment (SCE) can verify that a generated service meets the requirements of a customer. This requires functional tests which are derived from a finite state machine-based behaviour model being composed from so-called modular finite state machines by a new composition method. The derivation of the tests is fulfilled by the usage of an adequate path finding algorithm. The following execution of the generated tests is done automatically within a Testing and Test Control Notation (TTCN-3) test framework.

Key words: Functional Testing; TTCN-3; Behaviour Model; Finite State Machine

I. INTRODUCTION

In the near future, network operators and service providers aim for Service Creation Environments (SCE) that enable fast, easy and cost efficient provisioning of value added services. Currently, the building of such SCEs has been done in several research projects, as for instance in the TeamCom project [1; 2]. The TeamCom SCE offers a possibility for developers to design value added services with the help of a graphical user interface and the executable language BPEL (Business Process Execution Language). After the design is fulfilled it is analysed by a code generator and translated into the specific service code. Subsequently, the service can be deployed on an Application Server.

Current SCEs like the TeamCom SCE proved to work properly. However, a very important aspect is usually disregarded by SCEs: the integration of automated functional tests to validate and verify the created value added services.

The enhancement of integrating functional tests will have to be done, because the provider has to assure that the services are executed properly and do not affect other running services within the provider's service environment. Also the integration of testing procedures enables a service provider to check if the built service meets the demands of a customer.

The aim of this paper is to show how testing procedures can be automated. For this purpose, the ComGeneration [3] project has been established that should provide a consistent solution to support the life cycle of a service by simplifying development, testing and provisioning of multimedia communication services. This approach reduces the expenditure of time and cost.

The paper is structured as follows: Section II presents the overall concept of ComGeneration whereas Section III is concerned with the current approach to describe customer's

requirements and how to derive the behaviour model from the requirements. Section IV gives an overview of the needed components and mappings to TTCN-3 (Testing and Test Control Notation) [4] for the planned tool chain. Section V discusses the related concepts and works and Section VI concludes the paper.

II. COMGENERATION CONCEPT

Before looking at the detailed issues about how functional testing of value added services looks like, it is worthwhile having a look at the concept of the ComGeneration approach. Figure 1 gives an abstract overview.

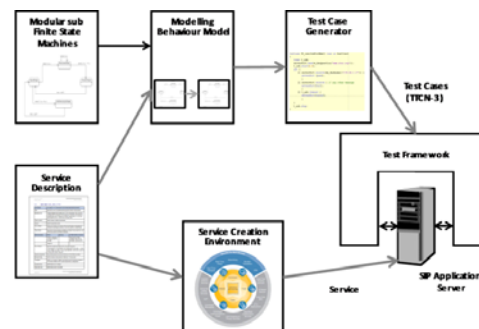


Figure 1. ComGeneration Architecture

The shown architecture can be divided into two main paths: The Service Development and the Test Development. In between there are tasks that are relevant to both paths.

The initial task, which concerns both Service and Test Development, is the definition of a “Service Description”. This is a document that can be understood as a requirements specification and is created by the service provider in consultation with a customer. It contains all possible demands a customer might have for a specific value added service.

After the “Service Description” is defined, both the “Service Development” and the “Test Development” are triggered in parallel. The “Service Development” part is fulfilled by a certain SCE. For the ComGeneration project, any SCE can be used to create a service automatically. An exemplary SCE is the TeamCom Service Creation Environment [2; 5]. The service creation within this SCE works as follows: a service designer describes the business

process of the corresponding value added service through a formal control logic based on BPEL. So that the modelling of the business process can be done correctly, it requires the usage of predefined communication building blocks which cover the functionality of typical service aspects. This concept of using elementary communication service components is a key advantage of the approach because it hides the underlying heterogeneous communication networks. Thus, the service designer does not need any detailed knowledge of certain communication protocols and is able to focus on the application logic instead. As BPEL has not been developed for control of real time communication services in heterogeneous networks, a code generator respectively “Service Generator” has been implemented to translate the business process description into Java code. The generated code is based on the Java APIs for Integrated Networks Service Logic Execution Environment (JAIN SLEE) [6] architecture, as this technology fulfils the necessity of communication services. The final step of the approach is the deployment of the code on a specific JAIN SLEE Application Server such as Mobicents [7].

In parallel with the “Service Development” process, the “Test Development” process is initiated by a test developer. First of all, the test developer has to interpret the “Service Description” properly and also has to extract the relevant service information for the test purpose. Afterwards, he has to choose the service related characteristics out of a repository of predefined modular finite state machines. These state machines cover typical service characteristics like protocol sequences for TCP (Transmission Control Protocol), SIP (Session Initiation Protocol) or HTTP (Hypertext Transfer Protocol). By composing the chosen predefined modular finite state machines, the test developer creates a so-called behaviour model, which describes the possible behaviour of a value added service. Depending on the service’s complexity, the behaviour model itself is also a more or less complex finite state machine. If the behaviour model is complete, an algorithm generates the service specific test cases by identifying every possible path through the finite state machine. After the generation is done, every identified test case is converted to TTCN-3 [4] within the “Test Case Generation” process. TTCN-3 is an abstract test scripting language which was standardized by ETSI [8; 9; 10] and ITU-T [11; 12] and supports the modularized creation of test scenarios for message and procedure based systems. In the ComGeneration approach, the execution of the generated TTCN-3 test cases on the deployed service is done within a TTCN-3 test framework.

One very significant aspect of this paper is on the one hand side the way how the behaviour model is modelled by a test developer from the content of the “Service Description”. On the other hand side, the automatic generation of test cases from the behaviour model and the following execution on the System under Test (SUT) is the main focus. The mentioned aspects should verify that a communication service meets the requirements of a customer. The following Figure 2 shows the relevant steps for the “Test Development” in detail.

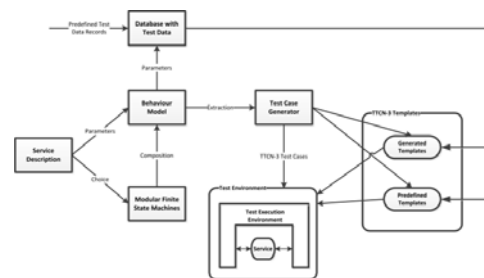


Figure 2. Test Development Process

On the basis of the description, the behaviour model is described by a finite state machine. Some important service related parameters can be specified within the “Service Description”. These parameters have to be integrated into the behaviour model. In TTCN-3, parameters and their values are defined as so-called TTCN-3 templates. This leads to the fact that every parameter within the behaviour model has to be transformed to a TTCN-3 template. An example for a relevant parameter within the behaviour model could be the name of a SIP instant message (e.g. “MESSAGE”). This could mean that during the service flow such a message is expected to be sent, maybe to a specific SIP User Agent. The information of possible message structures used within the behaviour model has to be available during the “Test Development” process. So, a database with test data is required. In this database, many possible test data records are predefined as TTCN-3 templates. Predefined templates already exist in the database, even before the behaviour model was created. Depending on which finite modular state machines are used within the model, the predefined templates are activated and integrated within the test framework. The generated templates are completely new. They are associated to the parameter inputs made by the test developer. The last step of the “Test Development” process is the testing of the service itself within the test framework.

III. SERVICE DESCRIPTION AND BEHAVIOUR MODEL

As depicted in the last section, the test developer has to design a behaviour model based on the information he could retrieve from the service description. Both the service description and the model will be explained in the following.

A. Service Description

The aim of the service description is to deliver a complete set of requirements from the view of a user which has to be fulfilled by the communication service. A user can on the one hand be a person or on the other hand be an external system.

For the ComGeneration project, a specific way of defining a service description has been developed. It has been derived from a standardised object oriented method and includes the following steps:

1. Short description
2. Identification of the roles
3. Requirements specification

Depending on the specification, each state machine can have several inputs and outputs. These interfaces are used to compose more state machines with each other and to enable the building of the behaviour model.

The state machine SIP UAS_INVITE which was derived from [13] describes the handling of an incoming SIP INVITE message for an User Agent Server (UAS). Every incoming and outgoing transition represents a message which either is received by the UAS or sent. The possible responses, which can be initiated by the User Agent Server, are defined as outputs. Besides the relevant protocol specific outputs like the SIP status codes (2xx, 3xx-6xx) and occurring transport errors, there is also a so-called “AnyEvent” defined. This output can be understood as a placeholder for any kind of message from any protocol. This technique enables the composing of all available state machines.

A test developer only knows about the available state machines from specific protocols. His main task to build a behaviour model is to handle the interfaces of the finite state machines.

B. Mapping to TTCN-3

The test machine generates messages and sends them to the system under test (SUT) or it checks the messages sent by SUT and responds to those messages. With the messages sent to the SUT, the test system tries to provoke errors on the SUT. This requires the test machine to understand the messages from the SUT and also provide test data that consist of positive and negative cases. To support these tasks TTCN-3 defines some elements that are required to create tests. In Table I, some of these elements are described.

The Test Case Generator (TCG) creates test cases from the FSMs and translates them into TTCN-3 code. So the TCG needs some knowledge about the TTCN-3 elements or generates these elements by himself. For the mapping of the elements from the TCG to TTCN-3 two concepts are introduced, the Connectivity Concept and the FSM Concept. To generate the required test cases the TCG needs to obtain the information from both concepts. Each concept provides other information for the TCG and the two concepts together offer all the required information to generate the TTCN-3 code.

TABLE I. TTCN-3 ELEMENTS

TTCN-3 element	Description
Type Definition	Defined data types to describe the exchange of data between test components and SUT.
Port Definition	Communication between the test components and the SUT is established by connecting local ports. e.g. SIP port, HTTP Port
Component Definition	Structure of the test components that represent the client and server protocol-specific endpoints e.g. UAS SIP, HTTP Client
Test Case	Runs individual test components Summarizes the test events
TTCN-3-Templates	Definition for the description of test data

Control Part	Describes the order and the conditions for the execution of individual test cases
TTCN-3-Codecs and Test Adapter	Adaptation for the SUT Converts TTCN-3 code in an understandable format for the SUT
Verdict	Judgement (Pass, Fail, Inconclusive, error)

The FSM Concept is presented in Figure 5.

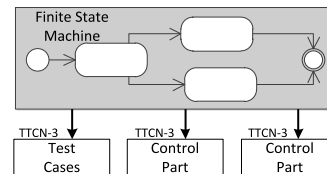


Figure 5. Finite State Machine Concept

From the FSM, different TTCN-3 test cases can be generated. The FSM also describes in what order and under what conditions the test cases are executed. From this information, the TTCN-3 Control Part is generated. The FSM represents only the positive “good” reactions of the service. All other cases, which occur in TTCN-3, are defined as invalid (failed). The TTCN-3 verdict can therefore also be derived from the FSM approach.

The Connectivity Concept is shown in Figure 6.

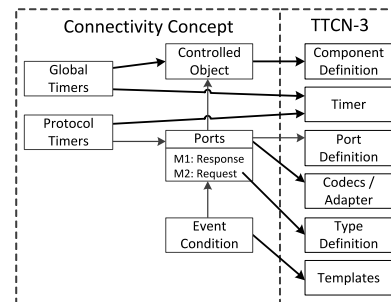


Figure 6. Connectivity Concept

Also from this concept, information for the mapping to TTCN-3 elements can be derived. The concepts are connected through the Controlled Object (CO). A CO is assigned to the FSM within the FSM concept. This CO holds information about the used ports and timers.

