Model-Driven Test Case Construction by Domain Experts in the Context of Software System Families

Dissertation

zur Erlangung des akademischen Grades
"Doktor der Ingenieurwissenschaften"
(Dr.-Ing.)
der Technischen Fakultät
der Christian-Albrechts-Universität zu Kiel

Stefan Bärisch

Kiel
2009

1. Gutachter	Professor Dr. Wilhelm Hasselbring
2. Gutachter	Professor Dr. Jürgen Krause
Datum der mündlichen Prüfung	23.02.2009

Abstract

This thesis presents MTCC (Model-Driven Test Case Construction), an approach to the construction of acceptance tests by domain experts for testing system families based on feature models. MTCC is applied to the application domain of Digital Libraries.

The creation and maintenance of high quality systems that fulfill both formal requirements and meet the needs of users is one of the primary goals of software engineering. A prerequisite for the quality is the absence of faults. Software engineering has defined a number of techniques for avoiding faults, identifying them or fixing them. Testing identifies faults by exercising an implementation artifact and comparing its actual and expected behavior. MTCC is an approach the automate acceptance tests for the members of system families.

The basic hypothesis of this thesis is that the involvement of domain experts in the testing process for members of system families is possible on the basis of feature models and that such a testing approach has a positive influence on the efficiency and effectiveness of testing. Application quality benefits from the involvement of domain experts because tests specified by domain experts reflect their needs and requirements and therefore can serve as an executable specification.

One prerequisite for the inclusion of domain experts is tooling that supports the specification of automated tests without formal modeling or programming skills. In MTCC, models of automated acceptance tests are constructed with a graphical editor based on models that represent the test-relevant functionality of a system under test as feature models and finite state machines. Feature models for individual testable systems are derived from domain-level systems for the system family. The use of feature models by the test reuse system of MTCC facilitates the systematic reuse of test models for the members of system families. MTCC is a Model-Driven test automation approach that aims at increasing the efficiency of test execution by automation while keeping independence from the implementation of the testee or the test harness in use. Because tests in MTCC are abstract models that represent the intent of the test independent from implementation specifics, MTCC employs a template-based code generation approach to generate executable test cases.

In order to validate the approach, MTCC is applied to the Digital Library application domain. Digital Libraries are Information Retrieval systems that aim to provide the scientific community with relevant information. A MTCC prototype is designed and realized for a system family of three Digital Libraries and an Information Retrieval system. The capability of representing tests relevant for the application domain, for reusing these tests for multiple systems and for generating executable tests from the abstract test models are validated. An assessment of the understandability by domain experts and of the usability of the editor is conducted. The feasibility and practicality are shown by a validation involving domain experts for a system family of Digital Libraries.

Acknowledgements

Many people deserve my gratitude for the support and encouragement they gave me during my work on this thesis.

My advisor, Professor Dr. Wilhelm Hasselbring, helped me to shape and refine the ideas underlying this work und supported me in developing the means to express, implement and validate this ideas in this dissertation. Thank you.

Professor Dr. Jürgen Krause gave me the opportunity to combine my studies and work on this thesis with the always interesting work for the GESIS. I thank him for his support and his belief in this work as well as for our interesting talks.

I thank all my colleges at the GESIS, both past and present. Max Stempfhuber gave me the freedom to combine this dissertation with my other responsibilities and pointed out areas of improvement. Patrick Lay and Holger Heusser gave me important feedback on my thoughts.

Thanks also all domain experts. I am grateful for the time and effort contributed by the members of the evaluation group, Vivien Petras, Maria Zens, Simone Weber and Jan-Hendrik Schulz.

Rielies Neitzke proved that at the heart of all research, a librarian can be found. Harald Deuer and Maria Zens always had time for some words. My deepest thanks to Vivien Petras for our discussions and for her support and encouragement.

It is always worthwhile to present ones ideas to a fresh audience. My thanks the ES-EC/FSE 2007 Doctoral Symposium for their time and insights, in particular for emphasizing the importance of a sound evaluation.

The Software Engineering Group at the Carl von Ossietzky University of Oldenburg gave me the opportunity to present and discuss many of the ideas that are present in this thesis. I thank you both for this and for the friendly atmosphere I could always count on.

My parents made this dissertation possible in more than one way. Elke and Wolfgang, you have my sincerest thanks and gratitude for all you have done in the past 31 years.

Contents

List of Figures

List of Tables

Chapter 1

Motivation and Background

Testing is the execution of a system with the intent to compare the expected and the actual behavior of a system (Bertolino, 2007) in order to find faults. In addition to the identification of faults (Myers *et al.*, 2004), testing supports the assessment of the quality of a system with some degree of confidence (Mugridge & Cunningham, 2005b; Beck & Andres, 2004). Testing benefits greatly from automation. Automated regression tests allow for a nearly continuous verification of the quality of a system. Automated tests can also take the role of executable, albeit limited, specifications.

This thesis describes the MTCC, an approach that facilitates the construction of automated tests by domain experts for the members of a system family based on abstract models of the testees in the system family.

The quality of an automated testing system can be evaluated based on two aspects: The ability of tests to reveal faults in the SUT (System under Test) and the resources necessary for the design, implementation, execution and maintenance of the test (Graham & Fewster, 2000). We argue that the involvement of domain experts in the testing process increases the probability to reveal faults, while test automation increases the efficiency of testing:

In order (1) to include domain experts in the testing process and thus increase the probability of finding faults relevant to them and (2) to support the automation of tests and (3) to facilitate the reuse of tests in a system family context, we propose the construction of test models by domain experts. The construction process encompasses (1) the modeling of the test-relevant aspects of a system family and the implementation of a test infrastructure by domain experts as well as (2) the use of an editor based on the modeled formal representation of systems and tests for the members of a system family by the domain experts and (3) the generation of test cases from these test models.

- If a fault is defined as an aspect of the system behavior that does not meet the requirements of customers, it follows that it is beneficial for testing this system and hence the quality of this system to involve domain experts in the testing process. Such involvement ideally allows the domain experts to directly express their requirements as acceptance tests.
- The resources necessary for test execution are reduced by the use of test automation software. Test automation facilitates the execution of more tests more often. Since human testers are no longer needed for the execution of routine regression tests, they can work on the identification of new faults. Test automation therefore not only reduces the costs of testing, but also benefits the quality of a system.

A software system family is defined by the commonalities of its members (Parnas, 1976). Since the systems within such a system family share common requirements and use cases, it is desirable to systematically reuse testing assets and tests among them.

We argue that the inclusion of domain experts is of particular importance for the application domain of Digital Libraries. The inherent vagueness of concepts like information need (Lewandowski & Hoechstoetter, 2007) and relevance (Schamber *et al.*, 1990) requires the involvement of human domain experts in the specification of tests.

One challenge to involving domain experts in automated testing is that domain experts cannot be assumed to have any skills in programming or formal modeling. This is of special significance when the subject of test automation is not a single system but the different members of a system family. In such situations, the test-relevant functionality of one system as well as the commonality and variability (Coplien *et al.*, 1998) between the members of the system family that are relevant to testing have to be formalized. Existing approaches involving domain experts in the automation of acceptance tests (Mugridge & Cunningham, 2005b) do not consider the use of models for the systematic reuse of tests in the context of system families. We therefore present the MTCC approach for the **m**odel-driven **t**est **c**ase **c**onstruction in Chapter 6.

MTCC applies domain analysis (Kang *et al.*, 1990) to the testees in the system family in order to capture the test-relevant functionality and the steps needed to exercise and test this functionality. The identified functionality and the corresponding Test Steps are formalized as feature models (Czarnecki & Kim, 2005; Czarnecki *et al.*, 2005b) that represent the variability and commonalities within the system family as they are relevant for testing. Specific systems within the system family are expressed by specializations (Czarnecki *et al.*, 2005b) of these feature models. System-level feature models describe the testable Service instances for one SUT. The MTCC editor uses (1) the feature model of the SUT in combination with (2) a state machine representing all possible sequences of Service invocations and (3) models of the applicable Test Steps for each Service to support the construction of tests in a graphical editor.

A test in MTCC is expressed as a sequence of fully specialized (Czarnecki *et al.*, 2005b,a) Test Step feature models. Each configured Test Step instance represents an action taken on the system or an assertion used to verify an expectation about the system. Each Test Step instance is represented by a feature model that represents the possible configuration of the Service instance exercised by the Test Step as well as the available parameterization. Service instances represent specific aspects of the functionality of the testee. Since the tests that are constructed with the editor are abstract models and are thus not suitable for direct execution on a testee, the MTCC approach employs code generation techniques to create the actual test code used for exercising the testee.

1.1 Testing and Quality

The goal of testing is to find faults in a software system by executing it and then comparing some expected behavior for the system with its actual behavior as encountered and recorded during execution.

1.1.1 Testing in Quality Assurance

Binder (Binder, 1999) writes the following about the definition of testing: *'Software testing is the execution of code using combinations of input and state selected to reveal bugs'*.

Binder's definition is more specific than the definition of testing given by the IEEE standard 610 (of Electrical & Staff, 1991): *'The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component'*.

According to Binder's definition, the subject of testing is program code that is executed. Binder also states that the goal of testing if to identify faults in a system, not to evaluate the system. For the purpose of this thesis, we are closer to the IEEE definition of testing. We argue that while tests can never prove the absence of faults and thus the quality of a system, they have more uses than solely the identification of faults. In particular, we consider tests as an executable and formal — if incomplete — specification of required system behavior.

