A Novel Test Creation Framework for Value-Added Services

Patrick Wacht and Ulrich Trick
Research Group for Telekommunication Networks
Frankfurt University of Applied Sciences
Frankfurt, Germany
Email: {wacht, trick}@e-technik.org

***Abstract*—Recent years have witnessed that standard telecommunication services evolved more and more to complex 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 focusing on technologies and protocols. Besides, service providers have to consider a fast transition from concept to market product as well as low prices for new services due to the increasing competition in the telecommunication industry. So, there is an urgent need for effective test solutions that can be integrated into current value-added service development life-cycles. This paper proposes a novel framework for functional testing which is based on a new sort of specification for value-added services, the Service Test Description (STD). Combined with the properties of predefined reusable test modules based on Statecharts notation, an algorithm parses STD instances and automatically generates so-called behaviour models from which functional test cases can be derived, generated and subsequently executed against the value-added service. Prototypical implementations of the concept have confirmed its usefulness and effectiveness.**

***Index Terms*—automated functional testing, value-added services, test automation framework**

I. Introduction

The demand for advanced value-added services in the telecommunication domain has increased over the last years. Major challenges arise especially for service providers who have to provide a fast transition from concept to market product and who have to offer low prices for their product to satisfy their customers. In order to face the situation, service providers have integrated so-called Service Creation Environments in their Service Delivery Platforms (SDP) to enable service developers to rapidly create new and individual value-added services and bring them to market. Current SCEs [1] have proven to work properly but the functional quality of the services relies only on the skills of the service developer. This lack of support for functional testing requires a novel framework to enable a consequent testing of value-added services before their deployment and provision. Service providers can then assure their customers of a proper execution of the delivered services and that they perform according to the specified requirements.

The related testing approaches in literature are not sufficient to fulfil the requirements on a novel test framework. In fact, most of them follow a model-based strategy [2] based on finite state machines or extended finite state machines. Test developers need to design complex formal models, an error-prone and inefficient procedure. Oftentimes, the integrated methods to derive test cases from the models are rather inefficient such as in [3] and [4]. This leads to an enormous amount of test cases which cannot be executed against a System Under Test (SUT) within a reasonable amount of time. Finally, a test framework should include the test execution of the generated test cases and should provide a test report to document the verdicts of the test case execution.

This paper proposes a novel so-called Test Creation Framework (TCF) which can be included in given SDPs and which allows test developers to automatically create and execute test cases. The creation of tests is done by defining a novel sort of description language specifically for value-added services, the Service Test Description (STD). The structure of the STD hides the complexity of underlying existing Statecharts-based formal models and includes information to realise the mapping of requirements and generated test cases.

The remainder of this paper is structured as follows: section II introduces the architecture of the proposed TCF. Section III deals with the structure of the STD and shows a simplified example STD instance. Afterwards, section IV discusses the aspects of reusability within the TCF and the underlying composition algorithm. Section V briefly describes the test case generation and, finally, section VI concludes with an evaluation.

II. Test Creation Framework architecture

This section provides an overview of the proposed TCF. Fig. 1 illustrates its 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 and erase so-called reusable test modules which are based on 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 behaviour as well as possible alternative behaviour. The test modules themselves are stored within a database, the Test Modules Repository

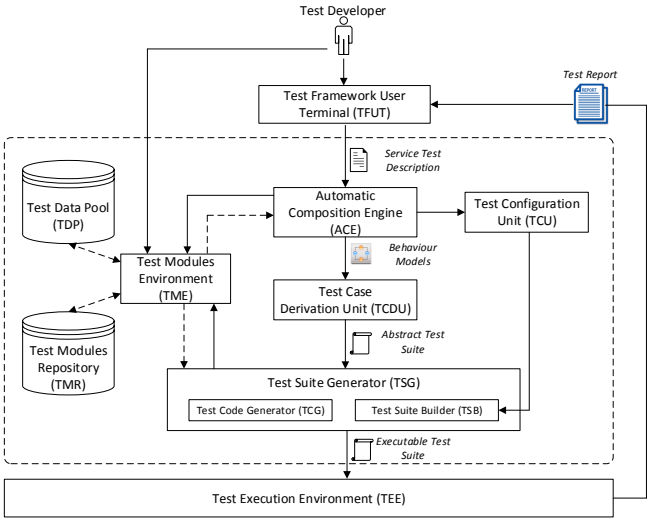

Fig. 1. Test Creation Framework architecture

(TMR) whereas the related test data is separately stored within the Test Data Pool. This has been established intentionally because the existing test data templates can then be used for diverse reusable test modules.

