Practical Model-Based Testing of User Scenarios

Sebastian Wieczorek
SAP Research
Software Eng. and Tools
Darmstadt, Germany
sebastian.wieczorek@sap.com

Vitaly Kozyura
SAP Research
Software Eng. and Tools
Darmstadt, Germany
v.kozyura@sap.com

Matthias Schur
SAP Research
Software Eng. and Tools
Darmstadt, Germany
matthias.schur@sap.com

Andreas Roth
SAP Research
Software Eng. and Tools
Darmstadt, Germany
andreas.roth@sap.com

***Abstract*—Increasing the efficiency of testing is a never ending industrial demand. In the software domain it has been tackled with automation of the test execution so far. Automation of the test design, as promised by model-based testing (MBT), is perceived as the next innovative leap. In order to introduce MBT in large-scale development projects, various challenges have to be addressed. Apart from the apparent need for a sophisticated modeling, embedding into established processes and tool chains as well as independence from single tool providers are desired features of an industrial MBT approach. In this paper we describe an MBT approach in the enterprise software domain. Our aim is to give insights not only into the technical details but also into the constraints and necessary considerations of a deployment of MBT in practice, e.g. the embedding into the existing testing process and framework.**

I. Introduction

In the past, a significant effort has been made by the software industry to increase the efficiency of testing. Test automation in this context has been regarded as the most promising way. State of the practice for enterprise software is the automation of the test execution process, while the test design i.e. the definition of abstract test cases and their concretization, is in general still a manual task.

Model-based testing (MBT) is regarded as the next step towards automating the test design and thus received a lot of attention both in the academic as well as the industrial communities recently. In the center of MBT are models, describing the behavior of the system under test (SUT). Mostly this is done directly by utilizing state-based system specifications, opposed to indirect specifications through environment modeling, e.g. user stories.

In the enterprise software domain, behavior modeling is widespread on system level, whereas activity-based modeling notations like BPMN are used extensively to document the software and ease communication with customers. Many companies are therefore interested in leveraging these models in the whole software life-cycle, i.e. during development, operation, and maintenance.

MBT is an integral part of such considerations, as it promises increased efficiency in the system level testing activities. However, commercial tool providers for MBT mostly focus on state-based modeling approaches. This is due to the fact that early industrial adoption of MBT took place in embedded software development like telecom and automotive. As a consequence, model-based test generation based on abstract state machines emerged as the de facto standard for MBT tools [1].

In this paper, we report on an industrial approach to utilize standard state-based MBT tools for system testing of enterprise software by introducing special purpose model transformations. We consider this work as a supporting step into the standardization of model-based techniques by showing how simple intermediate formats can enable seamless integration of common MBT tools even in proprietary test environments. The work has been conducted within SAP, the market leader in enterprise software [2].

The remainder of the paper is structured as follows. In Section II we provide an overview of our approach to model-based system testing. In Section III the architecture of our implementation is shortly described. In Section IV we describe the necessary model transformation for using common MBT tools. Finally, we discuss related work and conclude the paper.

II. Testing Approach

Test automation techniques are the most promising approach for increasing the efficiency of testing. As it was mentioned before, the test design, i.e. the definition of abstract test cases and their concretization, in general remains a manual task.

The transformation from abstract test cases to executable test scripts usually follows the keyword-driven testing principles. Keyword-driven testing uses keywords in the test cases, in addition to data. Each keyword corresponds to a fragment of a test script (the adapter code), which allows the test execution tool to translate a sequence of keywords and data values into executable tests [3].

For enterprise applications, the most important objective of testing is to gain confidence over the executability of the supported business processes. While functional errors may only affect a specific department of a customer, the general inability to run a business process will most likely have a severe impact. Therefore system testing is the predominant quality assurance activity in enterprise application

development. In practice, system testing is based on high-level usage scenarios and business requirements that have been defined by business analysts or customers. UI-based testing is most appropriate to carry out the tests, as the system should be validated as a whole, only using access points that are available to the users.

