Efficient Test Case Derivation from Statecharts-Based Models

Patrick Wacht[1,3], Ulrich Trick[1], Woldemar Fuhrmann[2], and Bogdan Ghita[3]
[1]Research Group for Telecommunication Networks, Frankfurt University of Applied Sciences, Frankfurt, Germany
[2]Department of Computer Science, University of Applied Sciences Darmstadt, Darmstadt, Germany
[3]Centre for Security, Communications, and Network Research, Plymouth University, Plymouth, United Kingdom
wacht@e-technik.org, trick@e-technik.org, w.fuhrmann@fbi.h-da.de, bogdan.ghita@plymouth.ac.uk

***Abstract*—This paper presents important aspects of a novel framework for the automated functional testing of value-added telecommunication services. Besides the introduction of a new approach to model the potential behaviour of a service by means of the Statecharts notation, the paper describes an efficient test case derivation algorithm.**

Keywords—automated functional testing; test case generation; test framework; value-added services

I. INTRODUCTION

The demand for advanced value-added services in the telecommunication domain has increased enormously over the last years. This fact is a major challenge especially for service providers who have to provide a fast transition from concept to market product and have to offer low prices for their products to satisfy their customers. In order to face these challenges, service providers have integrated Service Creation Environments (SCE) to allow their developers to rapidly create new and individual value-added services and bring them to market. However, relying on the quality of the SCEs and the skills of the developers to create value-added services is not sufficient to provide the services in best quality. Therefore, a novel test framework is required to enable consequent testing of value-added services before the deployment and provisioning. Then, service providers 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.

In literature, most testing approaches and frameworks follow a model-based strategy [1]. Here, a formal model of the implementation to be tested is created from which subsequently test cases are derived. Most popular models for the purpose of model-based testing are based on Finite State Machines (FSM), Extended Finite State Machines (EFSM) and sometimes Statecharts. Unfortunately, the existing approaches such as [2] and [3] often lack an efficient automated test case derivation method. In fact, the approaches either lead to an enormous amount of generated test cases because of the increasing complexity of the underlying formal model, or the test case derivation leads to infeasible path results. Here, the generated test paths can never be evaluated because the described behaviour never takes place. Finally, the approaches often lack flexibility regarding the test case generation method. Although different coverage criteria are offered, the structure of the formal models usually contains static.

In this paper, we propose a new method for test case derivation and generation based on a Statechart-based formal model notation. The method is embedded within the architecture of a novel framework for testing value-added telecommunication services, the Test Creation Framework (TCF). The Statecharts-based formal model in the approach is automatically generated based on a semi-formal requirements specification and is comprised of a number of reusable test modules. The approach supports different coverage criteria and considers the infeasible path issue.

The remainder of this paper is structured as follows: the following section 2 introduces the architecture of the mentioned TCF. Section 3 introduces the selected modelling notation of the formal model and illustrates how the reusable test modules are described. Afterwards, section 4 discusses the method to derive abstract test cases from the Statecharts-based models.

II. TEST CREATION FRAMEWORK ARCHITECTURE

This section provides an overview of the architecture of the novel Test Creation Framework (TCF) for the testing of value-added telecommunication services.

Fig. 1 illustrates the architecture components as well as the workflow of the methodology starting with the test developer who can access the Test Modules Environment (TME) and the Test Framework User Terminal (TFUT).

The TME enables the test developer to create, modify or erase so-called reusable test modules which are based on a modelling notation (Statecharts) and describe typical service characteristics such as sequences of multimedia protocols like SIP (Session Initiation Protocol) or 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 SIP, and cover both standard behavior as well as possible alternative behavior. The reusable test modules are stored persistently in a specific database, the Test Modules Repository (TMR). Furthermore, a reusable test module generally contains test data templates which are stored separately within the Test Data Pool (TDP). The databases are

separated because the existing test data templates can be used for diverse reusable test modules.

