Model-Driven Functional Test Engineering for Service Centric Systems

A.-G. Vouffo Feudjio
Fraunhofer Institute for Open Communication Systems (FOKUS)

Abstract

Functional Testing of service centric systems poses new challenges to traditional test design approaches. As a mean for achieving network and service convergence, such systems integrate various different notations and technologies into a common framework. Assuming that test artifacts targeting each of the technologies involved exist, designing functional tests at the service level, by integrating existing test artifacts is still a very complex engineering task. We propose to address those challenges with a model-driven test engineering approach based on test patterns. The approach allows tests to be designed at a higher level of abstraction as abstract test models, using a test specific, but platform independent modeling notation called UTML(Unified Test Modeling Language). We argue that our approach can be helpful in speeding up the production of new test cases through reuse of existing test artifacts and test infrastructure, as well as automated generation of executable test scripts from the abstract test models.

1 Introduction

With subscriber numbers quickly reaching saturation, the ability to create and deliver new services rapidly and flexibly has now become the centre piece of enterprise success in the telecommunication industry. The introduction of standardized and distributed Service Delivery Platforms (SDPs), also referred to as Service Mediation Platforms (SMP), is a key element of that new strategy. SDPs are built upon existing Business or Operation Support Systems (BSS/OSS) to provide horizontal integration between future, legacy and existing systems. This has lead to a trend towards Service Oriented Architectures (SOA), i.e. systems based on interacting coarse grained autonomous components called services [14].

Testing such services and service-centric systems poses new challenges to traditional testing approaches [4][13]. This is more the case for functional and integration testing.

SDPs are designed to facilitate access to services across different kinds of network infrastructures. Therefore they naturally involve several different protocols, trying as far as possible to use existing ones. The IP multimedia subsystem IMS is a good example of such an SDP. It makes extensive use of the Session Initiation Protocol (SIP) for signaling, the DIAMETER protocol for billing and accounting etc. Functional and conformance testing of SDPs can be a complex engineering task, because of that heterogeneity.

In most cases, although legacy test artifacts developed for individual protocols exist, reusing them for designing functional tests, that would cover the large variety of different components involved can still be either too costly or technically difficult to achieve. One reason is that different notations might have been used to design the test suites, thus making test system integration a very difficult task. Moreover, even in cases whereby the same notation may have been used, test system integration may still be difficult to achieve, because the mechanisms for reuse (e.g. by defining reusable libraries and export mechanisms), might not be present in the notation used for the individual test suites or in the test suites themselves. As a consequence, new test suites would have to be developed from scratch, thus causing extra avoidable costs. Which given the potential for reuse of existing artifacts would be quite a waste.

To address that issue, we propose to model tests at a higher level of abstraction, independently of any specific lower level test notation or scripting language. Our approach integrates concepts of Action- or Keyword-Based Testing (ABT) [2] with the concept of test design patterns we introduced in previous works [23][22]. This can ease the integration of legacy test artifacts into new test designs, out of which new executable test scripts can be generated automatically. This paper is organized as follows: In the next section, we will discuss the challenges in testing SDPs and present related work in that area. Section 3 provides a more detailed description of our Model-Driven Test Engineering approach, before 4 describes how it can be used to achieve test reuse and test system integration. Section 5 then describes a tool architecture for the approach, before Section 6 then concludes the paper and provides some outlook on future work.

2 Related Works: Challenges in Testing SDPs

The difficulties in testing SDPs on functionality can be classified in two categories. The first category of difficulties are those inherent to 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 stem from the fact that existing testing approaches for service centric systems are mostly immature or inappropriate to address those new requirements.

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 cannot be applied for 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 [6]) or xUnit (code-driven test frameworks) 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 development have already been demonstrated in many instances [15]. In previous work [23], we proposed to generate reusable code snippets and libraries of the target test notation based on test patterns to address those concerns, but 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, so that the conceptual gap between test design and product system design would be reduced.

Functional testing of SDPs is still a new field of research. While the challenges have already been identified, solutions for addressing them are yet to be proposed. The need for the type of high-level test modeling we are arguing for has already been acknowledged in the testing domain and was one of the drivers for the UML Test Profile (UTP[25]) and other earlier efforts around UML, SDL, MSCs [8] etc. The UML Testing Profile defines a language for designing, visualizing, specifying, analyzing, constructing and documenting the artifacts of test systems [12]. 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 dedicated language for Model-Driven Test Engineering based on a MOF meta-model enriching concepts proposed by UTP with test patterns identified in designing test systems for various domains and restricting its scope to the sole purpose of test modeling.