Keyword-based testing for UI is mostly done by utilizing capture/replay functionality, which is provided by standard test automation tools. These tools are monitoring user interactions on the interface that can reproduce the execution of the recorded sequence of events. These captured scripts commonly allow data flexibility by exchanging the concrete values (used during capturing) with variables that can be initialized independently. Further, the recorded scripts can be combined in so-called scenarios. A scenario is a sequence of recorded scripts working on a predefined set of data. Usually global variables are used in scenarios to organize the data flow. Their function is to store output values of a captured script and make it available as input for another.

The testing approach we introduce in this paper leverages a keyword-driven testing framework as described above. For details refer to [4]. On top we implemented a model editor to facilitate automated test generation. According to the taxonomy of Utting et al. [5], the model which is in the center of the MBT approach describes environment behavior, i.e. the users' interaction with the system in contrast to the systems' implemented routines. In more detail, the Test Model (TM) is an activity-based model, containing start and end points and a number of activities. An activity corresponds to a step in the business process that should be supported by the SUT. Each activity is linked to a recorded script and contains additional information necessary for test generation:

- data-flow information as described above
- a guard condition to specify constraints for the execution based on global variables
- a side effect for modifying global variables in case of activity execution

In order to be independent of a specific tool vendor, we integrated a general test generation approach. In contrast to our activity-based modeling language, most test generators rely on abstract state machine notations similar to ASM [6]. In more detail, test generators like SpecExplorer, Conformiq, and the ProB-Test-Generator [7], [8], [9] rely on input languages that include a general definition of the state space and actions, consisting of preconditions and effects. Preconditions restrict the state space in which an action is enabled. Effects describe how the current state is modified when the action is executed.

III. Architecture

In this section we give details on our architecture which is supporting the utilization of common MBT tools in a proprietary test environment. Figure 1 illustrates the main

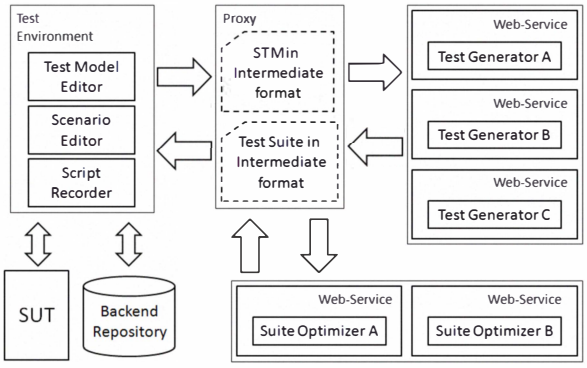

Figure 1. Architecture of the MBT environment.

blocks of this architecture. As described in the previous sections the *Test Model Editor* allows the creation and editing of activity-based test models, triggering test generation and visualization of the resulting test suite. It is embedded into a proprietary *Test Environment*. The *Test Environment* offers UI-based keyword-driven testing capabilities through a *Scenario Editor*, which allows to assemble captured test scripts and to visualize the generated executable scenarios (obtained from test cases). The scripts can be recorded through the *Script Recorder* component, which is connected to the SUT for this purpose. Besides the capturing of user interactions on the SUT, the *Script Recorder* offers replay functionality, which is also utilized for the stepwise execution of scenarios.

TM can be converted into an state transition machine (STM) according to given coverage criteria as described in Section IV. These model transformations are implemented on the client side, i.e. in the *Test Environment* and utilize an intermediate format for STMs. The intermediate format was defined general enough, such that it can be transformed to the concrete STM input languages in the web-services hosting the different test generators. A proxy is set up for routing the test generation requests in order to obtain a single communication partner, which allows to add and update generator components without additional configuration of the test environment.

