Availability testing for Web services

A.-G. Vouffo Feudjio, I. Schieferdecker

October 28, 2008

Abstract

New challenges have emerged regarding the robustness of convergence platforms against potential security threats and other types of failures which could affect service availability. Current approaches in testing such service centric systems target stateless unit-level testing and tend to focus on syntactical data analysis, while neglecting the semantical aspects. We propose a new pattern-driven test modeling approach to allow stateful test design on service level and to reduce the test development cycle through automated model transformation and code generation.

1 Introduction

Model-driven test development is seen as one of the most promising approaches for regaining control over the test process in an environment characterized by increasingly complex systems and ever shorter time to market of services. In fact, it is widely agreed, that a minimal amount of quality is expected from each deployed service for it to be successful. Therefore, to ensure that the testing activities would not become a bottleneck on the way to product delivery, it is essential for the test development process to provide at least the same level of agility as the product development process. While model driven software engineering has gained more popularity and helped significantly in speeding up system development through customization, reuse and automated code generation, a similar evolution is yet to take place for test development. Pattern-driven test design aims at applying the model-driven architecture (MDA) paradigm to the test development process, combining it with principles of pattern-driven design. We argue that, by allowing tests to be designed from higher abstraction level, it allows to address each of the aforementioned issues.

The Service Availability Forum (SAF) defines two main aspects of service availability: high availability and service continuity. With availability being defined as the probability for a service being up and running at any instant without a breakdown and the corresponding downtime, high availability refers to an availability of at least 99.999%. According to [18], service availability encompasses both exclusivity,i.e. the property of being able to ensure access to authorised users only, and accessibility, i.e. the property of being at hand and useable when needed. Service continuity is the ability of a service to maintain

customer sessions for uninterrupted services, i.e. even in case of failure of a given component in the service infrastructure and subsequent take-over by a redundant component.

Testing services and service-centric systems poses new challenges to traditional testing approaches [3][17]. This is more the case for integration testing and system-level testing of such services. As a requirement that is located at the system or the service level and which is subject to more than syntactical correctness, similar challenges are faced with, when testing service availability. Those difficulties can be classified in two categories. The first category stem from the specificities of distributed services and the development process they imply. Service-centric systems require fast development within increasingly complex and heterogeneous environments. Therefore test suites must be developed faster and at the same time fulfill their purpose, i.e. uncovering potential failures of the systems before they are deployed. The second category of difficulties come from the fact that existing testing approaches for service centric systems are mostly immature or inappropriate to address those new requirements. Most existing test approaches tend to focus on syntactical correctness, while neglecting semantical aspects as well as context of use, which are highly critical for service availability testing. Pattern-driven test design effectively allow tests targetting those semantical aspects to be design at a early stage to drive the product development process and to help uncovering failures prior to deployment of services.

This paper is organized as follows: The next session discusses some related works in the area of service testing. Then, Section2 provides a more detailed description of our pattern-driven test modeling approach, illustrating it with examples from our Parlay-X case study. Section 4 evaluates our approach in the light of the case study, before Section 5 concludes the paper and provides some outlook on future work.

2 Related Works

Existing approaches in testing service-centric systems are both white-box and black-box oriented. While white-box techniques based on automatic test case generation from FSMs models of the services would be quite efficient in testing in-house components as they are being developed, they can hardly be applied for Customer-Off-The-Shell (COTS) components widely used in this context. In fact, in most cases vendors would provide just the minimal amount of implementation details for their components, sufficient to allow interoperability with the rest of the platform. Furthermore, white-box techniques would not allow to address the distributed nature of such platforms.

On the other hand, existing black-box testing approaches such as TTCN-3 (Testing and Test Control Notation [7]) are specified typically at a lower level of abstraction, which makes their usage for testing services both less efficient and costly, although the benefits of using such test-specific notations for test devel-

opment have already been demonstrated in many instances [19]. In previous work [21], we proposed to generate reusable code snippets and libraries of the target test notation based on test patterns to address those concerns. However, as we evolved in our work, it became more and more obvious that, what was needed was a model-driven test development process allowing such tests to be specified at a high level of abstraction.

