Model-Based Testing in Legacy Software Modernization: An Experience Report

Marc-Florian Wendland,
Marco Kranz, Christian Hein, and Tom Ritter
Fraunhofer Institut FOKUS
Kaiserin-Augusta-Allee 31, 10589 Berlin
Germany
{marc-florian.wendland, marco.kranz,
christian.hein, tom.ritter}@fokus.fraunhofer.de

Ana García Flaquer
DOME Consulting & Solutions, S.L.
Parc BIT, Edificio U,
Local 14 07121 Palma de Mallorca
Spain
agarcia@dome-consulting.com

ABSTRACT

With the advent of cloud computing more and more vendors strive to modernize legacy applications and deploy them into the cloud. In particular when the legacy system is still applied in the field, the vendor must ensure a seamless change to the modernized system to not lose any economical assets and to keep the business running. As with normal development processes, testing is also inevitable for a modernization process to gain confidence that the modernized system behaves correctly. This paper describes an experience report from the FP 7 research project REMICS that deals with model-driven modernization of legacy systems to the cloud. We employed a model-based testing process for safeguarding the correct migration of the modernized system's functionality. As test modeling language, the UML Testing Profile was applied. The modernized system, called DOME, was one of the case studies contributed by one of the business partners of the project.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Software/Programm Verification – *validation, tracing, symbolic execution, testing tools.*

General Terms

Design, Standardization, Languages, Theory, Verification.

Keywords

Model-driven modernization, model-based testing (MBT), test automation, UML Testing Profile (UTP), Fokus!MBT

1. INTRODUCTION

The principles of Model Driven Architecture (MDA), specified by OMG [8], and concepts around domain-specific languages have gained much popularity in the recent years. The ideas of MDA initiated a still ongoing change of paradigms, were (semi-)formal models instead of code represent the main and centralized artifacts in the software engineering process. While MDA or in general model-driven software engineering (MDE) approaches usually start with models on a rather high level of abstraction, which are gradually refined into less abstract (or more platform-specific) ones, modernization of legacy systems start from the opposite direction, since the legacy system is already implemented and deployed in the field, potentially for a long time. The success of model-driven approaches in the development of systems led to initiatives like Architecture-Driven Modernization (ADM), also specified by OMG. In ADM (notice that ADM is just the inverse acronym of MDA), knowledge about the legacy system is preserved in models that serve as the starting point for re-building the system. As with any ordinary development process, quality assurance measures need also to be carried out in software modernization for safeguarding the transition from the legacy to the modernized system. Testing is an important, yet cost-intensive quality assurance measure in industrial software development due to several reasons [2]. Model-based testing (MBT) techniques bear the potential to reduce costs and to lead to more structured and systematic test design [15]. MBT stands for an umbrella of techniques that use (semi-)formal models for deriving test-relevant artifacts such as test cases, test data, test configuration, etc. [9] [1]. Since the REMICS project is aiming at model-driven modernization of legacy systems for making them deployable in the cloud, we decided to apply an MBT approach on the quality assurance side for safeguarding the functional compliance between the modernized and the legacy system.

This paper describes both our MBT methodology supported by the academic test modeling environment Fokus!MBT and its application to the DOME case study. Finally, we summarize the lessons we have learned from that case study. The migration itself will not be discussed in this paper. Throughout this paper, we use the term *methodology* as shortcut for "our methodology that is supported by the test modeling environment Fokus!MBT". In case of potential ambiguities we explicitly use the term *Fokus!MBT methodology*.

The remainder of the work is structured as follows: Section 2 summarizes key publications in the area of MBT in modernization. Section 3 provides a brief overview of the REMICS project and the DOME use case. Details of our methodology are presented in section 4. Section 5 describes the lessons we learned from the case study. Finally, section 6 reflects our work and provides a prospect on future work regarding methodology and tooling.

