Model-Driven Test Case Construction by Domain Experts

Stefan Baerisch[1,2] and Wilhelm Hasselbring[2]
[1]GESIS – German Social Science Infrastructure Services, 53113 Bonn, Germany
stefan.baerisch@gesis.org
[2]University of Oldenburg, Software Engineering Group, 26111 Oldenburg, Germany
Hasselbring@informatik.uni-oldenburg.de

April 29, 2008

Abstract

Model-driven construction of system level tests should involve domain experts. Our Model-driven Test Case Construction (MTCC) approach employs feature models and protocol state machines to represent systems under test (SUT) as well as the actions and assertions used to execute a SUT. An editor is instantiated from models of SUTs and actions and assertions for the SUT. This editor supports the construction of test models by domain experts without specific modeling knowledge. Reuse of test models for different systems within a system family is supported by comparing the actions, assertions and services available for one SUT with those of another SUT. The MTCC approach is implemented and evaluated at the GESIS institute in the application domain of scientific information retrieval systems.

1 Introduction

MTCC integrates model-driven testing with acceptance testing. Models of the test-relevant properties of a SUT are employed to support domain experts in the construction of test models. This approach involves domain experts into the testing process and supports model-driven testing in scenarios where it could otherwise not be applied in practice.

In order to identify defects, tests have to verify the correctness of the behavior of a system implementation according to a specification. The testing process must be both efficient and effective and the specification from which the tests are derived must represent the requirements of a system. Model-Driven Testing is potentially both effective and efficient. High test coverage can be achieved if formal models of sufficient detail for both the SUT and its environment are provided. Since test code is generated from models and can be executed without human interaction, testing is also potentially very effective in terms of the invested resources.

The availability of high quality models suited for automatic (test-) code generation is not given for all software development projects or organizations. The requirements for

such models are high: The models have to be testable [6] and have to correctly represent the requirements of the system as defined by experts for the application domain. This latter point must be emphasized since domain expert will often not have the skills needed for formal modeling. Close cooperation between modelers and domain experts is thus necessary [11].

The MTCC approach presented in this work addresses the issues outlined above as follows: MTCC does not assume the existence of detailed design models for SUT. The scope of the models used is limited to the description of the test-relevant features and test scenarios for the system family of the SUT. MTCC does not derive tests or oracles directly from models. Domain experts use a graphical editor based on the models to construct models of specific tests (`TestConfigurations`). `TestConfigurations` are used either for the generation of test code or are interpreted by a test runner. In order to facilitate the reuse of `TestConfigurations` for different SUTs and thus minimize the effort required from domain experts for modeling, MTCC addresses reuse of `TestConfigurations`. The variability of different SUTs within a system family [18] is described in feature models [10] and the features required to executed a `TestConfiguration` are compared with the features provided by a SUT.