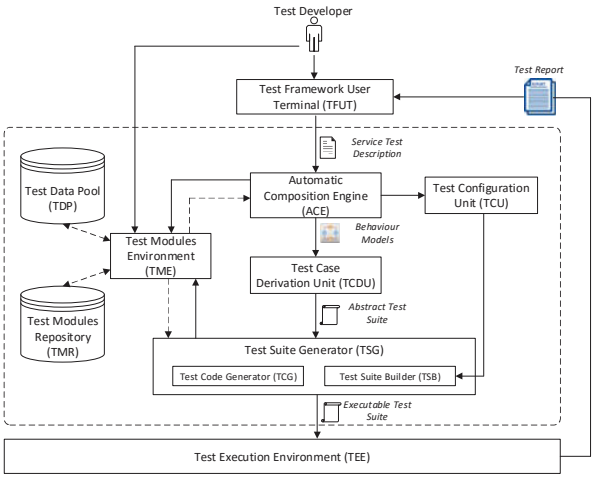

Fig. 1. Test Creation Framework architecture

Via the TFUT, the test developer can specify instances of a novel sort of specification or rather service description language which is named Service Test Description (STD). Once defined by a test developer, the STD triggers a fully automated process which results in the execution of generated test cases against the considered value-added service. In principle, the STD 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 service and their relationships as well as dynamic behavioural definitions specifying use-case related requirements. The specification of the behavior is performed by means of an applied pi-calculus approach [4]. The overall structure of the STD has already been published in [5].

On the basis of an STD instance, the Automatic Composition Engine generates Behaviour Models, complete formal models based on Statecharts notation which describe the desired possible behaviour of the specified value-added service. The process includes the automated selection of identified reusable test modules and their composition to more complex models as well as the automated instantiation of test data templates. As a result, the ACE generates one Behaviour Model for each specified requirement within the STD instance. This enables a direct mapping from specified requirements to the formal model and automatically, a mapping from requirements to test cases.

The Test Case Derivation Unit (TCDU) includes a test case finder which uses an algorithm and follows selected coverage criteria to enable the derivation of abstract test cases from the Behaviour Models. Depending on the selected coverage criteria and the selected reusable test modules, the amount of test cases differs significantly. The output of the TCDU is an abstract test suite which includes abstract test cases for each Behaviour Model.

The Test Suite Generator (TSG) creates a valid and executable test suite that can be imported into a TTCN-3 (Testing and Test Control Notation version 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 compilations as well as the Java compilation in order to generate an executable test suite.

The final step of the framework's underlying methodology takes place within the Test Execution Environment (TEE). Here, the generated executable test suite will be loaded and subsequently executed against the System Under Test (SUT), the value-added service. After all test cases have been executed, a test report is generated which includes information about successful and failed test cases.

This paper is mainly concerned with the TMR (see section 3) and the TCDU (see section 4).

III. Modelling Service Behaviour for the Purpose of Testing

To generate appropriate functional test cases, a formal modelling notation needs to be selected that enables a behavioural description of a value-added telecommunication service.

A. Requirements on a modelling notation

The ETSI standard [1] lists the following general requirements that have to be fulfilled by potential modelling notations for model-based testing approaches:

- 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, assignments and conditional statements.
- The notation shall support advanced control constructs such as loops.

Considering these general requirements, the ETSI standard [1] discusses that modelling notations for the specification of behaviour are limited to rule-based notations (such as EFSMs and abstract state machines), process-oriented notations (such as the Business Process Execution Language (BPEL)) and Statecharts [6]. If the properties of value-added services are taken into consideration, further specific requirements have to be met by the modelling notation:

- 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.

Based on these stated requirements, Statecharts were selected as modelling notation. First, Statecharts explicitly support modularity through the defined concept of hierarchical states. Within such a hierarchical state, the behaviour of reusable test modules can be specified. Second, the syntax of Statecharts is very similar to EFSM-based approaches. This aspect allows to include new transitions between reusable test modules and therefore enables compositions. Third, Statecharts support concurrency through so-called concurrent hierarchical states (so-called AND-states). Within such a concurrent hierarchical state, it can be more than one state executing simultaneously. This is a very important aspect for value-added services as they are running within concurrent environments. Fourth, the support for timers is provided as soon as a state is reached within a Statecharts model. Finally, the fifth requirement is met by Statecharts because of the existence of the State Chart extensible Markup Language (SCXML) [7], a formal language which has been defined as World Wide Web Consortium (W3C) recommendation.

