2010 17th IEEE International Conference and Workshops on Engineering of Computer-Based Systems

Establishing a Service-Oriented Tool Chain for the Development of Domain-Independent MBT Scenarios

Marc-Florian Wendland, Jürgen Großmann, Andreas Hoffmann

Fraunhofer Institut FOKUS
Berlin, Germany
{marc-florian.wendland, juergen.grossmann, andreas.hoffmann}
@fokus.fraunhofer.de

***Abstract*—Since software systems become more and more complex, the efforts for developing, documenting and executing meaningful test cases increases. Testing is a vital, but time- and resource-consuming activity. To avoid running out of time or budget, new test methodologies had to be established in order to increase reliable, yet maintainable test scenarios. In the last years the Model-Driven idea matures to the most promising approaches to solve current problems in the software development domain. Model-Based Testing adopts these concepts to exploit their benefits for testing area. In this paper an integrated tool chain (called FOKUS!MBT) is discussed, to enable a Model-based development of testing scenarios. It is based on a canonical metamodel for testing concerns and a service-oriented model storage and exchange infrastructure, that allows a flexible, yet extensible adaptation to different test process requirements. Its premise is to establish a tooling architecture for the specification and development of a domain-independent Model-based testing scenario.**

Keywords-FOKUS!MBT, MBT, UTP, Testing Metamodel, UML

I. INTRODUCTION *(HEADING 1)*

Software systems which are used in a critical environment have to face increased quality assurance requirements to avoid economical or human harm. Spending more efforts for quality assurance measures narrows the budget of software manufacturers. Such activities have to be well planed, organized, executed and documented to prevent running out of time and resources. The fundamental approach to evaluate a system's quality is testing. A testing process includes activities to validate or verify a software system with regard to its specification. These activities can be separate into static testing and dynamic testing.

In *Model-Driven Engineering* (MDE) the software development process is based on (semi-)formal artifacts, called models. Models are simplified constructs of an existing real-world system or a system to be. They are used on a higher level of abstraction to achieve both a better understandability and programming language and platform independency. *Model-Based Testing* (MBT), in addition, emphasizes a scenario, where the test artifacts are described partially or entirely by models to support a model-driven or model-based process for testing concerns. Such testing models should facilitate the specification, execution, evaluation or documentation of the test process, otherwise there is no rationale to use models for testing purpose. Utting, Preschner and Legeard [8] mentioned, that in particular the lack of documentation in traditional testing processes is still present.

Using models for testing purposes is not new. Several mature tools [22] [23] are available which produces test cases out of models, but mostly, they are intended to generate test cases solely, while other test relevant information like execution results or documentations are still text-based. Moreover the tools usually lack integration with other testing tools and the system development infrastructure. Unfortunately, there is neither a standardized definition of MBT itself nor a recommendation, what kind of models should be applied. In fact, it is totally up to a developer's preferences, respectively his needs, what kind of concepts a testing metamodel exhibits.

The contribution of this paper is to present a generic architecture for FOKUS!MBT, a model-based tool chain that provides a solid and flexible basis to realize MBT processes in a larger scale. The canonical model of FOKUS!MBT is described by a MOF-based testing metamodel, which proves the required degree of formalism. This paper is outlined as follows: Section 2 will recapitulate the state of the art of MBT as well as related work. Section 3 introduces the testing metamodel and the repository infrastructure briefly. Within section 4, minor case studies are described, where the prototype was applied to. Finally, we will summarize the discussed key concepts.

To distinguish between similar named elements of the discussed metamodels, we prefix each ambivalent element with the qualified prefix, so that *UML::Element* will differ from *TestingMM::Element*.