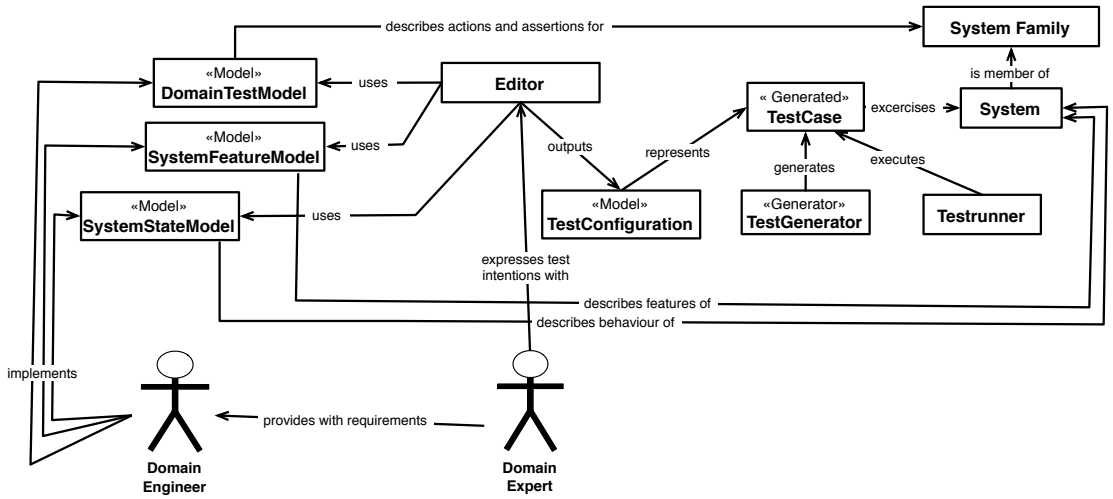

Figure 1: Overview of the MTCC testing process for a single SUT

MTCC combines manual test specification by domain experts with model driven techniques. Domain expert define tests and test success criteria while models are utilized to (1) support test construction via the domain expert's editor, (2) serve as an implementation independent representation of tests, and (3) allow the comparison of the features and behavior of different SUTs within a system family in order to support the reuse of compatible tests. We argue that MTCC allows model-driven testing in environments where it would otherwise not be suitable due to lack of resources or knowledge to create sufficiently detailed, formal and complete models. We also argue that MTCC is well suited to involve domain experts in testing. Figure 1 gives an overview of the testing process in MTCC and the artifacts and roles involved.

The test-relevant features of a SUT are represented as feature models in the `SystemFeatureModel`. The `SystemStateModel` describes possible traces of system execution

as a protocol state machine [19]. The `DomainTestModel` describes the *test steps* available to execute a SUT. Each test step is a model of either an action or an assertion. The specification of the `SystemFeatureModel`, the `SystemStateModel`, and the `DomainTestModel` requires modeling knowledge. It is thus assumed that the models are constructed by a domain engineer who has the required modeling skills. The domain engineer is supported by a domain expert. The construction of the `TestCases` is done by a domain expert using the graphical editor. The editor is instantiated based on compositions of the `SystemFeatureModel`, `SystemStateModel`, and `DomainTestModel`. The editor supports the domain expert in the definition of `TestConfiguration` support. In order to execute the test described by a `TestConfiguration`, the `TestConfiguration` is either transformed into an executable test case for a Commercial off-the-shelf (COTS) testing harness or is interpreted by a custom testing harness.

Our current implementation of MTCC was realized for the domain of scientific information retrieval portals, MTCC is evaluated for the SOWIPORT[1] portal of the GESIS and the IBLK portal[2] as well as for the back-end search engine the both portals. A core function of scientific information portals like SOWIPORT is information retrieval. The portals allow their users to search for scientific information, for example for research projects and for publications. Since the portals are information retrieval systems [3], their ability to return relevant documents for specific information needs is an important part of their functionality [23] and must thus be tested. Such testing requires the relevance assessments of a human domain expert.

Regarding their implementation technology and the underlying development process, the portals are web-based applications that are under continuous development. Since releases are frequent, testing has to occur often and must be automated to be efficient. When test automation is employed in this scenario, the automated tests must be maintained in parallel to the system and must reflect all changes that are relevant to the execution and the result of the test. We argue that models as used in MTCC decrease the effort necessary for maintenance. `TestConfigurations` used to express tests are decoupled from the implementation of the SUT, maintenance of test execution does not affect existing test models. The use of separate models for the SUT and for tests on the SUT allows to determine which test are affected by changes to the SUT and thus support the systematic maintenance of tests.

This paper is structured as follows: Section 2 discusses related work. Section 3 introduces the MTCC testing process, the models used in the process and the editing environment used for test construction by domain experts. Section 4 gives information about the current evaluation of MTCC at GESIS. Portals for scientific information as the application domain as well as the goals of the validation and preliminary results are discussed. Section 5 concludes the paper.

2 Related Work

MTCC is related to the fields of model-driven testing and model-driven engineering as well as acceptance test automation. The reuse of tests for different systems within a system family and the utilization of feature models are also relevant fields for MTCC.