To sum up, the Statecharts modelling notation meets all the stated specific requirements. Rule-based notations have not been taken into consideration because they do not support concurrency and they generally do not provide an existing standardised formal description. Process-oriented notations lack a concept for the composition of reusable test modules.

B. Principles of modelling potential service behaviour

As Statecharts have been selected as foundation for the description, the principles of modelling potential service behaviour have to be determined in the approach. The authors suggest a novel concept of modelling behaviour with a formal model which includes both system-specific and test-specific artefacts. The concept has been derived from the transaction user (TU) which is the fourth and topmost layer of the Session Initiation Protocol (SIP) [8] structure. In the context of the SIP protocol, the TU contains both User Agent Client (UAC) and User Agent Server (UAS) core. According to [8], a "core designates the functions specific to a particular type of SIP entity". Therefore, 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 test modelling approach, 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 SIP standard, also cores of other protocols that are dedicated to the Open Systems Interconnection Model (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 and is therefore able to control the service logic without having any information about the internal information of a value-added service. 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 Fig. 2.

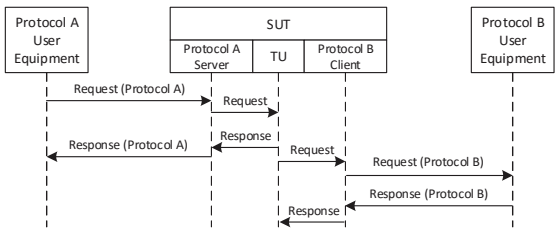

Fig. 2. Transaction user as mediator between client and server cores

The scenario shows that the TU as part of the SUT is informed about any incoming protocol message by the specified cores. It is also able to initiate messages through the cores. Based on this generic example, a Statecharts-based model can be defined to describe the behaviour. Such a model consists of states and transitions. Each transition can contain events and actions. In fact, the events as well as actions in the Statecharts notation are represented by protocol messages (both requests and responses). Fig. 3 shows the Statecharts model describing the behaviour illustrated in Fig. 2.

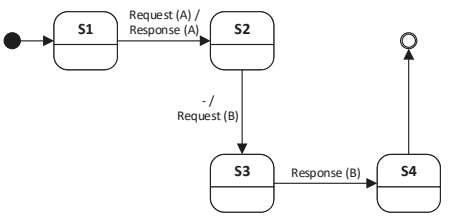

Fig. 3. Example simple Statecharts model

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 is always a request type (see Fig. 3, message "Request (A)"). Otherwise, if it is a client-based core, the received message is always a response type (see Fig. 3, message "Response (B)"). So, an event in the Statecharts notation always refers to an input the SUT has to process. In contrast, the actions refer to the reactions of the SUT through the corresponding cores (see Fig. 3, messages "Response (A)" and "Request (B)").

C. Reusable test modules

The recent example illustrated behaviour of generic protocols. For the concrete OSI application layer protocol SIP, two so-called reusable test modules have been derived for the server core, SIP UAS non-INVITE and SIP UAS INVITE. For the client core, there are also two, SIP UAC non-INVITE and SIP UAC INVITE. The "INVITE"-specific reusable test modules refer to transactions that include the initiation of a call using the INVITE method of the SIP protocol. All other SIP

methods (such as BYE, CANCEL and MESSAGE) are handled by the "non-INVITE" reusable test modules. In the following Fig. 4, the SIP UAS non-INVITE reusable test module is displayed as an example.