3 Model-Driven Test Engineering

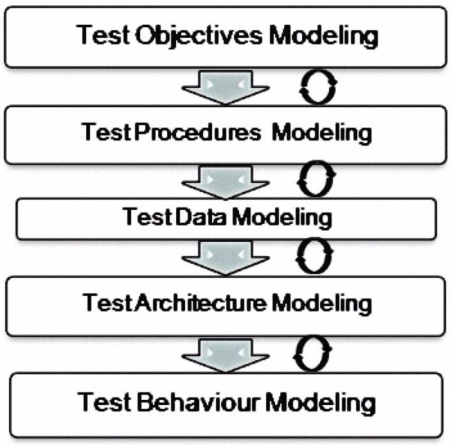

Figure 1. Overview of UTML Test Modeling Process

We define test modeling as 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]. Figure 1 depicts the test modeling process and the various phases it implies. Associated to each of those phases is a particular type of test model addressing a specific aspect of test design. Those various aspects have been have provided the base for a UML MOF(Meta Object Facility) meta-model representing a test-specific notation called UTML (Unified Test Modeling Language [22]). Based on that meta-model, a tool set is provided to guide test modelers in defining test specifications along the defined patterns. Therefore, figure 1 also provides an overview of the different types of test models UTML allows to define and how they relate to the test design process. As depicted on that figure, 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.

3.1 Test Objectives Modeling

The first phase 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.

3.2 Test Strategy Modeling

For each of the test objectives defined in the test plan and designed in the test objectives model, a test strategy must be designed, describing how that test objective will be assessed. Test procedures can be modeled as sequence of test steps, with each test step representing an action or an observation to be performed on one or more elements in the test setup. Each test step can be described with natural language. To ensure that all designed test procedures meet specific requirements with regard to their structure and semantics, model validation techniques can be applied on the resulting model. In fact that is the case for any UTML test model at every phase of test modeling. If any syntactic or semantic rule defined by the meta-model is violated in a test model, e.g. a mandatory field being omitted, than a warning message is issued for the test modeler to react accordingly and fix the issue. This ensures that conceptual errors in test design are discovered at the very early stage and that the resulting test models are both complete and consistent syntactically and semantically.

3.3 Test Data Modeling

The purpose of test data modeling is to precisely describe data that will be exchanged between elements of the SDP to implement the designed test procedures. Those data include stimuli, i.e. messages that will be sent to entities, as well as potential responses. Responses are modeled based on constraints dictated by semantic descriptions of the protocols. Generally, a static description of the protocol is available as a plain text document (IETF RFC), an XSD(XML Schema Descriptor) file or any other data description mechanism (e.g. ASN.1, IDL). Provided a machine-processable notation has been used for that purpose, the system data model could be imported automatically and provide a base for the test data modeling activity.

3.4 Test Architecture Modeling

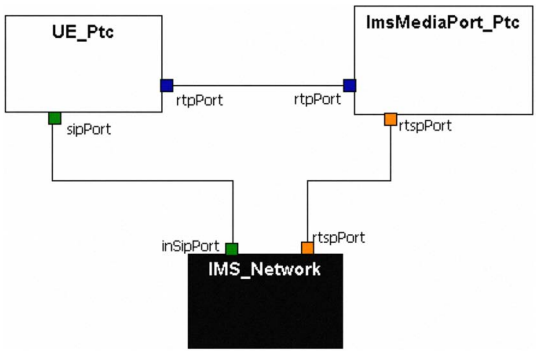

Figure 2. Test Architecture Diagram for sample IMS test case

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. Figure 2 displays a test architecture diagram for a sample IMS interoperability test case. 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). 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 are 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.

As depicted on figure 2, the designed test configuration for features a scenario involving two PTCs (Parallel Test Components) testing an IMS network for Video-On-Demand functionality. As visible on that figure, the test scenario requires 3 different protocols, i.e. SIP, RTSP and RTP.