2. RELATED WORK

Related work in the domain of MBT in ADM is very limited. More specifically there are, to our best knowledge, no other

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
JAMAICA'13, July 15, 2013, Lugano, Switzerland
Copyright 2013 ACM 978-1-4503-2161-7/13/07... $15.00.

methodologies published that are based upon the UML Testing Profile (UTP). The work presented in this paper is built upon experiences made during earlier projects Fraunhofer FOKUS was involved in, in particular during an extensive study funded by the European Space Agency ([12], [13]).

3. OVERVIEW OF REMICS AND DOME

The core objective of the REMICS[1] project is to develop efficient languages, methods and tools for a model-driven modernization of legacy systems to service cloud platforms. Three industrial case studies have been provided to REMICS by the business partners. In the work described by this article, we used the DOME case study for applying our MBT modernization methodology. The overall target was to safeguard the migration with respect to DOME's functional quality.

DOME Consulting is market leader in the development of cutting-edge technological solutions for companies in the tourism industry. Its clients portfolio are the major companies of the Spanish and international travel industry (such as Barceló Viajes Iberia, TUI etc.). The DOME application (henceforth simply called DOME) is a software system for excursion management. It allows the creation of different excursions based on a combination of different related services. The application provides all necessary modules to manage an excursion, from its definition to realization and execution. Furthermore, it eases the communication among the different service providers of each excursion. DOME is still in use in the field, thus, it represents an adequate case study for safeguarding the modernization process.

The current (legacy) DOME is realized as a typical Client-Server architecture. The client, i.e., the machine where the system is executed, offers the user interface. The server is realized as an Oracle Database that also is responsible for both the business rules and actual data storage. The code base consists of Delphi (for the UI) and PL/SQL (for the business rules). The migration of DOME into the cloud tackles some severe issues associated with the legacy system. The features and, thus, the code base of the system, evolved in terms of comprehension and complexity over time. Today, DOME is an almost eight years old application, and the continuous code evolution led to readability, maintainability and comprehensibility problems, inefficient code structure and performance issues. In particular the fact that the business rules were implemented as PL/SQL scripts in the database management system negatively influenced the systems maintainability. The strict procedural programming in PL/SQL forced to have different module implementation for different client needs and with a negative influence on database performance and scalability.

The REMICS work package *Validate, Control and Supervise new methodologies*, from which our work stems, is in charge of developing new or adopting existing methodologies for model-driven modernization.

4. MBT IN MODERNIZATION

The main target of the DOME case study was the development of a model-based system level test specification (henceforth referred to as *test model*) for safeguarding that the modernized system behaves like the legacy system in terms of functionality. A test model contains information pertinent to different testing activities such as test design etc. [9]. In our methodology, we treat the test model as the single source of truth (SSOT)[2] for all activities that affect or belong to the test process. As modeling language, we relied on UTP 1.1, and Unified Modeling Language (UML) 2.2. The tool that was employed is Fokus!MBT[3]. Fokus!MBT is an Eclipse-based test modeling environment that was adopted in REMICS for realizing MBT modernization methodology. As usual for research projects, only (a subset of) the functional requirements were considered.

Figure 1 briefly depicts an overview of the activities we carried out within the DOME case study, without going into details for now. The rounded rectangles represent artifacts being produced and consumed by activities. Activities are represented as labeled transitions that are executed either in a manually or automated manner. The dashed rounded rectangle (with caption *Test Model*) indicates that the artifacts and activities belong to or operate on the test model. The grey-shaded rectangles represent physical systems. The activities in Figure 1 belong (not visibly highlighted) to different test phases, which are obtained from and aligned with the fundamental test process proclaimed by the International Software Testing Qualifications Board (ISTQB) [7], consisting of test analysis (activities 1 and 2), test design (activities 3 and 4), test realization (activity 5) and test execution (activity 6). Moreover, the fundamental test process also includes the phases test evaluation and test closing. These phases, however, have not been addressed within REMICS. The following subsections describe in detail what has been done in the respective activities.