978-0-7695-4005-4/10 $26.00 © 2010 IEEE
DOI 10.1109/ECBS.2010.47

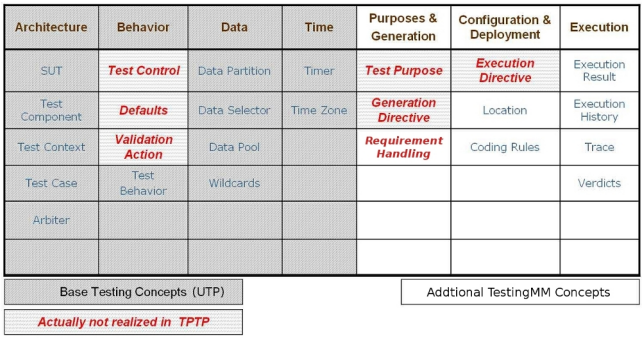

Figure 1. Conceptual Overview of TestingMM

II. Related Work

Pure testing metamodels are rarely proposed in the literature. The most prominent approach is the UML2 Testing Profile (UTP) [2], which represents an enhancement of the UML Superstructure [1] to provide test specific concepts within the UML. As its name suggest, the UTP was defined by using the UML-internal profile mechanism. Moreover, a standalone MOF-based metamodel [14] was specified too, in order to use the UTP outside of the UML scope. With UTP, tester and developer are intended to speak the same language for a better understanding of each other concerns. The UTP was highly influenced by TTCN-3 [4].

With the Test & Performance Tools Platform (TPTP) [3], a still ongoing project of the Eclipse Foundation, an open source and comprehensive testing metamodel is provided. TPTP was one of the first implementations of the UTP standalone MOF-based metamodel. Beyond that, it offers concepts for the test execution, execution results tracing over time, deployment and even more. There are some tools available which are based on the TPTP metamodel, but none of them deal with the comprehensive idea of MBT directly.

Schieferdecker and Din [5] presented a metamodel for TTCN-3. TTCN-3 stands for Testing and Test Control Notation and represents a standardized testing language, defined and maintained by the European Telecommunication Standards Institute (ETSI) [6] for testing complex, distributed systems. The TTCN-3 metamodel was established to integrate TTCN-3 into a MDE process natively. However, the TTCN-3 metamodel is not a comprehensive metamodel for testing but it eases the expression of test cases in a model-based or model-driven environment.

Dueñas et al. [15] discussed a metamodel which is capable to cope with model-based testing in a Product Family Engineering (PFE) environment. The metamodel they described is internally based on UTP but with dedicated enhancements according to the applied environment.

The Automatic Test Markup Language (ATML) embodies an XML-based specification for a better exchange of test data. It was developed and maintained by the IEEE [7] and is used especially in the ATE (Automatic Test Equipment) industry.

Another XML-based test description language is given with the Automated Generation and Execution of Test Suites for DIstributed Component-based Software (AGEDIS) [18]. It was developed during the AGEDIS-EU-Project.

Finally, the Unified Test Modeling Language (UTML) [9] had to be mentioned. It was developed by Alain Vouffo-Feudjio at the Fraunhofer Institut FOKUS [12] to support the creation, modification and maintenance of a dedicated testing model within a pattern-oriented test description approach.

To sum-up the concepts of the discussed approaches, in particular of the UTP and TPTP, there are neither concepts for essential testing aspects like test case generation rules or requirements association nor a recommendation, how to persit testing models within a repository.

III. Architectural Overview

One premise of the *Testing Metamodel* (TestingMM) was to provide concepts for a holistic test modeling and process approach. For the exchange and storage of TestingMM instances (as well as the correlated system models) and other related artifacts, FOKUS!MBT utilizes the integrated model repository provided by ModelBus [13] technology.

A. Structure of TestingMM

The TestingMM is a conceptual merge of TPTP and UTP, with specific additions of the UML. Fig. 1 shows a comparison among the UTP, the TPTP and the concepts newly introduced to the TestingMM. Staying close to the naming convention of the merged metamodels, facilitates import and export possibilities between the TestingMM and its related models (UML, UTP, TPTP). TestingMM is implemented in Ecore [10]. It consists of ten subpackages, where each of them deals with a separate concern of the test description. The packages are namely: Foundation: The foundation package provides basic concepts for object-oriented modeling. It is a subset of the *UML::Kernel* package with additions from the *UML::Components* package to specify components, ports, data types etc.