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

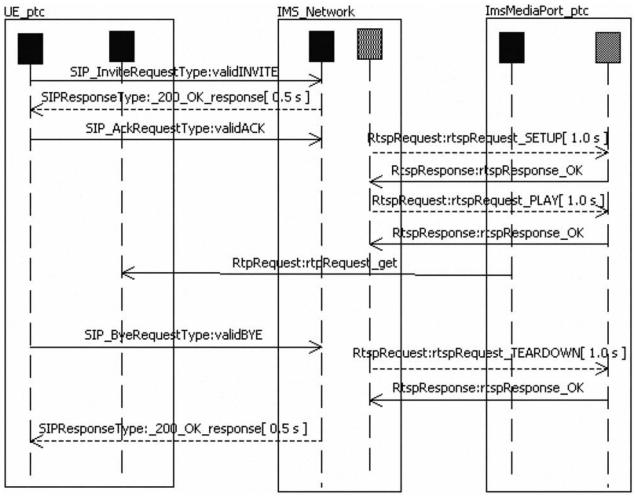

Figure 3. Test Behaviour Diagram for sample IMS test case

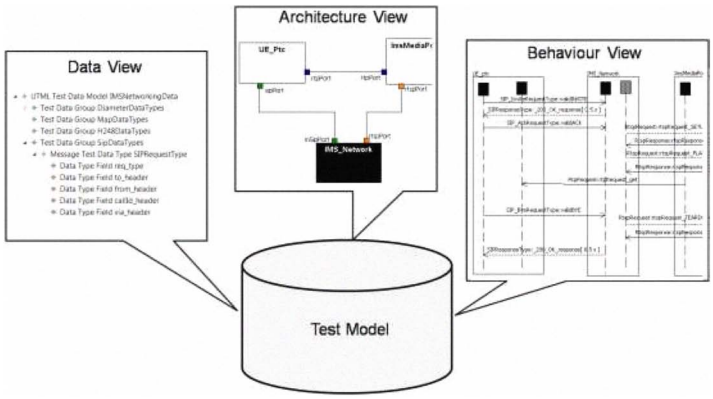

Figure 4. Different Views on the Test Design Model

test configuration previously defined in the associated test architecture model. A test behaviour model describes abstractly how each of the test procedures designed in the test strategy model will traduce in terms of actions on and between the entities in the associated test configuration. Figure 3 displays an example test behaviour diagram for an IMS Video-On-Demand (VOD) service . As depicted on figure 3, a UTML test behaviour diagram shows many similarities with a classical UML sequence diagram. The main differences 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 patterns into account. As shown in Figure 3, 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. Also visible on figure 3 is the equivalence between the architecture diagram and the layout for the associated behaviour diagram. In fact, as depicted on figure 4, the various diagrams merely provide a different view on the same test design model. Therefore, the initial setup for the behaviour diagram (i.e. the component instances and their connections) can be generated automatically from a previously defined architecture diagram to speed up test behaviour modeling, which would then simply consist in modeling the test actions. 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 SDP entities to be modeled in the same way as tests for services taken individually.

4 Test System Integration using Model Driven Test Engineering

Test development is a particular kind of software development. Therefore, the potential benefits of using domain-specific notations tailored for the processes it implies are high. However, introducing a new notation always poses a series of problems for organizations. It has to be ensured that the investment of learning the new notation, adapting the existing process and infrastructure to it etc. are worth the effort. To keep those investments as low as possible appropriate tools for the notation are required, that facilitates the usage of the notation, while integrating to the existing testing infrastructures.