4.1 Requirements Reverse Engineering

Testing heavily depends on the quality of an existing test basis. The software/systems requirements specification (SRS) constitutes an important artifact of the test basis, especially for system (and acceptance) testing as it was addressed by DOME. When we started investigating the SRS that was available for DOME, we quickly found out that the requirements were not sufficiently precise for testing purposes, especially in terms of verifiability and completeness [6]. There was actually a need for reverse engineering of DOME's functional requirements with the required level of details. We decided to extract the information from the legacy system together with a representative of DOME consulting. In a series of collaborative (including both face to face and telephone conferences) meetings, we performed an informal review of the business rules (decoded as constraints expressed in PL/SQL) and an explanatory walkthrough through the legacy system in the field. The gathered information was then compiled into a SRS**,** resulting in 35 functional requirements, each describing a single feature of the system. The requirements have been formalized with OMG's System Modeling Language (SysML) requirements concepts.

4.2 Test Requirements Derivation

The previously identified, specified and formalized (yet partial) SRS served as a starting point for test requirements derivation. Our methodology heavily relies on test requirements. ISTQB defines test requirements as "… an item or event of a component or system that could be verified by one or more test cases […]".

As its definition proclaims, a test requirements consequently describes a single feature or characteristics of a system

[1] http://remics.eu/; funded by the European Commission within the FP7 (contract number 257793); last visited 2013-04-15

[2] http://en.wikipedia.org/wiki/Single_Source_of_Truth

[3] http://www.fokusmbt.com

requirement. This implies that a single system requirement is verified by at least one, but usually many test requirements. Consider the following example from DOME:

One of the system requirements states that modifications of general attributes of every excursion (such as name, description, type etc.) will be executed in a transactional manner, i.e., only if all mandatory attributes are filled in correctly, the modification will go into effect. Thus, the system requirement essentially describes two alternative behaviors, one positive (successful modification) and one negative (unsuccessful modification) behavior. Based on this observation, two test requirements have been derived. Each test requirement precisely states the conditions on how to enforce their according behavior and how the system under test is expected to behave. This information is later on exploited for deriving test cases that realize the test requirements. In Fokus!MBT, system requirements are transitively verified by test requirements, which in turn are realized by test cases. Test requirements are only verified if all of its test cases were successfully executed. Consequently, a system requirement is only verified and accepted, if all of its verifying test requirements are verified. Out of the 35 reverse engineered requirements of DOME we selected four which dealt with the modification of very fundamental excursion data. From these four requirements, 10 test requirements have been derived.

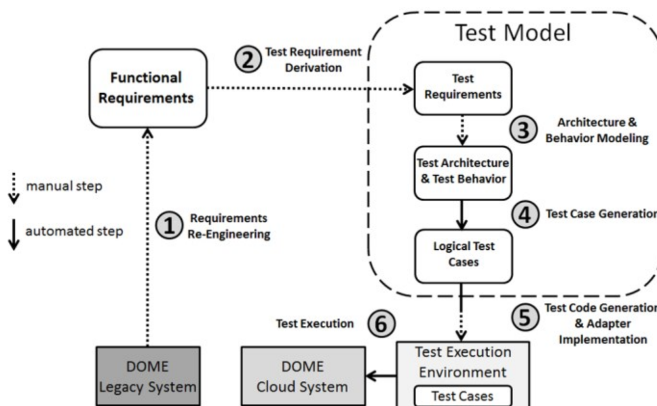

Figure 1: Overview of the test-related activities for DOME

4.3 Design of the Test Architecture

Our methodology follows the principle that behavior always relies on structure (inherited by UML). Thus, the first step in the test design phase is to derive the test architecture from the test requirements. The test architecture specifies the logical type system and interfaces of the SUT, as well as the test configuration including the test components and communication channels among the test components and the SUT.