The need for the type of high-level test modeling presented here has already been acknowledged in the testing domain and was one of the driving forces for the UML Test Profile (UTP[22]) and other earlier efforts around other modeling notations, such as the Unified Modeling Language (UML), the System Design Language (SDL), Message Sequence Charts (MSCs) [11] etc. The UML Testing Profile defines a language for designing, visualizing, specifying, analyzing, constructing and documenting the artifacts of test systems [16]. As a UML profile, UTP inherits all existing UML concepts without defining restrictions for their usage. While this makes UTP a very powerful notation for the purpose of test modeling, it does not facilitate its usage, because UML does not impose a specific process nor does UTP. However, test development (and even more so pattern-driven test development) implies a specific process. Therefore, we developed a dedicated language for pattern-driven test modeling based on a MOF meta-model enriching concepts proposed by UTP with test patterns identified in designing test systems for various domains, while restricting its scope to the sole purpose of test modeling.

3 Pattern-Driven Test Modeling

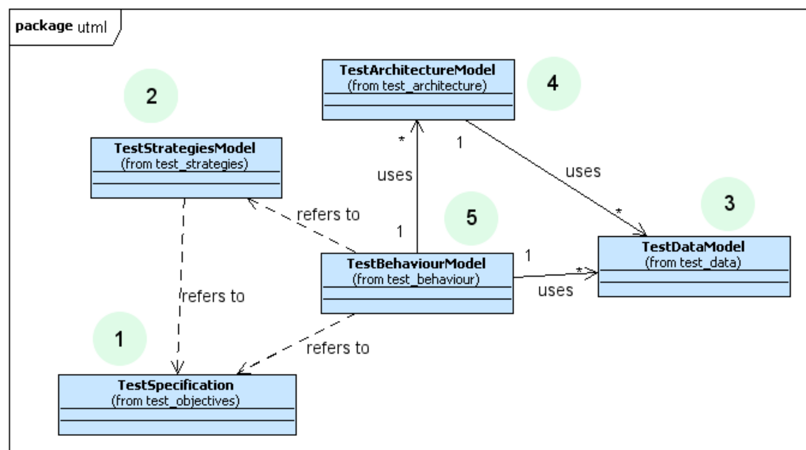

Figure 1: Overview of UTML Test Models

Test modeling is the application of the OMG's MDA (Model Driven Architecture) approach to the test development/specification process. It consists of using

modeling techniques to describe elements of a test specification. The resulting abstract platform-independent test models (PITs) are subsequently transformed into more concrete test models (called also platform-specific test models (PST)) until executable test scripts for specific test environments (the test code) are obtained [5]. We introduced the concept of test patterns as an attempt to apply the design pattern approach [1] broadly applied in object-oriented software development to model-driven test development. Patterns represent a form of reuse in development in which the essences of solutions and experiences gathered in designing and developing systems are extracted and documented to enable their application in similar contexts that might arise in the future.

To facilitate their reuse, the test patterns we have identified have been "implemented" into a UML meta-model representing a test-specific notation called UTML (Unified Test Modeling Language [20]). Based on that meta-model, a tool set is generated to guide test modelers in defining test specifications along the defined patterns. UTML comes together with a test modeling methodology embodying test patterns. Figure 1 gives an overview of the different types of test models UTML allows to define and the relationships between them. As

Figure 2: Overview of UTML Test Modeling Process

depicted on Figure 2, the goal of UTML test modeling is to design a test behaviour model, i.e. a complete description of the actions to be performed and observed on entities involved in each test case to verify that a system's behavior matches its requirements. Figure 1 also illustrates the UTML test modeling methodology through the numbers assigned to each type of model. Those numbers reflect the sequence of modeling phases towards a test model, out of which executable test scripts can then be generated.

3.1 Test Specification Modeling

A test specification is a document describing each of the test objectives that the system under test. The first step in UTML test modeling consists of identifying what the test objectives are going to be. A combination of both automatic and manual generation of test objectives is the most realistic approach. If the requirements on the system are expressed in a machine-processable notation, then automatic generation of test objectives could be achieved. Those would be completed by manually derived test objectives based e.g. on an analysis of potential failures of the system by test design experts. In the specific case of