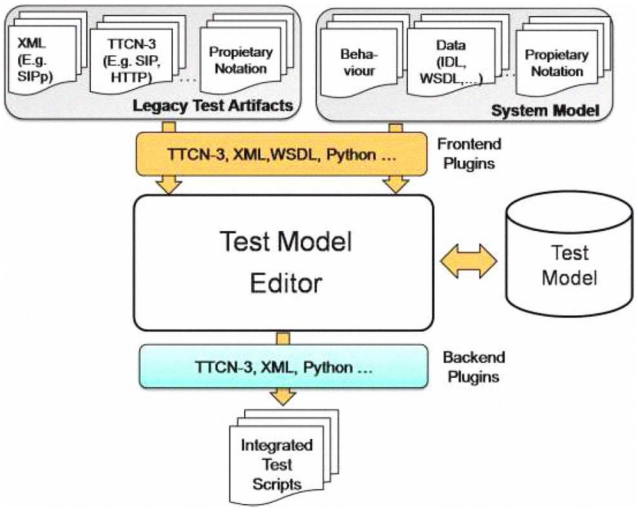

Figure 5. Possible Architecture of Test Modeling Tool

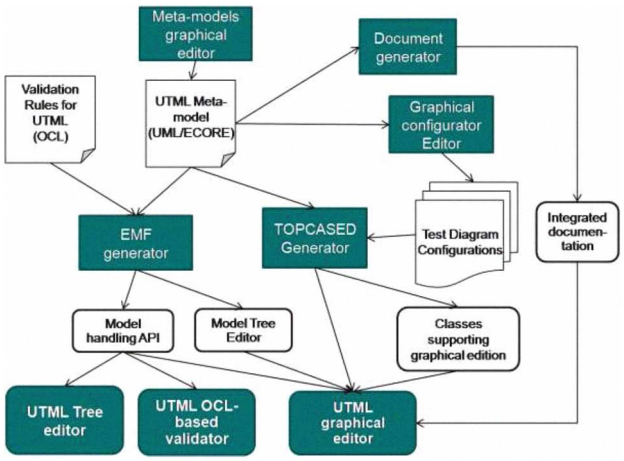

Figure 6. UTML Tool chain and relation to EMF and TOPCASED

Figure 5 depicts a possible architecture for a model-driven test engineering tool and how it could be used to design functional tests of SDP infrastructures. As displayed on that figure, the center-piece of the tool is a test model editor, in which the aforementioned views can be used to design the different aspects of the test model. Additionally the tool should define both a front-end- and a back-end plug-in interface. Using the front-end interface, legacy test artifacts (e.g. targeting single entities of the platform) can be imported via the appropriate front-end for reuse in test modeling. Using the same mechanism, elements of the system model (e.g. WSDL-files for SOAP web services, UML system models) can be imported for reuse in the test system as well. Furthermore, as depicted on the figure, the back-end interface provide the ability to export the integrated test model into an executable test scripting notation suitable for the target environment, via the appropriate back-end plug-in.

Furthermore, the tool is required to allow for external front-end-, and back-end plug-ins to be added, e.g. to import legacy test artifacts or to generate new test scripts written in proprietary notations.

5 Tool chain and Implementation Approach

Figure 6 depicts the implementation approach we used to develop a prototype visual editor for UTML test modeling. The prototype implementation is based on an ECore representation of the UTML meta-model. ECore is the Eclipse [19] Modeling framework's (EMF) variant of the Omega's Meta-Object Facility (MOF), i.e. a standardized meta modeling language. Through a set of tools hosted under the common umbrella of the Eclipse Modeling Project [18], EMF provided us with the means for following a model-driven development approach for the UTML modeling tool chain.

As depicted on figure 6, the UTML meta-model was defined using EMF's meta-model editor. Then, using the EMF generator, a set of APIs are generated automatically for handling the model, along with a basic model tree and tabular editor.

However, to increase usability and make test modeling and the resulting test model more accessible to people who might not be test experts, we also targeted a visual editor allowing for test models to be designed graphically. The TOPCASED [20] modeling framework combines EMF and Eclipse's Graphical Editing Framework (GEF) [17] to provide the means for defining graphical visual elements for an ECore/EMF meta-model. Those graphical elements are defined in so-called diagram configuration files, out of which the TOPCASED generator generates a graphical editor for the associated ECore meta-model automatically.

The tool chain has been designed and implemented to support the process depicted on Figure 1 by providing a set of wizards to guide the user through that process. Additionally, the tool chain defines an Eclipse plug-in API, via which back-ends can be provided for exporting the UTML test model, once it is ready. The current version of the tool chain provides back-end plug-ins for TTCN-3, XML and PHP. Those plug-ins use template-based model-to-text transformation to transform UTML models into the targeted notation. However, the choice of the transformation approach is left to the back-end provider. This high level of flexibility in Before each export operation the model is validated for consistency against the meta-model and some predefined Object Constraint Language (OCL) constraints. Therefore,

flaws in test design can be identified and addressed at early stage of testing. 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.