The TFUT (see Fig. 1) provides a graphical user interface for the test developer to specify instances of the STD. It also enables him to trigger a fully automated process which results in the execution of generated test cases against the relevant value-added service. In principle, the STD comprises elements of test specifications and service specifications and contains both architectural as well as behavioural content. The overall structure of the STD has been published in [5] and will be introduced in section III.

On the basis of an STD instance, the Automatic Composition Engine (ACE) (see Fig. 1) generates so-called 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. For each specified requirement of a value-added service, one behaviour model will be generated. 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) (see Fig. 1) 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 criterion 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) (see Fig. 1) 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 (TCG). The Test Suite Builder (TSB) will enhance the TTCN-3 code with specific test modules and includes also the configuration of TTCN-3 codecs and adapters. Furthermore, the TSB includes the TTCN-3 compilations as well as the Java compilation in order to generate an executable test suite.

The execution of all the test cases within the executable test suite takes place within the Test Execution Environment (TEE). Afterwards, a test report is generated which includes information about successful and failed test cases.

The architecture of the TCF contains several important components which will be investigated in the following. The structure of the STD is part of the upcoming section.

III. Service Test Description Language

The Service Test Description (STD) is the most relevant component of the proposed TCF as it specifies the basis on which test cases are later on generated. As mentioned before, the STD contains both service-specific and test-specific properties and can be seen as a combination of service specification defining service-related information and behaviour, and a test specification including the determination of test components and test data.

The structure of the STD is displayed in Fig. 2. It is subdivided into the architectural and the behavioural perspective. This principle has been derived from the concepts of Unified Test Modeling Language [6] and UML 2.0 Testing Profile [7].

The architectural perspective includes general information about the value-added service such as its “Service ID”, a unique identifier. The “Service ID” is important in order to differentiate between parallel projects the test developer is working on. The “Prose Description” documents the value-added services. It should be brief and concise but it does not play a role in the formal processing. The “Roles” field lists the

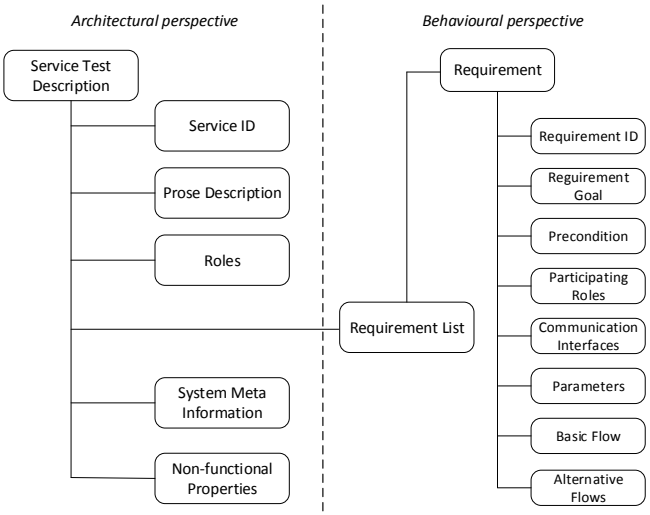

Fig. 2. Structure of Service Test Description

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 hand. The "Roles" usually represent specific external hardware or software that can interact with the service via communication protocols such as SIP and HTTP. Regarding the TCF, the "Roles" play a very relevant role as, based on their determination, sets of predefined reusable test modules for the test execution environment can be automatically derived and afterwards instantiated. An example for a "Role" can be a SIP phone (or rather VoIP phone) or a web browser. The "System Meta Information" field in the architectural perspective contains important parameters of the SUT such as its addressability (service URI, IP addresses and transport protocols). The final field "Non-functional Properties" allows the test developer to capture information that is important for the customer such as performance and usability. It does not describe specific functions.