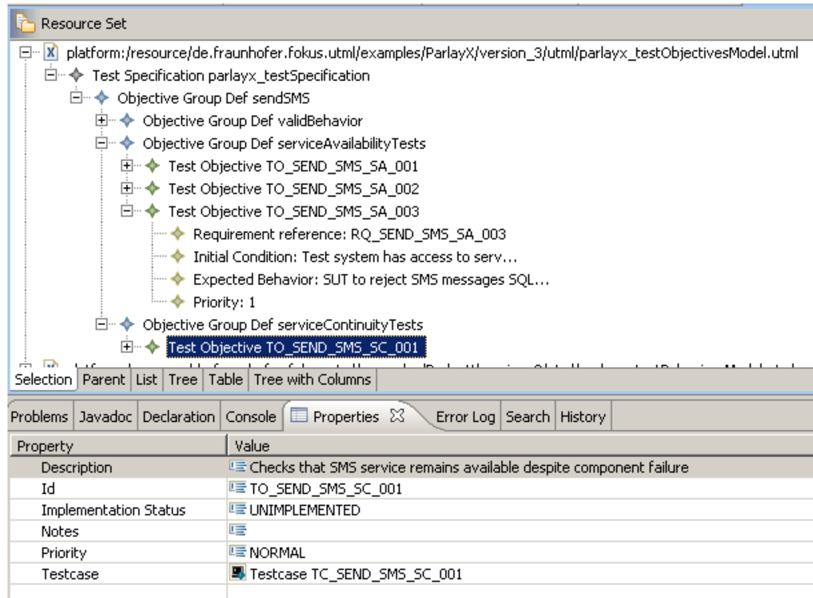

Figure 3: Example UTML Test Objectives Model for Parlay-X sendSMS Web Service

availability testing of services, test objectives are best derived from SLAs (service level agreements) and QoS (quality of service) constraints between service provider and consumer.

Figure 3 depicts the test objectives model we have designed for a Parlay-X sendSMS Web service for the sending of short text messages (SMSs). As mentioned earlier, we have organized the test objectives in two groups, targeting service availability and service continuity respectively. The test objective model displayed in Figure 3 also illustrates how test objectives can be linked with system requirements or elements of SLAs between service providers and service consumers.

3.2 Test Strategies Modeling

For each of the test objectives agreed upon between the parties, a test strategy must be designed, describing how that test objective will be assessed. Test strategies can be modeled as sequence of test steps, with each test step describing an action or an observation to be performed on one or more elements in the test setup. Each test step can be described by natural language. However, UTML provides the concepts to ensure that all test strategies follow the same pattern through model validation.

Figure 4 depicts the test strategies model for our Parlay-X sendSMS Web service case study. The link between test strategies and test objectives models

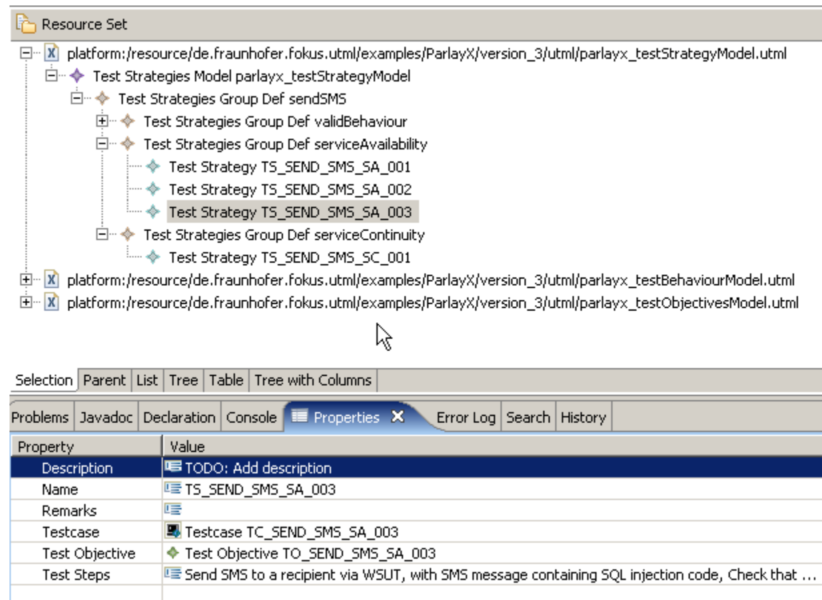

Figure 4: Example UTML Test Strategy Model for Parlay-X sendSMS Service

is also illustrated in that figure, with each test strategy being associated to a previously defined test objective from the test objectives model.

3.3 Test Data Modeling

```
   <wsdl:operation name="sendSms">
       <wsdl:input
 message="parlayx_sms_send:SendSms_sendSmsRequest"/>
       <wsdl:output
 message="parlayx_sms_send:SendSms_sendSmsResponse"/>
       <wsdl:fault name="ServiceException"
 message="parlayx_common_faults:ServiceException"/>
       <wsdl:fault name="PolicyException"
 message="parlayx_common_faults:PolicyException"/>
    </wsdl:operation>
```

Code snippet 1: Extract from Parlay-X WSDL File for sendSMS Web Service

The purpose of test data modeling is to precisely describe data that will be exchanged among service entities to implement the designed test strategies. Those data include stimuli, i.e. messages that will be sent to service providers, as well as potential responses. Responses will be modeled based on constraints dictated by semantic descriptions of the services. Generally, a static description of the service is available as a WSDL (Web service definition language) or similar document, based on which the type system for the test data can be extracted automatically. This procedure is rather straightforward and automating it saves a lot of efforts in test data modeling.