The TCG supports different timers (protocol timer and global timer). The protocol timers are pre-defined and belong to the respective ports. Depending on the protocols or the messages which are required by the test developer, the associated protocol timers are automatically added to the tests. The test developer also has the ability to define its own timers. These timers are called global timers, because they are defined and can be manipulated within the whole FSM.

The ports in the TCG map to the TTCN-3 ports. For all supported protocols, the ports are predefined in the TCG. Also, all possible protocol messages must be predefined within the respective ports. The timers which are defined within the protocol specifications are also added to the port definition. The definition of the test data in TTCN-3 is done with the help of templates. These TTCN-3 templates are generated by the TCG. For this purpose the TCG uses so called Event Conditions. The test developer uses these Event Conditions to choose from pre-defined test data or to define its own test data.

The data types which are used are predefined and correspond to the TTCN-3 Type Definitions. The ports used in the TCG are assigned to the corresponding TTCN-3 codec and TTCN-3 adapter.

C. Toolchain

The TCG generates the test cases and test data for TTCN-3 from the FSM which is created by the test developer. The TCG is composed by several elements, the GUI, the FSM Parser, the Connectivity Parser, the FSM Pathfinder and the TTCN-3 Code Generator, see Figure 7. In order to model the FSM in a simple manner, the test developer is supported by a GUI. This GUI consists of several views, the FSM View and the Connectivity View. In the FSM View, graphical FSM states and transitions can be defined.

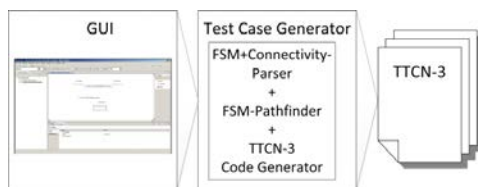


Figure 7. Tool chain

In the Connectivity View, ports, messages and timers are added to the Controlled Object and test data is defined or adjusted. Another element of the TCG is the parser. The parser analyses the FSM and gathers the information needed for the TTCN-3 code generation. All relevant test cases are required for the SUT test. A test case represents a path from initial state to end state within the FSM. This means that all paths within the FSM are discovered by the pathfinder. From each resulting path of the FSM a TTCN-3 test case is generated. With all of the test cases and the information obtained by the parser, a TTCN-3 code generator generates the complete TTCN-3 test.

V. RELATED WORK

The approach to describe tests with finite state machines is quite common. Conformiq [14] describes tests by UML state diagrams, but the focus is not to describe the service from the view of the SUT, but from the view of the test components. Furthermore, there are no predefined state machines which can be used to simplify and accelerate the modelling process.

A similar approach from Yuan [15] describes test case generation from UML activity diagrams, but the main focus is

about testing Web Service compositions with the help of TTCN-3. However, the functionality is quite limited, as only HTTP is supported which enables Web Service composition.

VI. CONCLUSION

In this paper, we have introduced a new approach to integrate automated functional testing within any SCE to validate that a created value added service meets the requirements of the customer who ordered the service. Therefore, we developed a technique to create a behaviour model for a service from predefined modular finite state machines with the help of a certain concept, the Transaction User. From the behaviour model, abstract test cases can be derived by the usage of an efficient path finding algorithm. The abstract test cases are then converted to executable test cases in TTCN-3, a standardised scripting language used in testing of communication services.

Further work should address the improvement of the path finding algorithm and validation of the concept.

ACKNOWLEDGMENT

The research project ComGeneration providing the basis for this publication was partially funded by the Federal Ministry of Education and Research (BMBF) of the Federal Republic of Germany under grand number 1724B09. The authors of this publication are in charge of its content.

REFERENCES

- [1] TeamCom project website: <http://www.ecs.fh-osnabrueck.de/teamcom.html>
- [2] A. Lehmann et al.: "TeamCom: A Service Creation Platform for Next Generation Networks", IEEE ICIW 2009, Venice/Mestre, Italy, 24-28 May 2009.
- [3] ComGeneration project website: <http://www.ecs.fh-osnabrueck.de/27619.html>
- [4] ETSI website for the TTCN-3 standards: <http://www.etsi.org/WebSite/technologies/ttcn3.aspx>
- [5] T. Eichelmann et al.: "Creation of value added services in NGN with BPEL", Proceedings of the Fourth Collaborative Research Symposium on Security, E-learning, Internet and Networking (SEIN 2008), Wrexham, United Kingdom, 5-9 November 2008.
- [6] Sun Microsystems, Open Cloud, JSR-000240 Specification, Final Release, "JAIN SLEE (JSLEE) 1.1", SUN, 2008.
- [7] Mobicents Open Source JAIN SLEE Server, <http://www.mobicents.org>
- [8] EG 201 873-1: Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part1: TTCN-3 Core Language. ETSI, September 2008.
- [9] EG 201 873-2: Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part2: TTCN-3 Tabular presentation Format (TFT), ETSI, February 2007.
- [10] EG 201 873-3: Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part3: TTCN-3 Graphical presentation Format (GFT), ETSI, February 2007.
- [11] Recommendation Z.140: The Tree and Tabular Combined Notation version3 (TTCN-3): Core Language. ITU-T, July 2001.
- [12] Recommendation Z.141: The Tree and Tabular Combined Notation version3 (TTCN-3): Tabular Presentation Format. ITU-T, July 2001.
- [13] J. Rosenberg et al. : RFC 3261 – SIP : Session Initiation Protocol. IETF, June 2002.
- [14] Conformiq website: <http://www.conformiq.com/>
- [15] Qilun Yuan et al. : A Model Driven Approach Toward Business Process Test Case Generation. IEEE, June 2008

Published in *ITG-Fachbericht Mobilfunk (Mobilfunktagung 2011)*, pp. 118-123,
University of Applied Sciences Osnabrück, Germany, ISB: 978-3-8007-3352-1

ComGeneration: die Dienstbeschreibung als Basis für automatisierte Tests

Patrick Wacht, Thomas Eichelmann,
Armin Lehmann, Ulrich Trick
Fachhochschule Frankfurt/M.,
University of Applied Sciences,
Nibelungenplatz 1, 60318 Frankfurt/M., Germany
E-Mail: {wacht, eichelmann, lehmann, trick}@c-
technik.org;

Rolf Lasch, Marten Fischer, Ralf Tönjes
Hochschule Osnabrück
University of Applied Sciences,
Albrechtstr. 30, 49076 Osnabrück, Germany
E-Mail: {r.lasch, m.fischer, r.toenjes}@hs-
osnabrueck.de;

Das dieser Publikation zugrunde liegende Vorhaben wurde mit Mitteln des Bundesministeriums für Bildung und Forschung (BMBF) unter dem Förderkennzeichen 1724B09 gefördert. Die Verantwortung für den Inhalt dieser Veröffentlichung liegt bei den Autoren.

Kurzfassung

Um sicherzustellen, dass ein Mehrwertdienst gemäß den Anforderungen eines Kunden realisiert wurde, ist die Durchführung von funktionalen Tests unerlässlich, welche aus einer vorliegenden Dienstbeschreibung bzw. Spezifikation abgeleitet werden. In diesem Aufsatz wird eine neue Art der Dienstbeschreibung vorgestellt, die nicht den Kontakt zum Kunden verliert, diesen mit einbindet und eine solide Grundlage für die schnelle, kostengünstige und qualitativ hochwertige Dienst- und Testentwicklung gewährleistet. Das Vorgehen beim Erstellen einer solchen Dienstbeschreibung wird vorgestellt und anhand eines konkreten Beispieldienstes gezeigt.

1 Einleitung

Die Nachfrage nach neuen Diensten, speziell Mehrwertdiensten, wächst ständig, daher besteht ein großes Interesse seitens der Netzbetreiber und Diensteanbieter, kostenoptimierte Lösungen zu finden, um neue Dienste einfach und schnell bereitstellen zu können. Durch das IMS (IP Multimedia Subsystem) werden neue Möglichkeiten der einfacheren und schnelleren Dienstbereitstellung gegeben, die zu neuen Einnahmequellen neben der normalen Telefonie führen. Um die Nachfrage der Kunden zu befriedigen, müssen neu entwickelte Dienste nicht nur bereitgestellt, sondern auch getestet werden, bevor sie in den Wirkbetrieb übergehen. Nur so kann die Grundvoraussetzung für die Zufriedenheit der Kunden garantiert werden. Das Testen von Diensten wird durch die steigende Nachfrage und Komplexität der neu entwickelten Dienste zunehmend an Bedeutung gewinnen. Im BMBF-Projekt ComGeneration (Test-gesteuerte Evolution und automatisierte Bereitstellung von Kommunikationsdiensten) [1] ist daher eine durchgängige Lösung erarbeitet worden, die beginnend mit der Dienstbeschreibung bis hin zum ausführbaren Test reicht.

Im nächsten Kapitel wird der in ComGeneration entwickelte Ansatz näher erläutert, der auf einer wohl definierten Dienstbeschreibung fußt. Diese Dienstbeschreibung wird in Kapitel 3 eingehend beleuchtet und

in Kapitel 4 anhand eines Beispiels veranschaulicht. Die Zusammenfassung und geplantes weiteres Vorgehen bilden den Abschluss dieses Aufsatzes in Kapitel 5.

2 ComGeneration-Ansatz

Wie in Bild 1 dargestellt, wird in diesem Ansatz Wert darauf gelegt, dass zu Beginn eine definierte Dienstbeschreibung vorliegt, aus der sowohl die Dienstentwicklung als auch die Testentwicklung hervorgehen.

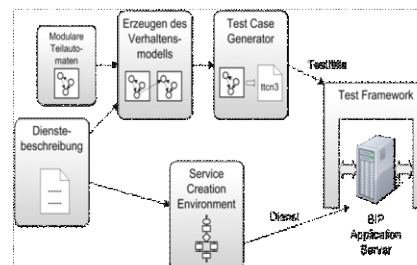


Bild 1 ComGeneration Framework

Für die Dienstentwicklung kann theoretisch jedes existierende Service Creation Environment (SCE) eingesetzt werden. Ebenfalls denkbar wäre eine manuelle Implementierung durch einen Dienstentwickler. Im Gegensatz dazu ist der Ablauf der Testentwicklung durch einen Testentwickler im ComGeneration-Ansatz wohldefiniert [2; 3]. Dieser muss zunächst die testspezifischen Informationen aus der ihm vorliegenden Dienstbeschreibung herausfiltern. Danach selektiert er aus einem Repository an vorgefertigten wiederverwendbaren modularen Teilautomaten die für den Dienst relevanten Bausteine. Diese modularen Teilautomaten beschreiben typische dienstspezifische Abläufe, z.B. bestimmte Sequenzen von Nachrichten für Protokolle wie SIP (Session Initiation Protocol) oder HTTP (Hypertext Transfer Protocol). Nach der Auswahl der relevanten modularen Teilautomaten komponiert der Testentwickler diese entsprechend den Angaben aus der Dienstbeschreibung und erhält in der Folge das sogenannte Verhaltensmodell, bei dem es sich um einen komplexen Zustandsautomaten handelt. Alle in diesem Zustandsautomaten möglichen zu durchlaufenden Zustände und Zustandsübergänge inklusive der diese hervorrufenden Events (Eingangssignale) und resultierenden Actions (Ausgangssignale) vom Startzustand bis zum Endzustand des Zustandsautomaten beschreiben die zu testenden Nachrichtenabläufe. Diese Nachrichtenabläufe stellen das mögliche Verhalten aus der Sicht des Dienstes dar. Um nun mittels des Test Case Generators aus dem Verhaltensmodell TTCN-3-Testfälle generieren zu können, welche anschließend in einem Test Framework auf das "System under Test", dem deployten Dienst, ausgeführt werden können, wird die Sicht invertiert. So handelt es sich bei den Eingangssignalen für den Dienst im Verhaltensmodell um Nachrichten, die von Testseite bzw. von den Testkomponenten gesendet werden müssen. Im Gegensatz dazu handelt es sich bei den Ausgangssignalen vom Dienst um Nachrichten, welche von der Seite des Tests empfangen werden.