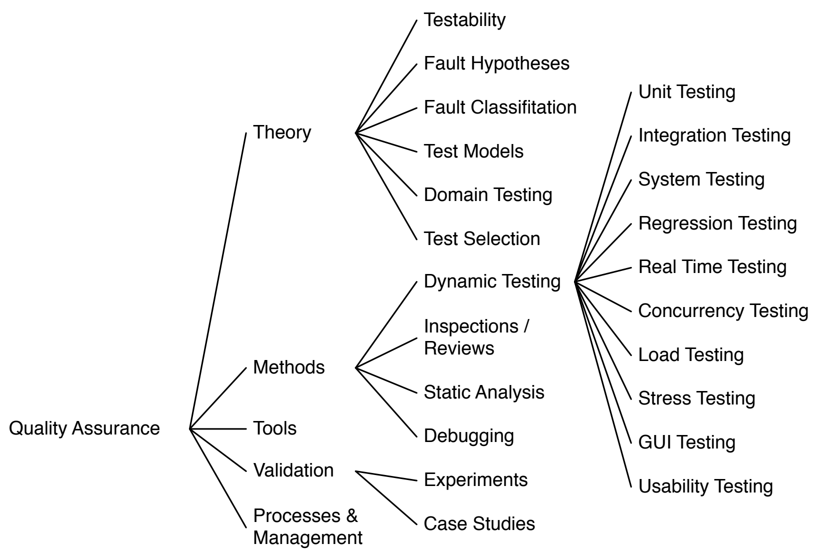


Figure 1: Testing in relation to other methods of software quality assurance (Winter, 1999)

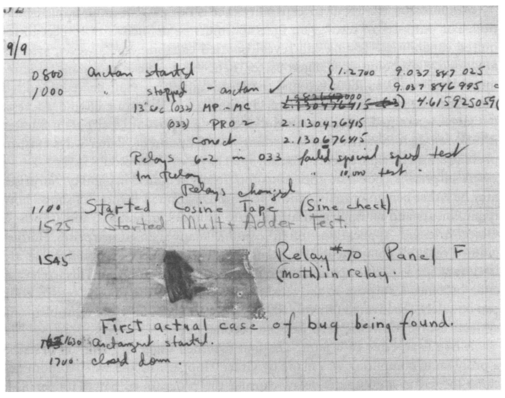
Based on this definition, testing can be distinguished from various other techniques that aim to assure the quality of a software system. Figure 1, adapted from Winter (Winter, 1999), gives an overview of some such methods, among them the following:

- Models, design documents, or code can be reviewed (Gilb *et al.*, 1993) for correctness or completeness. Since no artifact is executed, a review is not a test. This is also true for automated reviews such as model checking and static code analysis (Louridas, 2006).
- Debugging (Zeller, 2005) is not a testing activity. The goal of debugging is to find the cause of a fault in order to correct it. While a system is executed and examined during debugging, the existence of a fault must already be known in order for debugging to begin. Testing and debugging are thus closely related but different activities (Cleve & Zeller, 2000).
- Monitoring observes the execution of a system and records or controls data about the execution, for example the availability of the system or its resource usage. We do not consider monitoring a testing activity. While a system is executed and assessed, the motivation for testing and monitoring are different. Testing is performed in order to gain some confidence about the question whether a system is faulty, monitoring is carried out to ensure a reaction when faults occur during the execution of the system and to provide runtime statistics or uses other than the identification of faults.

1.1.2 The Goal of Testing

Two aspects about the definition of testing need to be discussed in more detail: (1) The question whether the only goal of testing should be the identification of bugs or whether it can be leveraged to ensure the quality of a system and (2) the definition of faults or 'bugs' in a software system.

The fact that testing cannot prove the absence of faults and therefore the quality of a system (Dijkstra, 1969) is widely accepted today (Winter, 1999). Exhaustive testing for all



9/9

0800 Arctan started
1000 " stopped - arctan ✓ {1.2700 9.037 847 025
9.037 846 995
13" sec (032) MP - MC ~~2.130476415~~ 4.615925059(
(033) PRO 2 2.130476415
correct 2.130676415
Relays 6-2 in 033 failed special speed test
in relay " 10,000 test.
Relays changed
1100 Started Cosine Tape (Sine check)
1525 Started Mult + Adder Test.
1545 Relay #70 Panel F
(moth) in relay.
First actual case of bug being found.
1630 Arctangent started.
1700 closed down.

Figure 2: Classic bug (Binder, 1999)

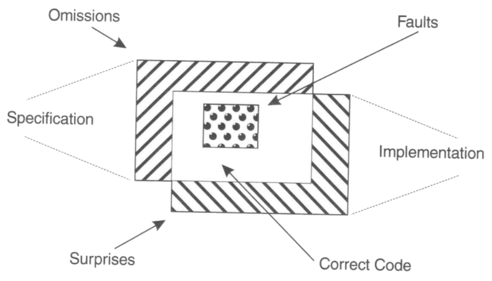


Figure 3: Implementation, specification and faults (Binder, 1999)

states and inputs of all but the most trivial systems is impossible (Binder, 1999), no useful system can thus be completely tested. Instead, testing focuses on those scenarios of system usage that have the highest probability to have faults.

Despite the fact that testing cannot prove the correctness of a system, we argue that testing has uses beyond finding faults. Depending on the test coverage of a system (Zhu *et al.*, 1997), testing can be used to gain some confidence in the correctness of a system. From this view, tests can also serve as an executable specification of some aspect of system behavior (Fowler, 2006). Regression testing, the execution of a set of standardized tests to ensure that modifications to a system did not introduce faults, is an example for the use of testing to specify the correct behavior of a system and judge its correctness.

Bugs and Faults

One prerequisite for the identification of faults is a definition of what defines a fault or bug for a given system under test.

Binder uses the term 'bug' as an euphemism for the term fault. A fault, as illustrated in Figure 3, is an aspect of the system implementation that differs from the specification. In this thesis, we use the term fault in a broader sense. Our definition also includes omissions and aspects of the implementation that differ from the requirements for the system, whether these are part of the specification or not. A defect is the underlying cause of a fault, for example a wrong passage in the source code of a system. The class of faults displayed in Figure 2 has mostly lost its relevance for today's systems.

As the definition of faults illustrates, in order to test a system for faults, a tester has to know what constitutes correct behavior for the system under test. In test-driven development (Beck, 2002), this fact is used to design and implement software systems. Before implementing some part of the functionality of a system, a test for this functionality is designed and realized. Writing these tests forces the developer to think about the intended functionality of the system in detail. The implementation of tests for a system and the system itself are done in parallel.

According to the definition of faults given above, testing must be based on a specification. Such a specification may be a formal document that describes the correct behavior of a system in great detail or it may be informal and incomplete, it may in fact only exist in the minds of engineers and domain experts. To a certain degree, tests can serve as examples (Gälli, 2006) of correct system behavior and therefore as a limited specification.

1.1.3 Dimensions of Testing

Different types of testing can be differentiated on the basis of their view of the SUT, the granularity of the SUT and the aspects of the system behavior tested.

Granularity of the Tested Artifacts

The most prevalent distinction of testing is made on the basis of the granularity of the tested artifact. System-level tests (Binder, 1999) exercise a complete system, unit tests in contrast are used to verify the correctness of a small number of classes or functions. While system-level tests require the whole system under development to be in a testable state and are therefore mostly used late in the development process, unit testing is feasible as soon as the first classes of the system are implemented.

Regression tests can be used both on the system level and on the unit level. As already described, regression tests are not used to find previously unknown faults in a system or verify new functionality but ensure that no faults are introduced into the already existing parts of a system.

Implementation-based Tests and Responsibility-based Tests

Tests can either verify the implementation of an artifact or its behavior. Implementation-based tests are written with explicit knowledge about the internal implementation of the system under test. An implementation-based test for a class would use information about private methods and instance variables to manipulate the state of a system and test it. Responsibility-based tests only use the public interface of an artifact under test to manipulate and verify its state and behavior (Binder, 1999).

An advantage of responsibility-based testing is the decoupling of the tests used to verify an artifact and the implementation of the artifact. It is therefore possible to reuse tests for different implementations of the same interface and change the implementation of an artifact without having to change its tests. Disadvantages of responsibility-based tests include the fact that direct manipulation of the internal state of the artifact under test is not possible. As a consequence, it may not be feasible to test the artifact for certain states. It is also impossible to use the internal data of the artifact under test to verify its correctness.

Functional and Non-Functional Tests

Testing approaches can be distinguished by the question whether they verify functional or non-functional attributes of the artifact under test. Functional tests ensure that the behavior of the artifact under test is correct. Non-Functional tests verify aspects like response time or system behavior under load.

MTCC is a system-Level, responsibility-based approach to test primarily functional requirements.

1.1.4 Test Automation

The testing of a software system can be separated into multiple phases:

- The goals of the test are defined during the first phase. The functionality that will be exercised by the test is selected and analyzed for likely faults.
- Test cases for previously selected functionality are defined in the second phase of testing. Each test case exercises the testee in a way that is likely to reveal a fault. This probability is estimated based on the analysis in the first phase.

- In the third phase, the test cases are executed on the testee, the behavior of the system for each test case is recorded.

- In the fourth phase, the recorded behavior of the testee is compared with its expected behavior. The test is successful, if the actual behavior of the test matches the expected behavior.

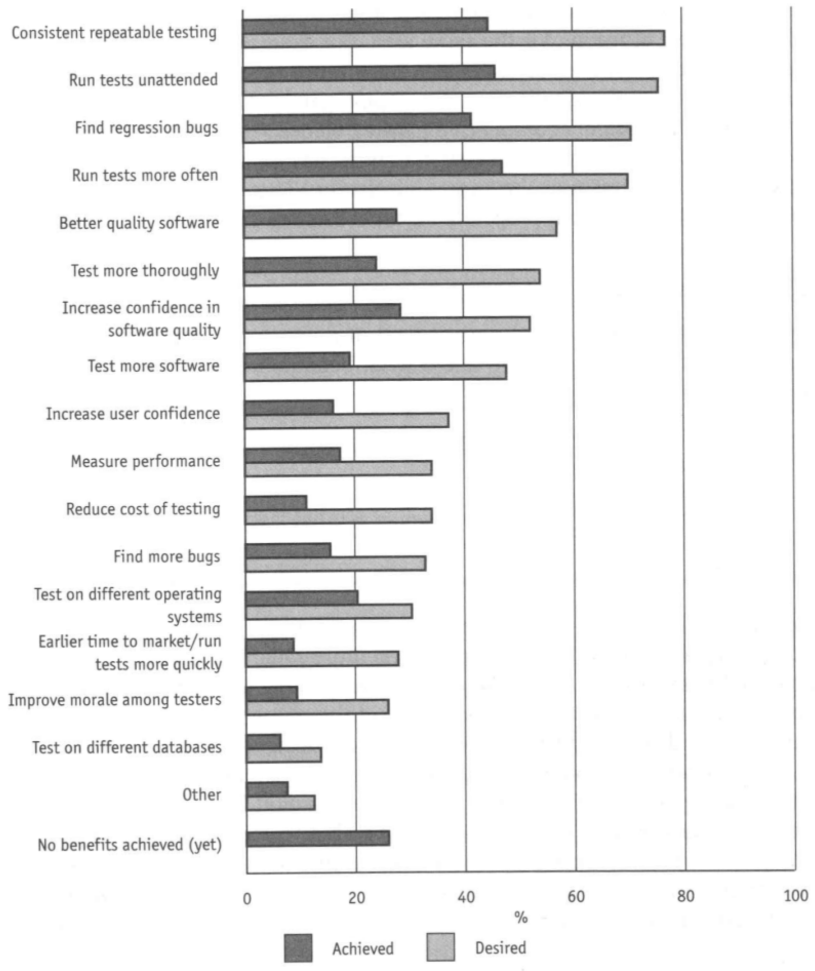


Figure 4: Goals of automated testing (Graham & Fewster, 2000)

Test automation aims to execute one or more of the phases of testing without the need for human involvement. It has the purpose to relieve humans from the need to repeatedly execute a test on a system. Test automation has the advantage of using less resources for the execution of a test and allows far more tests to be executed in a given period of time. Figure 4 examines a number of reasons for test automation that were given by practitioners and compares them to the already achieved goals. It can be seen that the reasons for test automation include both the need for more efficiency and more effectiveness.

