Model-Driven Test-Case Construction

Stefan Baerisch
Software Engineering Group
University of Oldenburg, Oldenburg, Germany
and
GESIS / Social Science Information Centre
Bonn, Germany
stefan.baerisch@gesis.org

ABSTRACT

Automatic system tests are frequently coupled to implementation details of the system under test. Such a tight coupling is problematic for a number of reasons: It prevents reuse of existing tests for multiple versions or variants of a system or for a number of systems in a system family and it makes the ability to construct executable tests for a system dependent on programming skills. In this paper we present an approach that decouples test specification and test execution by using system models and test models for the representation of systems under test and tests. We motivate the use of abstract test models and system models, introduce the relevant concepts of our approach and discuss the relationship to relevant fields.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Testing tools

General Terms

Design, Reliability

Keywords

Model-Driven Testing, Test Automation, Acceptance Testing

1. INTRODUCTION

Automatic system tests compare the implementation of system against its requirements. Since many requirements on the system level are not implementation dependent and systems within a system family frequently share requirements, the reuse of tests for multiple systems would be preferable to rewriting the tests. However, this tight coupling of tests with the systems under test (SUTs) and differences in the feature set and interfaces of the SUTs prevent such reuse.

ESEC/FSE'07, September 3–7, 2007, Cavat near Dubrovnik, Croatia.
ACM 978-1-59593-812-1/07/0009.

One method of software quality assurance is the use automatic system tests which compare the behavior of a system under test with its expected behavior. The expected behavior of a system represents functional or non-functional requirements for the system as defined by stakeholders or domain experts. This requirements are identified in an analysis phase and are then documented in an implementation independent way. In order for the requirements to be testable, they have to be encoded in executable software.

We call the information about the requirements encoded in an executable test the *intentions* of a test. Besides the intentions, an automatic system test must also include information necessary for its *execution*. Among this information are details about the implementation of the SUT and the testing harness used. While the combination of the intention of a test with implementation details is necessary for the execution of a test, a tight coupling of the intention of a system test and the specific implementation details of the SUT have negative consequences when compared to an approach that decouples test specification and test execution:

- It leads to increased costs for test creation and maintenance since changes to the SUT or test harness have to be reflected in the test.
- It prevents reuse of the test since the implementation and the intention can not be separated.
- It prevents the specification of test cases by domain experts in many domains, since programming skills are needed to write executable tests.

The disadvantages of tight coupling become more relevant in cases where a family of SUTs has to be tested. Such testing of multiple systems within a system family can be necessary for different reasons:

- Different versions of a system have to be tested. In cases where more than one version of a system exists, for instance a development version and a production version, system tests are needed for all versions.
- Different variants of a system have to be tested. Variants may differ in their interfaces, the functionality available, in the technologies used or in optimizations.
- Different systems with similar requirements have to be tested, for example during a system migration.

If implementation details necessary for test execution and test intentions are not separated in the above situations, automatic system tests have to be manually adapted to changes

in the interface, even if the tested requirements as expressed by the intentions are identical. When the different systems within a system family are not only different regarding their interfaces, but also differ in their features [5], the reuse of test cases without a clear separation of intention and execution details is hindered by the fact that the features exercised by a test case can not be determined from the test case itself. In order to allow the reuse of tests, an approach to automatic system testing must fulfill the following requirements:

- The specification of tests independent from the implementation of the tested system and the test harness must be supported.
- It is necessary to be able to select tests for a SUT based both on the features exercised by a test and the features available for a system.
- The system must be able to add information needed for test execution to the test intention in order to execute the test on the SUT.

2. RESEARCH QUESTION

Our research addresses the question how the intention and implementation of automatic system tests can be expressed in a way that allows the reuse of tests for different versions or variants of a system or for multiple systems within a family of systems and enables domain experts without programming skills to specify tests.

It is our hypothesis that it is possible to achieve this goal by modeling the intentions of tests and the relevant aspects of different SUTs in abstract, formal models. These models are used for the specification of abstract tests which are used for the generation of executable test code in a model transformation phase. By separating the development of a set of core assets for testing in an application domain and the development of the infrastructure needed for the execution of test for specific systems, the reuse of test intentions is supported.