[1] http://sowiport.de
[2] http://fachportal-iblk.de

The use of models to automatically test systems is a well established practice [6] in software engineering. Today, model-driven testing is a field which subsumes a number of different approaches [24]. Models used for testing include requirements specifications [13, 25] as well as structural and behavioral design models [21, 7]. MTCC is an approach that utilizes test-specific models to allow domain experts the specification of interactions with a SUT and is thus related to approaches that base testing on use cases or scenarios [20]. Since MTCC takes the perspective of a user of a system and thus describes tests in terms of interactions with the user interface, model-driven testing for GUIs ist also relevant [14, 2].

MTCC is related to the field of automated acceptance testing. Since the testing using capture-replay is fragile to changes of the SUT [16], current approaches abstract the GUI. One such approach is to express tests using the spreadsheet paradigm [17, 1] or as *Keyword* and *Action Words* [12]. MTCC is similar to these approches in the abstraction provided from the GUI and providing a way for non-developers to specify tests, MTCC differs in the systematic use of models to abstract both the SUT and the tests.

MTCC considers SUTs as members of a system family [18] or application domain [8]. The variants of test-relevant features of SUTs described as feature models [10, 9]. MTCC uses the feature models to identify test models that are reusable for different systems, it thus shares ideas with system family testing [22, 15].

3 Model-Driven Test Case Construction

The MTCC process can be subdivided in four steps, (1) the modeling of one or multiple SUTs as well as the actions and assertions that are relevant for the system family of the SUT, (2) the instantiation of an editor based on a composition of these models, (3) the use of the editor by domain experts to construct models of tests in form of `TestConfigurations` and (4) the interpretation of executions of the `TestConfigurations`. This section gives an overview of the steps and the relevant artifacts. Examples are taken from the application domain of scientific information portals.

In the first step, software engineers and domain experts analyse the test-relevant features of systems in the system family. Actions and assertions that execute identified features are expressed as *test steps* in the `DomainTestModel`. When the initial domain analysis is finished, the individual SUTs are modeled. Each system is represented by two models, the `SystemFeatureModel` is a specialization [9] of the `DomainFeatureModel`, the `SystemStateModel` is a protocol state machine that describes traces through the SUT.

A SUT in the `SystemStateModel` is represented as a number of *contexts*. While testing via the user interface, each context corresponds to a GUI screen or a page in a web application. Each context is a collection of *services*. A service is a system family-specific part of the interface of the SUT that can be executed by a test. Possible traces through the system are described by the `SystemStateModel`. Both contexts and services are expressed as states. A tests is executed by first entering the start context, it then exercises one of the services by entering a successor state for the context. The successors states of the exercised service are contexts. A service can have more than one context as a successor state, for example one state representing failures of service invocations and

one state for the success case.

Figure 2 shows the state model of a simple SUT. A test exercising the SUT would start in the `Search` context. `Search` only provides one service, `SearchForm 1`, this service sends a query to the information retrieval system and, on successful completion, enters the `Overview` context. From the `Overview` context, a test could for example inspect the list of result documents (`DocumentList 1`), could open the detail display for a document (`DocumentLink 1`) or could return to the `Search` context (`Link 1`). Each of the services in the `SystemStateModel` can have a different set of features, these features are described in the `SystemFeatureModel`.

A `DomainTestStep` model represents an action or assertion that execute or verify a specific kind of service. Each model includes of checks used to verify whether a particular service instance is compatible with the test step, references to the feature model of the service and information about the available configurations of the DomainTestStep. The `DomainTestStep`, `SystemStateModel` and the `SystemFeatureModel` are used by the editor to present a domain expert with the possible actions and assertions to test the services in the SUT. In order to allow the specification of tests for specific service instances, all test steps from the `DomainStepModel` are checked against the service description in the `SystemFeatureModel`. If the service referenced in the test step is of the same type as the service description from the `SystemFeatureModel`, and if all checks and references from the test step model can be satisfied by the service model, the test step becomes available as an action or assertion for this service. The available parametrization of the test step model is determined by the feature model of the test step and by referenced parts of the feature model for the service.