Like testing, the automated testing process consists of several phases. Figure 13 on page 35 illustrates these phases. Each phase can be automated to a certain degree. MTCC provides tool support for the Design phase and supports the automation of the Build, Execute and Check Phases.

Approaches to automate testing vary widely in the the format and notation used for the representation of the tests that will be automatically executed and the way in which this representation is created. Capture and replay tools record the interactions between a user and the system under test. The system is tested by replaying the actions of the user and comparing its current behavior with its behavior when the test was recorded. The various xUnit frameworks for different programming languages allow tests to be written as programs in a regular programming language. Fit (Mugridge & Cunningham, 2005b), a system for automated acceptance testing, uses a notation based on the spreadsheet paradigm to specify tests for a system.

Not all attributes of a system can be tested automatically. While function tests and tests for non-functional attributes like performance are routinely automated, tests for the usability or accessibility of a system are harder to automate.

A system for the execution of automatic tests is called a test harness. A test harness consists of a number of subsystems that execute tests, record the behavior of the SUT and decide whether the execution of each test was successful or not. The test driver, also called test runner, is the subsystem responsible for the execution of tests and for recording the behavior of the SUT. The test driver does not need to directly interact with the SUT, the Test Adapter is a library that provides a simplified interface for the SUT as well as commonly used functionality.

Since MTCC uses code generation to transform abstract test models into executable tests, it is not dependent on a specific test harness but assumes that the test harness is compatible with one xUnit (Meszaros, 2007) framework.

1.1.5 Testing System Families

A number of systems belong to a system family if the members of the family share so many properties that they are better discussed in terms of their commonalities than in terms of their differences (Parnas, 1976). System families pose specific challenges to testing, especially automated testing. An important issue in this context is reuse. In order to ensure optimal efficiency it would be desirable if both tests and artifacts used for test execution could be reused for the various members of a system family. The following points are relevant in this context:

- Similar requirements exist for the different members of a system family and it is likely that specific requirements are identical for some systems in the system family. In order to avoid the repeated specification of tests for multiple members of a product-line with shared requirements, tests should be reusable for different systems.

- The software used for test execution should be able to work with the different members of the product-line. The use of one test execution environment for all members of the system family supports the use of a common test representation format. That in turn makes the use of one repository for all tests in the system family possible and enables the reuse of tests.

A systematic approach to address the variability and commonalities of the systems within the system family is needed to enable the optimal reuse of both tests and of the infrastructure needed for test execution. The field of Software Product Line Engineering (Clements & Northrop, 2001; Weiss & Lai, 1999) provides both the theory and the methods for such systematic reuse. Feature modeling (Kang *et al.*, 1990; Czarnecki & Eisenecker, 2000; Czarnecki *et al.*, 2005a) in particular supports the representation of variability (Bachmann & Clements, 2005) between different testees within a system family. MTCC applies the Software Product Line concepts to support the systematic reuse of tests and testing core assets.

1.1.6 Acceptance Tests and the Inclusion of Domain Experts

The quality of a software system can either be judged on its conformance with a given specification or in relation to the requirements and needs of the project stakeholders. Ideally, the specification would be a complete and correct representation of all requirements that exist for a system. In practice, however, this cannot always be achieved. One problematic factor are changes to the requirements of a system that only arise or become clear during the project and are therefore not reflected in the initial requirements.

These problems do not affect the fact that a specification is needed for the development of all but the most trivial software systems. Such a specification must reflect the actual requirements for a system as completely and correctly as possible. We argue that tests constructed by domain experts can serve as a partial specification and that such tests must be automated. The following reasons for test automation are relevant:

- The construction of tests for a system necessitates the clarification of the requirements that a domain experts feels the system must fulfill. Since a test reflects a detailed interaction between the SUT and a user, the requirements reflected by the test must be elaborated on the same level of detail.
- Automated tests represent a type of executable specification. The execution of a test allows a definite statement about the success of the test, this in turn informs about the testee's compliance to the specification.
- Automated tests are repeatable and can be used to exercise a system without human involvement. This makes it possible for the domain experts involved in the project to use more of their time on the improvement of the system under development.

We argue that the fact that tests cannot replace a conventional specification does not put the advantages of acceptance tests into question. It is indeed the case that tests used for specification can only ever be a *Specification by Example* (Fowler, 2006). We argue, however, that the completeness and correctness of a given specification can only be formally proven to be correct in very few cases. The incompleteness of tests as a specification is therefore not specific for specification by example. It is equally impossible to proof that all requirements of a system are covered by, for example, use cases.

One limitation of tests as a means of specification must be pointed out: Tests cannot replace other forms of specifications for non-functional quality attributes such as maintainability. Tests can therefore never totally replace other methods of specification, even when the functionality of a system is completely covered.

1.2 Motivation

In the previous section we discussed the role of testing in the achievement and maintenance of quality in the context of software engineering. We put emphasis on three aspects of testing and discussed their potential to contribute to the quality of a system:

- Acceptance tests facilitate the inclusion of domain experts in the testing process. This inclusion allows for the specification of tests that reflect the requirements of domain experts regarding a software system under development.
- Test automation, when compared to manual testing, allows for the increase of both the frequency of test execution and the number of tests that are executed per unit of time. This supports the establishment of a regression testing regime that helps ensure the continuos quality of a system under test.

- Testing approaches for software system families facilitate the reuse of tests and testing infrastructure for various systems within a system family. Such approaches allow for the systematic treatment of the variably and the commonalities between the member of the system family and aim to establish a set of testing core assets.

We advance the hypothesis that acceptance testing, test automation and testing in the system family context can be addressed by a single approach. An approach supports testing in a way that is more likely to identify those faults where the system under test does not meet the expectations or requirements of domain experts. In order to achieve this goal we introduce the **M**odel-Driven **T**est **C**ase **C**onstruction process and the artifacts used in the process. In MTCC, tests are represented as abstract models that represent the members of a system family and that are constructed by domain experts by means of a Model-Driven editor.

Testing is closely interlinked with the application domain of the system under test. For this thesis, we introduce Digital Libraries as the application domain that is addressed by the automated acceptance tests realized with MTCC.

The discussion of the MTCC approach requires the consideration of the challenges that have to be met in order to allow the realization of automated acceptance testing in a product-line context as envisioned by MTCC. We introduce these challenges in the following and list potential solutions.

Building on our exploration of the application domain, the challenges to automated acceptance testing in a system family context and potential solutions to these problems, we formulate the hypothesis underlying MTCC and introduce the approach.

1.2.1 Digital Libraries

In this thesis, we use the term Digital Libraries to describe a specialized class of Information Retrieval systems. Digital Libraries are an important part aspect to supply researchers as well as other groups with relevant information, the field has therefore been the subject of considerable research, for example under the DELOS[1] efforts.

The quality of a digital library is determined by its ability to satisfy the information need of a user searching for relevant documents. In order to fulfill such an information need it is not sufficient for a Digital Library to merely provide query capabilities for data like a relational database and not exhibit any obvious faults. As a result of the vague nature of the concepts relevance and information need, any evaluation of the quality of a Digital Library has to consider the individual requirements and needs of the library users.

Vagueness in Digital Libraries

The field of Information Retrieval is complex and includes such different topics as distributed data structures, machine learning and usability. It is therefore nontrivial to give a complete and concluding definition. Manning, Raghavan and Schütze define Information Retrieval thus: *Information retrieval (IR) is finding material (usually documents) of an unstructured nature (usually text) that satisfy an information need from within large collections (usually stored on computers)* (Manning *et al.*, 2008). Baeza-Yates and Ribeiro-Neto give the following definition: *Information Retrieval(IR) deals with the representation, storage, organization of, and access to information items* and add in the following: *The emphasis is on the retrieval of information as opposed to the retrieval of data* (Baeza-Yates & Ribeiro-Neto, 1999).

Both definitions contrast Information Retrieval from data retrieval (as the domain for instance of highly structured relational databases). The difference between both types of

[1] `http://http://www.delos.info`

retrieval for testing becomes evident when we consider that the results of a data retrieval task are always decidable; results either satisfy the conditions defined by, for instance, a SQL select statement or they don't. In contrast there exists no automatic way to determine whether a given document from an Information Retrieval system satisfies the information need of a user. Consequently, a data retrieval system can theoretically be tested without any human involvement, testing an IR system needs humans to judge the relevance of results.

Human involvement in the evaluation and testing of IR systems becomes even more important in the face of semantic and structural heterogeneity (Hellweg *et al.*, 2001; Stempfhuber, 2003). Heterogeneity in Digital Libraries has some similarities to Schema Matching (Shvaiko & Euzenat, 2005; Rahm & Bernstein, 2001), Ontology Mapping (Kalfoglou & Schorlemmer, 2003) and Enterprise Application Integration (Hohpe *et al.*, 2004) but is differs in the significant role of semantic heterogeneity and the particularities of bibliographic metadata. In situations where information from different sources and of different quality are integrated into a common Digital Library, the involvement of domain experts in testing becomes even more important since the correctness of integrations efforts can not be judged from a technical perspective alone.

The Digital Libraries considered in this thesis provide information for scientific communities. They include bibliographic information as well as information about current research projects and scientific institutions. A number of specific requirements arise for Digital Libraries as Thurmeier (Thurmeier, 2007) and Heinold (Heinold, 2007) discuss.

User Interaction with Digital Libraries

For current Digital Libraries as introduced in Chapter 9, the interactions between a user and the retrieval systems during a search session may include the following steps:

- The user employs the user interface of the Digital Library to express his or her information need in the form of a query and sends this query to the retrieval system.
- The Information Retrieval system accepts the query, does further transformations on it or expands it. It computes the similarity of the query and the documents in the Digital Library. The way, in which such similarity values are derived for a query and the documents in the Digital Library, is determined by the retrieval model implemented.
- The Digital Library assembles a list of those documents that are most similar to the query and are thus deemed to be the most relevant with respect to the information need of the user. Most current Digital Libraries will return the documents in a ranked order, arranged with decreasing probability of relevance.
- The Digital Library represents the list of the documents, often enhanced with extra information such as statistics about the results, to the users.
- The user surveys the results returned by the Digital Library and investigates those documents that are the most likely to be relevant. In order to allow both a quick overview of the results and a detailed investigation of single documents, the Digital Library provides different views on the documents returned.
- Depending on the specific Digital Library, various value-added functions may be available to assist the user in her or his research. Examples for such functions include: Checking the availability of documents at local, physical libraries, ordering the electronic version of a document or exporting the documents to a reference management system.

Digital Libraries exhibit a number of particularities relevant for testing, namely their user interface and the documents held by the library.

Heterogeneity in Digital Libraries

A user interacts with the Digital Library by means of a graphical user interface. Since a domain expert for a given digital library will be familiar with the GUI of the system, an acceptance testing approach should emulate the patterns of interaction used by this GUI. If the GUI is the only interface available to interact with a system, the test runner must be able to exercise the GUI in order to exercise the system.