The behavioural perspective of the STD comprises a list of requirements to actually 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. Each "Requirement" specified in the behavioural perspective contains a "Requirement ID", a unique identifier. It allows to address the specific "Requirement" within the specification, e.g. as dependency within other "Requirements". The "Requirement Goal" contains in a very short natural language-based prose text the main objective of the corresponding "Requirement". Within the "Precondition" field, the statements indicate what has to have happened before the function of the current "Requirement" is activated. Through this field, the dependencies between "Requirements" can be determined. The architectural perspective already included the "Roles" field which holds the relevant participating entities to consume the service. In contrast, the "Participating Roles" field only contains selected "Roles" from the architectural perspective that are specifically playing a role in the current "Requirement". The "Communication Interfaces" (CI) field contains the most relevant information regarding the aspect of reusability in the proposed TCF. In principle, the CIs represent the points of interaction between the value-added service and the participating entities which are specified in the "Participating Roles" field. A "Role" provides a potential functionality 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". Considering a SIP phone as an example "Role", six different CIs have been identified on the side of the SUT. Four of the CIs are derived from the transaction state machines specified in the SIP standard RFC 3261 [8]. 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 apart 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 apart from INVITE requests). Besides these CIs for the SIP phone, there are also two RTP CIs, "RTP Source" and "RTP Sink".

Regarding the concept of reusability, each CI specified in the STD can be assigned to a predefined reusable test module which is stored in the TMR (see Fig. 1). The next field within a "Requirement", the "Parameters", are closely linked to the specified CIs and accordingly to the reusable test modules. Each test module includes variables that are instantiated from abstract data types which represent communication protocol messages (e.g. SIP requests and SIP responses). Through the "Parameters" field, each data field within a variable based on an abstract data type can be parameterised. The principle of the parameterisation will be demonstrated by means of an example. The "Basic Flow" and the "Alternative Flows" are the final fields within a "Requirement". In principle, the "Basic Flow" contains the description of steps that have to be taken to achieve the main target 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". As appropriate foundation of a language being able to specify the steps within the flows, the pi-calculus has been chosen. In principle, the pi-calculus can be seen as a model of communication systems which enables to express processes with changing structure. 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. 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 the channels. To apply the pi-calculus for the flow descriptions, the concept had to be reconciled by means of minor enhancements. Firstly, the names being sent and received over channels are substituted by terms. Such terms are placeholders for variables or even functions that expect input parameters and of course return a value to be either sent or received.

In order to illustrate the concept, a sample STD specification is discussed in the following. Therefore, a simple chat service is consulted, especially the exchanging of instant messages between participants. An architectural perspective of such a service is illustrated in Table I. Here, the "Service ID" is set and the "Prose Description" describes the main functionality the sample service has to deliver. Two different "Roles" have been identified for the service, the "[sender]" and the "[recipient]". Both "Roles" are acting as SIP phones. As the names of the "Roles" states, the "[sender]" is the initiator of the instant message whereas the "[recipient]" actually receives the message. Further information regarding the service addressability ("ServiceURI") and the used transport protocol

TABLE I
ARCHITECTURAL PERSPECTIVE OF CHAT SERVICE

Service ID	Chat Service
Prose Description	A chat communication should be provided. The service users are able to log into the system and log out again. While being logged in, the service user can enter chat rooms and leave chat rooms again. The service user can also send textual chat messages.
Roles	SIP phone: [sender] SIP phone: [recipient]
System Meta Information	ServiceURI: sip:chatservice@vas.de Protocol: UDP
Non-functional Properties	None

("UDP") are specified in the "System Meta Information" field. "Non-functional Properties" are not specified for the sample chat service.

The behavioural perspective in Table II includes the specification of one "Requirement" with the unique identifier "Req03". Furthermore, the "Requirement Goal" is described. The "Precondition" field contains the value "Req02". Although this "Requirement" is not determined in the example, the specified behaviour within its respective "Basic Flow" has to happen before the "Basic Flow" of "Req03" begins.

TABLE II
BEHAVIOURAL PERSPECTIVE OF 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	SIP phone: [sender] SIP phone: [recipient]
Communication Interfaces	SIP UAS non-INVITE: [sender1] ⇒ channel a SIP UAC non-INVITE: [sender2] ⇒ channel b SIP UAC non-INVITE: [recipient1] ⇒ channel c
Parameters	var initMsg = [sender1] ⇒ r_Request; var forwMsg = [recipient1] ⇒ s_Request; var okMsg = [sender2] ⇒ s_Request; var errorMsg = [sender2] ⇒ s_Request; timer t1 = [recipient1] ⇒ timerF; initMsg = {(Method, "MESSAGE"), (Text, "Hello Bob!")} forwMsg = {(Method, "MESSAGE"), (Text, initMsg.Text)} okMsg = {(Method, "MESSAGE"), (Text, "Ok!")} errorMsg = {(Method, "MESSAGE"), (Text, "Message not received!")}
Basic Flow	P ::= a(initMsg). c<forwMsg>. if(t1.timeout) then Q else. b<okMsg>. 0
Alternative Flow (AF1)	Q ::= b<errorMsg>. 0