1. Architecture: It is similar to the *UTP::Architecture*, with extra information outside of the scope of the UTP like relationship to requirements.
2. Data: The *TestingMM::Data* package is similar to the *UTP::Data* package. It supports the concepts for data pools and data partitions..
3. Time: A simple and intuitive timer mechanism.
4. Behavior: This package includes everything to model the behavior of test cases. Available behavioral aspects are simplified *UML::Interaction* models.
5. Generation: Information of how tools should derive test cases out of input models are located in the generation package.
6. Purpose: This package deals with the description of a test context's objectives and its relation to system's requirements to integrate the purposes of a test context.

7. Configuration: The configuration package specifies the deployment of the test artifacts to distributed locations and machines. It is similar to the *UML::Deployment* package.
8. Platform: The platform package describes how a test context and all its related test cases are applied to a specific platform. It provides mapping directives among interfaces as well as elements to initiate or shutdown the target platform.
9. Execution: Concepts to store the results of one or multiple test executions and traces between test results and requirements are defined in the execution package.

B. *TestingMM Core Concepts*

The essential package for the creation of test contexts (test suites) is *TestingMM::Architecture*. Fig. 1 has shown the core concepts of the UTP related to additional TestingMM elements. Fig. 2 represents a concise overview of the package, with cross-package references[1] (highlighted in a different color). The most relevant metaclass is *TestingMM::TestContext*. A test context collects a set of test case to satisfy the test context's purpose. Test cases are meant to run against several SUT instances, performed by test components. Each test component returns a local verdict to the test context they belong. These local verdicts had to be assembled to provide a unique and unambiguous verdict for the whole test case. An arbiter is responsible for this.
Arbitration will be managed by an optional arbiter. The idea corresponds to the arbitration mechanism of TTCN-3. Test designers can abandon this optional part, thus a virtual default arbiter will be used instead. In that case, the global verdict will be determined by defining a precedence order for verdicts, as the TTCN-3 core specification does. The order is established as *None* < *Pass* < *Inconclusive* < *Fail*, where < means the right verdict overrides the left verdict. Hence, if a test component states a *fail*, the global verdict will never be *pass*. However, some testing aspects, for example real-time testing, require a more fine-grained arbitration mechanism as the default one.

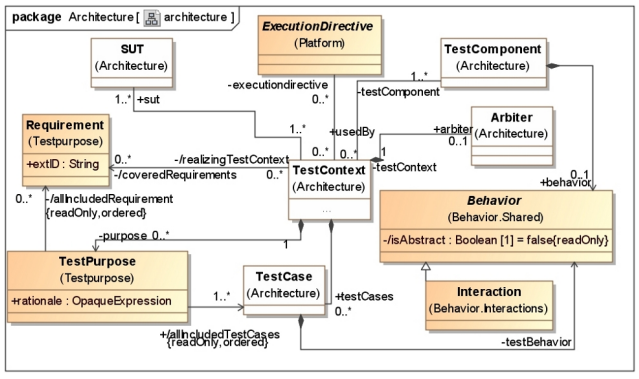

Figure 2. TestingMM Core Concepts Overview

[1] Due to readability, we have hidden some of the associations.