Two characteristics of the documents included in a Digital Library are relevant to testing:

- The documents in Digital Libraries are often semi-structured (Abiteboul *et al.*, 2000) or sparse. Many concepts in the documents are either optional and/or can occur multiple times.
- Structural and semantic heterogeneity (Stempfhuber, 2003) originates when multiple different document collections are integrated in a common Digital Library, either by data integration or by means of federated search technology. Documents are structural heterogeneous when the metadata schemes of the documents differ, semantic heterogeneity is defined by the use of different knowledge organization systems such as classifications or thesauri.

A Digital Library that integrates various different document collections is called a portal. The trend towards integration is motivated by the intent to provide user of the Digital Library with one place that allows an integrated search in a number of different collections.

While the data cleansing and integration methods used are main impact factors of the quality of the integrated, initially structurally heterogeneous data, testing can verify that all relevant information of a document was considered in data integration and that the results of the data integration process match the expectation of both the experts for the original and the integrated document schemas. Testing is highly relevant in the presence of semantic heterogeneity. Methods that address semantic heterogeneity such as the use of cross concordances in search, increase the complexity of an Information Retrieval system. Tests are needed to verify that a system works as expected.

Typical Architectures of Digital Libraries

Figure 5 illustrates one possible architecture for an Information Retrieval system, the concepts and system layers displayed in the figure and their relationships also apply to Digital Libraries as examined here.

As Figure 5 illustrates, a Digital Library is a modular system that consists of a number of different components. While MTCC is a responsibility-based approach and thus does not have any information about the implementation of components, the existence of the components is nevertheless relevant to testing:

- The Client Software determines how a user can interact with a system and which interfaces are available for testing. The Digital Libraries considered in this thesis are web applications whose interface is displayed using a web-browser.
- The Query-Processing Modules and the Result Processing Modules conduct transformations on the requests send to the IR system and on the results returned by the system. Since these transformations can greatly influence the behavior of a Digital Library and its ability to provide a user with relevant results, it is desirable that they are tested.

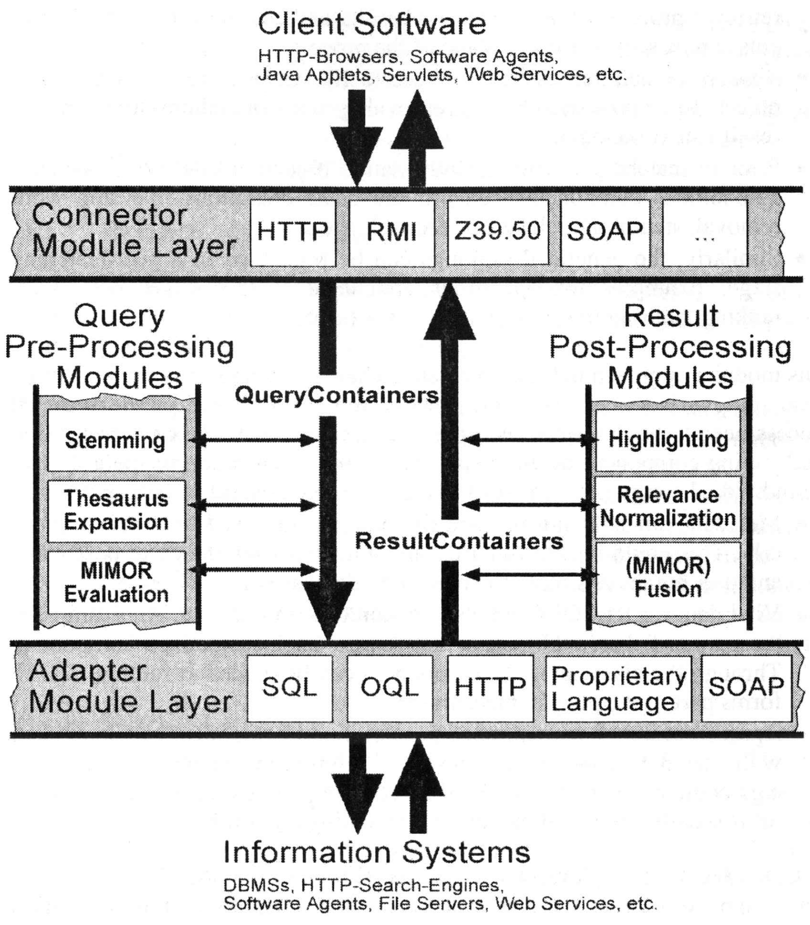


Figure 5: Architecture of an Information Retrieval system (Womser-Hacker, 2006)

- The Information System is the back-end layer that is responsible for the execution of a query. It has to determine which documents are the most likely to be relevant for the information need expressed in the query.

All the components listed above have great influence on the behavior of a Digital Library. While the automated acceptance tests conducted with MTCC do not normally exercise the components in isolation but rather the digital library as an integrated system, our discussion of the components serves to exemplify the complexity of a Digital Library and thus the need for tests.

An important aspect of a Digital Library is the information provided in the system. All digital libraries are build on metadata that describes the information in the library. The completeness and correctness of the metadata determines the usefulness of a Digital Library to its users to a significant degree. Tests for a Digital Library therefore have to include the option to test the metadata, for example to guard against unintended changes to metadata entries during updates of the data.

1.2.2 Motivation for the MTCC Approach

In this section, we discuss our motivation for the MTCC approach, namely the deficits we perceive in existing approaches to automate acceptance tests in the context of system families.

Model-Driven testing and specification-driven testing abstract from details of the implementation under tests and thus support the use of high-level test description languages or models of tests. While the abstraction of implementation details allows for the high-level modeling of tests instead of requiring tests to be programmed, the modeling tools used in these approaches are still not suited to involve domain experts in the testing process.

Testing in the context of Software Product Lines provides methods and tools for the systematic treatment of the variability of various testees in the product-line and thus facilitates the reuse of both tests and testing infrastructure. SPL-Testing assumes, however, that the testees themselves were constructed in a product-line context and thus are based on a common set of core assets. This assumption does not hold for the system families of Digital Libraries that are the subject of MTCC.

Approaches for the automation of acceptance tests support domain experts in the specification of tests that can be automatically executed on a system. However, such approaches do not address testing in a system family context.

Model-Driven Testing

Model-Driven testing generates automated tests from a model of the system under test and its environment (Blackburn *et al.*, 2004). In order to apply Model-Driven testing to a system, two preconditions have to be fulfilled:

- The models must support the identification of a functionality of the testee that is to be tested and then generate inputs for the system that exercise the testee.
- The model must contain enough information about the intended behavior of the system to decide whether a test executed on the system was successful or not.

When both these preconditions are met, tests can be automatically generated from models. Human involvement is neither needed for the design of tests nor for their execution. If such models would exist for all members of a software product-line, Model-Driven testing could also be applied to a family of systems.

We argue that applicability of Model-Driven testing is highly dependent on the application domain. More specific, Model-Driven testing is well suited for application domains where the requirements can be captured and formalized in a way that supports the instantiation of an oracle. Such an oracle has sufficient information about the expected behavior of a system to decide whether the actual behavior of a system is correct with respect to the model.

Such an oracle for Digital Libraries would have to be able to decide whether a document returned for a query is relevant to the information need expressed in the query. Since the concepts of information need and relevance are vague and can thus not be formalized in a way that supports an oracle to decide whether a query matches an information need, implementing an automatic oracle with these capabilities is not practical.

Aside from the fact that a fully automated oracle for Digital Libraries can not be implemented, another argument against Model-Driven testing is that domain experts cannot be assumed to have the modeling skills needed for the approach, even if a pragmatic approach and the reuse of existing design models is assumed (Bertolino *et al.*, 2004).

Testing based on formal Requirements Models

One approach to involve domain experts in the design of tests is the definition of tests during requirements analysis. In order to serve as tests, models used in requirement analysis, for example use cases, have to be formalized and extended to include all information necessary for automatic test execution.

A number of research efforts address different approaches to use functional requirements models for testing, an overview of the field can be found in Gutierrez *et al.* (2006).

Use case models are frequently used as a means of test description. Since use cases are not testable (Binder, 1999) models, several of the approaches investigate extensions and formalizations of use cases and the transformation of the resulting models into executable tests.

Two consequences arise form the use of formalized requirements models that lead us to argue that such approaches are not suited for the goals of this thesis:

- Since formal models are needed to express tests, users creating the models need formal modeling skills. It can therefore not be assumed that domain experts without modeling skills will be able to model tests.
- Since tests are modeled during requirements analysis and depend on artifacts and methods from this phase of the software development process, it is unclear how requirements models would have to be adapted for testing.

Testing of Software Product Lines

The Software Product Line approach to software engineering includes technical as well as organizational practices that facilitate the systematic development of products in a product-line context. Meister Meister (2006b) defines Software Product Lines as families of products that were explicitly designed and realized to support the derivation of new products with variable elements.

The application of Software Product Line Engineering methods offers potentially many advantages for testing, but also faces challenges specific to the domain. Software Product Line Testing (SPLT) is an independent field of research in software engineering (Geppert *et al.*, 2004a, 2005). MTCC is not a SPLT approach — SPLT is not well suited to achieve the goals of this thesis for two reasons:

- Most SPLT approaches assume that the systems under tests are themselves products of a SPL and that therefore a formal description of the variability of the different members of the product-line exists. It is also frequently assumed that the various systems under test are derived from a common set of core assets and thus share parts of their implementation. These assumptions prove to be problematic when they are not met, as is the case for the system families of Digital Libraries that are investigated in this thesis. Reuse for these systems is usually opportunistic.
- The focus of many SPLT approaches lies on the generation of tests based on a formal description of the variability within the product line (Geppert *et al.*, 2004b). This does not allow for the inclusion of domain experts in the testing process. Approaches that extend use case notation (Bertolino & Gnesi, 2003b) both to reflect the variability of a product line and to support the modeling of executable tests suffer from the problem that such an extended notation is no longer suitable for domain experts without modeling skills.

We conclude that SPLT aims for the systematic and automated testing of a product-line but the approach is not suited for our scenario, in which different Digital Libraries form a system family, not a product-line.

Automation of Acceptance Tests

Approaches for the automation of acceptance tests fall into two categories: Those that aim for the automation of the interaction between a user and the GUI of the testee and those that try to abstract from the GUI and base tests on a notation or model that expresses what is to be tested, not how it is to be tested.

The most basic approach to acceptance tests is the Capture-replay approach (Hicinbothom & Zachary, 1993). Capture-replay records the interaction between a user and the GUI of the system under test. When a test is executed, the actions of the user on the GUI are replayed and the behavior of the system is assessed based on the recorded behavior. Capture-replay tests are tightly coupled to the specifics of the GUI of the system under test, changes to the GUI often invalidate recorded tests, which than have to be re-recorded.

The fragility of tests created using the capture replay approach as well as other shortcomings have lead to the development of methods for the automation of acceptance tests that abstract from the GUI of the tested system.