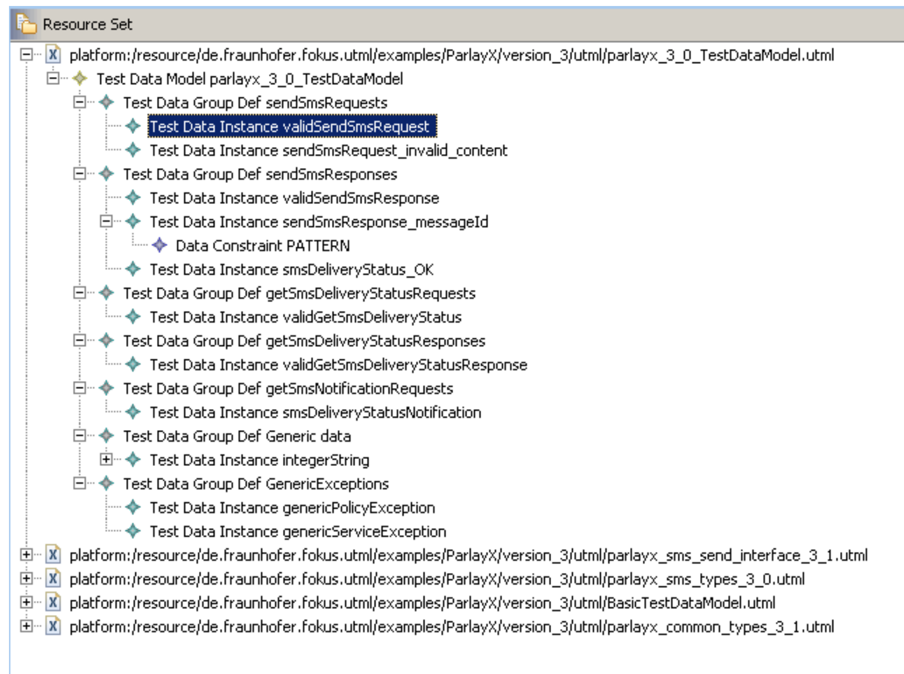

Figure 5: Test Data Model for Parlay-X sendSMS Web Service

Listing 1 displays an extract from the Parlay-X WSDL file defining the sendSMS Web service. More precisely, it shows the details of the sendSMS request message including parameters and potential faults it might generate. The sendSMS request message can be sent by an application to send a short text message to a recipient via the Web service. Figure 5 displays the test data model we defined based on the Parlay-X WSDL file. All WSDL files imported by the sendSMS' main WSDL file were transformed automatically into UTML test data models. Those transformed WSDL models provided the base for defining test data in the test model. Therefore the whole (System/Service Under Test)SUT's type system became accessible and could be used to model requests to be sent to the Web service under test, as well as to define syntactic constraints on responses we expect back from the Web service. Also visible in that figure is the integration of the automatically transformed test data models into the resulting main test data model. Furthermore, using domain partition methods such as the classification tree method (CTM) combined with heuristic to control the number of generated data, test data instances representing impulses and responses can be generated automatically as well.