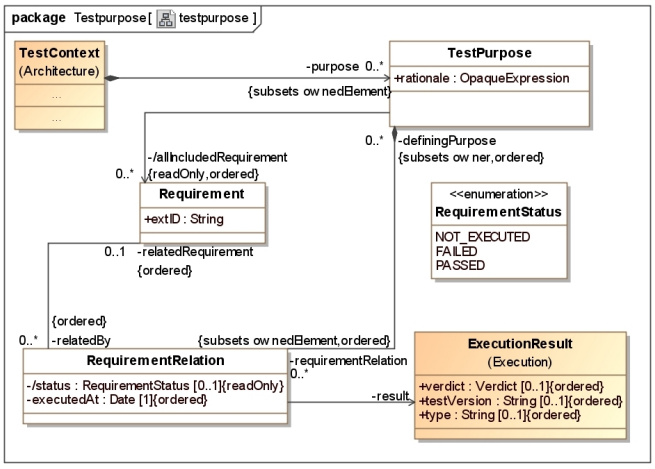

Figure 3. Test Purpose Package

Therefore, it is possible to specify a domain-specific arbiter within the testing model. It has to be said, that a default arbitration mechanism fits for the most test scenarios where TestingMM was involved in.
Test Cases are expressed as interactions. As mentioned before, the specification of interaction behavior is similar to, but in some details more intuitive than UML interactions. Anyway, they share basic concepts like defining messages, passing arguments to the receiver, executing invoked behavior and return operations results. A test case can either be created manually or be automatically derived from any behavioral description stored in whatever artifact.

C. *Test Purposes*

In UTP the rational of a test context or a test case is defined by a dependency relationship. Since dependencies connecting any element with no specific semantic (except that the source element depends on the target element), it might be ambiguous if a test case is the objective for a UML::Class instance. Commonly, a test purpose (or test objective in UTP nomenclature) express' what a test case or a couple of them aim for, otherwise its usage won't contribute any additional findings for the test analysis phase. If the intention of test case is not clear and it fails, what does this mean for the system? Or vice versa, a reviewer won't gain any confidence in the quality of the whole test process, if no unambiguous description for the rational of its test cases is available.
Test cases are intended to verify a system's requirements, elicitated and specified during the requirements engineering process. Therefore, a test case is related to at least one requirement of the system or its parts. TestingMM defines a more detailed structure for test purposes (see Fig. 3). A requirement is not contained by a test purpose, since there is a possible many-to-many relationship. One requirement can be included by a set of test purposes and vice versa. Instead of being a container for a requirement, the test purpose contains a *TestingMM::RequirementRelation* that relates a requirement to the execution result of a test case.