3 Dienstbeschreibung

Wie bereits erwähnt, ist die Grundlage für die Entwicklung eines neuen Dienstes und dessen Tests eine hinreichend genaue Dienstbeschreibung. Diese Dienstbeschreibung muss mehrere Kriterien erfüllen. Sie sollte möglichst unmissverständlich und eindeutig sein. Hierfür sollte ein eingeschränkter Wortstamm (mit Schlüsselwörtern) genutzt werden, um Doppeldeutigkeit an Begriffen zu vermeiden. Darüber hinaus muss die Dienstbeschreibung so geartet sein, dass sowohl der Test- als auch der Dienstentwickler diese verstehen und interpretieren können. Beide Entwickler sollten möglichst das gleiche Bild des Dienstes vor

sich haben, um spätere Differenzen zu vermeiden. Eine weitere Anforderung an die Dienstbeschreibung ist, dass diese hinreichend genau den Dienst bzw. das Dienstverhalten beschreibt. Hierzu bedarf es speziell geschultem Personal (sogenannte Service Agents), welches zusammen mit dem Kunden eine Dienstbeschreibung erstellt.

3.1 Aufbau der Dienstbeschreibung

Die Erstellung einer Dienstbeschreibung gliedert sich wie folgt:

1. Erstellung einer textuellen Dienstbeschreibung.
2. Identifizieren der am Dienst beteiligten Rollen.
3. Spezifizieren der Anforderungen aus Sicht der Benutzer/Kunden.
4. Ergänzen der spezifizierten Anforderungen
5. Identifizieren der Kommunikationsschnittstellen zwischen dem Dienst (System) und den Nutzern.
6. Entwickeln der Tests und des Dienstes.

Die genannten Punkte 1 bis 3 werden werden durch den Service-Agenten mit dem Kunden erarbeitet. Die Schritte 4 und 5 werden darauf durch den Service-Agenten fortgeführt und Schritt 6 wird dann durch die Entwickler (Test- bzw. Dienstentwickler) durchgeführt. Bis zum Schritt 6 muss die Dienstbeschreibung allen Anforderungen genügen. Die einzelnen Schritte der Dienstbeschreibung bis hin zur Erstellung der Tests und des Dienstes werden im Folgenden näher erläutert.

3.2 Textuelle Dienstbeschreibung

In der textuellen Beschreibung wird der Dienst mit all seinen Eigenschaften aus Sicht des Benutzers beschreiben. Hier werden auch Vereinbarungen genauer definiert wie zum Beispiel die SIP URI (Uniform Resource Identifier) des Dienstes. Während der Erstellung der textuellen Beschreibung muss der Service Agent genau darauf achten, dass keine Ungenauigkeiten in der Beschreibung auftreten. Ein Beispiel hierfür wäre der Satz: „Alle weiteren Fehler werden ebenfalls übermittelt“. Dieser Satz ist unzureichend genau und muss ergänzt bzw. geändert werden in z.B.: „Bei Nichterreichbarkeit und falsch angegebener Zieladresse wird dem anfragenden Teilnehmer über Statusmeldungen der aufgetretene Fehler signalisiert“.

3.3 Identifikation der beteiligten Rollen

Aus der bereits erstellten textuellen Beschreibung des Dienstes können Rollen (z.B. SIP-Endgerät) identifiziert werden. Alle Rollen müssen aus der textuellen Beschreibung ableitbar sein. Hierfür sorgt der Service Agent, indem er mit dem Kunden die Beschreibung komplettiert. Die Rollen müssen immer aus Sicht auf das System definiert werden, wobei unter dem System der erwünschte Dienst zu verstehen ist.

3.4 Spezifikation der Anforderungen

Dieser Abschnitt der Erstellung der Dienstbeschreibung unterteilt den zu entwickelnden Test und Dienst in einzelne Bereiche, die als sogenannte Dienstpfade verstanden werden können. Unter einen Dienstpfad ist ein bestimmter Ablauf bzw. eine Variante innerhalb eines Tests bzw. eines Dienstes zu verstehen. Zum Beispiel wird auf Grund eines Ereignisses (z.B. Eintreffen einer SIP-Nachricht INVITE) ein bestimmter Pfad betreten. Beim Eintreffen eines anderen Ereignisses (andere SIP-Nachricht) wird ein anderer Pfad eingeschlagen. Die Dienstpfade dienen dazu, den Test- bzw. Dienstentwickler zu unterstützen. Beide Entwickler können sich anhand dieser Spezifikationen orientieren um den Dienst bzw. die Tests zu erstellen. Die Spezifikation der Anforderungen besteht aus vier Elementen (in Tabelle 1 exemplarisch dargestellt).

- Rollen: Die bereits identifizierten Rollen, welche aus Sicht auf das System (Dienst) definiert wurden.
- Preconditions: Hier wird die Situation beschrieben, welche zu der vom Kunden gewünschten Folgesituation (Target) führen soll. Voraussetzung für das Erreichen des Targets ist, dass der gesamte Dienstverlauf fehlerfrei ist. Der Begriff Situation soll in diesem Zusammenhang die Summe aller Informationen bezeichnen, die dem Benutzer in seiner momentanen Rolle bezüglich seines Auftrags an das System zugänglich sind (z.B. Rufnummern oder Statusinformationen über sein Endgerät).
- Postconditions: Die Postconditions repräsentieren eine Liste aller möglichen Situationen, die nach Bearbeitung des Benutzerwunsches vorliegen können. Hier lassen sich auch die äusserlich sichtbaren Reaktionen des Systems bei gestörtem oder fehlerhaftem Ablauf des Dienstes beschreiben. Auch das Target zählt zu den Postconditions, es ist allerdings gesondert zu beachten und kenntlich zu machen.
- Prosa: Hier soll noch einmal das Target beschrieben werden. Dazu sollen allerdings

kurze und präzise Formulierungen verwendet werden. Hierdurch erhalten die Entwickler grundlegende Informationen zur Funktionalität des Dienstes.

Tabelle 1 Beispielhafte Anforderungsspezifikation

Rollen	Anrufer [a]	Zuordnung der Rollen
Pre-conditions	Anrufer kann beliebigen Teilnehmer erreichen	[a]
	Anrufer hat keine aktive Verbindung	[a]
Post-conditions	Anrufer wird nicht verbunden	[a]
	Empfänger ist unbekannt	[a]
	Empfänger ist nicht Verbindungsbereit Anrufer wird verbunden	[a]
Prosa	Teilnehmer [a] möchte ein Gespräch mit einem anderen Teilnehmer führen	

Wie bereits erwähnt, ist die gewünschte Folgesituation besonders hervorgehoben (hier: „Anrufer wird verbunden“) und wird als Geradeausfall definiert.

3.5 Ergänzung der Anforderungen

Falls nötig definiert der Service Agent Erweiterungen für die einzelnen Anforderungen. Darüber hinaus besteht in diesem Schritt der Erstellung einer Dienstbeschreibung die Möglichkeit, auch dienstinterne Eigenschaften aufzuführen. Dies war in den vorherigen Schritten nicht möglich, da der Dienst immer als Black Box betrachtet wurde. Die folgende Liste soll ein Beispiel für ergänzende Anforderungen sein.

- SIP URIs dürfen keine Sonderzeichen enthalten (ausgenommen @).
- SIP URIs dürfen nicht mehr als 64 Zeichen lang sein.
- Der Inhalt einer Instant Message darf nicht leer sein.

3.6 Identifizierung der Kommunikationsschnittstellen

In diesem Schritt werden die Kommunikationsschnittstellen zwischen dem Dienst und den einzelnen Rollen bestimmt. Auch hierbei ist darauf zu achten, dass die Identifizierung der Schnittstellen immer aus Sicht auf den Dienst geschieht. Hierfür werden vorab definierte Schnittstellen verglichen und ausgewählt. Die Schnittstellen repräsentieren grundlegende Funktionalitäten

verschiedener Protokolle z.B. SIP UAS (User Agent Server) oder SIP UAC (User Agent Client). Das folgende kurze Beispiel soll dies verdeutlichen.

Ein SIP-Endgerät sendet eine Nachricht (Instant Message) an das System. Das SIP-Endgerät repräsentiert hierbei die Rolle, welche auf das System zugreift. Aus Sicht dieser Rolle stellt das System die inverse Funktion dar. Das bedeutet für dieses Beispiel, dass das Endgerät die Funktionalität eines SIP UACs besitzt und das System das dazu passende Gegenstück, den SIP UAS, darstellt.

Jeder der einzelnen Schritte kann dazu führen, einen oder mehrere Schritte zurückgehen zu müssen, da Teilaspekte nicht genügend genau beschrieben wurden. Hierzu kann es auch notwendig sein, den Kunden wieder mit einzubeziehen, um im Sinne des Kunden zu handeln. All diese Schritte unterstützen die Entwickler, den Test- sowie Dienstentwickler bei der Umsetzung. Ein Testentwickler kann viele notwendige Informationen daraus schließen. Er kann z.B. einzelne Testpfade aus den Postconditions zur Erstellung des Zustandsautomaten-basierten Verhaltensmodells ableiten. Auch die zu nutzenden Schnittstellen (repräsentiert durch modulare Teilautomaten) und weitere wichtige Parameter (Ergänzungen der Anforderungen) zur Bestimmung der Übergänge zwischen den Zuständen im Verhaltensmodell sind bereits definiert worden und können leicht adaptiert werden.

Anhand eines Beispiels soll das Vorgehen zur Erstellung einer solchen Dienstbeschreibung im nächsten Abschnitt mittels eines Beispieldienstes detailliert erläutert werden.

4 Dienstbeschreibung für den Beispieldienst Click2IM

In dem vorigen Kapitel wurden der Aufbau und die Struktur einer Dienstbeschreibung in einer abstrakten Weise erläutert. Die Konkretisierung des Ansatzes erfolgt nun durch die Anwendung auf einen konkreten Beispieldienst mit dem Namen Click2IM (Click-to-Instant-Message).

Der erste Punkt der Dienstbeschreibung für den Click2IM-Dienst ist die grobe textuelle Beschreibung.

4.1 Textuelle Dienstbeschreibung

Es soll möglich sein, nach dem Aufruf einer Webseite zwei Eingabemasken auf der Seite auszufüllen. Eine Eingabemaske beinhaltet die Adresse bzw. Telefonnummer eines beliebigen Textanzeigergerätes bzw. Softphones, die andere soll einen beliebigen Text aufnehmen. Sobald die Eingaben in den Masken getätigt

wurden, können sie durch Betätigung eines dafür vorgesehenen Buttons bestätigt werden. Falls die Zieladresse nicht erreichbar ist oder der Text nicht übermittelt werden konnte, soll dies der Person via Webseite mitgeteilt werden. Auch die erfolgreiche Versendung des Textes soll bestätigt werden.

4.2 Identifikation der am Dienst beteiligten Rollen

In der textuellen Dienstbeschreibung sind die beiden beteiligten Rollen bereits implizit erwähnt:

- Rolle 1: Webseite (Browser)
- Rolle 2: Textanzeigergerät (Softphone)

4.3 Spezifikation der Anforderungen aus der Sicht des Benutzers

In der nachfolgenden Tabelle 2 wird die für den Click2IM-Dienst relevante Anforderungsspezifikation angegeben. Die Rolle des Browsers wird nachfolgend als Initiator bezeichnet, da diese den Dienst ursprünglich konsumiert. Die Rolle des Textanzeigergerätes bzw. Softphones wird hingegen als Empfänger beschrieben.