General-purpose MBT tools rely on varying strategies to reduce the large initial test suites they produce during test generation. Therefore we decided to offer unified test suite optimization independent of the chosen test generator. This allows us to consider custom requirements for the enterprise software domain, as described in [9]. After the test generation succeeded, the resulting traces are transformed into an intermediate test suite format and sent to the proxy component, which forwards it to an appropriate *Suite Optimizer*.

The test reduction is implemented on the intermediate format for test suites. Therefore a further transformation of the results in the *Suite Optimizer* is not necessary. The proxy

takes the reduced test suite and routes it back to the *Test Model Editor* where it will be used to create the concrete test suite, containing executable scenarios.

Because the test environment is relying on a detached backend repository for the storage of test artifacts, we were able to implement the *Test Model Editor* as a web-based tool, running in standard browsers. Having such a browser-based modeling environment and a service-based test generation brings the following advantages:

- Usability: Utilizing a generic input and output format, we are able to hide the complexity of the specific model transformations into the input format of concrete test generators, thus making MBT accessible as a service to other test environments.
- Reusability: Having a browser-based front end promises a light-weight integration of the *Test Model Editor* into other testing environments as long as it can be connected via back-end services.
- Performance: Decoupling computationally expensive functionality like test generation and test suite reduction promises better system performance and does not block front-end users. Replication of the web-services and the introduction of load balancing to the proxy increases scalability.
- Maintenance: The service-based decoupling in combination with a proxy allows to maintain and upgrade test generation components in a non-intrusive way.

IV. Transformations

As explained in the previous sections, the main motivation for transforming an activity-based TM into a state-based STM is to enable the usage of common MBT tools for test generation. In this section we describe different transformation strategies from a TM into a STM. This becomes necessary, because we want to leverage standard action coverage of STM-based test generators for various coverage goals on activity-based TM. As defined in Section II the term action refers to abstract transitions, i.e. defined on abstract states. Our concrete implementation is based on SAP's proprietary modeling languages [10]. Because we cannot introduce them here, we will describe the core concepts of our transformations utilizing a pseudo modeling language instead.

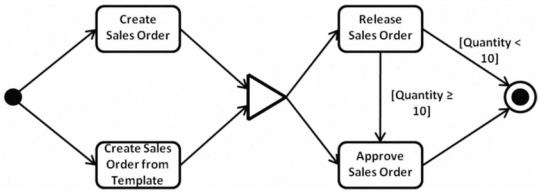

Figure 2. Activity diagram of the Sales Order example

The example TM depicted in Figure 2 will be used for illustrating the various transformation descriptions in this section. The activity diagram describes a sales order process. The diagram contains start and end points, activities (rounded rectangles) and a hub element (triangle) as well as connecting flows (arrows). The hub element is a graphical shortcut for a connection of each input element with each output element. Further, diamond-shaped conditional elements are supported by our editor, which can be used similarly to hubs. However they allow to specify guards on each outgoing flow. Their transformation will be discussed in a separate example.

The behavior described in the example is interpreted in the following way: The SUT supports two ways of creating a sales order, either from scratch or by utilizing a template. After the sales order is created, the user is able to choose either to release the sales order or to explicitly approve it. While released sales order with a quantity ≥ 10 must be approved after releasing it, for the others this activity is optional.

If the test suite generated from the TM covers every activity in the TM at least in one scenario, we have complete *activity-coverage*. Analogous, *flow-coverage* means that the test suite covers all flows of the TM at least once. As the hub element is a graphical shortcut only, we attempted to resolve it during test transformation, but the resulting suite appeared to contain redundancy for users. Therefore we introduced *extended-flow-coverage*, as each flow of the TM with resolved hub elements being covered at least once. For the given example, the minimal test suite for *flow-coverage* contains 2 tests, while the suite for *extended-flow-coverage* contains at least 4 tests.