In order to answer the research question and to validate our hypothesis, the following details have to be considered:

- How can the tests be identified that are relevant for the quality assurance of a system and can be expressed by abstract models?
- On what level of abstraction must SUTs and tests be represented to be independent from implementation details but suitable for automatic testing? How can information about implementation details that is necessary for the execution of tests be managed?
- How can test models be represented in a way that allows their instantiation by domain experts without specific technical skills?
- How can the variability of different SUTs be expressed and when in the test specification and execution process should this variability be considered? Is it more feasible to define tests completely independent of the specifics of a system under test and add details in a separate step or should test be specified under consideration of the specifics of multiple systems as represented by their system models?

3. APPROACH

In this section we give an overview of our approach, called Model-Driven Test Case Construction (MTCC). We introduce the different types of models and metamodels, discuss their use in the testing process and illustrate how reuse of tests can be achieved by separating the development of system independent tests and the generation and execution infrastructure for specific infrastructures.

3.1 Use of Models in MTCC

MTCC uses two models to represent SUTs and the tests executed on the SUTs:

- Test models express the intention of the test, they represent the domain-specific requirements that are verified by the test.
- System models describing the SUT must include information about the features and interfaces that are relevant for the definition and the execution of tests, but still abstract from implementation details.

In order to transform test-models into executable tests for specific systems, both information from the test model and the system model is necessary. We consider two approaches to combine information from the two types of models:

- In an early binding approach, system models for one or multiple SUTs are used to instantiate a specific test metamodel that only supports the definition of tests that can be executed on these SUTs.
- Using a late binding approach, tests are modeled using only the test metamodel. Test models are then selected based on exercised features and expected interfaces.

Since tests are expressed as models and not as executable code, our approach uses a separate model-transformation step following the specification of the test models and their binding to a specific system model. We use a transformation approach based on templates [8] to include both details about the SUT and the testing harness.

3.2 Process

MTCC separates the testing-process into the five steps presented in Figure 1. In the following, we will give an overview of the steps and the relevant concepts and than present the activities and methodologies used in the realization of the approach.

Metamodel Definition In a first step, metamodels are defined. The metamodels define the concepts that are relevant for testing systems within a specific domain and the constraints to apply to these concepts [7]. The `TestMetaModel` describes basic structures for test within the domain of interested. The `SystemMetaModel` includes concepts for the description of the features offered by different SUTs and for the description of the interfaces used for testing.

System Modeling The `SystemMetaModel` is instantiated in a second step. This model instance represents the features and interfaces of one specific system under tests. The `SystemModel` is used to instantiate a specialized variant of the `TestMetaModel` restricted to those test which can be executed on the modeled system.

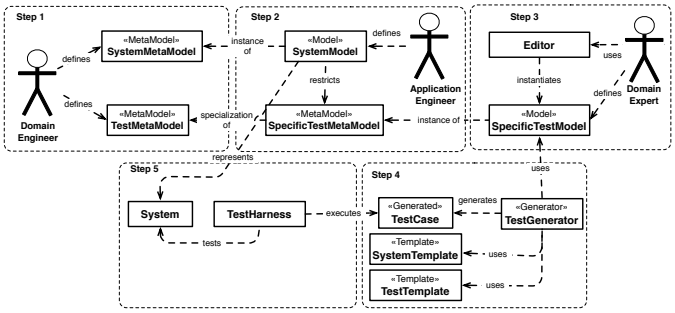

Figure 1: Steps and concepts of the MTCC approach to model-driven test case construction

Test Modeling A test is modeled by a domain expert using an editing environment.

Test Generation The previously defined test model is transformed into an executable test case. Details about a SUT and the test runner used for testing are supplied by templates. Templates are source code fragments without an underlying metamodel.

Test Execution The test case generated by the test generator are executed on the system.

The approach shown in Figure 1 uses an early binding strategy to manage the variability between different SUTs. Information about a SUT as represented by the system models is introduced into a test metamodel before it is instantiated. Test modeling is done based on an restricted `SpecificTestMetaModel`. We also consider alternative approaches where the `TestMetaModel` is either restricted with as `SystemModel` representing a superset of multiple systems.

3.3 Reuse of Tests

We consider the specification of test intentions and the preparation of the execution of tests as two separate activities, similar to core asset and product development in the context of software product lines.

- A number of core assets are central to our approach. These include test intentions for a domain of systems expressed in abstract test models. The models are instances of test metamodels. These metamodels, together with metamodels for the systems under test and the infrastructure for system model instantiation and test specification form the core assets of our approach.

- The executable tests for the SUT in combination with the infrastructure used for test generation and execution are a product based on the core assets.