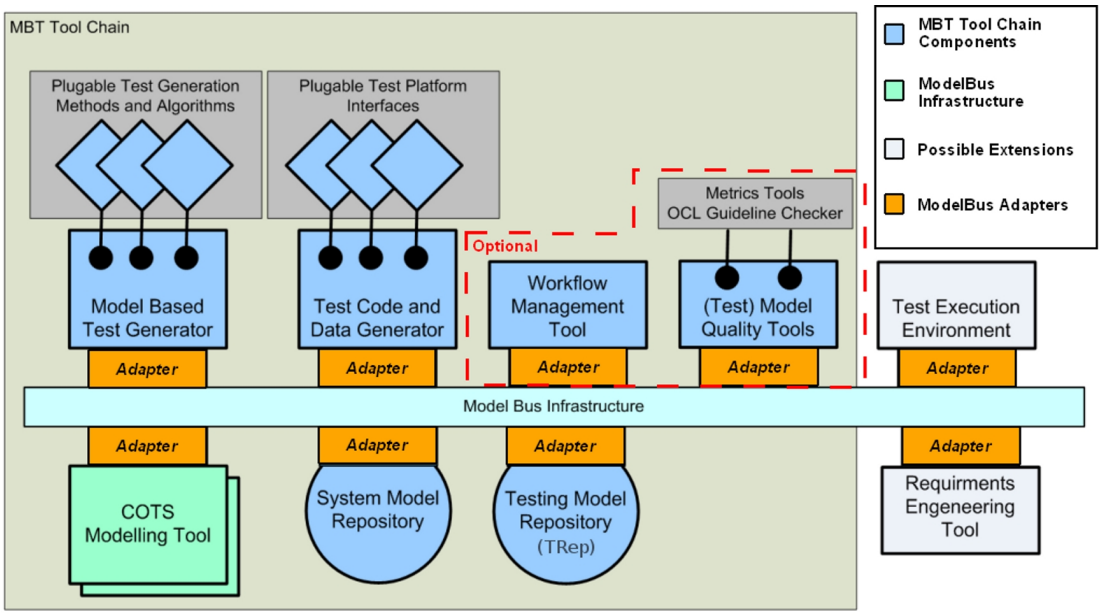

Figure 4. FOKUS!MBT High-level Architecture

Such a relation states the execution status of a requirement (and implicitly of the related test case). Possible values are NOT_EXECUTED, PASS and FAIL. The concrete value for a requirement status is derived from the related execution result. Since a test case is arbitrated to either pass, inconclusive or fail, a requirement will only be set to pass, if the arbiter states *pass*, otherwise *fail*. Inconclusive makes no clear statement from requirements point of view, because a requirement has to act like it is intended to, thus an inconclusive verdict will fail the whole requirement.

Since a execution result of a test case is traceable over a period of time, each new invocation of the test case will create a new *TestingMM::RequirementRelation* instance, which will be connected to the new execution result. An assigned date for each new invocation enables the reviewer to follow a requirements status over time.

D. Process Workflow and Model Exchange

Artifacts are stored in repositories to allow them being integrated in a distributed development process. The ModelBus is a model-driven tool integration framework which enables a seamlessly integrated tool and model environment for a model-based engineering process [28][29]. The idea is to separate participating tools and models (testing models as well as system models) in a loosely coupled tool chain infrastructure. A general overview of the FOKUS!MBT infrastructure is depicted by Fig. 4.

The ModelBus acts as a middleware component to provide a transparent access to the development artifacts in a distributed, yet integrated environment. Participating tools exchange their data through specific web service adapters. An adapter exhibits the tool's services and required parameters via the *Web Service Description Language* (WSDL). This enables a tester to work with his desired modeling tool, as long as it is adopted.

The foundation of FOKUS!MBT are the integrated model repositories, especially for the TestingMM instances. Test case and test data generators can extend the tool chain by simply be plugged into the ModelBus infrastructure via their specific adapters.

The process flow orchestration of a specific scenario is commonly described (but not limited to) by the *Business Process Modeling Notation* (BPNM) [25]. Workflow and quality services won't be discussed in this paper, since they are not a vital for testing purposes.

IV. Case Study

The proposed architecture has been used and evaluated in several research projects. It has been shown that the integrated test repository could be successfully adapted to the different tooling infrastructures and project requirements.

In the ModelPlex-Project [24], an interdisciplinary project funded by the EU, FOKUS!MBT and its canonical test model TestingMM have been developed and applied use cases from SAP and Telefonica.

The RTE-Space-Project [19], initiated by the European Space Agency (ESA), explores software migration problems and methodologies for the space domain. The MBT tool chain was applied to safeguard the modernization of a legacy file archiving and versioning system, by focusing on test generation from UML::Interaction models in particular [20]. In the following we will present an example that presents a MBT approach to generate tests for a simplified phone conference application. The artifacts necessary to generate, execute and manipulate the tests have been stored exclusively in the repositories of FOKSUS!MBT.

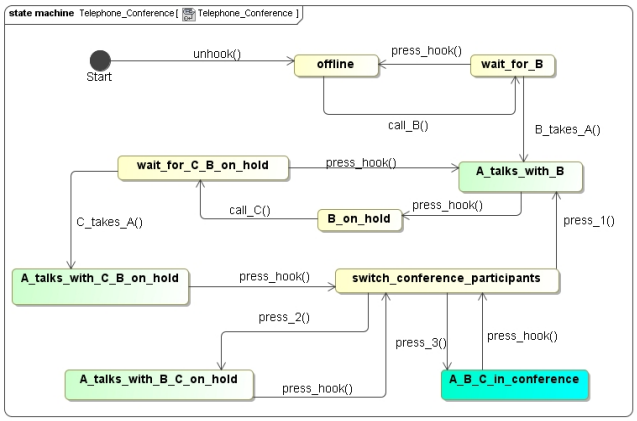

Figure 5. Telephone Conference Scenario

A. Scenario Description

The first step was to create a behavior model out of a given specification. In contrast to the RTE project [19][20], an *UML::StateMachine* was chosen for this scenario. Fig. 5 shows the (simplified) behavior model of the system for subsequently applied test case generation.