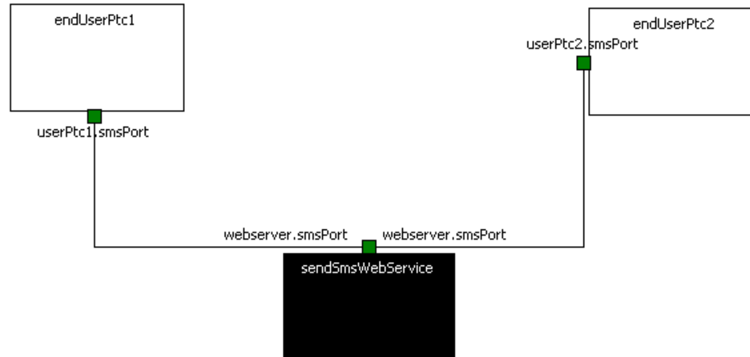

Figure 6: Test Architecture Diagram for sendSMS Service

3.4 Test Architecture Modeling

The test architecture describes the topology of the test system, i.e. its composition in terms of parallel test components and the points of communication between those and elements of the SUT. UTML concepts of test architecture modeling are inherited from TTCN-3 and UTP. A test architecture model consists of a collection of test configurations, i.e. predefined setups of a test system as a composition of parallel test and system components interconnected with each other via ports over which they exchange messages. The communication mode between such test components can be synchronous (request/reply) or asynchronous (message based). Figure 6 displays a test architecture diagram for the sendSMS Web service to test service availability. As depicted in that figure, the sendSMS Web service is modeled as a SUT component in the test configuration. Depending on the objective of the test, any of the components defined in a test configuration can be labelled as part of the SUT and will be displayed with a black color accordingly to underline the fact that we follow a black-box testing approach. This facilitates the creation of new test configurations as variants of existing ones, since the same base configuration can be used or adapted for additional test objectives.

3.5 Test Behaviour Modeling

Based on previously defined test data and test architecture models, semantic requirements on the service can be expressed as UTML test behaviour models using their graphical representation called UTML test behaviour diagrams. Each UTML test behaviour diagram is built upon a test configuration previously defined in the associated test architecture model. Figure 7 displays an example test behaviour diagram for testing service continuity of a Parlay-X sendSMS Web service. As depicted in that figure, a UTML test behaviour diagram shows many similarities with a classical UML sequence diagram. The main differences

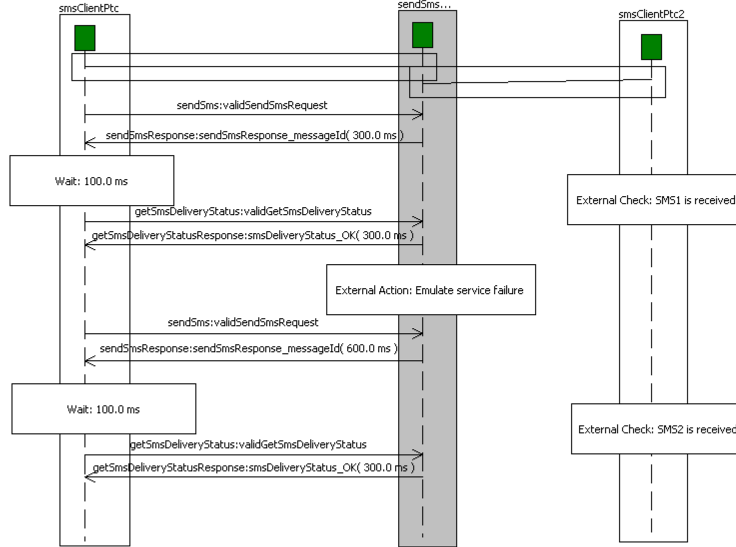

Figure 7: Test Behaviour Diagram for sendSMS Service Continuity

lie in the fact that UTML behaviour diagrams provide a concept of test components, with a test component containing one or many lifelines representing its communication ports. Also, UTML behaviour diagrams take black-box testing concerns into account. As shown in Figure 7, selected test components can be labelled as being part of the SUT or being parallel test components, which is the default assumption. This has implications on the operations allowed on a component and its owned ports in term of semantics. For instance a send action, i.e. an action modeling the sending of a request or the invocation of a method on an interface, will not be allowed from an SUT component, because that would violate the black-box testing paradigm, according to which we do not have such access from the test system. Additionally, the test behaviour model provides the means for modeling constraints on actions performed by service providers or users. For example, for an action describing that the test system expects a response from a service provider, the test behaviour can set a timing constraint for that response along with other constraints on the data contained in the response itself.

Furthermore, if available, system behaviour models expressed using UML sequence diagrams or state machines can be transformed into test behaviour models, either manually or automatically using the aforementioned automatic test generation techniques based on EFSMs. However, it should be taken into consideration that this might lead to a large number of test cases being generated, to the extend that the trade-out would outweigh the expected benefits.

Another benefit of modeling test behaviour at such a high level of abstraction is that it allows tests for complex combinations of distributed services to be modeled in the same way as tests for services taken individually.