4.3.1 On Different Abstraction Levels

Identifying the most appropriate level of abstraction for the test model is one of the most important decisions, since it influences the entire activities throughout the test process. The level of abstraction describes how close the logical type system and the interfaces of the SUT encoded in the test model resemble the technical type system and interfaces of the actual implementation [10]. Current literature mainly distinguished between two most extreme levels of abstraction, i.e., high and low level. Of course, there can be any shades of abstraction in between, however, both having distinct advantages and disadvantages. Finding the right level of abstraction is a non-trivial task.

One possibility is to choose a low level of abstraction where the logical type system and interfaces resemble as much as possible those of the actual implementation. This has the obvious advantage that the logical test cases within the test model are already very close to the actual implementation and, thus, almost executable. This would, in turn, simplify the implementation of the test adapter that is in charge of ultimately executing the test cases. The test adapter would only be considered as a thin layer delegator forwarding information from and to either side (i.e., the test execution system and actual implementation).

Using a lower level of abstraction also has, of course, some disadvantages. Models are by definition abstracted views on a real world entity that serve a certain purpose [14]. Test models commonly abstract from technical details of the SUT and focus only on these aspects that are pertinent for test design in the first place. Adding technical details into the test model result in a more complex test model, what might counteract the idea of MBT in terms of understandability, comprehensibility and maintainability. In addition, the actual design activity would be slowed down due to the larger number of details that need to be respected. Another fact that hinders the creation of lower level test models in an early testing approach is that the required technical details are often not known at that point in time. The earlier a test process starts the less reliable technical information of the actual implementation will be available. Another challenging side-effect of lower level test models is that changes to the design of the implementation would also entail changes in the test model itself, even if the system requirements would not have changed at all.

The second possibility is to choose a higher layer of abstraction what manifests in a logical type system and interface description that are derived from the information obtained from the SRS. This has the advantage that the test engineers can concentrate on building a test model that is not blurred or becoming overcomplicated by implementation-specific details. Such a test model is supposed to be less complex and better understandable. Since it is not reliant on the actual implementation, test-related activities can start as soon as the SRS is available and consolidated. The most significant disadvantage of higher level test models is that the technical details omitted in the early modeling phase have to be faced during test realization and test execution. In contrast to lower level models, the test adapter would become much more complex since it would be in charge of mapping concepts of different abstraction levels. Such complex test adapters often require resource-consuming manual work.

4.3.2 Abstraction Level for DOME Test Model

In order to find the most appropriate abstraction level for DOME, we investigated what information about the modernized system was available from the other work packages. The migrated system is a cloud service running on external servers of a cloud service provider. The user interface is web based, the complete business logic is contained within the cloud service, however, the database remains the same as for the legacy system. The cloud service is based on Java providing a Remote Method Invocation (RMI) interface as a means of communication with the UI. Since Fokus!MBT is currently not suited for doing UI testing and a complete specification of the RMI interfaces was not available, we decided to target a rather high level of abstraction for the test model. Another reason was that we wanted to achieve fast progress in the test design and test generation stages in order to get an early impression of the amount of test cases for the implementation.

4.3.3 Logical Type System and Interfaces

In order to abstract as far as possible from the technical details of the implementation, we decided to build the complete logical type system on enumerations and to rely on interface operations. The enumeration literals represent distinct logical data partitions (equivalence classes) of possible input data. Each literal, therefore, implies a certain constraint for later to be realized actual data, such as ranges of values. The constraints have not been specified in a formal manner, though. An example is presented in Figure 2**.** It shows an operation *login()* that takes a parameter of type *LoginRequest* and that returns a certain response. Due to page limitation, the parameters of the interface operations are in some cases partly omitted. Furthermore, it shows a specification the test environment consisting of a test component (*TestComponent*) and the logical SUT (*DOME_SUT*) that offers the *SUT_Interface*. Both test component and SUT are parts of the test configuration, which is part of the test context (*DOME_TestSuite)*. Test context, test component and SUT are stereotypes provided by UTP. The test configuration is a UML composite structure that connects the SUT and test components.