As described above, our approach assumes that each test generator supports *action-coverage* of an STM, which means that generated test suites visit each reachable action in at least one test case. In order to utilize such a test generation algorithm for the three defined coverage criteria, we illustrate three different translations from activity diagrams to STMs using the example from Figure 2.

Activity-Coverage: The transformation, which is necessary to utilize *action-coverage* on the resulting STM to simulate *activity-coverage* on the TM, works as follows. We define a state type t_s containing the elements $start$ and end as well as $s_1..s_n$, where n is the number of activities in the TM. Furthermore, we define a variable $state$ of type t_s and initialize it with $start$. Each activity is translated into one action, which sets $state$ to one distinct value of t_s. The precondition of each action is expressed as a guard over $state$ and enforces that the action is only executable, if $state$ holds the value of an associated enabling activity. Further an *End* action is introduced, which is enabled in states related to final activities in the TM. The actions *Approve Sales Order* and *End* have additional guard conditions: $Quantity \geq 10$ and $Quantity < 10$.

Figure 3 illustrates the result of this transformation for the given TM. For example the *Release Sales Order* activity

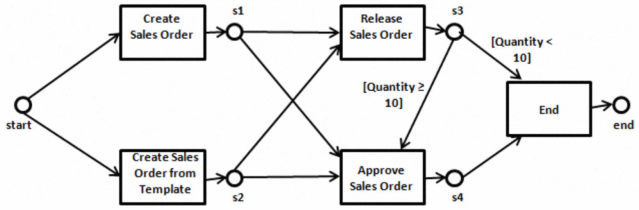

Figure 3. Translation of Sales Order example in order to use the activity-coverage criteria

has been translated to an action with identical label, which is enabled when $state$ is either assigned to s_1 (indicating that *Create Sales Order* has been executed) or s_2 (indicating preceding execution of *Create Sales Order from Template*) and results in assigning $state$ to s_3.

As each activity of the TM is translated to exactly one action in STM, *action-coverage* of the STM results in the desired *activity-coverage* for the TM. For the STM in Figure 3 the obtained test suite is given below:

I Create Sales Order, Release Sales Order.
II Create Sales Order from Template, Approve Sales Order.

Flow-Coverage: Figure 4 illustrates the result of the transformation of the example using the *flow-coverage* criteria. Like for *activity-coverage*, we define a state type t_s containing the elements $start$, end, $s_1..s_n$, but in this case the actions correspond to the arrows (flows) and the states to the modeling elements of the diagram in Figure 2. For example, the action *Start To Create* corresponds to the arrow from *start* to *Create Sales Order* and the states s_1 and s_3 correspond to the activity *Create Sales Order* and the *Hub*-element.

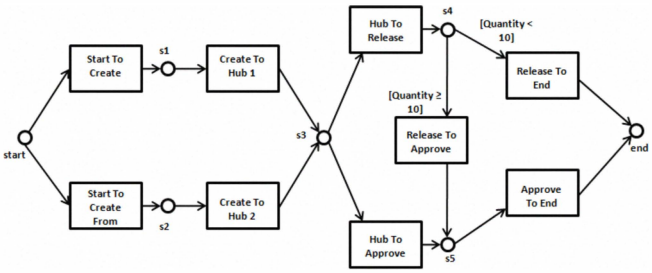

Figure 4. Translation of Sales Order example in order to use the flow-coverage criteria

The obtained *action-coverage* in STM therefore simulates the *flow-coverage* of the original TM. The test cases can be translated back to the TM using the fact that each action relates to a flow in TM connecting two activities or an activity and *start* or *end*. For the STM in Figure 4 the obtained test suite is given below:

I Create Sales Order, Release Sales Order.
II Create Sales Order from Template, Approve Sales Order.
III Create Sales Order, Release Sales Order, Approve Sales Order.

Extended-Flow-Coverage: Figure 5 illustrates the result of the transformation of the example using the *extended-flow-coverage* criteria. The actions in STM also correspond to the arrows (flows) and the states to the modeling elements of TM, but in this case the hub-elements are resolved and the obtained arrows are translated instead. As shown in Figure 5 this results in six states instead of the seven states for *flow-coverage* in Figure 4.