In this example (see Table II), "Req02" could indicate the entering of both "Participating Roles" in the same chat room to be able to exchange messages. The "Requirement" identifies three different CIs. Every CI definition includes the type of CI (e.g. "SIP UAS non-INVITE"), the "Participating Role" it belongs to as well as a channel identifier. The channel stands for the actual communication channel between the SUT and the "Roles" as external entities. In the example, 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 "Parameters" field includes the definition of several variables all representing SIP MESSAGEs, either being sent from the sender to the SUT ("initMsg"), from the SUT to the sender ("okMsg", "errorMsg") or from the SUT to the recipient ("forwMsg"). Additionally, the "timerF" of the "SIP UAC non-INVITE" CI is defined. Subsequently, the "Basic Flow" is determined. Each process in a pi-calculus description has to have a unique identifier (here: "P"). The steps within a process are separated through the "." operator. Each process step includes one of the specified channels as well as a parameter. The sort of brackets determines whether the SUT receives a message over a specific channel ("( )") or whether it send a message ("< >"). In the "Basic Flow", it initially denotes the SUT to receive the SIP MESSAGE "initMsg" over channel "a" and then consequently sends the SIP MESSAGE "forwMsg" over channel "c". In the next step, the state of the timer "t1" is checked. If it has not timed out (if-else-then condition), the SUT sends out the SIP MESSAGE "okMsg" over channel "b" and the "Basic Flow" terminates afterwards with the "0" step. Otherwise, if the timer has timed out, the "Alternative Flow" is activated. Here, a different SIP MESSAGE "errorMsg" is sent by the SUT over channel "b". Then, also the "Alternative Flow" terminates.

IV. REUSABLE TEST MODULES AND BEHAVIOUR MODEL GENERATION

Each reusable test module is defined within a hierarchical OR-state in Statecharts notation. For each CI specified in the STD, there is an existing reusable test module which is either classified as client core (UAC) or server core (UAS). This classification determines in which role the SUT is acting towards the participating entities. Client core reusable test modules always specify outgoing request types and incoming response types whereas server core reusable test modules specify incoming request types and outgoing response types. For the SIP protocol, two sever core reusable test modules have been identified ("SIP UAS non-INVITE" and "SIP UAS INVITE") as well as two client core reusable test modules ("SIP UAC non-INVITE" and "SIP UAC INVITE").

Fig. 3 illustrates the "SIP UAC non-INVITE" test module as example. It consists of states and transitions which again can contain events and actions. In fact, the events as well

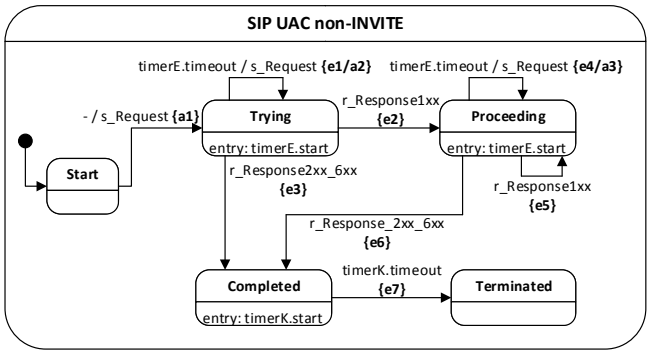

Fig. 3. SIP UAC non-INVITE reusable test module

as the actions in the Statecharts notation are represented by protocol messages (both requests and responses). The entry point into the reusable test module is the "Start" state which contains a transition to the state "Trying" which holds the action "s_Request". The prefix "s" is an abbreviation for "send" and refers to the SUT that actually sends a message by this statement. Once in the "Trying" state, there are three valid optional paths that can be taken, either to the "Proceeding" state with the "r_Response1xx" event, to the "Completed" state with the "r_Response2xx_6xx" event, or to the "Trying" state again with the timeout event of "timerE". The "r" prefix here represents messages that are actually received by the SUT. In principle, the alternative paths describe potential behaviour of the SUT. It could happen that based on the "s_Request", the SUT directly has to receive either a provisional SIP message (e.g. "100 Trying") or a successful one (e.g. "200 OK"). All these cases are considered within the reusable test module and might occur. Therefore, the different optional paths are the foundation of the test cases to be generated.