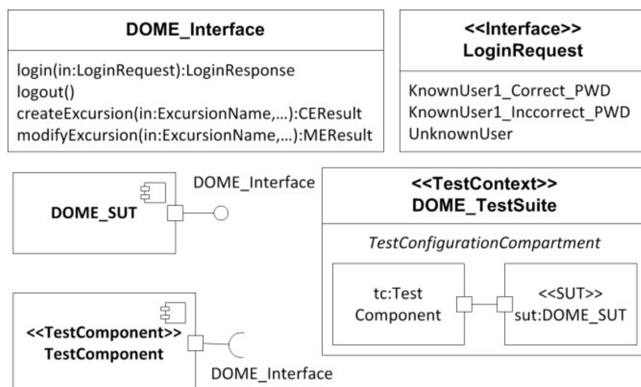

Figure 2: Use of Enumerations as equivalence classes

4.4 Automated Test Generation

After we had finished the specification of the test architecture, we started elaborating the test design. Our primary intention was to employ test generation in order to be more efficient in the test design phase. Fokus!MBT provides an integration with the test generation engine Spec Explorer and an according modeling environment that operates on UML state machines (SM). These SMs are referred to as *test basis behavior*, an extension of the term *test basis* as defined by ISTQB. The test basis behavior serves as the input for a test generator that derives test-relevant artifacts (including both test cases and test data) from it. Fokus!MBT provides a transformation from SMs into Spec Explorer input files that consists of a so called model program (actually a slightly changed version of a C# class) and cord (configuration) script. A Spec Explorer model program is a representation of an Abstract State Machine (ASM) [5]. The Spec Explorer first explores the ASM through symbolic execution, synthesizes concrete data values by using symbolic execution and its internal constraint solver Z3 and ultimately creates a Finite State Machine (FSM) out of the ASM ([3], [4]). Then, Spec Explorer derives test cases from this FSM as follows: each test case starts at the starting state, covers at least one uncovered transition, and ends in the first accepting state it reaches. The test case derivation ends when all transitions have been covered at least once. This explanation leads to the conclusion that the employed coverage criterion is transition coverage.

For the specification of the test basis behavior, Fokus!MBT employs SMs as *Extended Finite State Machines* (EFSM) to describe the communication between test environment and SUT from a global point of view. This means that neither the internal behavior of the SUT nor of the test environment is described explicitly. A transition either stands for a stimulus to or response from the SUT. In an EFSM, the global state of the SM is represented by both the states and an arbitrary number of state variables. The state variables convey necessary information for calculating the actually exchanged data. A SM has one initial state and one or more accepting states, representing final states for the generation process. Fokus!MBT tailors transitions as follows:

Each transition has at most one *trigger*. This trigger represents the operation or signal that is called/sent by either the SUT or the test environment. Triggering events appear on ports that either belong to the SUT or a test component, thus, a trigger defined on an SUT port represents the invocation of the SUT by a test component, and vice versa. A transition may have a *guard.* A guard is a logical expression and determines whether the transition can fire after it was triggered. Only when the guard expression evaluates to *true* the transition is taken. In Fokus!MBT, Guards are expressed as Spec# code. A transition can have an *effect* that describes what happens when the transition fires. The effect is used for modifying the global state variables. Besides, the output of either side related to a certain stimulus is calculated in the effect. Effects are expressed as Spec# code.

Figure 3 shows an example state machine that represents the behavior related to the modification of the core excursion data. The starting state shows the user being logged out. The user then logs into the system and creates a new excursion. After creation, the possible paths split up modeling both the successful modification case (all mandatory fields are set) and the unsuccessful case (one mandatory field not set). For the sake of simplicity, we hid the Spec# code in the model and only displayed a narrative text. This information is exploited by Spec Explorer for generating both test cases and adequate test data.

The majority of DOME's system requirements were of limited complexity regarding alternative behaviors, so that for each system requirement we targeted a single SM. These SMs, hence, realized all the test requirements for the system requirement they realize. As result of the test case derivation step we generated 46 test cases. Figure 4 shows a test case as sequence diagram completely derived from the SM depicted in Figure 3.