Tabelle 2 Anforderungsspezifikation für Click2IM

Rollen	Browser [b], Softphone [s]	Zuordnung der Rollen
Preconditions	Initiator kann beliebige Zieladresse angeben	[b]
	Initiator kann Textinhalt angeben	[b]
	Initiator bestätigt Eingaben	[b]
Postconditions	1. Empfänger ist unbekannt	[b]
	2. Empfänger erhält keine Textnachricht	[b]
	3. Empfänger erhält Textnachricht	[b,s]
Prosa	Initiator möchte eine Textnachricht an ein Softphone senden.	

In der Tabelle sind ferner die relevanten Preconditions und Postconditions aufgeführt. Die Preconditions umfassen hierbei alle für den Click2IM-Dienst relevanten Schritte, um zu der geforderten Folgesituation bzw. Postcondition zu kommen, welche zuvor als Target definiert wurde. Neben dem eigentlichen Target werden noch weitere Postconditions spezifiziert, welche die sichtbaren Reaktionen des Dienstes bei gestörtem

oder fehlerhaftem Ablauf beschreiben. Sowohl dem Dienstentwickler als auch dem Testentwickler sollte klar sein, wie es zu derartigen Postconditions kommen kann. Hierfür werden die nachfolgenden ergänzenden Anforderungen angegeben.

4.4 Ergänzung der Anforderungen

Diese Anforderungsergänzungen gelten ausschließlich für die Postconditions und werden jeweils diesen zugeordnet.

- Standardfehler werden signalisiert (z.B. Timeouts)
- Maximale Länge der Zieladresse (128 Zeichen)
- Sonderzeichen in der Zieladresse sind nicht zulässig (exklusive @)
- Maximale Länge der Texteingabe (256 Zeichen)
- Leere Texteingaben sind nicht zulässig

Der Bezug zu den in Tabelle 2 definierten Postconditions wird im nachfolgenden Bild 2 deutlich.

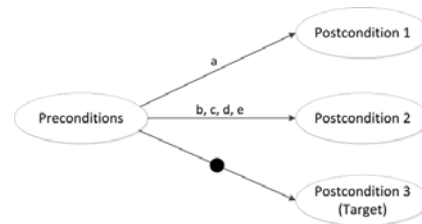


Bild 2 Zuordnung von Anforderungsergänzungen zu Postconditions

Die Postcondition 1 („Empfänger ist unbekannt“) wird hier über die Anforderungsergänzung a erreicht, gleiches gilt bei der Postcondition 2 für die Anforderungsergänzungen b, c, d und e. Wenn also diese fehlerhaften Angaben gemacht werden, wird laut Spezifikation erwartet, dass Postcondition 2 eintritt, der Empfänger sollte also keine Textnachricht erhalten. Postcondition 3 stellt das Target dar und wird hier gesondert dargestellt.

Indem man also die Anforderungsergänzungen den Postconditions zuordnet, erhält man aus der Dienstbeschreibung die relevanten Dienstpfade. Insgesamt können für den Click2IM-Dienst sechs Dienstpfade identifiziert werden, die sowohl der Dienstentwickler als auch der Testentwickler berücksichtigen müssen.

4.5 Identifizierung der Kommunikationsschnittstellen

Die Kommunikationsschnittstellen stellen jeweils das Gegenstück der klassifizierten Rollen dar. Sie repräsentieren die realen Schnittstellen des Dienstes nach außen. Für den Click2IM können die folgenden Schnittstellen identifiziert werden:

- HTTP Server
- SIP UAC (User Agent Client) initRequest

Die Kommunikationsschnittstelle HTTP Server besagt, dass die vom Browser initiierten HTTP Request-Nachrichten (z.B. HTTP POST) dienstseitig empfangen und weiterverarbeitet werden. Die zweite Kommunikationsschnittstelle SIP UAC initRequest regelt hingegen die SIP-Kommunikation zwischen dem Dienst und dem Softphone. Bei der Beschreibung des ComGeneration-Ansatzes in Kapitel 2 wurden bereits die sogenannten Teilautomaten erwähnt, welche als Bausteine zur Erzeugung eines Verhaltensmodells herangezogen werden können. Für die Kommunikationsschnittstellen wurden im Rahmen des Projektes derartige Teilautomaten definiert. Die für den Click2IM-Dienst relevanten Teilautomaten kann der Testentwickler direkt aus den in der Dienstbeschreibung definierten Kommunikationsschnittstellen ableiten und als Basis für das Verhaltensmodell verwenden.

5 Zusammenfassung und Ausblick

In dieser Veröffentlichung wird ein Ansatz zur Spezifizierung und Beschreibung von Mehrwertdiensten vorgestellt. Im Rahmen des Projekts ComGeneration wird eine durchgängige Lösung zum Testen von Mehrwertdiensten von der Dienstbeschreibung bis zum ausführbaren Test entwickelt. Der oben gezeigte Überblick über das Framework lässt bereits auf die Mächtigkeit des hier präsentierten Lösungsansatzes schließen.

Die in diesem Aufsatz vorgeschlagene Dienstbeschreibung stellt die Grundlage für die Dienstentwicklung dar, ganz gleich mit welcher SCE der Entwickler arbeitet oder ob er den Dienst sogar manuell erstellen muss. Aus der geforderten Unabhängigkeit von einer bestimmten SCE folgt eine gewisse universelle Einsetzbarkeit des hier vorgestellten Ansatzes.

Auch für den Testentwickler stellt die Dienstbeschreibung die Grundlage für die Entwicklung von Testfällen dar. Der Testentwickler ermittelt aus der Dienstbeschreibung die benötigten modularen Teilautomaten und verknüpft diese, wie in der Dienstbe-

schreibung aus den Dienstpfaden ablesbar ist, miteinander. Daraus entsteht das Verhaltensmodell des Dienstes, aus dem automatisch TTCN3-Testfälle abgeleitet werden. Ein weiterer Vorteil dieses Vorgehens ist die automatische Generierung der Testfälle. Der Testentwickler muss nur die relevanten Teilautomaten auswählen und miteinander verknüpfen. Je umfangreicher und vollständiger der Pool an vordefinierten Teilautomaten ist, desto geringer fällt der Aufwand für den Testentwickler aus. Modulare Teilautomaten können dabei auch vom Testentwickler aus einfacheren Teilautomaten komponiert werden, hier entfaltet dieser Ansatz sein größtes Potential.

Dieser Aufsatz fokussiert hauptsächlich auf die Dienstbeschreibung. Der Aufbau, die Regeln und Anforderungen für die Erstellung dieser werden definiert und erläutert. Das Vorgehen beim Erstellen einer Dienstbeschreibung wird vorgestellt und anhand eines Beispiels gezeigt.

Für das hier vorgestellte Framework wird ein Prototyp implementiert. Der momentane Stand der Framework-Entwicklung erlaubt dem Testentwickler bereits die Erstellung der Statemachine für das Verhaltensmodell. Modulare Teilautomaten werden im Moment noch nicht unterstützt. Die automatische Generierung von TTCN-3-Testfällen aus dem Verhaltensmodell konnte aber bereits für einfache Dienste ähnlich dem hier vorgestellten durchgeführt werden. Der Test Case Generator kann das Verhaltensmodell analysieren und findet die möglichen Testfälle aus dem Verhaltensmodell heraus. Für jeden gefundenen Testfall generiert der Test Case Generator den entsprechenden TTCN-3 Code.

Die Dienstbeschreibung muss hinreichend genau sein. Eine zu genaue Beschreibung des Dienstes führt zu umfangreichen Dokumenten, andererseits kann eine ungenaue Dienstbeschreibung zu Fehlinterpretationen durch den Dienstentwickler oder den Testentwickler führen. Durch die bisherige Formalisierung werden diese Schwierigkeiten weitgehend vermieden. Dennoch soll in der nächsten Entwicklungsstufe ein Meilenstein- und Versionskonzept in den ComGeneration-Ansatz aufgenommen werden.

An der Integration der modularen Teilautomaten wird im Moment gearbeitet. Der Testentwickler kann dann direkt aus der Dienstbeschreibung die benötigten Teilautomaten erkennen, diese im Framework auswählen und zu einem Verhaltensmodell zusammenfügen. Damit wäre das Ziel einer durchgängigen Lösung erreicht, die nebenbei noch viele reizvolle Vorteile bietet.

6 Literatur

- [1] <http://www.ecs.fh-osnabrueck.de/27619.html>
- [2] Wacht, P.; Eichelmann, T.; Lehmann, A.; Trick, U.: "A New Approach to Design Graphically Functional Tests for Communication Services". Fourth IFIP International Conference on New Technologies, Mobility and Security, Paris, Frankreich, 2011
- [3] Wacht, P.; Lehmann, A.; Eichelmann, T.; Fuhrmann, W.; Trick, U.; Ghita, B.: "Integration of Model-Based Functional Testing Procedures within a Creation Environment for Value Added Services", Sixth Collaborative Research Symposium on Security, E-learning, Internet and Networking (SEIN 2010), Plymouth, United Kingdom, 2010

Published in *Proceedings of the Fourth International Conference on Internet Technologies & Applications (ITA 2011)*, pp. 262-269, Wrexham, United Kingdom, ISBN: 978-0-946881-68-0

A NEW APPROACH TO MODEL A FORMALISED DESCRIPTION OF A COMMUNICATION SERVICE FOR THE PURPOSE OF FUNCTIONAL TESTING

Patrick Wacht^{1,2}, Thomas Eichelmann^{1,2}, Armin Lehmann^{1,2}, Woldemar Fuhrmann³, Ulrich Trick¹ and Bogdan Ghita²

¹Research Group for Telecommunication Networks,
University of Applied Sciences Frankfurt/M., Germany

²Centre for Security, Communications and Network Research,
University of Applied Sciences Plymouth, United Kingdom

³University of Applied Sciences Darmstadt, Germany
wacht@e-technik.org

ABSTRACT

This paper presents a concept of how a service provider can verify that an implemented communication service meets the requirements of a customer. This requires functional tests which are derived from a finite state machine-based behaviour model being composed from predefined modular sub finite state machines. As the composition of these modular finite state machines to a behaviour model is done by retrieving information from the requirements specification or rather Service Description, the model reflects the business logic of the communication service. A case study of the modelling procedure is shown in this paper by means of an example service.

KEYWORDS

Functional Testing; Behaviour Model; Finite State Machine; Model-Based Testing

1. INTRODUCTION

The complexity of value-added communication services is ever increasing. In the telecommunication domain, a malfunctioning of services may be costly or even compromises the reputation of a specific service provider. Therefore, functional testing procedures have to be executed consequently before the delivering of the service to a customer, because the provider has to assure that the service is executed properly and does not affect other running services within the provider's service environment.

Functional testing is considered a sub-category of black-box testing and the construction of the test cases is solely done manually by a test developer from the information given in a specification which is supposed to define the behaviour of a system or rather service. In general, the test developer has to spend a significant amount of time on test case design, test data selection, and test evaluation because there are no adequate tools available to automate these tasks for testing of communication services. So, new mechanisms have to be evolved to help overcome this situation, thus increasing both efficiency and effectiveness of the testing process.