Key Words and Actions Words (Blackburn *et al.*, 2002) are textual representations of actions on the GUI or steps in a workflow. Key Words and Action Words may represent fairly low-level actions, such as the filling out of forms, or higher-level actions, such as the placing of an order. Besides manipulating the SUT, key words and actions words also exist to verify the state of the SUT.

An even higher degree of abstraction is provided by a category of tools for the automation of acceptance tests that do not present a test as a sequence of actions on the GUI but allow the declarative specification of tests where only the inputs and expected results of a workflow are specified. FIT (Mugridge & Cunningham, 2005b) allows for tests both based on actions taken on the testee and declarative tests. All FIT tests are represented as tables, tests can be edited by domain experts as long as they are familiar with the general concepts of spreadsheets.

Regarding the goals of this thesis, these established approaches for the automation of acceptance tests exhibit the following deficits:

- The different approaches are not based on internal models of the concepts of the application domain. The approaches are either based on an abstraction of the GUI or, in the case of FIT, use generic models based on tables. The lack of formal models on the function level hinders the transformation of test models and leads to the coupling with implementation details.
- The lack of models — as opposed to code or spreadsheets — is adverse to the reuse of test models for different systems within a system family. Without models of a sufficiently high level of abstraction, the functionality that is exercised by a test and thus must be present for all systems that are to be tested cannot be determined.

Given that Model-Driven testing relies on complex formal models that are not usable by domain experts and that most existing approaches to acceptance test automation are not build on a formal modeling approach, this work proposes the use of simple, test-specific models that support the construction of tests, not their automatic selection, generation and assessment.

1.2.3 Hypothesis and Research Question

The hypothesis underlying this work is that the model-based abstraction of tests and tested systems and the instantiation of an editor for acceptance tests facilitates the inclusion of domain experts in the construction of tests for the members of a system family.

We argue further that the generation of executable test code is feasible from the editor-constructed Test Configurations and that the quality of a tested system is improved because of the increased number of tests relevant to the users of the system. Further the necessary resources for test execution are lowered since test models can be automatically and repeatedly leveraged to generate and execute test code to exercise the testee. We consider the application domain of Digital Libraries, more precisely portals for scientific information, primarily bibliographic data from the social sciences.

In order to evaluate our hypothesis we present the MTCC approach. MTCC is implemented for a system family of four systems. MTCC uses feature models to represent the Services of the tested systems both on the level of the system family and on the level of individual systems. Actions and checks one these Services are also represented as feature models.

MTCC is based on a set of basic assumptions regarding the role of testing and the software development process:

- Automated, Model-Driven acceptance tests have a positive influence on the quality of a number of systems within a system family. Such automated tests facilitate continuous testing of systems with minimal expenditure of resources and support the formalization of existing requirements as tests.
- Within the examined application domain, there exists a system family that consists of a number of systems with similar requirements. The requirements shared by the various systems can be formalized as tests.
- Requirements for a system as well as the tests derived from these requirements can be decoupled from the specific implementation that is to be tested. Tests as well the functionality exercised by the tests can be expressed by abstract, formal models.
- The software development is characterized by different groups of persons with different roles in the software development process. The groups differ in their required skills and knowledge. While domain engineers are familiar with implementation technologies, domain experts have detailed knowledge about the functional requirements of a system.
- The quality of a software system under development benefits from the inclusion of domain experts. This is particularly true for the information retrieval domain where the inherent vagueness of concepts like relevance or information need necessitate human judgement.

We pursue the goal for MTCC to realize activities and artifacts that support the modeling of automated acceptance tests for systems in the context of a system family of Digital Libraries.

This goal leads to the basic question of our thesis: How can tests and systems under tests from a system family of Digital Libraries be represented in a way that supports (1) the instantiation of an editor for tests and thus (2) the construction of tests by domain experts and (3) the generation of test code?

1.3 Model-Driven Test Case Construction

We now sketch the MTCC approach (Baerisch, 2007), its underlying process and the artifacts used.

The main hypothesis underlying MTCC is that the use of formal models supports both the generation of test code and the execution of tests and the construction of tests by domain experts. Formal models can represent functions and functional requirements within the application domain while abstracting from implementation details. This allows for the high-level specification of tests and the decoupling from the specific interfaces used to exercise a given testee.

A precondition for the use of models to test software systems are transformations that support the generation of executable test code from the models.

MTCC stipulates a process that includes the analysis of a system family for test-relevant features both on the level of an individual system within the system family and on the level of the system family itself. This process is based on the foundations of domain engineering (Czarnecki, 2005; Eichmann, 1997).

The contribution of MTCC to the field of testing lies in the approach itself as well as in the implementation used for evaluation and in the results of the analysis for test-relevant features within the application domain.

In order to support domain experts in the construction of tests, allow the reuse of configured tests for different systems and support the generation of test code, MTCC includes a number of different activities and artifacts on the level of systems and the system family that we discuss in the remainder of this chapter. Before the MTCC testing process is covered however we discuss the different categories of tests that MTCC must support and how the apply to Digital Libraries.

1.3.1 MTCC Tests Categories for Digital Libraries

MTCC supports three categories of tests. *Technical* tests verify that a system can be executed without obvious errors, such as abnormal terminations. *Functional* tests verify that usage scenarios can be executed as expected by domain experts. *Information Retrieval* Tests are specific the the Digital Libraries Domain. These Tests verify that the SUT is capable to identify and return relevant documents for an information need. For the Digital Libraries, the categories above cover the following aspects:

- Technical Tests for Digital Libraries verify that no obvious errors like crashes occur. These tests also verify that the automated testing system itself is functional and that all operations necessary to execute the testee can be executed.
- Functional Tests verify that the functionality of a Digital Library conforms to the expectations of domain experts and that the metadata such a system is complete and correct. The following scenarios would be covered by functional tests.
  - A search for a known document returns the expected document, the metadata for the document is complete and correct.
  - The list of results can be manipulated as expected. Changes to the ordering of the list for example return the list in the right order.
- Information Retrieval Tests a kind of functional test that are specific for digital libraries. This tests verifies that an information need, expressed as a query, can be satisfied by a SUT. In MTCC, Information Retrieval Tests are are represented as Information Retrieval validation scenarios for which user-specified Recall or Precision values have to be reached.

MTCC supports *system-centric* Information Retrieval tests (Manning *et al.*, 2008) that consider individual queries and the results returned for that queries. *User-centric* tests are not supported because MTCC abstracts from the specifics of the GUI of the SUT.

A test in MTCC is successful if all Test Steps specified for the test can be executed on the SUT and if the behavior of the SUT conforms to all assertions specified in the tests. A tests fails if the execution of any Test Step instance is not possible for technical reasons or if the system behavior deviates from the behavior expected from the system and expressed in the test.

1.3.2 Overview of the MTCC Approach

Figure 6 illustrates the process of modeling a test in MTCC from the perspective of a domain expert. The figure also illustrates the generation of test code and the execution of the generated test cases by the test runner.

In a first step, the domain expert uses the editor to construct a Test Configuration. Such a Test Configuration is a fully specified model for a test. The Test Configuration then serves as input for the Test Generator. This is a code generator that outputs a test case. The test runner executes this test case on the system under test.

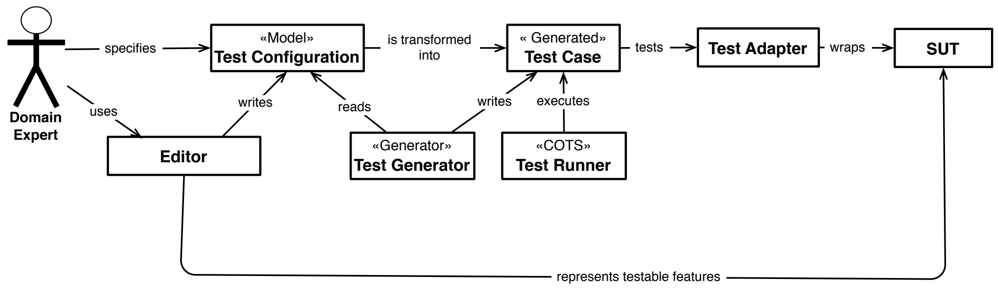


Figure 6: Overview of the test construction process

The editor facilitates the construction of tests by domain experts without programming or modeling skills. It represents a model of the testee and the available tests in a simple GUI and abstracts from implementation details as they would be needed to manually implement a test case. The editor only supports the construction of those tests that can in fact be executed on the system represented by the models used by the editor. The editor supports the test construction process by partitioning a test into a number of Test Steps. Each Test Step represents a single action or assertion that a user should be familiar with, from his or her past interaction with the system under test itself.

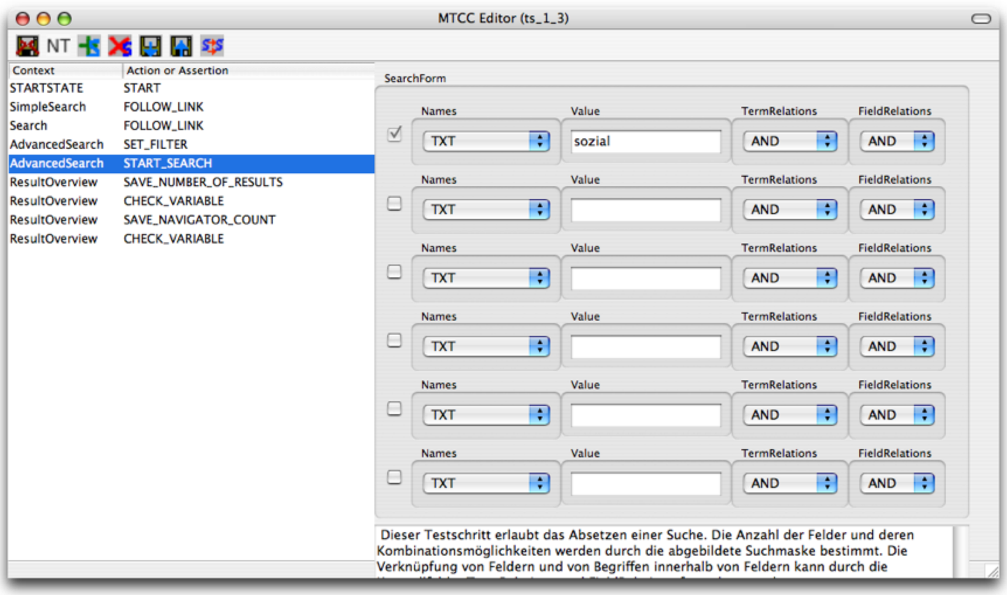
The editor displays the possible configurations for each Test Step. Since the various systems in the system family differ with respect to their test-relevant features and supported interactions, Test Steps for the same actions will have differences in their possible configurations. If for instance one Digital Library supports more options for searching than another, these differences will manifest themselves into corresponding differences, for the Test Steps representing the sending of a query to the system. As the editor only displays those Test Steps that can be executed on a system, it also only displays legal configurations for the available Test Steps.