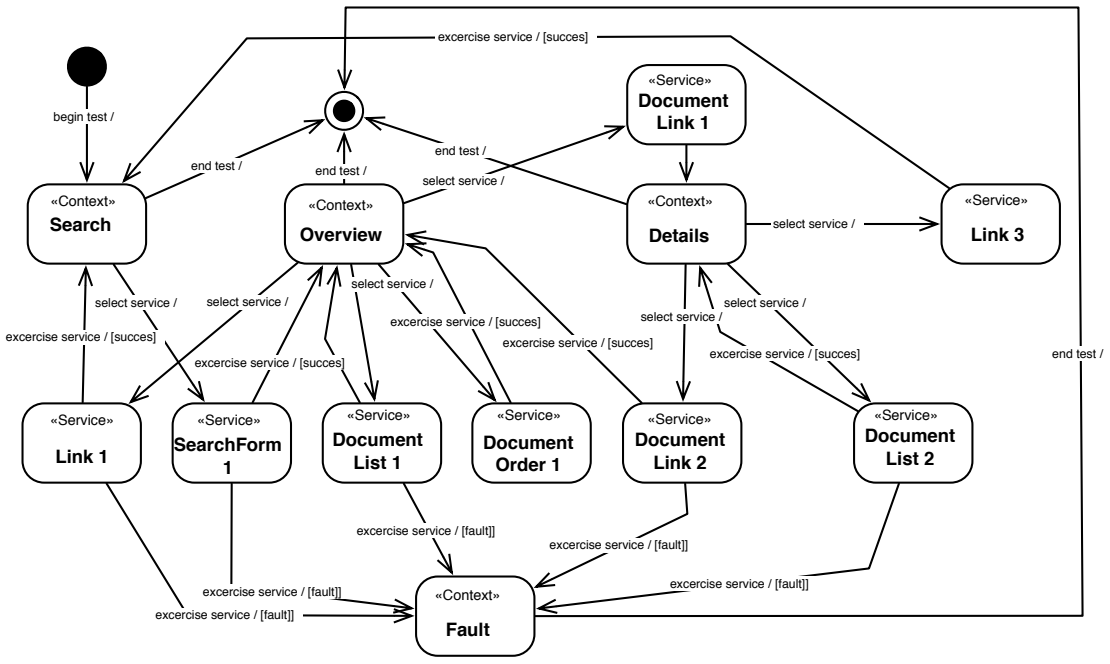

Figure 2: MTCC State model

The editor (1) has to guide the construction of tests based on the behavior of the SUT and (2) has to represent the available parameterization for each test step that is used to execute a service. Users can construct sequences of test steps for the SUT by

selecting test steps from a list of possible successors for the currently selected test step as determined by composition of the `SystemStateModel` with the `DomainStepModel`. The available parameterization for each test step is represented by its feature model. When values for the test step are set in the editor, the corresponding configurations are applied to the underlying feature model. To allow the serialization of test models and to support the identification of tests reusable for other SUTs, configurations are not immediately applied to the feature model but are stored as configuration objects. In order to construct the GUI for each test step, the editor traverses the feature model of that test step and identifies GUI elements for each node. The relationship between the feature model and the GUI as illustrated in Figure 3 is independent from the application domain of the SUT. The controls used in the editor are determined by the structure of the feature model. The tree structure of the feature model is traversed top-down. Each node and potentially its children are checked against a set of rules. The first rule that matches determines the GUI control, for example a checkbox, a radio box or a text entry area, that is used to represent the node in the editor.