This paper which demonstrates concepts of the corresponding project ComGeneration [1] proposes a new approach to compose a so-called behaviour model from predefined building blocks which can be created by a test developer from the information he could retrieve from a certain requirements specification. Both the behaviour model and the predefined building blocks

are finite state machines (FSM) whose paths represent possible message flows. To create the behaviour model, the test developer will use a graphical editor to design the model itself and provide the test configuration and test data. The whole modelling procedure will be the focus of this paper.

The retrieving of test cases from the behaviour model and the generation and execution on the System under Test (SUT) will not be discussed. Further information on this purpose can be read in [2].

The remainder of this paper is structured as follows: Section 2 introduces the related research. In Section 3, the consistent concept of the approach is described. Section 4 contains a case study which introduces the development of a behaviour model by means of a service example. Finally, a conclusion is provided in Section 5.

2. RELATED WORKS

Among the existing researches, test case generation based on models – model-based testing (MBT) – describing the intended behaviour of a system is proposed by different authors. The MBT approach is used for various kinds of software.

Conformiq [3] describes tests by UML state diagrams, but the focus is not to describe the service from the view of the SUT, but from the view of the test components. A similar approach from Yuan et. al [4] describes test case generation from UML activity diagrams, but the main focus is about testing Web Service compositions with the help of TTCN-3. Gönczy et. al [5] address the testing of service infrastructure components against their specifications. They proposed a technique to synthesise a compact Petri net representation for the possible interactions between the service under test and the test environment. Concrete test cases can be defined by a sequence of controllable actions in this Petri net. Brucker et. al [6] report on their experience on how model-based descriptions can be used to derive tests for security policies. They were able to derive tests from models for stateless and statefull firewalls. Ali et. al [7] discuss closely related work to those of the author. In their approach, the use of UML 2.0 state machines, composed of several sub machines, is proposed. They implemented a model-based testing tool with the name TRUST (Transformation-based tool for Uml-baSed Testing) which is able to flatten the complex state machine and transform it into a test model. However, their approach still requires a lot of interactions with a user and therefore is not applicable for automated test execution as desired by the author's research project. Wiczorek et. al [8] developed an infrastructure in which complex software systems are described by the usage of model-driven engineering (MDE). The main goal of this project is to develop a standardised solution to improve the quality and productivity. Testing aspects are limited to model verification techniques and model-based simulations, allowing only the observation of the service's behaviour. Zhang Xiaoyan et al. [9] research generates test cases based on the OWL-S requirement model in order to test the interaction of several Web Services. Pretschner et. al [10] give a general overview about common approaches and challenges model-driven testing is facing. Tretmans et. al [11] highlight the benefits for a completely automated test support originating from the test code generation from a model, over the test execution to the analysis of the test results.

The introduced research activities have in common that they only support artefacts of the whole software testing process. Most of the concepts do not provide a consequent procedure from the requirements specification to the execution of tests and, furthermore, do not provide any predefined building blocks like the modular FSMs in the ComGeneration approach.

3. CONCEPT OVERVIEW

Before looking at the practical steps being demonstrated in the upcoming case study in Section 4, it is worthwhile having a look at the ComGeneration approach [2; 12]. Figure 1 gives an abstract overview.

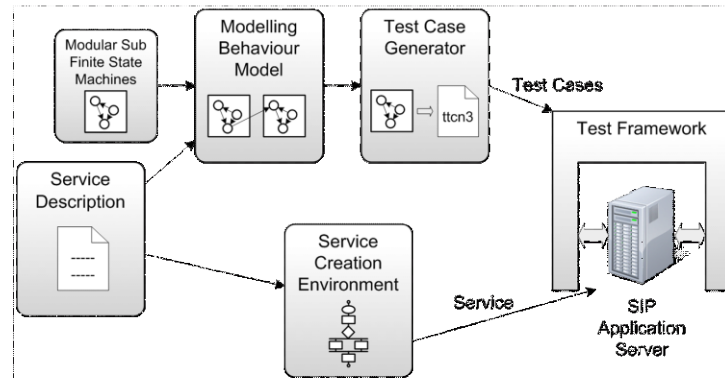


Figure 1. ComGeneration Architecture

The shown architecture can be divided into two main paths: The Service Development and the Test Development. Both the paths have their origin in the initial Service Description that can be seen as a sort of requirements specification for communication services. The Service Description consists of a document containing specific use-case related information and is created by the service provider in consultation with a customer. It contains all possible demands a customer might have for a communication service.

Once the Service Description is defined, both the Service Development and the Test Development can begin in parallel. In the presented approach, it is not ultimately defined which Service Creation Environment (SCE) is used to develop and subsequently deploy a specific service. However, the base for the development and deployment of services with any SCE is the Service Description. The output will then be a service which is deployed on a SIP Application Server.

The Test Development process starts with the test developer who has to interpret the Service Description properly and extracts the relevant service information for the test purpose. Afterwards, he chooses the service-related characteristics out of a repository of so-called predefined modular FSMs. These state machines cover typical service characteristics like protocol sequences for SIP (Session Initiation Protocol) or HTTP (Hypertext Transfer Protocol). By composing the chosen predefined modular FSMs, the test developer creates a behaviour model, which describes the possible behaviour of a value added service. Once the behaviour model is created it is passed to the Test Case Generator (TCG) which contains an algorithm to automatically generate the service-specific abstract test cases by identifying every possible path through the FSM. After the generation of these abstract test cases is done, they are afterwards converted into executable TTCN-3 (Testing and Test Control Notation) test cases. TTCN-3 is an abstract test scripting language which was standardised by ETSI [13] and ITU-T [14; 15] and supports the modularised creation of test scenarios for message and procedure based systems. In the ComGeneration approach, the execution of the executable TTCN-3 test cases on the deployed service is done within a TTCN-3 test framework.

The significant aspect of this paper is the way how the behaviour model is created by a test developer from the information he can retrieve from the Service Description.

4. CASE STUDY: THE EXAMPLE SERVICE CLICK2IM

In this section an evaluation of the modelling approach is presented with the help of an example service Click2IM (Click-2-Instant-Message).

4.1. Scenario Description

The function of the Click2IM service is to send a SIP MESSAGE with a specified text to a SIP phone having a specified SIP URI. Both the text and the SIP URI are input parameters in a form on a website. Once the parameters are sent by actuating a button, the service creates the SIP MESSAGE which contains the input text and sends it to the SIP phone with the SIP URI.

4.2. Service Description

As described in Section 3, the Service Description is maybe the most important aspect within the process of creating a service on the one hand and testing it against the requirements on the other hand. The Service Description can be seen as a kind of contract between the customer and a service provider.

In the following Table 1, an exemplary Service Description for the Click2IM service is illustrated. It contains a short textual description of the main functionality of the service. Furthermore, the participating roles are defined as well as the preconditions which have to be fulfilled to trigger the service. Also, the possible postconditions are defined within the table which show the possible consequences of the preconditions.

Table 1. Service Description for Click2IM Service

<u>Short Description</u>		
A website should deliver two input masks. The first input mask should contain the address or telephone number (SIP URI) of any participant and the second one should carry any kind of text. A button should be integrated on the website. When submitting it, the text included in the second input mask should be transferred to the address that was filled in the first input mask. If the SIP URI is not reachable or the text could not be transferred an error should occur on the website. If the transfer worked, a success message should occur.		
<u>Roles</u>	Web Browser [b], SIP UAS [s]	Assignment of the roles
<u>Preconditions</u>	Initiator sets any destination address Initiator sets text input Initiator confirms inputs	[b] [b] [b]
<u>Postconditions</u>	1. Receiver is unknown 2. Receiver does not get text message 3. Receiver gets text message	[b] [b] [b, s]
<u>Prosa</u>	Initiator wants to send a text message to a SIP phone.	

The third postcondition in the table is specially highlighted, because it represents the required functionality of the service which is defined as target. The other postconditions define how the service should behave when certain errors or malfunctions occur. For both the test developer and the service developer, it is necessary to define additional requirements which demonstrate how the postconditions are reached:

- a. Standard errors will be signalised, for instance timeouts (leads to → Postcondition 1)
- b. Maximal Length of the destination address can be 128 characters (leads to → Postcondition 2)
- c. Maximal length of the text input can be 256 characters (leads to → Postcondition 2)
- d. Special characters (except of '@') are not allowed in the destination address (leads to → Postcondition 2)
- e. Empty text inputs are not allowed (leads to → Postcondition 2)

For the target (Postcondition 3), there are no additional requirements defined as the limitations are already covered.

4.3. Tool Support

Once the Service Description is available the test developer can start to create the behaviour model. For this purpose, the ComGeneration approach provides three editors which have to be used:

1. Connectivity Editor
2. Test Data Editor
3. Behaviour Model Editor

To enable the modelling of the behaviour model, a lot of preliminary work has to be done. Both the Connectivity Editor and the Test Data Editor can contain service-specific conditions which have to be defined before adding further properties to the Behaviour Model. Within the Connectivity Editor, the test developer can define certain parameters for the service like a component, ports and timers. These parameters are derived from typical TTCN-3 test configurations. A component represents one test component within the test. The communication between the component and a system under test is realised through the connection of the local ports, which can be seen as well-defined interfaces [13]. The defined timers can be typical protocol timers or certain global timers.

In the Test Data Editor, the definition of the test data is done with the help of so-called templates. In dependency of the protocol, the test developer can define the inputs of the headers for certain protocol messages. In SIP, this would be the SIP Requests (e.g. INVITE, MESSAGE) and SIP Responses (e.g. 200 OK, 404 Not Found).

Within the Behaviour Model Editor, a test developer can create the referring behaviour model for a specific service by means of a FSM. This FSM describes in what order and under what conditions the test cases are executed. It represents the predicted reactions of the service which were defined as postconditions in the Service Description.

4.4. Creation of the Behaviour Model

Based on the information the test developer retrieves from the Service Description, he first does the test configuration within the Connectivity Editor. Independent of the service he wants to test, he first has to define a so-called component which can be seen as the central element within the Connectivity Editor. Because the two roles Web Browser (HTTP) and SIP UAS (SIP) are mentioned, the test developer exactly knows that he will need the two ports SIP and HTTP to cover the sending and receiving of messages from these protocols. Then, he defines two global timers which may be used to protect the test from infinite waiting for service responses.

Finally, the test developer has to define so-called message variables. Such message variables usually represent protocol messages, for instance SIP or HTTP Requests and Responses which might occur within the test procedure. Because of the message variable's acquisition to protocols, they are usually used in combination with the defined ports. There are other elements

like guards and actions which are not yet supported completely by the editors. Nevertheless, the modelling can be done. Figure 2 shows the configurations for the Click2IM service within the Connectivity Editor.

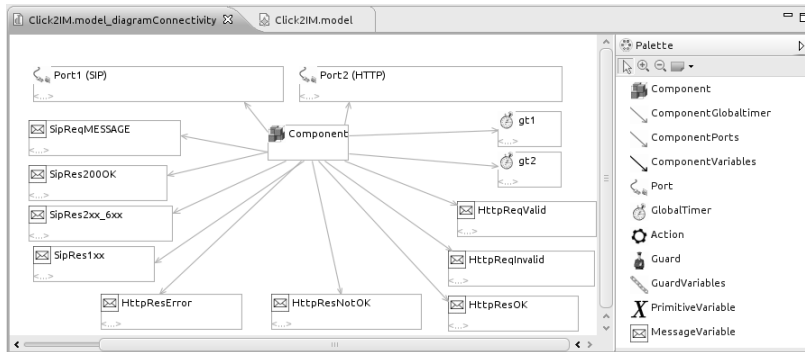


Figure 2. Connectivity Editor for Click2IM

All the Message Variables that were defined in the Connectivity Editor are connected to so-called test data templates. These templates can be edited in the Test Data Editor. For both the protocols SIP and HTTP it is possible to configure a lot of values for headers that were defined in the protocol specifications. Figure 3 demonstrates the configuration of a template belonging to the Message Type SIP Request. Here, the headers for the SIP MESSAGE are edited.