Figure 7 displays a screen shot of the editor. The Test Steps of the current tests are displayed in the left half of the screen, the GUI elements that represent the possible configurations of the currently selected Test Step are displayed in the right part of the screen.

A Test Configuration that has been constructed with the editor is an abstract model, it cannot be used to exercise and verify a testee. The transformation of the Test Configuration into an executable Test Step is a separate step in the MTCC testing process.

The Test Generator transforms the Test Configuration into a test case. A test case is an executable test that can be executed by the test runner. Because test cases for different test runners or testing frameworks can generated, COTS (Commercial of the shelf) (Morisio & Torchiano, 2002) testing frameworks or test execution systems developed by the testing organization can be used.

The test runner is not a MTCC artifact but a COTS software system that is used by MTCC. MTCC can generate code for different test runners depending on the one that is best


MTCC Editor (ts_1_3)

Context	Action or Assertion
STARTSTATE	START
SimpleSearch	FOLLOW_LINK
Search	FOLLOW_LINK
AdvancedSearch	SET_FILTER
AdvancedSearch	START_SEARCH
ResultOverview	SAVE_NUMBER_OF_RESULTS
ResultOverview	CHECK_VARIABLE
ResultOverview	SAVE_NAVIGATOR_COUNT
ResultOverview	CHECK_VARIABLE

SearchForm

	Names	Value	TermRelations	FieldRelations
✓	TXT	sozial	AND	AND
☐	TXT		AND	AND
☐	TXT		AND	AND
☐	TXT		AND	AND
☐	TXT		AND	AND
☐	TXT		AND	AND

Dieser Testschritt erlaubt das Absetzen einer Suche. Die Anzahl der Felder und deren Kombinationsmöglichkeiten werden durch die abgebildete Suchmaske bestimmt. Die Verknüpfung von Feldern und von Begriffen innerhalb von Feldern kann durch die

Figure 7: Representation of Test Steps and their parameterization in the MTCC editor

suited for the system family investigated by MTCC. The execution of the generated tests is independent from the MTCC testing process, when and how often tests are executed is determined by the testing regime in place for the SUT, not by the MTCC approach.

Since the Test Configuration is an abstract model, it does not include information about the implementation of the system under test. MTCC provides this information in the Test Adapter, a library that is used by the test script and the test runner to exercise the SUT. The Test Adapter encapsulates implementation details for each Test Step supported by the system under test. MTCC uses a different Test Adapter for each pair of SUT and test runner.

The last step, the execution of tests, is equal to test execution in conventional approaches to test automation and decoupled from MTCC both in terms of the underlying technology and their organization.

1.3.3 Phases and Activities in the MTCC

From a users point of view, the test construction and test execution process in MTCC can be regarded as two user-visible phases: (1) The modeling of abstract test models and (2) the generation of test code and the subsequent execution of these test cases. Such a user-centered view on testing, however, does not cover all activities necessary for testing with MTCC. The user-visible phases depend on the existence of a modeling language for the description of tests and the availability of infrastructure that allows for the creation, maintenance, and transformation of these models.

The MTCC testing process structures the activities above and those needed to implement the necessary infrastructure for testing in three phases, each of which is defined by a number of activities.

Figure 8 illustrates the different phases and activities of the MTCC testing process.

We call the test construction and execution process *Test Engineering*. The focus of Test Engineering is the development of tests for one specific system. The activities necessary for the realization of the infrastructure and the modeling languages at the system family and system level are part of *Domain Engineering* or *Application Engineering*.

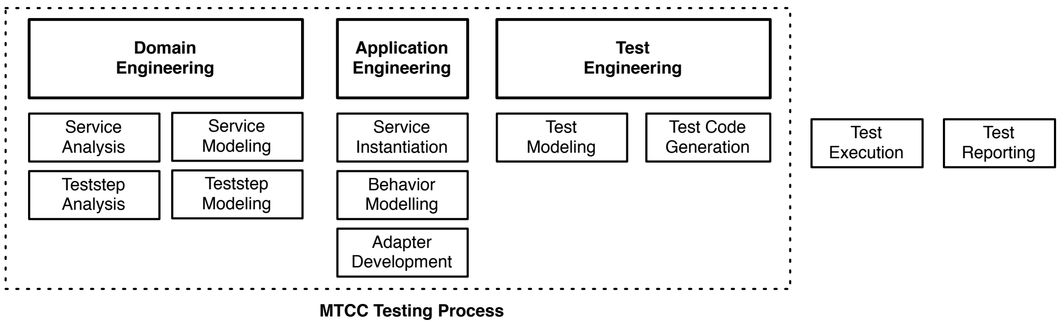


Figure 8: Phases and activities of the MTCC testing process

- Domain Engineering considers all SUTs in the tested system family. Hasselbring (2002a) defines Domain Engineering in the context of component-based software engineering as *an activity for building reusable components, whereby the systematic creation of domain models and architectures is addressed.* This definition can also be applied to the testing models and infrastructure used in MTCC. The goal of the Domain Engineering phase in MTCC is the analysis and modeling of the Services of the system family that are relevant for testing and of the Test Steps that are needed to interact with these Services. A Service in MTCC is a limited aspect of the functionality of a system in a certain Context. A search form that can be used to submit queries to a Digital Library and the list of potentially relevant documents returned by the system would be examples of Services. Test Steps represent actions that can be taken on a Service for example counting the results in a list of documents or changing the ordering of such a list.

- The goal of *Application Engineering* is the representation of one specific system from the system family considered in Domain Engineering. MTCC represents testees as a set of Service instances that are arranged in Contexts. A Service instance is a specialization of one of the Services identified and modeled during Domain Engineering. For the search form Service described above, a Service instance would describe the exact fields and operators available for one particular search form. A Context in MTCC is a set of Services, each of which can be independently exercised. The result overview presented after a search by a Digital Library is a Context that includes Services like the results list for the search, information about the search and links to other pages of the application. The behavior modeling activity of Application Engineering describes how exercising the Services of a system affects its currently active Context. For example, when a search is submitted to a system, the current Context of the system changes from a Context containing an instance of the search Service to a context with a Service instance that represents a list of result documents.

  Adapter development is the realization of a code library that is used by the test runner and the generated test cases to interface with the system under test.

- The purpose of Test Engineering is to construct tests based on the models implemented in Domain Engineering and Application Engineering. Test Engineering consists of two activities, Test Modeling and Test Code Generation. Test Modeling is done by domain experts using an editor that is instantiated based on the Service and behavior models for system realized in Application Engineering and the Test Steps realized in Domain Engineering. The editor uses standard GUI-widgets to represent the possible Test Steps that can be executed on the SUT at any given time as well as the legal configurations for this Test Steps. The result of Test Modeling is set of Test Configurations, high-level models of tests. These high level models are transformed into executable test cases during test code generation.

- The final activities of testing, Test Execution and Test Reporting, are not part of the MTCC approach. Once test cases are generated, test execution is no different from any other automated testing regime. MTCC does not assume that a particular test runner or a specific process is employed.

1.3.4 Modeling of Systems and Tests in MTCC

Figure 9 illustrates the models used in the MTCC test process, the interaction of the different roles in the development process with the models as well as the various artifacts that use the models.

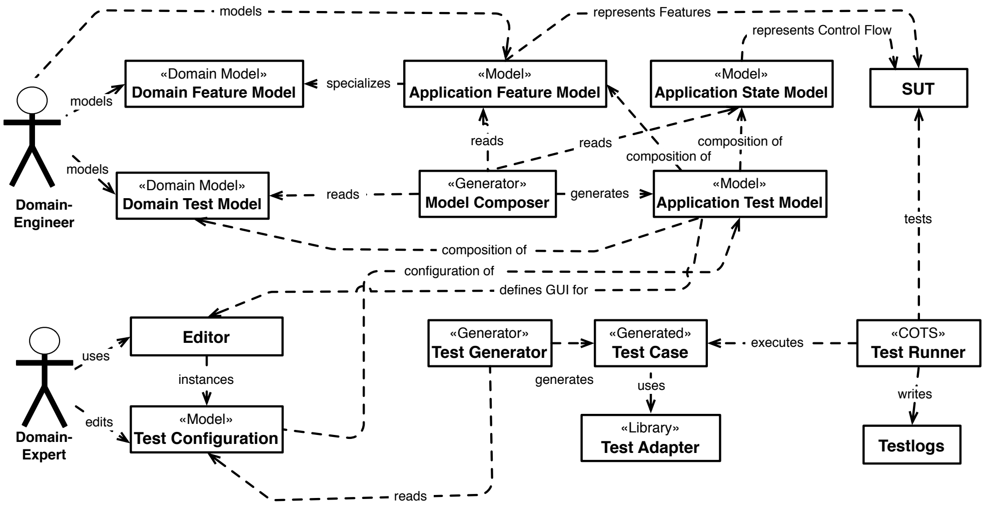


Figure 9: Models and roles in MTCC

The construction of Test Configuration-instances using the editor is based on the Application Test Model, a feature model (Kang *et al.*, 1990) that describes all tests that can be executed on the testee represented by the model and the valid configurations for these tests.

The Domain Feature Model describes all Test Steps that are relevant for the system family. Each Test Step references a Service that is exercised by the Test Step and includes a description of the parameterization of the Test Step. The Domain Feature Model is collectively realized by domain experts and domain engineers. The model is realized before the testing of a system starts but incremental work on the models continues during the whole testing process. The model can be described as a catalog of test-relevant actions that can be taken on the Services of the systems and thus the systems themselves in the system family under consideration.

Modeling of SUTs

The Domain Feature Model represents all test-relevant Services for the system family. Each Service is represented as a feature model that formalizes the variability between different instances of the Services. The focus of the Domain Feature Model is on the representation of the testable interface of each Service In contrast to the utilization of feature models in Generative Programming (Czarnecki & Eisenecker, 2000), feature models in MTCC are used in a descriptive way to represent an existing system, not in a prescriptive way to describe a system to be built. The Domain Feature Model is realized by the domain engineer and the domain expert based on the Test Steps identified for the Domain Test Model.

The Application Feature Model is a specialization (Czarnecki *et al.*, 2005b,a) of the Domain Feature Model. It represents the specific Service instances of one system from the considered system family as a feature model. The Application Feature Model and Application Test Model are realized by a domain engineer. Since MTCC assumes that domain engineers have the necessary skills to realize this model without special tool support, MTCC does not postulate the use of an editor for the realization of this model.

Test Modeling

The Application Test Model is a composition of the Domain Test Model, the Application State Model, and the Application Feature Model.

Each of the Test Steps represented in the Domain Test Model, contains references to the Service instance that is exercised by this Test Step. The use of these references decouples the Test Steps from the Services and supports the use of a single Test Step for multiple different instances of Services. In order to instantiate the Application Test Model, these references are resolved and Test Step instances are created for each Service in the Application Feature Model. Possible sequences of Test Step instances are derived from the Application State Model that represents the dynamic behavior of the SUT. The Application Test Model is an integrated representation of all tests that can be executed on a given system, the model is optimized for use by the editor to support a domain expert in the construction of tests.