Since in our evaluation this technique did not lead to satisfying results in all cases, some special transformation rules were implemented for features specific to the application domain. One example for such a special case is the control for complex search forms that allows the use of predicate logic to formulate queries for multiple fields as shown in Figure 3. This approach allows to hide parts of the feature model that cannot be configured, such as the `FieldRelations` feature in the figure. Event handlers are registered for each node that translates user interactions with a GUI element into configuration actions on the feature model. When the user has constructed a test model by defining a sequence of test steps, the test model is saved as a `TestConfiguration` that can either be executed or used to generate executable tests.

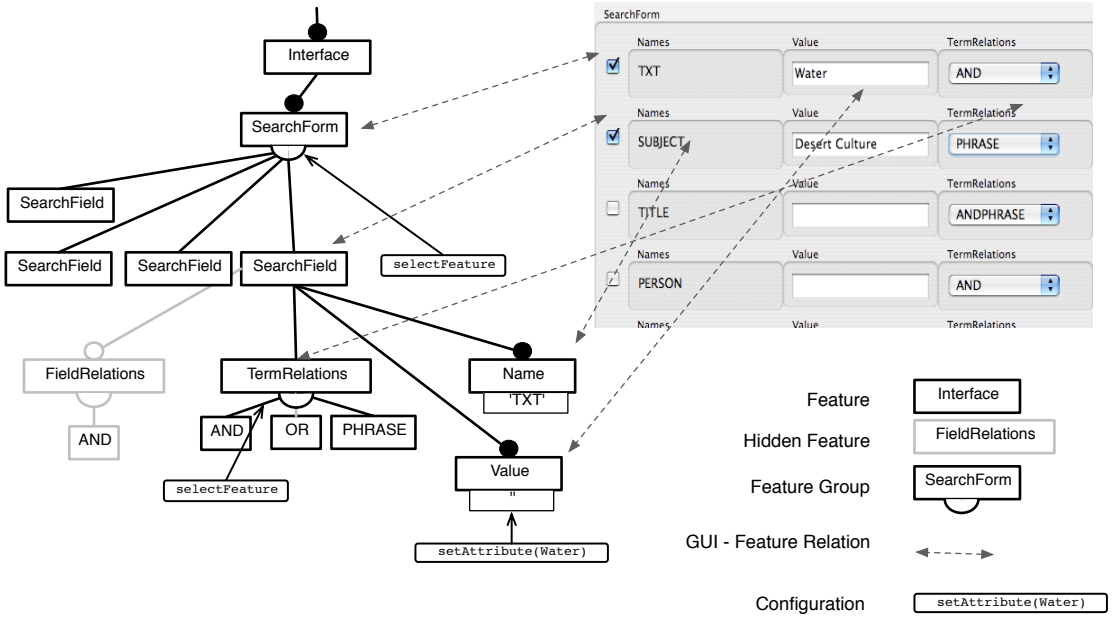

Figure 3: Feature model of a test step and its representation in the MTCC editor

MTCC does not assume a specific mode of test execution, nor does the approach include a test runner. A method of test execution for the MTCC `TestConfigurations` is

selected based on the specific interfaces of the current SUT. Whether test configurations are interpreted or whether code generation techniques are better suited of test execution depends on the specific circumstances of a testing project. The structure of the MTCC test models, a sequence of actions and assertions, is compatible to format in which tests for other acceptance test frameworks are defined. Test frameworks such as FIT [17] or Selenium[3] can thus be used to execute MTCC test models by using a wrapper library. The resources required to allow the execution of tests for a specific SUT depend on the number of services for the SUT and the number of test steps that execute these services. The code to allow the execution of each test step has to be implemented separately for each SUT. We argue that this approach is useful since the infrastructure to execute tests has only to be implemented once for a SUT.