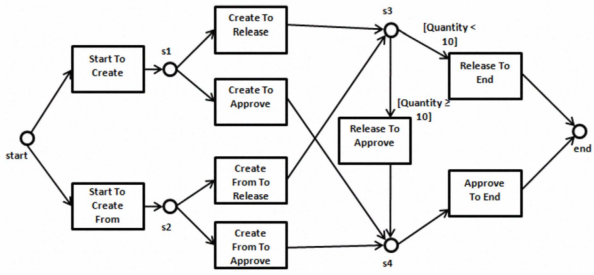

Figure 5. Translation of Sales Order example in order to use the extended-flow-coverage criteria

The hub-element is not relevant for the test suite of the TM, because it does not relate to any executable activity. This means that the translation of generated test cases back to the TM is similar to the one used in the case of *flow-coverage*. For the STM in Figure 5 the obtained test suite is given below:

I Create Sales Order, Release Sales Order.
II Create Sales Order, Approve Sales Order.
III Create Sales Order from Template, Approve Sales Order.
IV Create Sales Order from Template, Release Sales Order.
V Create Sales Order, Release Sales Order, Approve Sales Order.

Note that it is possible to use *flow-coverage* or even *extended-flow-coverage* transformations in order to obtain a test suite which realizes *activity-coverage*. However, for minimizing test effort the suite must then be reduced after test case generation in order to eliminate residual test cases. As described before, test suite optimization usually has to be performed anyway (e.g. when lots of test cases are generated because of TMs containing loops in combination with guards) but tend to introduce performance issues in larger models. Further, also the test generation itself is slower for *flow-coverage* compared to *action-coverage*.

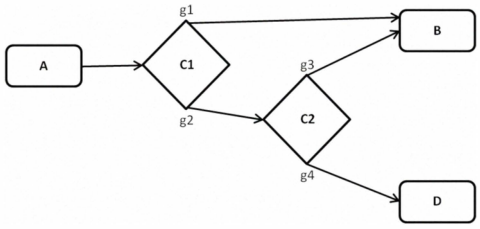

Figure 6. Conditional elements of activity diagrams

Conditional Element: in the remainder of this section we will discuss the translation of the conditional element. Let us therefore consider another example, depicted in Figure 6. It contains two conditional elements, $C1$ and $C2$, both having two output flows with the corresponding guard conditions g_i.

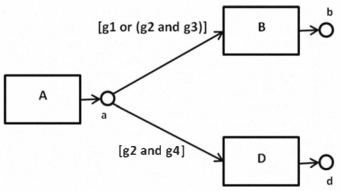

Figure 7. Resolving conditional elements

For calculating *activity-coverage*, the conditional elements could be resolved similar to hub elements. The resulting STM fragment for the given example is depicted in Figure 7. However we do not advice such an approach, because deriving traces of generated test cases on the original TM (e.g. for highlighting) is much easier when preserving the original structure of the model. Further, the debugging of the resulting STM becomes much more convenient. From a theoretical standpoint, also for *extended-flow-coverage* the conditional elements should be resolved similar to hub elements. However in practice conditional elements are seldomly used as graphical shortcuts but for explicitly visualizing a conditional choice after the execution of an activity. Therefore our users do not expect that *extended-flow-coverage* might result in a smaller test suite than *flow-coverage*.