Before focusing on the test case derivation, the composition algorithm performed by the ACE (see Fig. 1) to generate behaviour models is introduced. The following steps are included in the algorithm:

1) Reading STD instance.
2) Selection of relevant reusable test modules and their instantiation based on the behavioural perspective.
3) Reading of relevant variables and parameterisation of test module instances.
4) Composition of test module instances.

The first three steps are rather static and just refer to the identification of the test modules based on the CI definition in the STD as well as their parameterisation through the "Parameter" definitions. The final step 4 considers the pi-calculus-based flow descriptions in the STD instance. For each flow, the algorithm categorises the steps within it and performs accordingly. A flow step can describe the sending (see Table II, "c<forwMsg>") or the receiving (see Table II, "a(initMsg)") of messages. It can also contain conditions or it can be the so-called "null step". There is also a fifth category, the definition of concurrent behaviour. Such a pi-calculus definition uses the statement "—" to specify concurrency. After categorising the steps, the test module instances are composed or connected according to the sequence of steps within the flows. As illustrated in Fig. 3, each test module contains a "Start" state and a "Terminated" state. These states are the direct connector states between test modules. If there is, for instance, a sequence of a sending and a receiving step, the algorithm generates a new transition from "Terminated" state of the test module instance related to the sending step until the "Start" state of the test module instance related to the receiving step. In this simple case, the behaviour model instance would include two test module instances. If the step includes a condition, guards are included within the transitions that lead to the upcoming alternative steps. If a concurrent step is parsed, the algorithm creates a new instance of a Statecharts hierarchical AND-state. Within the AND-state, the two corresponding test module instances are included. If a null step is detected by the algorithm, an end state is included in the corresponding behaviour model instance.

V. Test Case Generation

After the behaviour model generation is completed, the TCDU (see Fig. 1) derives test cases from the models. For transition-based models such as Statecharts, generally structural coverage criteria are applied. Based on the selected criterion, a test case generator automatically generates a set of paths within the model from an initial state to an end state. Well-known structural criteria are for instance "All-States" (every defined state within a given model is visited at least once) and "All-Transitions" (every transition of the model must be traversed at least once), but there are many others, too. The selection of a proper structural coverage criterion depends a lot on the underlying formal model. If the model contains many alternative branches and also loops, typical All-Paths-based criteria (such as "All-Paths" and "All-k-Loops-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. 3). Therefore, the structural coverage criterion "All-Round-Trips" was selected. Its application generates a test case for each loop in the model and that it only has to iterate once around the loop. In comparison to the All-Paths-based criteria, it can be satisfied with a linear number of test cases and it is able to detect faults more thoroughly than criteria such as "All-States" and "All-Transitions". Furthermore, it is recommended in literature, e.g. by [9].

The principles of test case derivation following the "All-Round-Trips" structural coverage criterion will be exemplified by means of the "SIP UAC non-INVITE" reusable test module (see Fig. 3). Based on its behavioural description, the derived test cases are illustrated in Fig. 4.

The state names within Fig. 3 have been abbreviated in Fig. 4 ("Start" to "S", "Trying" to "Tr", "Proceeding" to "P", "Completed" to "C" and "Terminated" to "Te"). "TC1" and "TC2" in Fig. 4 are based on the "All-Transitions" criterion without loops. Both test cases describe a standard behaviour of a SIP request being sent from the SUT to the participating

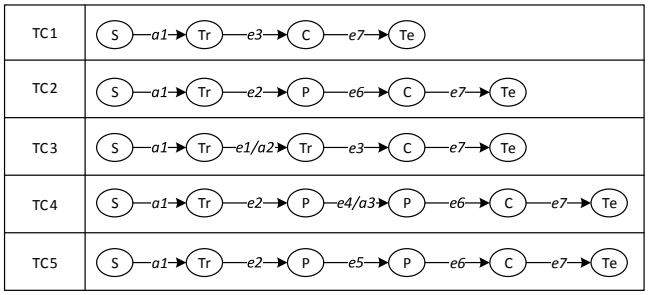

Fig. 4. Derived test cases from SIP UAC non-INVITE

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 example test case derivation illustrates how the test case derivation for a reusable test module would look like. Considering a value-added service, the output of the test derivation phase is an abstract test suite which includes sets of abstract test cases (see Fig. 4) for each generated behaviour model. In the following step, the TCG reads the abstract test suite and generates a test configuration based on the architectural information defined in the STD (e.g. identification of test components). The TCG then continues with the generation of the test data which requires a connection to the TDP. As a result, for each variable instance, a so-called TTCN-3 template will be generated. Afterwards, the abstract test cases belonging to specific behaviour models will be transformed to TTCN-3 modules which include TTCN-3 test cases. The test behaviour creation process includes all particularities that are integrated in the abstract test cases 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 which will subsequently create an executable test suite (ETS). This ETS is executed within a TTCN-3-based test execution environment.