In the current version of MTCC, reuse of `TestConfigurations` for different SUT in a system family is achieved by identifying those test configurations of a SUT that are compatible with the SystemStateModel and the SystemFeatureModel of another SUT. The Identification of such reusable `TestConfigurations` is done in two steps: (1) It is determined whether the sequence of test steps defined for SUT A is valid for SUT B and (2) the feature configuration for each test step in the `TestConfiguration` and the feature models of the test steps of SUT B are compared. The first step is done be constructing a tree in which each node is a test step for SUT B and where all traversals from the root of the tree to a leaf yield the same sequence of test steps as in the TestConfiguration for SUT A. In all situations where this tree has more than one leaf node, the user selects the sequence of test steps for SUT B that best reflects the intended behavior. Once a sequence of test steps is selected, the compatibility of the feature model of each test step for SUB B with the configuration of the corresponding feature model in SUT A is verified by applying the existing configuration objects from SUT A to the nodes of the feature model starting from the root.

4 Evaluation

MTCC is currently under evaluation at the GESIS institute for a system family of three information retrieval systems: two scientific information portals and the search engine with the transformation services that are used by these two portals. The portals were selected for the validation since they share enough properties to motivate the reuse of tests, but also differ sufficiently in their contents as well as in their interface to make reuse of tests nontrivial. The third SUT used in the evaluation is the search engine layer used by both portals, this SUT was included in the validation since it shares the requirements of the portal software but uses an HTTP-based interface instead of a GUI, thus allows the evaluation of the MTCC modeling approach for non-GUI applications.

The technical feasibility as well the the usability of the editor by domain experts are evaluated in type I and type II evaluations, as discussed in [5]. The Goal-Question-Metric (GQM) [4] approach is applied to each evaluation type. Three aspects of the MTCC approach, test modeling, test reuse and test execution, have to be considered in order to evaluate the approach. The type I valuation demonstrates that the MTCC is feasible for testing systems in the considered system family of scientific information retrieval. The type II valuation demonstrates that the approach is practical not only for

[3] http://selenium.openqa.org/

the author but also can be applied by domain experts. The type I evaluation for feasibility of the modeling approach is currently under way and the evaluation of the practicality of the editor by the current testers has started. The evaluation of the test execution and test reuse aspects of the approach will start once the evaluation of the models and the editor is finished.

For the type I evaluation of the modeling aspect of MTCC, we constructed models of the three SUTs described above. We then asked domain experts to document both their current testing practices and to describe test scenarios that they would consider for automation. We collected 40+ scenarios for functional tests for each of the portals and 7 test scenarios to evaluate recall and precision of the information retrieval aspects of the system. Since the evaluation is not yet finished we will only outline our preliminary results.

We can represent more than 80 percent of the test scenarios described by the domain experts using MTCC models. Those test scenarios that cannot be modeled with the current implementation of MTCC fall mostly in two categories: (1) Test scenarios that are specific to functionality of the SUTs that is not covered by models and (2) test scenarios that are not feasible for test automation. The problem of test scenarios that rely on functionality not covered by the MTCC model could in most cases be solved by adding the missing functionality to the model. Those test scenarios that cannot be automated either lack details needed for automation or would require a person to decide on the success of the test. We argue that most of the test scenarios that cannot be automated in their current form cover functionality for which test could be modeled with the editor. This research question will be investigated in the type II evaluation.

5 Conclusions

We presented our approach to model-driven test case construction (MTCC). MTCC allows domain experts the construction of test models that can be used to execute SUTs in the context of system families. MTCC uses models to represent SUTs, actions relevant to test the SUT and the constructed tests. Systems in MTCC are represented as a collection of contexts, each context providing a number of testable services. The types of services are determined by the system family of the SUT, each SUT can contain different instances of these services. Feature models are used to represent variants of services, protocol state machines describe possible traces through a SUT. To allow the construction of tests by domain experts without the need for modeling skills, an editor for tests is instantiated based on the models. An MTCC prototype is implemented and currently evaluated at the GESIS institute with three systems in the domain of scientific information retrieval.

References

[1] Jennitta Andrea. Generative Acceptance Testing for Difficult-to-Test Software. In Jutta Eckstein and Hubert Baumeister, editors, *Extreme Programming and Agile Processes in Software Engineering*, volume 3092 of *Lecture Notes in Computer Science*, pages 29–37. Springer, 2004.