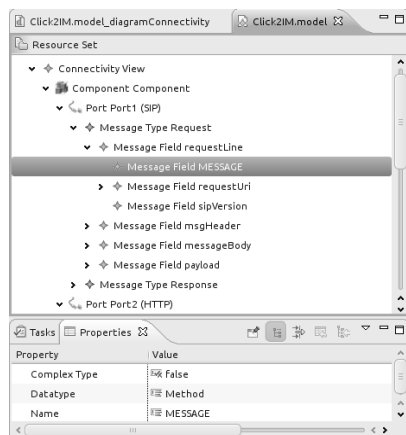


Figure 3. Test Data Editor for Click2IM

Once all the configuration and definition of protocol messages and ports is done, the test developer can start to model the behaviour model. At first he has to choose the relevant sub FSMs for the Click2IM service. Within the Service Description, the roles have been defined. The counterparts of these roles represent the relevant groups of sub FSMs. This would be on the

one hand the group of Web Server (HTTP) which consists of two sub FSMs and SIP UAC nonInvite which consists of three sub FSMs.

After choosing the correct sub FSMs, the test developer has to follow the instructions within the Service Description and has to reproduce the behaviour within the composition of the sub FSMs. This composition is done by a concept called Transaction User (TU) which acts as a mediator between possible client and server roles. The whole concept is described in [2].

Altogether, the behaviour model consists of five sub FSMs which have to be composed due to the specification. In the following, the FSM compositions referring to the case “Success” is introduced. From the initial state in the behaviour model, a HTTP POST Message is expected which contains the input text and SIP URI as parameters. Once this is done, the current state is “HttpRequest_Server”. Then, the service initiates the sending of the SIP MESSAGE which contains the input text to a SIP phone with the SIP URI. The transition from “HttpRequest_Server” to “SipUAC_nonInvite_init” contains a guard which compares the text inputs of the POST Message and the SIP MESSAGE to verify that they are identical. Afterwards, the transition from “SipUAC_nonInvite_init” to “SipUAC_nonInvite_term” refers to the expecting SIP 200 OK response message which verifies that the SIP MESSAGE has been successfully received by the SIP phone. The last transition is integrated between the states “SipUAC_nonInvite_term” and “HttpResponse_Server”. It represents the response of the HTTP Web Server to the originating HTTP POST request of the web browser. Figure 4 demonstrates the complete behaviour model.

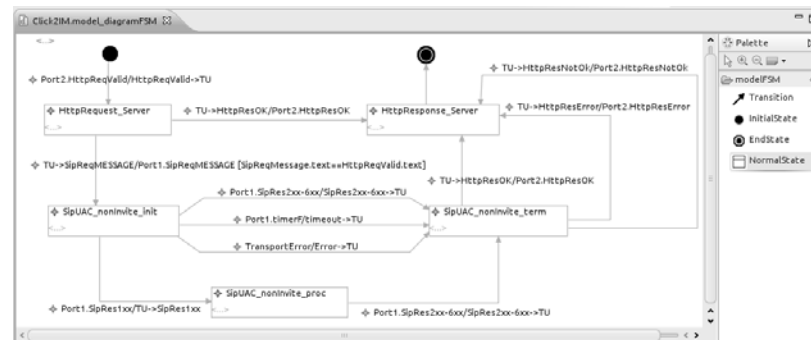


Figure 4. Behaviour Model for Click2IM

5. CONCLUSION

In this paper, we have introduced an approach to automate functional testing of communication services by means of creating a so-called behaviour model from which the relevant test cases are derived. For the modelling purpose, a test developer has to get a deep knowledge about the service requirements from the Service Description and then has to build the behaviour model by composing the sub FSMs. This procedure is simplified and accelerated due to the supply of three editors, the Connectivity Editor, the Test Data Editor and the Behaviour Model Editor. Besides, enhancements like the support of additional protocols and sub FSMs can be easily integrated as the software is modular-based.

Further work should address the improvement of the handling and the support of additional elements within the Connectivity Editor like actions and guards to increase the readability of the

Behaviour Model. Moreover, the evaluation of the concept from the Service Description to the execution of test cases on the System under Test, which was not the focus of this paper, is planned near-term.

ACKNOWLEDGEMENTS

The research project ComGeneration providing the basis for this publication was partially funded by the Federal Ministry of Education and Research (BMBF) of the Federal Republic of Germany under grant number 1724B09. The authors of this publication are in charge of its content.

REFERENCES

- [1] ComGeneration project website: <http://www.ecs.fh-osnabrueck.de/27619.html>
- [2] P. Wacht et al., “A new Approach to design graphically Functional Tests for Communication Services”, in *Proceedings of the Fourth IFIP International Conference on New Technologies, Mobility and Security*, Paris, France, 2011
- [3] Conformiq website: <http://www.conformiq.com/>
- [4] Quilu Yuan et al., “A Model Driven Approach Toward Business Process Test Case Generation”, in *Proceedings of the Tenth IEEE International Symposium on Web Site Evolution*, Beijing, China, 2008
- [5] Gönczy et al., “Model-based Testing of Service Infrastructure Components”, in *Proceedings of the Nineteenth IFIP TC6/WG6.1 International Conference*, Tallinn, Estonia, 2007
- [6] Brucker et al., “Verified Firewall Policy Transformations for Test Case Generation”, in *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST2010)*, Paris, France, 2010
- [7] S. Ali et al., “Model Transformations as a Strategy to automate Model-Based Testing – A Tool and Industrial Case Studies, Version 1.0”, Technical Report, 2010
- [8] S. Wiczorek et al., “Enhancing Test Driven Development with Model Based Testing and Performance Analysis”, in *Proceedings of the Practice and Research Techniques Academic & Industrial Conference TAIC PART*, Cumberland Lodge, Windsor, UK, 2008
- [9] Zhang Xiaoyan, Huang ming, Yu Ying, “OWL-S Based Test Case Generation”, in *Journal of Beijing University of Aeronautics and Astronautics*, Beijing, China, 2008
- [10] A. Pretschner et al., “One Evaluation of Model-Based Testing and its Automation”, in *Proceedings of the Twenty Seventh International Conference on Software Engineering*, St. Louis, USA, 2005
- [11] G. Tretmans, H. Brinksma, “Automated Model-Based Testing”, in *Proceedings of the First European Conference on Model-Driven Software Engineering*, Nuremberg, Germany, 2003
- [12] P. Wacht et al., “Integration of Model-Based Functional Testing Procedures within a Creation Environment for Value Added Services”, in *Proceedings of the Sixth Collaborative Research Symposium on Security, E-learning, Internet and Networking (SEIN 2010)*, Plymouth, United Kingdom, 2010
- [13] EG 201 873-1: Methods for Testing and Specification (MTS): The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language . ETSI, 2008
- [14] Recommendation Z.140: The Tree and Tabular Combined Notation version 3 (TTCN-3): Core Language. ITU-T, 2001
- [15] Recommendation Z.141: The Tree and Tabular Combined Notation version 3 (TTCN-3): Tabular Presentation Format. ITU-T, 2001

Published in *Proceedings for the Second International Conference on Future Generation Communication Technology (FGCT 2013)*, pp. 59-64, London, United Kingdom, IEEE, ISBN: 978-1-4799-2974-0

A New Service Description for Communication Services as Basis for Automated Functional Testing

P. Wacht, U. Trick
Research Group for Telecommunication Networks
University of Applied Sciences Frankfurt
Frankfurt, Germany
{wacht, trick}@e-technik.org

W. Fuhrmann
University of Applied Sciences Darmstadt
Darmstadt, Germany
w.fuhrmann@fbi.h-da.de

P. Wacht, B. Ghita
Centre for Security, Communications, and Network research
University of Plymouth
Plymouth, United Kingdom
{patrick.wacht, bogdan.ghita}@plymouth.ac.uk

Abstract—The advances in the telecommunication domain to support complex communication services has resulted in a need for a new approach to automatically verify that the communication services meet the demands of the customers. This paper presents a concept for automated functional testing by means of a novel test framework. Within the framework, the tests are automatically derived from a proposed new sort of requirements specification for communication services, the Service Description, and afterwards generated by means of predefined test modules. Finally, the test cases are executed against the System under Test, the communication service.

Keywords—automated functional testing; communication services; requirements specification; test framework; testing methodology

I. INTRODUCTION

In the telecommunication domain, network operators and service providers aim for fast, easy and cost-efficient provisioning of value-added communication services. A fast transition from concept to market product and low price of new communication services is necessary due to the increasing competition in the telecommunication industry. The sum of these demands leads to reducing complete and sufficient functional testing which has a bad impact on the quality of the service. Moreover, functional testing procedures have to be executed consequently before the delivering of a communication service to a customer because the service provider has to assure that the communication service is executed properly and according to the specified requirements and that the communication service may not cause undesired behaviour within the provider's service environment.

In order to avoid these problems, the whole test development cycle specifically for communication services has to be improved. This starts with the requirements analysis which is usually standard Unified Modelling Language (UML) use case design [1; 2], with mostly natural language-based descriptions. This often leads to a lack of clarity as it is difficult

to use language in a precise and unambiguous way. Besides the requirements analysis, the service testing is oftentimes manually done by test developers who gain their knowledge about a service's functionality from the natural language-based requirements specification. The test developer has to spend a significant amount of time on test case design, test data selection, and test evaluation.

In this paper, we propose a new test framework in order to do automatic functional testing of communication services. The foundation of the testing methodology is based on the definition of a new sort of requirements specification for communication services, the Service Description. After the Service Description is specified for a new communication service, it is parsed by a special test framework artifact which reads out the significant content and generates a formal behaviour model by composing predefined parameterised test modules. From the behaviour model, the functional test cases are derived and generated into executable TTCN-3 test cases which are subsequently executed against the communication service within a TTCN-3 test execution environment.

The remainder of this paper is structured as follows: the related work is presented in section 2. Section 3 introduces the novel architecture of the test framework and describes the testing methodology. Afterwards, section 4 discusses the demands on a new sort of requirements specifications for communication services and introduces the proposed Service Description. A simplified example of a communication service specification using the Service Description is discussed in section 5 and section 6 concludes the paper.

II. RELATED WORK

Our survey of the related literature shows that many different methodologies have been developed in the field of automated testing, mostly in the field of process-based systems. These approaches either generate test paths directly from code, based on data and control flow information [3; 4], or translate the code into formal specification languages like Petri nets [5],

[6], to perform the model checking and test derivation. A major disadvantage of these approaches is that the tests cannot detect the deviations from the functional specifications as the tests are directly derived from the code.

A series of methods is proposed in [7] to capture requirements and then manually transform them into conceptual models composed of object models (e.g. class diagrams), dynamic models (e.g. state machines and sequence diagrams) and functional diagrams. The authors introduce a set of techniques for users to precisely specify requirements and describe rules how the users can derive conceptual models from these requirements. The paper [7] does not mention a complete transformation method. Besides, the effort for a human to define both requirements and conceptual models seems to be very high.

An approach to generate finite state machines from use cases in restricted Natural Language (NL) is proposed in [8]. The approach needs the existence of a domain model which serves two purposes: a lexicon for the NL analysis of use cases, and the structural basis of the state transition graphs being generated. The domain model acts as the lexicon for NL analysis of the use cases because the model elements of the domain model are used to document the use cases. It is imaginable that an enormous user effort is needed to obtain a domain model containing classes, associations, and operations which are indispensable for generating state machines. There is no proof that the restricted NL is sufficient to describe the use cases. Yet no case study is presented to evaluate the approach.