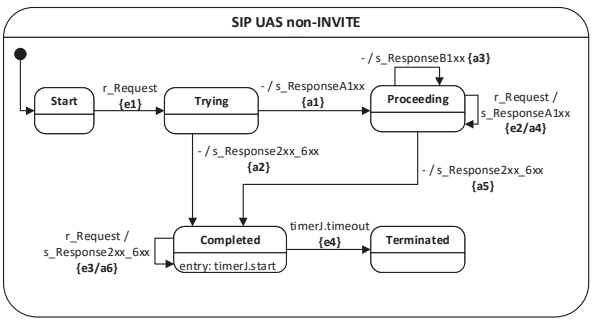

Fig. 4. Statecharts model of SIP UAS non-INVITE reusable test module

The entry point into the reusable test module is the "Start" state which contains a transition to the state "Trying" which holds the event "r_Request". Here, the "r" prefix is an abbreviation for "received" and refers to the SUT that actually receives a message by this statement. Once in the "Trying" 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" action. Both actions contain the prefix "s" for "send", which states that the SUT actually sends the message back to the initiator of the "r_Request" message. The alternative paths describe the potential behaviour of the SUT (the value-added service). It could happen that based on the "r_Request", the SUT directly acknowledged 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 by means of the Service Test Description [5]. 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. 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 the specified behaviour in the SIP UAS non-INVITE reusable test module, test cases can be later on derived by a specific test case derivation algorithm. In general, this algorithm will be performed on the resulting behaviour models, which are compositions of several reusable test modules.

IV. Test Case Derivation

The test case derivation from formal models is widely discussed in literature (such as in [9], [10], and [11]). Generally, so-called structural coverage criteria are applied for transition-based models such as Statecharts.

A. Selection of a proper structural coverage criteria

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 has been specified in [12] and is illustrated in the following Fig. 5.

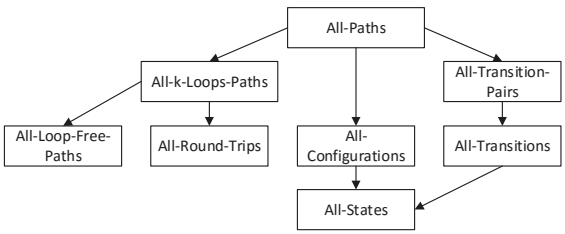

Fig. 5. Hierarchy of structural coverage criteria

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 criterion c_1 (arrow source) subsumes another criterion c_2 (arrow destination). The meaning of the diverse structural coverage criteria is as described in [10] [12] [13]:

- *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.

The selection of a proper structural coverage criteria depends on the underlying formal model. If the model contains many alternative branches and also loops, the All-Paths-based criteria (such as All-Paths, All-k-Loops-Paths and All-Loop-Free-Paths) lead to an infinite number of test paths. In fact, the underlying models applied in this approach can contain quite a lot of branches and self-transitions (see Fig. 4). The All-Paths-based criteria cannot be applied here. Alternatively, the structural coverage criterion All-Round-Trips can be satisfied

with a linear number of test cases. It has been selected as coverage criterion because in comparison to standard structural coverage criteria such as All-Transitions and All-States, it is able to detect faults more thoroughly. Furthermore, it is recommended in literature by [9], [13] and [14] for model-based testing approaches.

B. Representation of test cases in proposed approach

Although a proper structural coverage criterion has been selected, the properties of value-added telecommunication services have to be taken into consideration. Of course, it would be possible to apply the All-Round-Trip structural coverage criterion on the Statechart-based notation, but most of the derived abstract test cases will run results in an inconclusive verdict as soon as they have been made executable. This has to do with the fact that resulting from the coverage criterion, 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 the SIP protocol is considered [8]. The test execution environment sends an INVITE request in order to establish a session to a 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, 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, 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 Statecharts-based 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 $v_i \varepsilon V$ can theoretically have an infinite number of outgoing edges. However, according to the test case representation, there is a restriction defined. A vertex $v_i \varepsilon V$ can only have more than one outgoing edge if it specifies an action and not an event. This has to do with the fact that within the proposed approach, events can be definitely predicted whereas actions cannot.