This ability of flexible mapping, that allows for mapping rules to be provided on-the-fly for a notation is particularly interesting for heterogeneous systems, as it allows the new notation to be integrated seamlessly in existing infrastructures. Furthermore, the tool chain provides a similar mechanism for importing test artifacts into the test modeling environment. The current version provides two plug-ins for importing TTCN-3 and WSDL files into UTML test models

The TTCN-3 code generated automatically from the UTML test model for the IMS 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.

One of the additional benefits of our approach is that the resulting test models can be understood by test designers as well as system designers and dedevelopers alike. Furthermore, by decoupling test system engineering into a design and implementation activities, it allows a more flexible organization of the work to avoid bottlenecks. This is especially important when agile test driven development (TDD) is practised, whereby the test design drives the whole product development process[1]. Also, for large projects in which design-by-contract is applied, UTML test models could provide the common ground for understanding the requirements on each entity in the architecture and for verifying that, that entity's implementation behaves accordingly. Although we assume that the learning curve for such a visual test modeling notation should be lower than that of a functional programming or a test scripting notation, a more in-depth qualitative analysis is required to validate that assumption. Furthermore the flexibility of the approach is also underlined by the fact that transforming the test model into a specific notation for a particular test infrastructure would only require providing the back-end plug-in for that notation.

6 Conclusion and Outlooks

We have presented a new approach for model-driven test engineering based on black-box test design patterns and described how it could be used to enhance reuse of system models and legacy test artifacts. Although a full quantitative evaluation of the approach is still ongoing, first results clearly indicate that the test development cycle is shortened significantly. The fact that the approach is also applicable to other kinds of service centric architectures, independent of the protocols being used. However some challenges are yet to be addressed. 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] Jennitta Andrea. Envisioning the next-generation of functional testing tools. *Software, IEEE*, 24(3):58–66, 2007.

[2] H. Buwalda. Action figures. *Software Testing and Quality Engineering*, pages 42–47, 2003.

[3] H. Buwalda and M. Kasdorp. Getting automated testing under control. *Software Testing and Quality Engineering*, pages 39–44, 1999.

[4] G. Canfora and M. Di Penta. Testing services and service-centric systems: Challenges and opportunities. *IEEE Journal*, 2006.

[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] 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.

[7] ETSI ES 202 553: Methods for Testing and Specification (MTS). Tplan: A notation for expressing test purposes. Technical report, European Telecommunications Standards Institute, Sophia Antipolis, 2007.

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

[9] Matthias Grochtmann and Klaus Grimm. Classification trees for partition testing. *Softw. Test., Verif. Reliab.*, 3(2):63–82, 1993.

[10] omg. *Meta Object Facility (MOF) Core Specification Version 2.0*, 2006.

[11] B. Panwar and K. Singh. Ims sip core server test bed. *IP Multimedia Subsystem Architecture and Applications, 2007 International Conference on*, pages 1–5, Dec. 2007.

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

[13] 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.

[14] A. Rothem-Gal-Oz. What is SOA anyway? *Journal*, Year.

[15] 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.

[16] Stephan Schulz, Anthony Wiles, and Steve Randall. Tplan-a notation for expressing test purposes. In Alexandre Petrenko, Margus Veanes, Jan Tretmans, and Wolfgang Grieskamp, editors, *TestCom/FATES*, volume 4581 of *Lecture Notes in Computer Science*, pages 292–304. Springer, 2007.

[17] Eclipse Open source Project. The eclipse graphical editing framework(gef). http://www.eclipse.org/gef.

[18] Eclipse Open source Project. The eclipse modeling project. http://www.eclipse.org/modeling.

[19] Eclipse Open source Project. The eclipse project. http://www.eclipse.org.

[20] TOPCASED Open source Project. Toolkit in open source for critical applications and systems development. http://topcased.gforge.enseeiht.fr/.

[21] M. Vignesh Karthik and S. Prateek. Building an ims client test bed with open source tools. *IP Multimedia Subsystem Architecture and Applications, 2007 International Conference on*, pages 1–5, Dec. 2007.

[22] 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.

[23] 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.

[24] Daping Wang. An xml-based testing strategy for probing security vulnerabilities in the diameter protocol. *Bell Lab. Tech. J.*, 12(3):79–93, 2007.

[25] 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.