4.5 Test Realization and Execution

The generated test cases as sequence diagrams were transformed into TTCN-3 test scripts. TTCN-3 itself requires a test adapter for executing the test cases against the actual SUT. Due to the decision we made regarding the test model's higher abstraction level the test adapter for the modernized DOME system was in charge of mapping the logical interfaces and type system to the actual interfaces SUT. Since we did not formalize this mapping, the test adapter had to be implemented manually offsetting a lot of the time savings achieved thanks to automated test derivation. Figure 5 shows the actual test execution system based on the TTCN-3 execution system TTworkbench. The test execution results were actually of less interest in the DOME case study. Since the modernization tasks was not completed at the time when this paper was written, the modernized DOME implementation was far away from being complete.

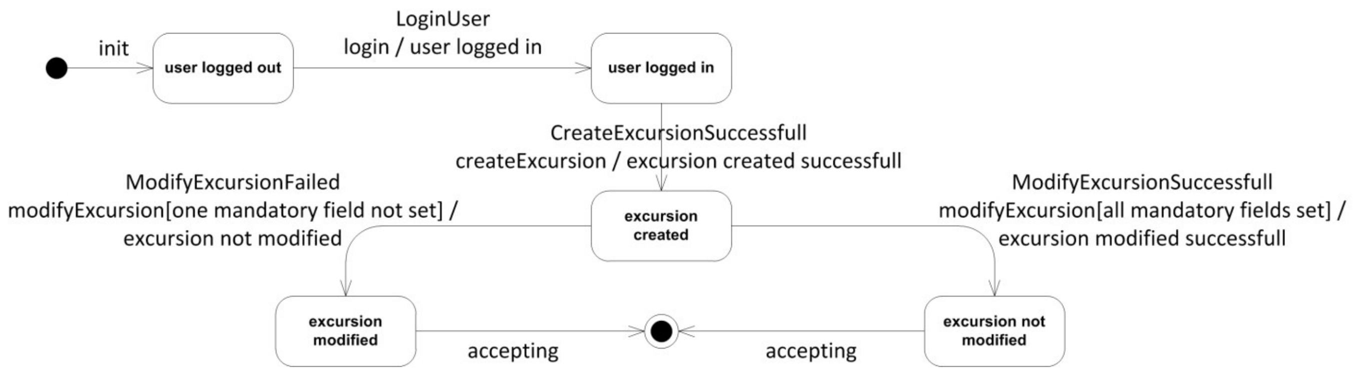

Figure 3: State machine describing the excursion modification behavior

5. LESSONS LEARNED

5.1 Adequacy of Methodology and Tooling

When we started working on DOME, we already had a partially elaborated methodology and tooling available. The question was to what extent the existing tools would be feasible for model-driven modernization. Reflecting the test process we carried out, we can say that we did not have to apply significant changes to both methodology and tooling for modernization. Of course, the reverse engineering and specification of requirements were not part of the former methodology. However, these activities were only integrated for the purpose of providing the required level of details to start doing testing and will not become part of the Fokus!MBT methodology. So far, we can say that our methodology and tooling seems stable and mature enough for being applied to more complex case studies. Certainly, there are still many open challenges in the realm of MBT we have not addressed in DOME like regression testing, change management, versioning etc. These open tasks have to be inevitably tackled in an industrial environment. We consider our achievements in REMICS (besides DOME there was also another case study we have worked on, but that was not finished at the time writing this paper) as starting point for working on these challenges.