C. Exemplified test case derivation from Statecharts models

The principle of test case derivation will be exemplified by means of the SIP UAC non-INVITE and the SIP UAS non-INVITE reusable test modules. In the following Fig. 6, the SIP UAC non-INVITE behavioural description is illustrated with a special identification of the transitions (e.g. "{a1}").

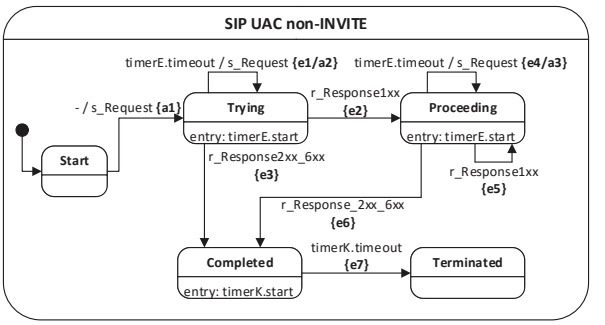

Fig. 6. Statecharts model of SIP UAC non-INVITE reusable test module

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, the following five test cases can be derived (see Fig. 7).

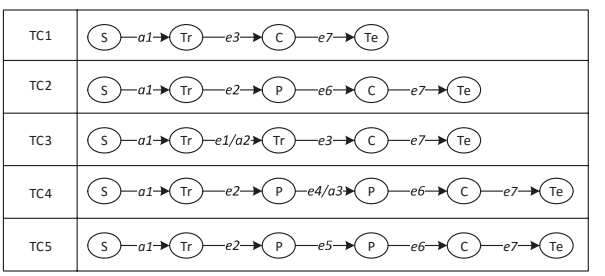

Fig. 7. Test case derivation from SIP UAC non-INVITE

The state names within Fig. 6 have been abbreviated in Fig. 7 ("Start" to "S", "Trying" to Tr", "Proceeding" to "P", "Completed" to "C" and "Terminated" to "Te). "TC1" and "TC2" in Fig. 7 are based on the All-Transitions criteria without loops. Both test cases describe a standard behaviour of a SIP request being sent from the SUT to the participating entities. 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. 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 will 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 following Fig. 8, the test case derivation algorithm is applied to the SIP UAS non-INVITE reusable test module (see Fig. 4) which leads to three test cases.

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 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. A further action "a3" describes a provisional response which can be retransmitted a not specified number of times. Therefore, a self-loop is included in all three test cases.

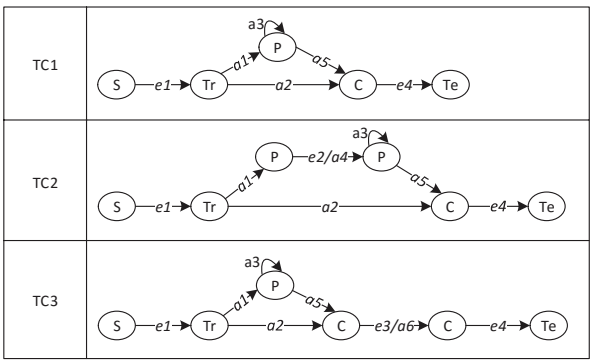

Fig. 8. Test case derivation from SIP UAS non-INVITE

To sum up, the proposed approach enables an efficient test derivation method by applying the All-Round-Trip structural coverage criteria. It offers significantly better coverage than the weaker standard coverage criteria All-Transitions and All-States without the disadvantage of generating an infinite number of test cases. Furthermore, because of the introduced graph-based test case structure, there are no possible infeasible path results. Even if some paths within a test case are not reached, the execution can still be evaluated as "pass". However, this is only valid if optional paths are included. Finally, the proposed approach also supports further flexibility regarding the structure of the reusable test modules. Theoretically, it is possible to manipulate the Statecharts-based models describing recurring behaviour. An example would be to erase the "Proceeding" state of the SIP UAC non-INVITE and the SIP UAS non-INVITE reusable test modules. Then, the provisional messages are not considered anymore. For certain behaviour this can be a useful feature (such as in instant messaging, where provisional messages are not used).