The result of test modeling by a domain expert using the editor is a Test Configuration. A Test Configuration is a sequence of configured (Czarnecki *et al.*, 2005b,a) Test Step instances as defined by the Application Test Model. Each Test Step instance is a feature model without any remaining variability.

Test Modeling for Digital Libraries

After the introduction of the basic MTCC models above, we now discuss how tests for Digital Libraries and Information Retrieval systems in general can be expressed. We focus on the representation of the quality attributes of an Information Retrieval system by MTCC.

For the purpose of testing Digital Libraries, the particular quality attributes of such systems must be considered. One means to verify the ability of an Information Retrieval system to return relevant results for an information need is an evaluation based on specialized document collections and predefined information needs with given relevance assessments. Given such data, the information needs can be used with the system, and the IR system can be evaluated based on metrics for retrieval evaluation such as Recall/Precision (Baeza-Yates & Ribeiro-Neto, 1999; Manning *et al.*, 2008).

In addition to the effectiveness of the Information Retrieval aspect of a Digital Library, the quality of such a system as a whole is also determined by the quality of the information in the system (Mandl, 2008; Quix, 2003) and by its usability, especially the degree to which it fulfills the expectations of its users (Thurmeier, 2007; Heinold, 2007).

MTCC does not include any functionality to evaluate the usability of a system nor do we consider any extension or changes to the approach to add such functionality as useful. We argue that usability testing conflicts with the purpose of MTCC since MTCC is a Model-Driven approach that explicitly abstracts from the specifics of the user interface. In contrast to usability testing, the ability of an IR system to return relevant results for an information need as well as the quality of the information provided by the system can be evaluated within the constraints of the MTCC approach. Information needs are expressed as Test Steps that interact with the testee, for example by sending a query to the system. Recall, precision and data quality can be evaluated by Test Steps that examine the results returned by the tested system.

If the quality of IR for a system is to be evaluated with an evaluation collection, it is necessary for this collection to be part of the information contained in the Digital Library. The corpus must be searchable in isolation from all other documents in the Digital Library. When the effects of additions to the retrieval process such as the utilization of cross-concordances are to be evaluated, whether these additions are used must be controllable on a query basis. Both of the assumptions are not specific to testing with MTCC, they also apply for manual testing and other forms of retrieval evaluation.

The following steps illustrate how MTCC Test Steps can be employed for an Information Retrieval evaluation:

1. The set of documents from which the Information Retrieval system may select relevant documents is limited to those documents in the corpus used for evaluation.
2. The information need is expressed as a query. This encoded information need is sent to the system.
3. The documents that are returned by the IR system as the most likely to fulfill the information need are recorded. They are then compared with existing relevance assessments for the document collection in a Information Retrieval evaluation (Manning *et al.*, 2008).
4. Metrics of the retrieval performance of the system are calculated based on the results of the previous step.

MTCC provides a number of specialized Test Steps for the evaluation of IR effectiveness, for example to record the IDs of the documents in a list of results and to calculate recall and precision values.

1.3.5 Test Code Generation

Since the Test Configuration models specified in the test modeling phase abstract from the implementation specifics of a system under test and from the test runner used for test execution, they are not suitable by themselves to be executed.

The purpose of the Test Generator is the transformation of Test Configuration models into tests that can be executed by the test runner. MTCC uses a simple code generation approach based on the Templates and Filters Stahl & Völter (2006) method. Generated tests use the routines in the Test Adapter library to actually execute a test. Figure 10 gives an overview of the test generation process in MTCC.

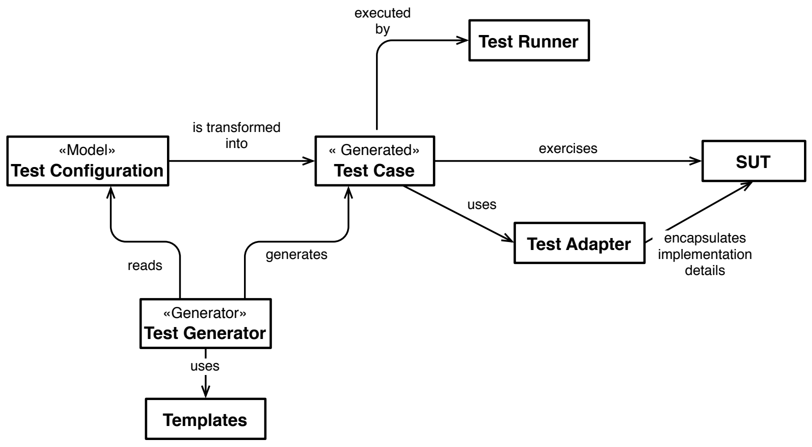


Figure 10: Test code generation in MTCC

In order to allow the generation of test cases from abstract Test Configuration models and the execution of the generated test scripts on a SUT, the templates used by the Test Generator and the Test Adapter must provide the following information about the TestRunner and the SUT:

- Information about the interface by which the SUT will be exercised. Testees may be exercised via Web Services, HTML/Javascript interfaces, GUIs, or other interface concepts. In order to execute a test a mapping is needed for all Test Step instances to the appropriate interactions with the interface of the SUT. In MTCC, this information is provided at runtime by the Test Adapter.

- Information about the structure and implementation of the test cases, for example, about the xUnit implementation used. This information is provided by the Templates.

MTCC utilizes a different Template and Test Adapter for each pair of TestRunner and SUT. Template and Test Adapter are implemented in the Application Engineering Phase. By concentrating all implementation details necessary for test execution on these artifacts, changes to the interface of the testee are decoupled from the test models.

1.3.6 Test Execution

Test execution is not part of the MTCC approach insofar as no particular test regime or test execution infrastructure is required or assumed. The Test Runner is not an artifact of the MTCC approach but COTS software. The only condition that a test runner must meet to be usable in the context of MTCC is that it must be capable of exercising the GUI and that the generation of test code for the test runner must be possible.

The decision not to implement a specific test runner for MTCC was made because a great number of COTS test runners already exists. It is unlikely that a test runner specific to MTCC would be more effective or efficient than existing systems, especially since such a test runner would need to replicate much more functionality than is already implemented in such systems.

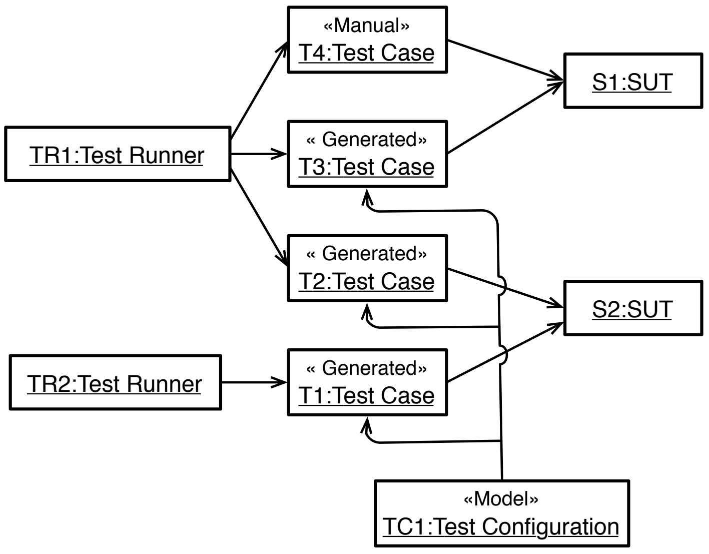


Figure 11: Execution of MTCC tests by multiple test runners is a system family context

The fact that the MTCC approach can be used with various test runners is beneficial to both the effectiveness and efficiency of testing in multiple ways. Figure 11 illustrates a scenario where four test cases are executed on the systems by two different test runners.

- Test scripts generated by the Test Generator can be executed in combination with test cases that are realized by hand. This allows the integration of MTCC into an already established test regime since test cases can be generated for the test runner already in use. The same reporting and bug-tracking tools can be used for MTCC tests and regular tests.
- A Test Configuration can be used to generate test cases for different test runners. Such a scenario can arise when one test runner exercise the SUT via the GUI while another uses the API.

1.4 Structure of the Thesis

This thesis consists of four parts. Part I introduces the foundations of our work and gives an overview of relevant fields of research. Part II discusses the MTCC approach and its application to digital libraries in detail, Part III introduces the MTCC validation. Part IV concludes.

Part I introduces the fields relevant to MTCC. Chapter 2 discusses the role of testing in software quality assurance. The purpose of testing and various testing methods are considered, special focus is given to acceptance tests and test automation. Chapter 3 discusses the role and application of models in software engineering. Based on the definition of models as formal abstraction with a specific purpose used in the context of this thesis, we introduce Model-Driven development and Model-Driven testing. Chapter 4 considers software product-lines and software families. The concept of variability is introduced and feature models are explored as a means for the systematic treatment of variability. The consequences that software families and software product-lines have on the testing process are examined. Chapter 5 introduces Digital Libraries as the application domain of MTCC. The purpose of Digital Libraries as a specialized information retrieval system is discussed. The meaning of the concepts information need and relevance for the definition of quality in the Information Retrieval context is discussed in relation with its implications for testing.

Part II of this thesis presents the MTCC approach and its underlying process in detail. The models and artifacts that are utilized in the context of MTCC are described. Chapter 6 gives an introduction to the testing process with MTCC and the roles relevant to this process. The activities of domain engineers and domain experts during the different phases of test construction and the purpose and interdependencies of the various phases are discussed. Chapter 7 gives a detailed treatment of the various models used in MTCC. The derivation of system-Level models from system family-level models is examined, as is the composition of models used for the representation of systems with those representing the Test Steps relevant for the system family. Chapter 8 examines the question how the editor facilitates the MTCC models to support the construction of tests. The representation of test models using a graphical user interface is examined. Further topics of the chapter include the generation of test code and the reuse of Test Configurations for different systems within a system family. In Chapter 9 we present the application of the MTCC modeling approach to a system family of Digital Libraries. The members of the Digital Library are introduced and the Services and Test Steps relevant for testing the system family are examined.

In Part III, we present the evaluation of MTCC and discuss related work. Chapter 10 and Chapter 11 examine the MTCC plan and the conducting of the MTCC validation. Based on the GQM approach, we derive questions that verify both the plausibility of various claims on the approach and its usability. The validation process and its results are introduced, the result are discussed in terms of their consequences with respect to our hypothesis. Chapter 12 discusses related work. We analyze the relation of other approaches to acceptance test automation or testing in a system family context to MTCC and point out similarities and differences.

Chapter 13 in Part IV summarizes the results of this thesis and discusses out contributions. Chapter 14 concludes and gives an outlook of future work.

1.5 Chapter Summary

High quality software depends on the involvement of domain experts in the software development process. This involvement is especially significant for Digital Libraries. Digital Libraries are Information Retrieval Systems whose quality is in part defined by the degree to which they can satisfy the vague information need of an user. One way to involve domain experts in the software development is acceptance testing. In automated acceptance testing, domain experts specify automated tests in a language understandable for non-programmers. Automated testing facilitates more efficient testing and leads to more repeatable results than manual testing.