In [9], the authors use a behaviour engineering methodology to formalise and validate a requirements specification and extend it with appropriate test activities. It is shown how testing information may be weaved into behaviour trees by identifying the system's boundaries and the definition of test action. The approach lacks the information of how tests are generated and executed but it seems necessary to transform the behaviour trees in state machines.

The generation of test cases from complete system requirements models is discussed in [10]. The model is described in a requirements specification language (SpecTRM-RL) which is based on a formal state machine model. Nevertheless, according to the authors, the notation is simple to read and understand for non-experts. However, the approach allows the definition of requirements models in different degrees of abstraction. Besides, the generation of tests and their execution is indeed discussed but not further defined.

A tool-based methodology to model-driven system testing of service-oriented systems is introduced in [11]. Additionally, it provides full traceability between the requirements, the system and test model. This aspect, however, leads to an amount of work for the human as the requirements have to be specified, the system model has to be defined as well as the test model. For the execution of test cases, the outdated technologies RMI (Remote Method Invocation) and CORBA (Common Object Request Broker Architecture) have been applied.

Whittle and Schumann proposed an approach [12] to automatically generate UML state machines from UML

sequence diagrams. However, widespread modeling techniques like UML are too generic and lack the formalism required for domain modelling such as the requirements modeling of communication services.

Most approaches described in literature lack the definition of a testing methodology from the definition of the requirements until generation and subsequent evaluation of the functional tests. There is no discussed framework covering these steps specifically for communication services. Besides the standard way of defining requirements of communication services through UML use case design no further approaches are discussed so far.

III. TEST FRAMEWORK ARCHITECTURE

This section provides an overview of the underlying components and principles within the developed test framework.

Fig. 1 shows the artifacts of the test framework. The workflow of the testing methodology is triggered by a Test Developer whose role is the compilation of the Service Description. The Service Description is a new sort of requirements specification for communication services for the purpose of specification-based functional testing. It contains static architectural definitions describing the participating roles involved in the consumption of a communication service and their relationships as well as dynamic behavioural definitions specifying use-case related requirements on the part of the communication service. In the compilation phase, the Test Developer has to follow a well-defined guideline to define a Service Description for a communication service. Within the testing methodology this is the only task being carried out by a human, the subsequent process performs automatically. A more detailed introduction regarding the Service Description will follow in section 4.

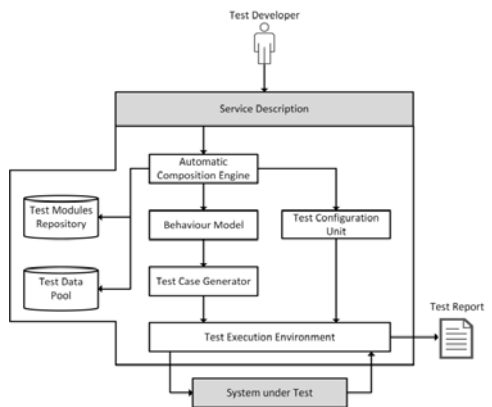


Fig. 1. Test framework artifacts and methodology

According to Fig. 1, the Service Description will be delivered to a very significant component within the testing

methodology, the Automatic Composition Engine (ACE). The main task of the ACE is the generation of a system model, the Behaviour Model, which is a complete formal model or rather Extended Finite State Machine (EFSM) describing the desired and possible behaviour of the specified communication service.

In order to generate the Behaviour Model, the ACE first parses the architectural definitions from the Service Description and forwards them to the Test Configuration Unit (TCU). The TCU thereupon extracts the relevant information for the Test Execution Environment (TEE) such as the System under Test (SUT) addressability, the participating test components and the data structures being exchanged between SUT and test system.

The ACE parses the behavioural part of the Service Description and identifies the participating roles within the specified requirements to select suitable test modules from the so-called Test Modules Repository (TMR). The TMR is a database containing predefined modular EFSMs, so-called test modules, which cover typical communication service characteristics such as sequences of multimedia protocols like SIP (Session Initiation Protocol) or RTP (Real-Time Transport Protocol) and other important protocols, e.g. HTTP (Hypertext Transfer Protocol). The test modules usually define a protocol-specific behaviour of a certain use case, e.g. the sending of an instant message by using the SIP protocol, and cover both standard behaviour as well as possible alternative behaviour like timeouts. To sum it up, the test modules define the standard compliant behaviour of a certain use case. Additionally, the test modules are parameterised in order to configure the test data.

After selecting the appropriate test modules from the TMR, the ACE connects to the Test Data Pool, a database containing collections of test data templates for each test module within the TMR. Then the ACE chooses the adequate test data templates and the parameters from the Service Description are included. After that, the ACE starts with the composition of the test modules. Each test module has interfaces which are linked to the existing states within the underlying EFSM of a test module. If two test modules are to be combined, the originating test module and the destination test module are connected with a transition between their interfaces. The task of the ACE is to realise the connection according to the use-case related information within the Service Description. Obviously, not all the interfaces within one test module has to be operated, it depends on the descriptions. However, at least the interfaces within the start state and the end state of a test module have to be activated excepting the first and the last test module to be composed.

After the composition of the chosen test modules is fulfilled the dependencies of the parameterisations for each test module have to be dissolved. This is necessary in order to reuse and change parameter values being defined in one test module for the other test modules within the composed model. This is important if certain parameter values defined in test module A have to be reused in test module B. This could be for instance a SIP URI which was defined in A and has to be reused in B.

As mentioned before, the result of the composition is the Behaviour Model which is then delivered to the Test Case

Generator (TCG). The TCG contains a test case finder which uses an algorithm to enable the derivation of abstract test cases from the Behaviour Model. This algorithm optimises the traversal of the EFSM by combining depth-first search and breadth-first search. After the extraction of the abstract test cases, a test code generator translates them into executable test cases by means of a special mapping concept which is described in [13]. The executable test cases are defined in TTCN-3 (Testing and Test Control Notation), a test scripting language which was standardised by ETSI [14] and ITU-T [15; 16], and supports the modularised creation of test scenarios for message and procedure based systems.

The final step of the methodology takes place within the TEE which receives both the executable TTCN-3 test cases from the TCG and the relevant information about the SUT and the participating test components from the TCU. Based on the information, the TEE selects the appropriate system adapter and codecs. The system adapter [17] adapts the communication of the TTCN-3 test system to the specific execution platform of the SUT whereas the codec is responsible for the encoding and decoding of TTCN-3 values into bitstrings so that the data can be sent to the SUT. Finally, the system adapter and codecs are added to the Test Suite and the generated TTCN-3 test cases are executed against the SUT. A test log is written which documents the test case execution and the reactions of the SUT. The data is formatted and integrated into a Test Report to demonstrate if all the tests were successful due to the defined requirements specified in the Service Description. If there are mismatches the whole process has to be verified, the SUT as well as the Service Description.

IV. SERVICE DESCRIPTION

A well-defined requirements specification is the critical component when it comes to functional testing as it represents the foundation for every derived test case. Especially with reference to the proposed test framework introduced in the last section, several demands on the Service Description were discussed.

A. Demands on the Service Description

The Service Description should meet some general demands which are relevant for any kind of specification document. First of all, the Service Description should be complete and has to contain all the requirements which describe exactly the desired behaviour of a communication service. The specified requirements should be understandable and not ambiguous. The Service Description should not contain any contradictions and changes can be done without difficulties. It should be machine-readable and interpretable so that the ACE is able to parse the content.

Besides the general demands, the proposed test framework requires some further specific demands with reference to the artifacts within the testing methodology. The ACE for instance requires the description of behavioural aspects which can be described in terms of use-case related requirements. Each requirement has to be traceable throughout the whole testing process from its definition within the Service Description by the Test Developer until the execution of the automatically

derived test cases. Therefore, a formal semantic relationship between the requirements and test cases has to exist. Also, the requirements have to contain information about the participating roles so that the ACE can select the appropriate test modules while parsing the Service Description. As the requirements describe the specification of a communication service they address a subset of the protocol-specific behaviour defined in the test modules. Possible relations and dependencies between requirements can lead to compositions of test modules. Another important demand on the Service Description is the support for applying the test data from the Test Data Pool. Within the requirements it should be easily possible to parameterise and address the test data sets.

Finally, with reference to the test configuration within the Test Execution Environment, the Service Description has to contain all the relevant information about the test architecture, which is a representation of the structural aspects of the test system, such as SUT information, test components and required codecs.

B. Service Description structure

As mentioned in section 3, the proposed Service Description is subdivided into architectural and behavioural definitions. Besides, some further information is given. Fig. 2 illustrates the structure of the Service Description.

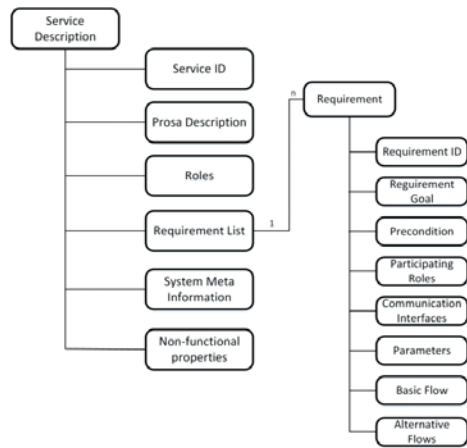


Fig. 2. Structure of the Service Description

The *Service Description* element is the root of every instance of a requirements document. It contains the *Service ID*, a unique identifier for the communication service to be specified. This is an important attribute as it determines the name of the test suite to be generated. The *Prosa Description* contains an abstract description of the communication service’s functionality. In the *Roles* attribute, all the participating users who consume the communication service are listed. Roles could be for instance Web Browsers or SIP Softphones. The definition of the roles is the basis for the selection of the test

components within the test configuration. Further test configuration properties are defined in the *System Meta Information* containing SUT information in order to build up the test configuration. Here, the service addressability is defined such as the service URI, IP addresses and port numbers. A predefined variable list is available to assure that the relevant parameters are set. The *Non-functional properties* contain non-functional requirements like costs.

A further important part of the Service Description is the *Requirement List* which defines the behavioural part and contains all the relevant requirements a communication service has to fulfill. The specification of each requirement in the *Requirement List* is well-defined by the following components in Table 1.

TABLE I. SPECIFICATION OF A REQUIREMENT

Component	Description
Requirement ID	Unique identifier for a requirement.
Requirement Goal	Prosa description of the requirement’s target.
Precondition	Determination of depending flows within other requirements that have to be carried out before the Basic Flow of this requirement can take place.
Participating Roles	List of the roles involved in this requirement.
Communication Interfaces	Definition of the relevant system side communication interfaces.
Parameters	Determination of the required parameters within this requirement.
Basic Flow	Description of the steps that have to be taken to achieve the target of the requirement.
Alternative Flows	Description of exceptional behaviour. Each step within a Basic Flow can lead to an Alternative Flow.

The significant part of a requirement is the use case description of the *Basic Flows* and *Alternative Flows*. In standard textual UML use case design, natural language-based descriptions are used. There are many documented approaches [18], where restriction rules for textual use case design are applied to reduce the imprecision and incompleteness. However, even if restricted vocabulary is used, formulation oftentimes is confusing and error-prone. The larger a requirements specification is the more problems arise disproportionately with natural language-based specifications. With reference to the testing methodology, the *Basic Flows* and *Alternative Flows*, the descriptions should be machine-readable so that the ACE can parse and interpret them. Therefore, a new formal approach is required which enables the description of behaviour flows and realises the reference to the test modules within the TMR and the test data.