[2] Anneliese Andrews, Jeff Offutt, and Roger Alexander. Testing Web Applications by Modeling with FSMs. *Software Systems and Modeling*, 4(3):326–345, 2005.

[3] Ricardo A. Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[4] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. *Encyclopedia of Software Engineering*, chapter The Goal Question Metric Approach, pages 528–532. John Wiley & Sons, 1994.

[5] Steffen Becker. *Coupled Model Transformations for QoS Enabled Component-Based Software Design*. PhD thesis, Carl von Ossietzky Universität Oldenburg, Oldenburg, Germany, 2008. (submitted).

[6] Robert V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 1999.

[7] Alessandra Cavarra, Charles Crichton, and Jim Davies. A Method for the Automatic Generation of Test Suites from Object Models. *Information & Software Technology*, 46(5):309–314, 2004.

[8] Krysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming. Methods, Tools and Applications*. Addison-Wesley, 2000.

[9] Krzysztof Czarnecki. Overview of Generative Software Development. In J.-P. Banâtre et al:, editor, *Unconventional Programming Paradigms (UPP) 2004*, volume 3566 of *Lecture Notes of Computer Science*, pages 313–328, Mont Saint-Michel, France, 2005. Springer.

[10] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Formalizing Cardinality-based Feature Models and their Specialization. *Software Process Improvement and Practice, special issue of best papers from SPLC04*, 10(1):7 – 29, 2005.

[11] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2004.

[12] Dorothy Graham and Mark Fewster, editors. *Software Test Automation: Effective Use of Test Execution Tools*. Addison-Wesley, 2000.

[13] Javier J. Gutierrez, Maria J. Escalona, Manuel Mejias, and Jesus Torres. Generation of test cases from functional requirements. A survey. Technical report, Department of Computer Languages and Systems, University of Seville, 2006.

[14] A. Kervinen, M. Maunumaa, T. Pääkkönen, and M. Katara. Model-based testing through a GUI. In *Formal Approaches to Testing of Software*, pages 16–31, Edinburgh, Scotland, 2005. Springer.

[15] John D. McGregor, Prakash Sodhani, and Sai Madhavapeddi. Testing Variability in a Software Product Line. In *Proceedings of the International Workshop on Software Product Line Testing (SPLiT 2004)*, 2004.

[16] Gerard Meszaros. *xUnit Test Patterns*. Addison Wesley, 2007.

[17] Rick Mugridge and Ward Cunningham. *FIT for Developing Software. Framework for Integrated Tests*. Prentice Hall PTR, 2005.

[18] David L. Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, 2(1):1 – 9, March 1976.

[19] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual.* Addison Wesley Professional, 2nd edition, July 2004.

[20] Johannes Ryser. *Szenariobasiertes Validieren und Testen von Softwaresystemen*. PhD thesis, Wirtschaftswissenschaftliche Fakultät der Universität Zürich, 2003.

[21] Ina Schieferdecker, Zhen Ru Dai, Jens Grabowski, and A. Rennoch. The UML 2.0 Testing Profile and its relation to TTCN-3. In *TestCom 2003 : testing of communicating systems*, volume 2644 of *Lecture Notes of Computer Science*, pages 79–94, 2003.

[22] Antti Tevanlinna, Juha Taina, and Raine Kauppinen. Product Family Testing: a Survey. *SIGSOFT Softw. Eng. Notes*, 29(2):12–12, 2004.

[23] M. Thurmeier. Erwartungen an wissenschaftliche Fachportale : Ergebnisse einer qualitativen Befragung von Wissenschaftlern. Technical report, in to mind: Institut für Marketingforschung, 2007.

[24] M. Utting, A. Pretschner, and B. Legeard. A Taxonomy of Model-Based Testing. Technical report, Department of Computer Science The University of Waikato, 2005.

[25] Mario Winter. *Qualitätssicherung für objektorientierte Software: Anforderungsermittlung und Test gegen Anforderungsspezifikationen*. PhD thesis, Fernuniversität Hagen, 1999.