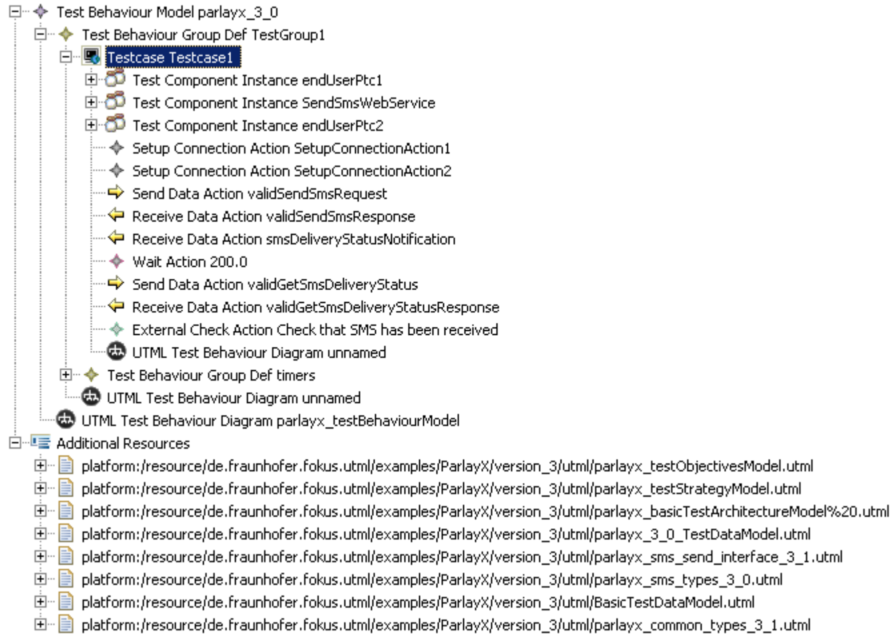

Figure 8: Test Behaviour Model for sendSMS Service Continuity

Figure 7 depicts the test behaviour diagram for a test case targeting service continuity of the Parlay-X sendSMS Web service. Figure 8 depicts the same model in its tabular form, including the other test models for test architecture and test data models partly derived from the SUT's WSDL specification. As depicted on both pictures, the test behaviour involves two parallel test components and the sendSMS Web service as SUT component. The represented test semantics is as follows:

- In a first phase, the smsClientPtc component sends the Web service a valid sendSMSRequest containing an SMS message for smsClientPtc2.
- The smsClientPtc component expects an SMSResponse from the Web service containing the message ID within a delay of 300.0 ms.
- The smsClientPtc component pauses for 100.0 ms
- smsClientPtc component sends a GetSmsDeliveryStatus request to check wether the first SMS was delivered successfully.
- Again the response for that request is expected within 300.0 ms

- Test operator is requested to trigger the serving component on the Web service to emulate a failure.
- smsClientPtc sends another SMS and checks that it was forwarded successfully to smsClientPtc2, meaning that service continuity has been ensured by the Web service, e.g. with a redundant component taking over after failure of the first one