V. Conclusion

In this paper, we have introduced parts of a novel framework for the automated functional testing of value-added telecommunication services. As the approach includes a model-based testing process, the requirements on an appropriate modelling notation have been stated and the Statecharts notation has been selected after evaluation. Furthermore, we developed a new principle of modelling behaviour with Statecharts. Through their server and client cores, recurring behaviour of application layer protocols can be specified for further usage. Finally, an efficient way to derive abstract test cases from the models has been demonstrated.

Further papers to be published will focus on the missing aspects of the proposed TCF, such as the relevance of the Service Test Description within the process as well as the transformation of the derived abstract test cases into executable TTCN-3 test cases.

References

[1] ES 202 951: "Methods for Testing and Specification (MTS); Model-Based Testing (MBT); Requirements for Modelling Notations", ETSI Standard, 2011.

[2] P. Wacht, T. Eichelmann, A. Lehmann, and U. Trick, "A new Approach to design graphically functional Tests for Communication Services", Proc. 4th IEEE International Conference on New Technologies, Mobility and Security (NTMS 2011), IEEE press, Feb. 2011, pp. 1-5, doi:10.1109/NTMS.2011.5721068.

[3] J. Ernits, A. Kull, K. Raiend, and J. Vain, "Generating TTCN-3 Test Cases from EFSM Models of Reactive Software using Model Checking", Proc. 36th Jahrestagung der Gesellschaft für Informatik e.V. (INFORMATIK 2006), GI, Oct. 2006, Vol. 94, pp. 241-248.

[4] R. Milner, J. Parrow, and D. Walker, "A calculus for mobile processes", Information and Computation, Elsevier, 1992, Vol. 100, Issue 1, pp.1-40.

[5] P. Wacht, U. Trick, W. Fuhrmann, and B. Ghita, "A new Service Description for Communication Services as Basis for automated functional Testing", Proc. 2nd IEEE International Conference on Future Generation Communication Technology (FGCT 2013), IEEE press, Dec. 2013, pp. 59-64, doi:10.1109/FGCT.2013.6767211.

[6] D. Harel and M. Politi, "Modeling Reactive Systems with Statecharts: The Statemate Approach (Software Development)", McGraw-Hill Inc., 1998, New York, USA, ISBN: 978-0-070-26205-8.

[7] W3C, "State Chart XML (SCXML): State Machine Notation for Control Abstraction", W3C Recommendation, 2015.

[8] IETF RFC 3261: "SIP: Session Initiation Protocol", Request For Comments, IETF, 2002.

[9] M. Utting and B. Legeard, "Practical Model-Based Testing: A Tools Approach", Morgan Kaufmann Publishers Inc., 2007, San Francisco, USA, ISBN: 978-0-1237-2501-1.

[10] P. Ammann and J. Offutt, "Introduction to Software Testing", Cambridge University Press, Cambridge, 2008, UK, ISBN: 978-0.521-88038-1.

[11] L.H. Tahat, B. Vaysburg, B. Korel, and A.J. Bader, "Requirement-based automated black-box test generation", Proc. 25th Annual International Computer Software and Applications Conference (COMPSAC 2001), IEEE press, Oct. 2001, pp. 489-495, doi:10.1109/CMPSAC.2001.960658.

[12] S. Haschemi, "Model transformations to satisfy all-configurations-transitions on statecharts", Proc. 6th International Workshop on Model-Driven Engineering (MoDeVVa 2009), ACM press, Oct. 2009, doi:10.1145/1656485.1656490.

[13] R. Binder, "Testing Object-Oriented Systems: Models, Patterns, and Tools", Addision-Wesley, 1999, Boston, USA, ISBN: 0-201-80938-9.

[14] G. Antoniol, L.C. Briand, M. Di Penta, and Y. Labiche, "A case study using the round-trip strategy for state-based class testing", Proc. 13th International Symposium on Software Reliability Engineering (ISSRE 2002), IEEE press, Nov. 2002, pp. 269-279, doi:10.1109/ISSRE.2002.1173268