As an appropriate formal method the usage of a process algebra notation has been found, the pi-calculus [19]. In general, the pi-calculus is a simple language to specify interactive message-passing programs. It provides mathematical foundations of some modern workflow languages like the Business Process Execution Language (BPEL) and is more concise than automata, very expressive

and even easier to develop. However, the pi-calculus is so minimal that it does not contain primitives such as numbers, booleans, variables, functions, or even usual flow control statements such as if-then-else constructions. The syntax just consists of a set of prefixes and process expressions which is illustrated in Table II.

TABLE II. BASIC PI-CALCULUS SYNTAX

Syntax	Description
$P ::= 0$	Process P is a null process.
$P \mid Q$	Parallel composition of processes P and Q .
$!P$	Replication of process P .
$'a\langle x \rangle.P$	x is sent along channel a , then process P starts.
$a(x).P$	Channel a receives x , then process P starts.

The mentioned limitations of pi-calculus may justify why it has not been applied as a requirements specification language for functional testing methodologies so far. Therefore, we propose an applied pi-calculus language in order to describe the *Basic Flow* and *Alternative Flows* within the requirements properly. The conceptual idea was derived from [20]. In that approach the grammar for processes is similar to the one in the pi-calculus, except that messages can contain terms. In our proposed pi-calculus, we reuse the ideas of terms to define flow control statements, variable usage and method invocation. Furthermore, we reuse the channels to express possible outputs and inputs on the part of the system side communication interfaces.

Our proposed enhancements of the pi-calculus syntax are illustrated in Table III.

TABLE III. ENHANCED PI-CALCULUS SYNTAX

Syntax	Description
$\text{if } x = \{value\} \text{ then } P \text{ else } Q$	If the variable x contains the value the process P starts otherwise process Q .
$'a\langle httpServer \rightarrow response(200) \rangle$	Through channel a , a 200 response is sent from the communication interface $httpServer$.
$response \rightarrow statusCode=200$	The attribute $statusCode$ of the complex variable $response$ is set to the value 200.

The complex variables such as *response* within the description of Table III are parameters which can be loaded from the Test Data Pool. The arrow symbolises the access to the attributes of the complex data structure. In standard programming languages this would be the dot operator. As the dot has a different meaning in pi-calculus, namely the separation of process steps, the arrow is used in our approach.

With reference to the description of the requirements, each *Basic Flow* and each *Alternative Flow* can be defined by one pi-calculus process in the Service Description. Again, each process contains n channels where each is representing the communication between the communication service and the components which are depending on the defined communication interfaces. The mentioned enhancements of the

pi-calculus show that certain conditions can lead to different behaviour which is specified through different following processes.

In the following, an example of a communication service specification with the Service Description will be described.

V. EXAMPLE

The example communication service being specified by the Service Description is called Click-2-Instant-Message. The service flow starts with a text message and a destination SIP URI being typed in by a user on a website. By actuating a button the message is sent via HTTP protocol to an application server with the deployed Click-2-Instant-Message service. Subsequently, the service sends a SIP Message containing the text message from the website to the SIP phone with the stated SIP URI.

The architectural part of the Service Description enables the building of the test configuration and is illustrated in Table IV.

TABLE IV. EXAMPLE SERVICE DESCRIPTION ARCHITECTURAL PART

Service ID	Click-2-Instant-Message
Prosa Description	A website should deliver two input masks. The first input mask should contain the address or telephone number (SIP URI) of any participant and the second one should carry any kind of text message. A button should be integrated on the website. When submitting it, the text included in the second input mask should be transferred to the address that was filled in the first input mask. If no text was typed, the user should be informed with "No text input" on the website. If the SIP URI was invalid the user should be informed with "No valid SIP URI". If the transfer worked, a success message should occur, "Message sent successfully".
Roles	Web Browser, SIP Softphone
System Meta Information	ServiceURI: sipclick2IM@sip.de
Non-functional properties	None

The defined roles leads to the fact that two test components are required with one understanding the HTTP protocol (Web Browser) and the other one understanding the SIP protocol (Softphone). The SUT is reachable through the ServiceURI which is specified in the *System Meta Information*.

Furthermore, the behavioural part of the Click-2-Instant-Message Service Description contains one requirement. The requirement determines amongst others the communication interfaces which describe the communication channels from the SUT to the test components. The declaration of the communication interfaces automatically leads to the selection of the test modules, in this example *HTTP_Server* and the *SIP_UAClient_MESSAGE*. A more detailed description of the structure of the test modules is described in [21]. Every test module contains a set of parameters. The relevant ones for the specified flows in the requirements have to be determined like in Table V.

TABLE V. EXAMPLE SERVICE DESCRIPTION BEHAVIOURAL PART

Requirement ID	1
Requirement Goal	Initiator wants to send a text message from a website to a SIP softphone.
Precondition	None
Participating Roles	Web Browser, SIP Softphone
Communication Interfaces	HTTP Server [w] → channel a SIP UAClient MESSAGE [s] → channel b
Parameters	[w]→httpRequest; [w]→httpResponse; [s]→sipRequestMessage; [s]→sipResponse2xx 6xx
Basic Flow	$P ::= a([w] \rightarrow \text{httpRequest}(\text{text}, \text{targetURI}))$; if $\text{text} = \text{NULL}$ then Q else ; if $\text{isValidURI}(\text{targetURI}) = \text{false}$ then R else ; $b([s] \rightarrow \text{sipRequestMessage}(\text{targetURI}, \text{text}))$; $b([s] \rightarrow \text{sipResponse2xx } 6xx(200))$; $a([w] \rightarrow \text{httpResponse}(200, \text{"Message sent successfully"})) > .0$
Alternative Flow 1	$Q ::= 'a<[w] \rightarrow \text{httpResponse}(200, \text{"No text input"}) > .0$
Alternative Flow 2	$R ::= 'a<[w] \rightarrow \text{httpResponse}(200, \text{"No valid SIP URI"}) > .0$

The behavioural flows are described in the proposed pi-calculus syntax. The Basic Flow specifies the process P with an incoming HTTP request over the channel a containing the parameters text and targetURI . Then the content of both parameters is checked. If text does not have content, process Q is triggered, otherwise if targetURI contains an invalid SIP URI, process R is triggered. If both parameters are correct, a SIP Message with the text is expected to be sent over the SIP channel b and acknowledged. At the end, a HTTP response is sent over channel a to inform that the transfer was successful. The sum of flow descriptions in this example define a specification of a service which describes a certain subset of the flows being contained in the test modules.

VI. CONCLUSION

Automated functional testing of communication services directly from a requirements specification requires its understandability, completeness and machine-readability. Such a requirements specification, the Service Description, was introduced and exemplified in this paper. Besides, its significance was discussed with reference to the proposed test framework.

The presented approach empowers network operator and service providers to deliver high quality communication services in a cost and time optimised way to their customers. The services undergo a continuous testing procedure based on a new functional testing methodology.

REFERENCES

- [1] O. Ryndina, P. Kritzing, "Improving requirements specification for communication services with formalised use case models", Proceedings of the Southern African Telecommunication Networks and Applications Conference (SATNAC 2004), Spier Wine Estate, Western Cape, South Africa, September 2004.
- [2] A. Eberlein, M. Crowther, F. Halsall, "Development of new telecommunications services using an expert system", BT Technology Journal, vol. 15, pp. 2137-222, 1997.
- [3] J. Yan, Z. Li, Y. Yuan, W. Sun, and J. Zhang, "BPEL4WS Unit Testing: Test Case Generation Using a Concurrent Path Analysis Approach", Proceedings of the Seventeenth International Symposium on Software Reliability Engineering (ISSRE 2006), Raleigh, North Carolina, USA, November 2006.
- [4] Y. Yuan, Z. Li, and W. Sun, "A graph-search based approach to bpel4ws test generation", Proceedings of the International Conference on Software Engineering Advances, Papeete, Tahiti, French Polynesia, October 2006.
- [5] J. Garcia-Fanjul, J. Tuya, and C. de la Riva, "Generating test cases specifications for BPEL compositions of web services using SPIN", Proceedings of the International Workshop on Web Services: Modeling and Testing, Palermo, Italy, June 2006.
- [6] Y. Zheng, J. Zhou, and P. Krause, "A model checking based test case generation framework for web services", Proceedings of the International Conference on Information Technology (ITNG 2007), Las Vegas, Nevada, USA, April 2007.
- [7] E. Insfrán, O. Pastor, R. Wieringa, "Requirements Engineering-Based Conceptual Modelling", Requirements Engineering Journal, vol. 7, pp. 61-72, June 2002.
- [8] S. Somé, "An approach for the synthesis of state transition graphs from use cases", CSREA Press, vol. 1, pp. 456-462, 2003.
- [9] M.-F. Wendland, I. Schieferdecker, A. Vouffio-Feudjio, "Requirements-driven testing with behaviour trees", Proceedings of the Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW 11), Berlin, Germany, March 2013.
- [10] K. Kelley, "Automated Test Case Generation from Correct and Complete System Requirements Models", Proceedings of the IEEE Aerospace Conference, Big Sky, Montana, USA, March 2009.
- [11] M. Felderer, P. Zech, F. Fiedler, R. Breu, "A Tool-based methodology for System Testing of Service-Oriented Systems", Proceedings of the Second International Conference on Advances in System Testing and Validation Lifecycle (VALID 2010), Nice, France, August 2010.
- [12] J. Whittle, J. Schumann, "Generating statechart designs from scenarios", Proceedings of the Twentysecond International Conference on Software Engineering (ICSE 2000), Limerick, Ireland, June 2000.
- [13] P. Wacht, T. Eichelmann, A. Lehmann, and U. Trick, "A New Approach to Design Graphically Functional Tests for Communication Services", Proceedings of the Fourth IFIP International Conference on New Technologies, Mobility and Security (NTMS 2011), Paris, France, February 2011.
- [14] EG 201 873-1: Methods for Testing and Specification (MTS): The Testing and Test Control Notation version 3; Part 1; TTCN-3 Core Language, ETSI, 2008.
- [15] Recommendation Z.140: The Tree and Tabular Combined Notation version 3 (TTCN-3): Core Language. ITU-T, 2001.
- [16] Recommendation Z.141: The Tree and Tabular Combined Notation version 3 (TTCN-3): Tabular Presentation Format. ITU-T, 2001.
- [17] S. Blom et al., "TTCN-3 for Distributed Testing Embedded Software", Proceedings of the Sixth International Andrei Ershov Memorial Conference on Perspectives of Systems Informatics (PSI 2006), Berlin, Germany, 2007.
- [18] T. Yue, S. Ali, L. Briand, "Automated Transition from Use Cases to UML State Machines to Support State-based Testing", Proceedings of the Seventh European Conference on Modelling Foundations and Applications (ECMFA 2011), Birmingham, United Kingdom, June 2011.
- [19] D. Sangiorgi, "The Pi-Calculus: A Theory of Mobile Processes", Cambridge University Press, 2008.
- [20] M. Abadi, C. Fournet, "Mobile Values, New Names, and Secure Communication", Proceedings of the Twentieth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, United Kingdom, January 2001.
- [21] P. Wacht, T. Eichelmann, A. Lehmann, U. Trick, "A New Approach to Design Graphically Functional Tests for Communication Services", Proceedings of the Fourth IFIP International Conference on New Technologies, Mobility and Security (NTMS 2011), Paris, France, February 2011.