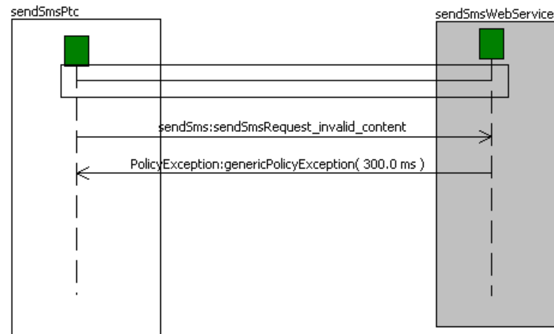

Figure 9: Test Behaviour Diagram for sendSMS Robustness

Figure 9 features another example of test case, this time targeting service robustness to ensure availability. The test case emulates the introduction of disallowed characters in one of the parameters for the sendSMS request, e.g. an SQL-injection attack on the Web service. It then checks that the service responds with a PolicyException as requested and does not crash. Which would indicate vulnerability to such attacks.

```
1   . . .
2   /**
3     * Functions for test component behaviours
4     */
5   group TC_Service_Continuity_functions {
6
7    function f_TC_Service_Continuity_smsClientPtc_behaviour()
8    runs on SendSmsEndUserType {
9     map (smsClientPtc: sendSMSPort, sendSmsWebService: sendSMSServicePort);
10    sendSMSPort.send (validSendSmsRequest);
11    T_Guard.start;
12    alt {
13     [] sendSMSPort.receive (sendSmsResponse_messageId) {
14      T_Guard.stop;
15      log ("***  F_TC_SERVICE_CONTINUITY_SMSCLIENTPTC_BEHAVIOUR():
16      sendSmsResponse message received as expected ***");
17      setverdict (pass);
18     }
19     [] T_Guard.timeout {
20      log ("***  F_TC_SERVICE_CONTINUITY_SMSCLIENTPTC_BEHAVIOUR():
21       Time out while expecting sendSmsResponse message ***");
22      setverdict (fail);
23     }
24    }
25    log ("***  F_TC_SERVICE_CONTINUITY_SMSCLIENTPTC_BEHAVIOUR():" &
```

```
26         "start waiting for 100 ms. ***");
27     v_timer.start (100);
28     alt {
29      [] v_timer.timeout {
30       log ("***  F_TC_SERVICE_CONTINUITY_SMSCLIENTPTC_BEHAVIOUR():
31       finished waiting for 100 ms. ***");
32      }
33     }
34     sendSMSPort.send (validGetSmsDeliveryStatus);
35     T_Guard.start;
36     alt {
37      [] sendSMSPort.receive (smsDeliveryStatus_OK) {
38       T_Guard.stop;
39       log ("***  F_TC_SERVICE_CONTINUITY_SMSCLIENTPTC_BEHAVIOUR():
40        getSmsDeliveryStatusResponse message received as expected ***");
41       setverdict (pass);
42      }
43      [] T_Guard.timeout {
44       log ("***  F_TC_SERVICE_CONTINUITY_SMSCLIENTPTC_BEHAVIOUR():
45        Time out while expecting getSmsDeliveryStatusResponse message ***");
46       setverdict (fail);
47      }
48     }
49     action ("SMS1 is received");
50     . . .
51   } //end f_TC_Service_Continuity_smsClientPtc_behaviour
52
53   . . .
54
55 } //end TC_Service_Continuity_functions
56 /**
57   *
58   * @purpose
59   *     TP version:
60   * @desc:
61   *     TODO: Add description Test strategy:
62   */
63  testcase TC_Service_Continuity() runs on SendSmsWebServiceType
64   system ParlayX_MainTestComponentType {
65   //Instanciate test components
66   var SendSmsEndUserType smsClientPtc := SendSmsEndUserType.create;
67   var SendSmsEndUserType smsClientPtc2 := SendSmsEndUserType.create;
68
69   //Preamble
70   //Test body
71   smsClientPtc.start (f_TC_Service_Continuity_smsClientPtc_behaviour());
72   smsClientPtc2.start (f_TC_Service_Continuity_smsClientPtc2_behaviour());
73  } //end TC_Service_Continuity
```

Listing 1: Automatically Generated TTCN-3 Code for Service Continuity Testing

Using model transformation techniques we successfully export the UTML test behaviour model into TTCN-3, thus making it executable upon compilation using a TTCN-3 compiler. Our approach for model transformation is a template-based model-to-code transformation [4], whereby we use the Freemarker Template Language (FTL)[8] to define the transformation rules. Alternatively, we could have used another template-ba sed model transformation such as Java Emitter Templates (JET)[14][15], which is part of the Eclipse Modeling Framework (EMF). The only reasons for not doing so was our lack of practical experience in using JET and the existence of an integration framework for FTL that could be reused for rapid prototyping. Code snippet 1 displays a sample from

the TTCN-3 source code automatically generated through model transformation from the UTML test behaviour model depicted on Figure 7. Alternatively, we could have generated test scripts in any other general purpose programming language or test-specific notation, fulfilling the same purpose. This alternative is particularly interesting for real-time and embedded systems whereby the test system must meet specific constraints due to their execution environment.

4 Evaluation of the Approach

Using a prototype implementation of UTML based on the Eclipse TopCased modeling framework, we have modeled a set of availability tests featuring the OSA Parlay-X version 3.1 sendSMS service. Designing a test data model and an architecture model was straightforward, once the service's WSDL was transformed to UTML automatically using model transformation techniques. Using the test behaviour diagram editor we could design the complete test cases' behaviour graphically. Although a full quantitative evaluation of the approach's benefits is still ongoing, first results clearly indicate that the test development cycle is shortened significantly. Given that the approach implies a specific process and thanks to the model validation facilities used to check the test model's consistency, we could identify flaws in test design at early stage. Also, by ensuring that the designed test model is complete and consistent, the approach enables the automatic generation of fully executable test scripts from the test model. The TTCN-3 code generated automatically from the UTML test model for the OSA Parlay-X is valid and ready for compilation in terms of its behaviour. The additional effort required consisted in completing the definition of TTCN-3 templates generated from the UTML data instances defined in the test data model.