B. Test Case Generation

The intention was to generate test cases out of state machines by using the MBT tool generator for structural coverage. The generator was parameterized by the concepts of the *TestingMM::Generation* package. The successfully integrated MBT tool [17] can be configured to use either the Dijkstra algorithm [26] or a random algorithm to achieve its coverage goal. In most situations it will be either state or transition coverage. Table 1 figures out the currently available structural coverage criteria of the TestingMM.
The combination of coverage criteria and their parameter act as stop conditions. They can be combined by logical operators like *AND*, *OR*, *XOR* to span a tree of fine-grained generation stop criteria. The result of such a derivation is an ordered list of transition names, that had be traversed. UML denotes such a list as an trace and notates it like this: *<event1, event2, event3…>*. These string traces had to be converted to concrete operation calls or signal sendings to interact with the SUT. A possible test case, resulting from a 50 percentage transition coverage goal, consist of the following trace: *<unhook(), call_B(), B_takes_A(), press_hook(), call_C(), C_takes_B(), press_hook()>*.

C. Deadlock Prevention

As Fig. 5 shows, it is possible to run into an infinite loop during the derivation process. Assume that a 100 percentage transition coverage goal was defined. If the algorithm selects the trace *<unhook(), call_B(), B_takes_A(), …>* its stop criteria won't be reached, since the *press_hook()* transition between the states *offline* and *wait_for_B* is unreachable. To avoid deadlocks like this, it is necessary to define a default stop criterion, which restricts the resulting length or the process time of the derivation. In this scenario, the default stop criterion restricts the test length to maximal 30 execution steps.

D. Test Execution

After the derivation process, the resulting traces are transformed into instances of *TestingMM::Interaction* models. A specific model-to-text transformation generated executable TTCN-3 code out of the resulting traces. Each test context was transformed into a separate TTCN-3 module, containing the required type system for execution. The transformator was realized with Acceleo [27], an implementation of the OMG's MOFM2T [27] specification.
In contrast to the RTE scenario, a real SUT was unavailable. In order to get the test cases running, the SUT was simulated within TTCN-3 by a simple component. The generated test cases were successfully executed with the TTworkbench [16].

TABLE 1. STRUCTURAL COVERAGE CRITERIA

Coverage Criteria	Description	Parameter
State Coverage	Denotes the quantity of traversed states of the state machine, expressed in percent	0-100% (integer)
Reached State	Stops as soon as the state with the given name was reached	State name (string)
Transition Coverage	Same as *State Coverage* but concerning transitions	0-100% (integer)
Reached Transition	See *Reached State*	Edge named (string)
Test Length	Restricts the length of the resulting sequences	No of steps (integer)
Test Duration	Restricts the test case's length by a maximal process duration	Duration in milliseconds (integer)

V. CONCLUSION AND FURTHER WORK

In this paper an approach of an integrated metamodel and repository for testing was presented, which merges the specific concepts of the UTP and the TPTP. The resulting TestingMM is intended to cope with the formalization of an entire test process. Needful additions to achieve this goal were elaborated and described. Afterwards the service-oriented infrastructure FOKUS!MBT was presented and briefly described.
It has been shown that FOKUS!MBT was able to handle the variety of different domains due to its flexible and to the different tooling infrastructures and project requirements. FOKUS!MBT was successfully applied into real world case studies by following the major MDE concepts. Test cases

were automatically derived from appropriate behavioral aspects of UML system models and translated into executable TTCN-3 test cases. These test cases were compiled into Java executables subsequently and executed by the TTworkbench. The implemented tool chain showed that the entire testing process can be automated even if test execution and result analyzation had still to be performed manually. Further work will address an adaption of the execution environment to invoke the compilation and execution process automatically, orchestrated via BPMN as well as to establish requirements traceability.

ACKNOWLEDGMENTS

This research has been co-funded by the European Commission within the 6th
Framework Program project Modelplex contract number 034081 (cf. http://www.modelplex.org).