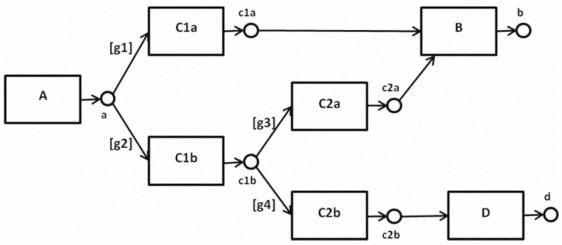

Figure 8. Translation of conditional elements

As a consequence we decided that our transformations preserve each conditional element. This means that for conditional elements the translation rule is the same for *activity-coverage*, *flow-coverage* and *extended-flow-coverage*. Basically, for each branch of the conditional element a separate action and an additional element for the state type is created in the STM. The guard of the action contains the condition of the associated branch in addition to the description of enabling states. As depicted in Figure 8, the branch to activity B at the conditional element C_2 in Figure 6 is translated to action C_2a. It is enabled, if g_3 is evaluated $true$ and $state$ is assigned to c_1b, the state element assigned with the branch of conditional element C_1, which leads to C_2. After the execution of action C_2a, $state$ is set to the associated state element c_2a.

V. Related and Future Work

The motivation for this paper is to describe an industrial application of model-based testing. Therefore we do not intend to distinguish ourselves from academic work in terms of novelty. Instead, we will discuss related work that influenced our design decisions or offers interesting perspectives for future enhancement of our framework.

Over recent years, model-based testing has gained more and more momentum. Various companies such as Smartesting[1] or Conformiq[2] are offering mature MBT tools. Further, the large software vendors Microsoft and IBM developed MBT solutions [7], [11]. Also the MBT customer base increased consistently over the past and meets in a dedicated MBT User Conference[3]. In order to increase interoperability between MBT tools, various new standards are under way [1], [12]. This work describes the industrial perspective of an MBT user in the enterprise software domain. We further hope that, by giving evidence for the interoperability we leveraged, we support the started standardization effort.

In this paper, we concretely discussed model transformations towards suitable input formats for test generators. Most work on test transformations has been carried out on code level. Such approaches derive control-flow graphs, which are then used for verification and test generation [13]. In a way we attempt something similar, as we transform formal behavior specifications (on a higher abstraction level) such that they can be interpreted by test generators. In academia, additional test transformations have been discussed to increase the efficiency of test generation tools, which rely on heuristic searches for deriving test cases, e.g. in case of platoons in the control-flow graph [14]. Whether such techniques are relevant for our work needs further investigation. On the modeling level, transformations towards input formats for MBT tools have been proposed by various research groups, e.g. from UML system models [15]. Special purpose transformations to simulate advanced coverage criteria on state machines with basic MBT tools, offering transition coverage only, is proposed by [16]. Even though our work is based on the same idea, it was impossible to use the cited results directly, because our testing approach is demanding action coverage on abstract state machines instead.

In Section II we mentioned that the test generation activity is followed by a test suite reduction in our MBT approach and motivated, why we do not rely on built-in algorithms of the tool vendors. Even though this reduction is not a direct contribution of this paper, we would like to provide some

[1] www.smartesting.com
[2] www.conformiq.com
[3] http://www.model-based-testing.de/mbtuc11

pointers to this topic. Most test generators rely on some heuristics (e.g. embedded in model checkers) or exhaustively search the state space until they produced a test suite that covers all desired model properties. Usually this test suite contains a subset of tests, which still preserves the targeted coverage but further considers objectives like the minimization of test steps in the test suite [17]. For single objective optimization, branch and bound algorithms are known to perform well, while for multi-objective optimization Pareto algorithms seem promising [18]. We will continue our investigation on this topic and hopefully are able to report on the applicability of the existing academic approaches.

VI. Conclusion

In this paper we introduced an industrial approach to MBT and described various practical aspects like the embedding into the general testing process, the design decisions for our tool chain and the architecture. Further we gave details about model transformations that are necessary to utilize common MBT tools. The described artifacts have been implemented and integrated as described and are currently evaluated for productive use inside SAP.

Due to the limited size, we were not able to discuss various aspects of our work in detail but we described its general directions. Apart from the test suite reduction that we sketched in Section V, we also already investigated into test data provisioning and inclusion of product-line concepts into the test generation. From the technical perspective, we plan to enhance our approach with test generation for hierarchical models.