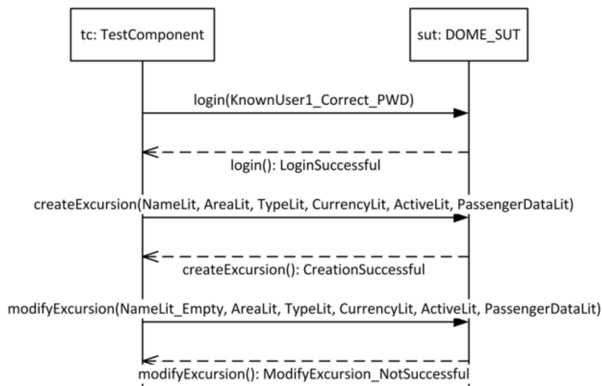

Figure 4: Example of a test case sequence diagram

5.2 Benefits of Test Requirements

As described earlier, our methodology is strictly based on test requirements. The benefits of using test requirements is twofold: At first, they serve as the basis for creating the test model in an autarkic manner, thus, the availability of other models such as a requirements model or system model is not necessary. Secondly, they are an important means of tracking the overall progress in test design and test execution in terms of requirements coverage throughout the test process. From what we learned from DOME, test requirements are an appropriate means to further decouple the testing and development side. As a downside, the derivation is of sufficiently precise test requirements is pure manual work, unless the SRS is represented in a formal way that allows automated processing. To the best of our knowledge, such SRS are not the majority in industrial software development, so that manual activities need to be planned for deriving the test requirements. This, however, positively influences the quality of SRS, because potential ambiguities in the system requirements are identified early in the development process.

5.3 Reflecting the Chosen Abstraction Level

As already said, we opt for a rather high level of abstraction for the DOME test model by just using enumerations that manifested in a rather simple logical type system and interface description. As this way of modeling came in most convenient while developing, validating, communicating and maintaining the test model, the task of making the generated test cases executable became labor intensive. We had to face the technical complexity of the implementation that was hidden in the test model in the test adapter. Since no formal mapping between the test model and the implementation [11] was available, the test realization task was pure manual work and rather annoying. It resembled in efforts and duration almost the efforts of developing the test model. Our finding here is that there is a clear trade-off between simplicity and maintainability of the test model and complexity of the test adapter. This leads to our next finding, namely that the test model and the applied tool should offer capabilities for gradually refining the test model.

5.4 Gradual Refinement Might be the Key

The overall aim we see for MBT approaches is that the test adapter has no further behavior than delegating information from the test cases to the implementation and vice versa. In an ideal world, any human activities would be necessary to execute the test cases. This, of course, can only be achieved if the test model contains a formal mapping between artifacts on different level of abstractions. We refer to this as *refinement of abstraction levels*. Such a refinement would enable test engineers to still rely on high level test models and to gradually introduce technical details pertinent for test execution. We believe this could be the key for continuous test automation processes. However, tools would have to guide the refinement process to avoid manually introduced errors as much as possible, thus, tools must become more powerful than they are currently are.

5.5 Size of UML State Machines

MBT approaches are often said to lead to unmaintainable behavioral models that result in state space explosion. We counteracted this situation by building rather concise SMs based on logically interrelated test requirements. We believe that understanding and modifying such concise SMs is less demanding for the test engineers. Of course, we considered only parts of the entire DOME functionality, so further research is required for finding out whether more complex systems are also feasible for such a functional decomposition for test generation. As a general lesson learned we can state that big, unmanageable behavioral models for test generation might be avoidable. A behavioral model can be designed to cover only specific parts of the expected functionality of SUT and might be composed to larger pieces of behavior, if needed.