The separation of abstract test models from test generation and test execution allows the reuse of existing tests for different systems if all tested system share the features that are exercised by a test model. Differences in the interfaces of the systems become only relevant in the test generation step where templates include the specific information for systems and test harnesses needed to execute the tests.

4. EVALUATION

We plan to implement and evaluate MTCC in the domain of scientific information retrieval systems. This domain has a number of properties that motivate this choice:

- The domain is rapidly evolving, new features are frequently introduced into existing systems and new systems are implemented. These frequent changes increase the importance of automatic tests.

- Interfaces, data formats and procedures are not standardized. The resulting heterogeneity makes it possible to evaluate MTCC in the context of multiple systems.

- The specific functional and non-functional requirements for scientific information retrieval systems are highly dependent on the use cases of different stakeholders. This motivates the involvement of domain experts in the testing process, especially since concepts like the relevance of a document can not be automatically verified.

MTCC will be evaluated in terms of efficiency and effectiveness. This will be done by constructing automatic system tests for a number of scientific search portals. The SUTs form a system family in that they offer search and browsing functionality over information from the same topic. Since the SUTs share a number of functional and non-functional requirements, but differ in the detailed requirements as in the features offered and in their implementation.

MTCC will be compared to manual testing as well as the automated testing using capture replay techniques. Of central interest is the cost of test specification and execution compared to those approaches and the ability of MTCC to test requirements as defined by domain experts.

5. RELATED WORK

MTCC addresses similar fields of research as model-driven testing (MDT) and automatic acceptance tests. It builds on the concepts of software product line engineering and domain engineering and extends these for the specification of automatic system tests.

MTCC is similar to model-driven testing [3] in that tests are not implemented manually but are generated from a formal model. An important difference between MDT and MTCC is that the latter is based upon a test-complete [2] model of the SUTs and its environment, while MTCC models represent tests and those aspects of a SUT that are necessary for test execution. The semantics of the tests are not derived from the model.

Approaches for the automation of acceptance tests share MTCC's goal to allow test specification by domain experts. In order to do so, FIT [6] or the approach described by Andrea [1] separate the test specification phase from the test execution phase. The difference between MTCC and those approaches is that MTCC addresses the reuse of test specifications for multiple systems.

Testing for Software Product Lines (SPLT) is related to MTCC in that it addresses the generation and execution of tests for systems within a system family. A difference between both approaches is that MTCC does not assume that SUTs are created in a SPL process using the same core assets.

MTCC shares many aspects with software product lines [4] and domain engineering [5], in particular the goal to generate a family of products, which in the case of MTCC are test programs, from a set of common core assets. The concepts of a features and feature modeling are important to MTCC in order to describe the properties of a system under test.

We expect the greatest benefit of MTCC to be in the reuse of existing test knowledge for different systems within a domain and in the development of models and metamodels for the descriptions of tests.

6. CONCLUSION AND FUTURE WORK

We presented MTCC, an approach for model-driven test case construction. In order to allow reuse of automatic system tests and to support test specification by domain experts. MTCC's goal is to separate test intentions from implementation specific aspects of test. MTCC builds on research from the fields of software product lines, model-driven development, domain engineering and automatic acceptance testing. Implementation and evaluation of MTCC are done in the domain of scientific information retrieval systems.

7. REFERENCES

[1] Andrea, J. Generative Acceptance Testing for Difficult-to-Test Software. In *Extreme Programming and Agile Processes in Software Engineering* (2004), J. Eckstein and H. Baumeister, Eds., vol. 3092 of *Lecture Notes in Computer Science*, Springer, pp. 29–37.

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

[3] Blackburn, M., Busser, R., and Nauman, A. Why Model-Based Test Automation is Different and What You Should Know to Get Started. In *International Conference on Practical Software Quality* (2004).

[4] Clements, P., and Northrop, L. M. *Software Product Lines : Practices and Patterns*, 3rd ed. Addison Wesley, 2001.

[5] Czarnecki, K., and Eisenecker, U. W. *Generative Programming. Methods, Tools and Applications*. Addison-Wesley, 2000.

[6] Mugridge, R., and Cunningham, W. *FIT for Developing Software. Framework for Integrated Tests*. Prentice Hall PTR, 2005.

[7] Stahl, T., and VÃűlter, M. *Model-Driven Software Development*. Wiley & Sons, 2006.

[8] Voelter, M., and Bettin, J. Patterns for Model-Driven Development. EuroPLoP, 2004.