VI. Evaluation of the Concept

A prototypical implementation has been developed based on the TCF architecture. The prototype includes a web-based front end which enables a test developer to define STD instances and initialise the testing process. Furthermore, he can maintain already existing STDs and retrieve information about the test monitoring. In the back end, each of the TCF architecture components have been implemented in so-called OSGi bundles within a Java-based OSGi

Several example value-added services have been specified using the prototype such as a more extensive example of the chat service. A further example service is the so-called "pizza service", a service which automatically established an audio call between the service consumer and a nearby pizzeria.

The result of the validation of the concept has shown that our proposed approach has a ROI of 128% compared to traditional manual testing approaches and a ROI of 32% compared to model-based testing approaches such as described in [8].

VII. Conclusion

This publication presents a novel framework for the automated functional testing of value-added services. For the TCF methodology, a new service description language has been developed, the STD. It comprises both service-specific and test-specific properties and is the only manual task in the process which has to be done by the test developer. The structure of the STD considers aspects of reusability through so-called reusable test modules and is based on a simple pi-calculus description. The derivation of test cases in the TCF is based on formal Statecharts-based models which are automatically generated by an efficient composition algorithm. A well-known problem from other test frameworks, the generation of an infinite number of test cases, has been solved by applying the efficient "All-Round-Trip" structural coverage criterion. The concept has been evaluated by means of a prototype implementation.

References

[1] A. Lehmann, T. Eichelmann, U. Trick, R. Lasch, B. Ricks, and R. Toenjes, "Teamcom: a service creation platform for next generation networks," in *Proc. IEEE International Conference on Internet and Web Applications and Services (ICIW09)*, Venice, Italy, May 2009, pp. 12–17.

[2] *Methods for Testing and Specification (MTS); Model-Based Testing (MBT); Requirements for Modelling Notations*, European Telecommunications Standard Institute Std., Rev. ETSI ES 202 951 V1.1.1, 2011. [Online]. Available: http://www.etsi.org

[3] P. Wacht, T. Eichelmann, A. Lehmann, and U. Trick, "A new approach to design graphically functional tests for communication services," in *Proc. IEEE International Conference on New Technologies, Mobility and Security (NTMS11)*, Paris, France, Feb. 2011, pp. 1–5.

[4] J. Ernits, A. Kull, K. Raiend, and J. Vain, "Generating ttcn-3 test cases from efsm models of reactive software using model checking," in *Proc. Jahrestagung der Gesellschaft fuer Informatik e.V.*, vol. 94, Oct. 2006, pp. 241–248.

[5] P. Wacht, U. Trick, W. Fuhrmann, and B. Ghita, "A new service description for communication services as basis for automated functional testing," in *Proc. IEEE International Conference on Future Generation Communication Technology (FGCT13)*, London, United Kingdom, Dec. 2013, pp. 59–64.

[6] A.-G. Feudjio, "Model-driven functional test engineering for service centric systems," in *Proc. IEEE International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities and Workshops (TridentCom09)*, Washington D.C., USA, Apr. 2009, pp. 1–7.

[7] *UML Testing Profile (UTP)*, Object Management Group Std., Rev. OMG UTP Version 1.2, 2013. [Online]. Available: http://www.omg.org/spec/UTP/1.2

[8] *SIP: Session Initiation Protocol*, Internet Engineering Task Force Std., Rev. IETF RFC 3261, 2002. [Online]. Available: https://www.ietf.org/rfc/rfc3261.txt

[9] G. Antoniol, L. Briand, M. D. Penta, and Y. Labiche, "A case study using the round-trip strategy for state-based class testing," in *Proc. IEEE International Symposium on Software Reliability Engineering (ISSRE02)*, Annapolis, USA, Nov. 2002, pp. 269–279.