Acknowledgment

This work was partially supported by the EU-FP7 project Deploy[4] (EC-grant no. 214158).

References

[1] ETSI-MTS-MBT, "Final draft of MBT ES on requirements for modeling notations, version 0.6.1," http://docbox.etsi.org/MTS/MTS/05-CONTRIBUTIONS/2011/MTS, 2011.

[2] R. Hayen, *SAP R/3 Enterprise Software: An Introduction*. McGraw-Hill/Irwin, 2006.

[3] M. Utting and B. Legeard, *Practical model-based testing, a tools approach*. Morgan Kaufmann, 2007.

[4] S. Wieczorek and A. Stefanescu, "Improving Testing of Enterprise Systems by Model-Based Testing on Graphical User Interfaces," in *2010 17th IEEE International Conference and Workshops on Engineering of Computer-Based Systems*. IEEE, 2010, pp. 352–357.

[5] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing," Department of Computer Science, The University of Waikato (New Zeeland), Tech. Rep. 04/2006, 2006.

[6] E. Börger, "The Abstract State Machines method for high-level system design and analysis," *Formal Methods: State of the Art and New Directions*, pp. 79–116, 2010.

[7] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson, "Model-based testing of object-oriented reactive systems with Spec Explorer," *Formal Methods and Testing*, pp. 39–76, 2008.

[8] A. Huima, "Implementing Conformiq Qtronic," *Testing of Software and Communicating Systems*, pp. 1–12, 2007.

[9] S. Wieczorek, V. Kozyura, A. Roth, M. Leuschel, J. Bendisposto, D. Plagge, and I. Schieferdecker, "Applying model checking to generate model-based integration tests from choreography models," in *Proc. of the 21st IFIP Int. Conf. on Testing of Communicating Systems (TESTCOM'09)*, ser. LNCS, vol. 5826. Springer, 2009, pp. 179–194.

[10] S. Katker and S. Patig, "Model-driven development of service-oriented business application systems," in *Business Services: Konzepte, Technologien, Anwendungen*. Oesterreichische Computer Gesellschaft, 2009, vol. Band 1, pp. 171–180.

[11] A. Stefanescu, S. Wieczorek, and A. Kirshin, "MBT4Chor: A model-based testing approach for service choreographies," in *European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'09)*, ser. LNCS, vol. 5562. Springer, 2009, pp. 313–324.

[12] OMG, "UML 2.0 testing profile, final adopted specification," http://www.omg.org/spec/UTP/1.0, 2005.

[13] S. R. Vergilio, J. C. Maldonado, and M. Jino, "Constraint based criteria: An approach for test case selection in the structural testing," *Journal of Electronic Testing*, vol. 17, no. 2, pp. 175–183, 2001.

[14] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper, "Testability transformation," *IEEE Transactions on Software Engineering*, vol. 30, no. 1, pp. 3–16, 2004.

[15] M. Vieira, J. Leduc, B. Hasling, R. Subramanyan, and J. Kazmeier, "Automation of GUI testing using a model-driven approach," in *AST '06: Proceedings of the 2006 international workshop on Automation of software test*. New York, NY, USA: ACM, 2006, pp. 9–14.

[16] S. Weißleder, "Semantic-preserving test model transformations for interchangeable coverage criteria," in *Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme*, 2009, pp. 26–35.

[17] G. Rothermel, M. Harrold, J. Von Ronne, and C. Hong, "Empirical studies of test-suite reduction," *Software Testing, Verification and Reliability*, vol. 12, no. 4, pp. 219–249, 2002.

[18] S. Yoo and M. Harman, "Pareto efficient multi-objective test case selection," in *Proceedings of the 2007 international symposium on Software testing and analysis*. ACM, 2007, pp. 140–150.

[4] http://www.deploy-project.eu