5 Conclusion and Outlooks

We have presented a new approach for test modeling based on test patterns. Our case study involving a Parlay-X Web Service showed that our approach can considerably fasten the test development process for service availability. The fact that the approach could be demonstrated on a service described with WSDL indicates that it could be applied also to other family of services, independent of the notation being used. One major challenge that is inherent to model-driven development and that also needs to be addressed in this context is that of model consistency. While model validation and a well-defined process help avoiding errors in test modeling, mechanisms for ensuring model consistency have not yet reached the same level of maturity. Further work will aim at improving that aspect to avoid problems of unresolved references between inter-dependent test models which would be a major hampering factor for the adoption of test modeling.

References

[1] Ch. Alexander. *The Timeless Way of Building*. Oxford University Press, 1979.

[2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture, A System Of Patterns, Volume 1*. Wiley Series in Software Design Patterns, 2001.

[3] G. Canfora and M. Di Penta. Testing services and service-centric systems: Challenges and opportunities. *IEEE Journal*, pages 10–17, March-April 2006.

[4] K. Czarnecki and S. Helsen. Classification of model transformation approaches. In *OOPSLA03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

[5] M. Busch et al. Model transformers for test generation from system models. *Proceedings of Conquest 2006, 10th International Conference on Quality Engineering in Software Technology, September 2006, Berlin, Germany, Hanser Verlag*, 2006.

[6] M. Frey et al. Draft report: Methods for testing and specification (MTS); patterns for test development (PTD). Technical report, European Telecommunications Standards Institute (ETSI), 2004.

[7] European Telecommunications Standards Institute (ETSI). Multipart Standard 201 873: Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; - Part 1 (ES 201 873-1): TTCN-3 Core Language. European Telecommunications Standards Institute (ETSI), Sophia-Antipolis (France), August 2008.

[8] Ftl - freemarker template language. http://freemarker.org, 2008.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1994.

[10] D. Geneiatakis, T. Dagiuklas, G. Kambourakis, C. Lambrinoudakis, S. Gritzalis, Karlovassi, S. Ehlert, and D. Sisalem. Survey of security vulnerabilities in session initiation protocol. *IEEE Communications Surveys*, 8(3), 2006.

[11] J. Grabowski, D. Hogrefe, and R. Nahm. Test case generation with test purpose specification by MSCs, 1993.

[12] Hao He. What is service-oriented architecture? http://webservices.xml.com/pub/a/ws/2003/09/30/soa.html, 2003.

[13] Z. Micskei, I. Majzik, and F. Tam. Robustness testing techniques for high availability middleware solutions. In *Proc. of International Workshop on Engineering of Fault Tolerant Systems (EFTS2006)*, Luxembourg, Luxembourg, June December–JanuaryMarch 2006. University of Luxembourg.

[14] R. Popma. Jet - java emitter templates tutorial part 1: (introduction to jet). http://www.eclipse.org/articles/Article-JET/jet_tutorial1.html, 2008.

[15] R. Popma. Jet - java emitter templates tutorial part 2: (write code that writes code). http://www.eclipse.org/articles/Article-JET2/jet_tutorial2.html, 2008.

[16] OMG ptc. Unified Modeling Language: Testing profile, finalized specification. Technical report, Object Management Group, April 2004.

[17] L. Ribarov, I. Manova, and S. Ilieva Ph.D. Testing in a service-oriented world. *Proceedings of the International Conference on Information Technologies (InfoTech-2007)*, 2007.

[18] J. Rossebe, M. Lund, K.E. Husa, and A. Refsdal. A conceptual model for service availability. *Quality of Protection, Security Measurements and Metrics, Advances in Information Security, Vol 23*, 2006.

[19] I. Schieferdecker, D. Vega, and C. Rentea. Import of WSDL definitions in TTCN-3 targeting testing of Web services. *Proceedings of IDPT 2006, 9th World Conference on Integrated Design and Process Technology, June 2006, San Diego, California, USA*, 2006.

[20] A. Vouffo-Feudjio. A unified approach to test modelling. *Proceedings of MoTIP 2008, 1st Workshop on Model-based Testing in Practice, Berlin, Germany, June 2008, IRB Verlag*, 2008.

[21] A. Vouffo-Feudjio and I. Schieferdecker. Test pattern with TTCN-3. *Proceedings of FATES 2004, 4th International Workshop on Formal Approaches to Testing of Software, Linz, Austria, Sept. 2004, Springer*, 2004.

[22] J. Zander, Z.R. Dai, I. Schieferdecker, and G. Din. From U2TP models to executable tests with TTCN-3 - an approach to model driven testing. *Proceedings of the IFIP 17th Intern. Conf. on Testing Communicating Systems (TestCom 2005)*, 2005.