REFERENCES

[1] Object Management Group (OMG): OMG Unified Modeling Language (OMG UML) Superstructure, Version 2.2. http://www.omg.org/cgi-bin/doc?formal/2009-02-02.

[2] Object Management Group (OMG): UML2 Testing Profile, Version 1.0. http://www.omg.org/cgi-bin/doc?formal/05-07-07.

[3] Eclipse Foundation: Test & Performance Tools Platform (TPTP). http://www.eclipse.org/tptp.

[4] European Telecommunications Standards Institute (ETSI): The Testing and Test Control Notation version 3 (TTCN-3). http://www.ttcn-3.org.

[5] Schieferdecker, Ina; Din, George: A Metamodel for TTCN-3. In: Applying Formal Methods: Testing, Performance, M/E-Commerce. LNCS, vol. 3236, pp. 366-379. Springer, Heidelberg (2004). ISBN 978-3-540-23169-1.

[6] European Telecommunications Standards Institute (ETSI). http://www.etsi.org.

[7] IEEE: Automatic Test Markup Language (ATML), http://grouper.ieee.org/groups/scc20/tii/

[8] Utting, M.; Pretschner, A., Legeard, B.: A Taxonomy of Model-Based Testing. ISSN 1170-487X, 2006. http://*www.cs.waikato.ac.nz/pubs/wp/2006/uow-cs-wp-2006-04.pdf.*

[9] Unified Test Modeling Language (UTML), http://www.fokus.fraunhofer.de/de/motion/ueber_motion/technologien/utml/index.html.

[10] Eclipse Foundation: Eclipse Modeling Framework (EMF). http://www.eclipse.org/emf.

[11] Eclipse Foundation: UML2 Project (MDT-UML2), http://www.eclipse.org/modeling/mdt.

[12] Fraunhofer Institut für Offene Kommunikationssysteme (FOKUS), Berlin. http://www.fokus.fraunhofer.de

[13] MoldeBus 2.0, http://www.modelbus.org.

[14] Object Management Group (OMG): Meta-Object Facility (OMG MOF). http://www.omg.org/mof/

[15] Dueñas et al.: Model driven testing in Product Family Context. Universidad Polytécnica de Madrid, http://modeldrivenarchitecture.esi.es/pdf/paper4-2.pdf.

[16] TTworkbench, Testing Technologies IST GmbH, http://testingtech.de

[17] Model-Based-Testing Tool (MBT), http://mbt.tigris.org

[18] AGEDIS Project, Automated Generation and Execution of Test Suites for DIstributed Component- based Software, http://www.agedis.de.

[19] Sadovykh et al.: On Study Results: Round Trip Engineering of Space Systems. Fifth European European Conference on Model-Driven Architecture Foundations and Applications (ECMDA) 2009, Twente, Netherlands.

[20] Sadovykh et al.: Architecture-Driven Modernization in Practice – Study Results. 14th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS), 2009, Potsdam, Germany.

[21] Eclipse Foundation: Acceleo (MTL), http://www.eclipse.org/modeling/m2t.

[22] All4Tec: MaTeLo (Markov Test Logic), http://www.all4tec.net/index.php/All4tec/matelo-product.html

[23] TGV, http://www.inrialpes.fr/vasy/cadp/man/tgv.html

[24] ModelPlex Project, http://www.modelplex.org

[25] Business Process Modeling Notation. http://www.bpmn.org/

[26] Object Management Group (OMG): MOF Model to Text Transfortmation Language (MOFM2T). http://www.omg.org/docs/formal/08-01-16.pdf.

[27] Aldazabal et al.: Automated Model Driven Development Processes. Proceedings of the ECMDA workshop on Model Driven Tool and

[28] Process Integration, Fraunhofer IRB Verlag, Stuttgart 2008. ISBN: 978-3-8164-7645-1.

[29] Hein, Christian; Ritter, Tom; Wagner, Michael: Model-Driven Tool Integration with ModelBus. Workshop Future Trends of Model-Driven Development, 2009.