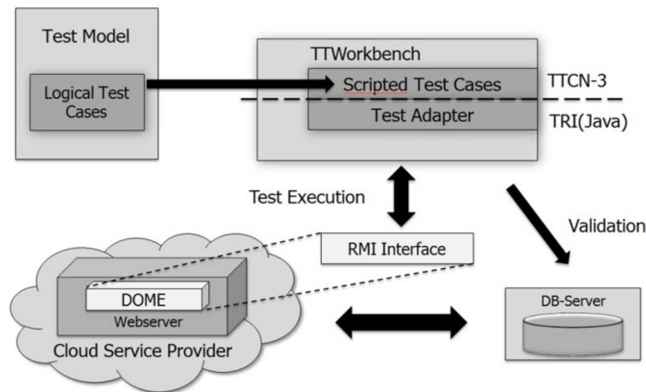

Figure 5: Connection of Test Execution Environment and SUT

6. CONCLUSION

In this paper, we have described a functional system testing approach for safeguarding the model-driven modernization of parts of the DOME case study in the context of REMICS project. We described the principles of our methodology, supported by our academic research tool Fokus!MBT and discussed the effects of decisions being made along the test process on the test design and test execution. Our approach utilizes a central test model, expressed with UTP and UML that contains all relevant information required or produced throughout the test process, ranging from test requirements over test architecture and test basis behavior to test cases. For test execution we employed TTCN-3 and the TTCN-3 test execution system TTworkbench.

Our findings from the DOME case study are summarized in lessons we have learned about both the applicability of our methodology and more general assumptions regarding MBT approaches.

Future research will target in particular the gradual refinement of test models from higher abstraction levels down to the lowest level. The refinement capability needs to be manifested in both the methodology and the tool. Our assumption is that such a mapping, in case it is guided by a mature tool, will help our methodology to become more efficient and to further stress the single source of truth principle.

7. ACKNOWLEDGEMENTS

This work was funded by the EU FP7 project REMICS (project no. 257793).

8. REFERENCES

[1] European Telecommunications Standardisation Institute (ETSI): ES 202 951: Requirements for Modeling Notations. ETSI Standard, Methods for Testing and Specification (MTS); Model-Based Testing (MBT). V1.1.1 (2011-07)

[2] Grieskamp, W., Model-Based Testing in the Field: Lessons Learned, *Lecture Notes in Informatics*, Vol P-94 (2006), Pages 189- 196.

[3] Grieskamp, W. et al., Interaction Coverage Meets Path Coverage by SMT Constraint Solving, *Lecture Notes in Computer Science*, *Vol. 5826*, Pages 97-112.

[4] Grieskamp, W. et al., Generating finite state machines from abstract state machines, in *Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis (ISSTA '02)*, (Rome, IT, 2002), ACM.

[5] Gurevich, Y., Evolving Algebras 1993: Lipari Guide, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[6] IEEE Standards Association (IEEE): 830-1998 – IEEE Recommended Practice for Software Requirements Specifications. http://standards.ieee.org/findstds/standard/830-1998.html

[7] International Software Testing Qualifications Board (ISTQB): ISTQB/GTB standard glossary for testing terms. http://www.software-tester.ch/PDF-Files/CT_Glossar_DE_EN_V21.pdf

[8] Object Management Group (OMG): MDA Guide, version 1.0, Doc: omg/2003-05-01, 2003.

[9] Object Management Group (OMG): UML Testing Profile. URL: http://www.omg.org/spec/UTP

[10] Prenninger W., and Pretschner, A., Abstractions for Model-Based Testing, in *International Workshop on Test and Analysis of Component Based Systems* (TACoS 2004), (Bonn, GER, 2004), Pages 59-7.

[11] Pretschner, A. and Philipps, J., Methodological Issues in Model-Based Testing. in *Model-Based Testing of Reactive Systems*. Springer, 2004, Pages 281-29.

[12] Sadovykh, A. et al, Architecture Driven Modernization in Practice – Study Results: in *14th IEEE International Conference on Engineering of Complex Computer Systems*, (Paris, FR, 2009), Pages 50-57.

[13] Sadovykh, A. et al., On Study Results: Round Trip Engineering of Space Systems. In: *Fifth European European Conference on Model-Driven Architecture Foundations and Applications (ECMDA)*, (Enschede, NL, 2009), Springer.

[14] Stachowiak, H.: Allgemeine Modelltheorie, Springer, Wien, 1973, ISBN-10: 3-211-81106-0.

[15] Utting, M., Pretschner, A. and Legeard, B., A Taxonomy of Model-Based Testing, *Software Testing, Verification and Reliability*, Vol. 22 (5), Pages 297-312