MTCC is an approach that facilitates automated acceptance testing in a system family of Digital Libraries. MTCC can express technical tests that verify that a system works without obvious errors, functional tests that allow the testing of a SUT against test scenarios and information retrieval tests that compare the result returned by a Digital Library against a collection of documents known to be relevant.

A system family is a set of systems that are defined by their common properties. In contrast to Software Product Lines, the members of system families are not based on a common set of Core Assets that can be leveraged to support testing. MTCC addresses the variability in a system family by representing the testable features of the members of a system family with feature models. Feature models are used to instantiate an editor that supports the construction of automated tests by domain experts.

The MTCC process consists of three phases: Domain Engineering addresses the analysis and modeling of the common and variable test-relevant features of a system family. In Application Engineering, models for specific systems are derived from the domain level system and the testing infrastructure for the system is implemented. Abstract tests representations are constructed by domain experts in Test Engineering. Code generation is used to transform these test representations into executable test cases.

Part I

Foundations

Chapter 2

Testing of Software

Testing is defined as the execution of a system with the purpose of finding faults in the system. It encompasses a wide range of different approaches with different goals and motivations. In this chapter, we discuss both the role of testing for software quality and the role of MTCC in the field of testing. We start with an examination of the role of testing in Section 2.1.

In order to be more efficient and effective, often, various parts of the testing process are automated. The extent, to which testing is automated, varies with different approaches, but the execution of tests implemented as test scripts and the recording of test results form the core of test automation. Section 1.1.4 examined the goals of test automation and the challenges that must be met to establish an automated testing regime.

Acceptance tests involve domain experts in system testing. Section 2.2 motivates the involvement of domain experts into the testing process and discusses acceptance tests and their automation.

The quality of a test is defined by its ability to find faults. Tests must therefore be based on the requirements as defined by the domain experts for a system and must be able the assess a testee based on this requirements. Section 2.3 discusses the meaning of quality for tests and test software and applies the quality attributes of software engineering to test software.

2.1 Testing and Software Quality

Testing is a method of quality assurance that verifies the behavior of a system against a set of requirements that are expressed as tests. The notion of quality and its relation to the requirements of a system, the specifications that represent these requirements and the needs of the users of a system are central to testing.

If quality is defined as the capability of a system to fulfill the requirements of its users or customers (Juran & Gryna, 1988), the question must be considered how these requirements can be captured and formalized as tests. One source of requirements are domain experts. Domain experts are customers or users of a system who are familiar with the functions that the systems needs to fulfill and who are often experienced in the use of similar systems.

The direct and continuous involvement of domain expert in the software development process as a methodology belongs to the group of agile approaches to software development (Beck & Andres, 2004) but is considered as part of other approaches to software development as well (Evans, 2004).

One method to make the knowledge of domain experts available for the software development process is the definition of acceptance tests. In acceptance tests, domain experts either test a system themselves or specify the functionality to be tested and the conditions for the success of a test in a format that is either suitable for execution by a human tester or for execution by an automated testing system.

One purpose of test automation is to allow the execution of tests without human involvement. Automation is desirable both for economic reasons and to allow for greater effectiveness in testing.

2.1.1 Software Quality

Juran & Gryna (1988) defines quality as *fitness for use*. This definition raises a number of questions: fitness for what use? What constitutes fitness? How can it be determined, if and to what degree a system is fit to be used? In order to evaluate the fitness of a system, some sort of specification is needed. A specification is a document that describes the requirements for a system. The quality of a system can also be defined as the degree to which its implementation complies with its specification. The specification must therefore reflect the requirements for a system.

Functional as well as non-functional quality attributes are relevant to software systems. Software engineering defines a number of methods that serve the purpose of creating a high-quality products or evaluating its quality.

Different Notions of Quality

Two questions have to be addressed in the discussion of quality assurance. At first, quality has to be defined. Based on this definition, methods and tools have to be identified that allow for the evaluation of quality and thereby provide a prerequisite to increase quality.

Two basic notions of quality exist (Reeves & Bednar, 1994): The view that quality is the adherence to a specification or that quality is the degree to which a product or Service is useful to its users or customers. The definition of quality given by Crosby (1979) is exemplary for the former view on quality while Edwards (1968) and Juran & Gryna (1988) are proponents of the latter. We call these different notions of quality specification-centric and user-centric.

Applied to the field of software engineering, the specification-centric and the user-centric notions of quality are reconciled (Kan, 2002) when the specification of a system represents the needs of its users completely and correctly. The quality of a specification is instrumental for the quality of the system built based on this specification. The quality of a specification is determined by the degree to which it reflects the users requirements, needs, and to a certain degree wishes. In order to build high-quality systems, both high-quality specifications and ways to compare a specification and the implemented systems are needed.

Validation is the activity to ensure that *the right things are done*, that is, a specification represents the requirements of a system. Verification aims to ensure that *things are done right* and that a system reflects all points of the specification. Testing is foremost a verification activity. It systematically compares a specification with a system; tests are examples for the expected behavior of the system defined by the specification.

We argue that tests can also serve as a partial specification, a *specification by example* (Fowler, 2006), in particular in situations where the actual specifications are incomplete or missing. In such situations, tests cannot be derived from specifications but must be derived from the requirements of a system and must be specified by users or customers. It follows that such acceptance testing by users who are experts for the subject domain is a validation activity.

Attributes of Software Quality

The concept of software quality has multiple dimensions or attributes. The ISO standard 9126-1 (for Standardization, 2001) defines the quality of software along six attributes: Functionality, Reliability, Usability, Efficiency, Maintainability, and Portability.

We distinguish between functional and non-functional quality as well as between user-observable quality and not user-observable quality (McConnell, 2004):

- Functional quality is the capability of a system to correctly fulfill a specific task. Non-functional quality attributes describe general conditions that the system must meet, for instance limits to the time and resources used or the reliability of a system. The final five of the six ISO quality attributes given above describe the non-functional qualities of a system.
- The user-observable quality of a system includes those attributes that effect a user working with the system. User-observable quality subsumes both functional and non-functional quality attributes but excludes quality attributes such as maintainability or server-side resource usage that are not visible to a user.

The notion of non-user-observable quality appears contradictory based on our definition that quality is, either direct or indirect, the degree to which a system meets the requirements of its users. It is indeed the case that quality attributes like maintainability are not visible to a user and are yet crucial for the sustained quality of a system. The existence of non-user-observable quality attributes is one reason why acceptance tests are only a partial specification. We argue that this limitation does not challenge the suitability of tests as a specification for the user-observable quality attributes

Methods of Quality Assurance

There is no method or tool in software engineering that can serve as a silver bullet (Brooks, 1987) and guarantee high quality throughout the whole software development process and for all products. As a consequence, software engineering employs a bundle of techniques that support the construction of high-quality artifacts or evaluate the quality of existing products and processes.

Among the measures that directly influence the quality of a product under development are methodologies and software process models, for instance the V-Model (Broy & Rausch, 2005), the Rational Unified Process(Kruchten, 2003; Larman, 2004) or agile methods (Beck & Andres, 2004) and the utilization of established best practices like design patterns (Buschmann *et al.*, 1996; Gamma *et al.*, 1995) as well as the use of tools that support the software development process like version control systems (Pilato *et al.*, 2004).

While the measures above address the creation and increase of quality in the product development process, complementary means are necessary to verify the quality of a product and identify quality deficits. Testing is one means for this purpose.

2.1.2 Definition of Testing

A basic question about the role of testing is whether it serves to ascertain the quality of a product or identify faults. Fault-oriented testing and conformance-oriented testing are contradictory views on the role of testing and give rise to the question exactly how testing benefits the quality of a software system and what characterizes good testing.

Conformance and Fault-oriented testing

Conformance-oriented testing is based on the assumption that the purpose of testing lies in proving or at least demonstrating the quality of product. Approaches that advance a conformance-oriented view of testing usually place testing towards the end of the development of a system or subsystem, for example, at the end of a naive implementation for the waterfall (Royce, 1987) model of software development.

The focus of fault-oriented testing, in contrast, lies on the identification of faults in the system. Since testing can never prove the absence of faults (Dijkstra, 1969), the fault-oriented view on testing only regards those tests as useful that find faults (Myers *et al.*, 2004).

The current view on testing is a compromise of the conformance-oriented and the fault-oriented view. While the notion that correctness cannot be proven by testing is not challenged, the usefulness of tests that do not find faults is also accepted (Chen, 2005). While, for instance, regression tests are not expected to find faults in a system, they provide an assurance against faults that were introduced in existing systems.

While tests cannot prove the absence of faults, high test coverage of a code base both increases the probability to find faults and allows for some degree of confidence about the likely faultiness of a system or subsystem.

Beside the fault-oriented and conformance-oriented views on testing, yet another view is held by the Test Driven Development (Beck, 2002; Janzen & Saiedian, 2005) approach. In TDD, tests serve to clarify the requirements that exist for a software and ensure that the intended behavior of an implementation artifact matches its actual behavior. The goal of testing in TDD is not to find faults but to guide the development of a system. *"One of the ironies of TDD is that it isn't a testing technique. It's an analysis technique, a design technique, really a technique structuring all the activities of development"* (Beck, 2002).

In the context of MTCC we advance the view that automated acceptance tests both serve to describe the intended behavior of a system and to find faults in a system. MTCC tests also serve as regression tests and thus, under the precondition that test coverage is sufficiently high, provide some confidence that a system is free of defects. MTCC shares one aspect with TDD in the fact that tests serve as a means to express requirements for a system, it is however no test-first approach since a testable system must exist prior to acceptance testing. Also, acceptance testing with different roles does not support the *red-green-refactor* cycle that Beck describes as the mantra of TDD. We regard testing with MTCC as a supplement to fault-oriented testing that raises quality both by supporting acceptance tests and by facilitating to identify conflicts between the requirements of domain experts and an implementation.

Quality of Tests

The ability to identify faults in a system and to provide means to assess the likeliness of faults with some confidence do not in themselves lead to criteria to judge the quality of a test. In addition to the effectiveness of a test, its ability to find faults, non-functional criteria like efficiency and maintainability also contribute to the quality of a test.

Graham & Fewster (2000) introduce four attributes of quality for automated tests that can be applied to tests in general: A test is *effective*(1) when it is likely to find faults in a system, it is *exemplary*(2) when it does not only cover highly specific functionality of a system but tests multiple aspects of a system at once.

The resources needed to design, implement, execute and debug the test describe how *economical*(3) a test is. A test is *evolvable*(4) when it can be maintained and adapted to changes in the testee with high costs.

Figure 12 illustrates how the four quality attributes defined by Graham and Fewster apply to manual and automated testing. The attributes help to illustrate that testing is an activity that has economic as well as purely functional aspects. The economic side of testing is one of the main motivations behind the automation of tests.

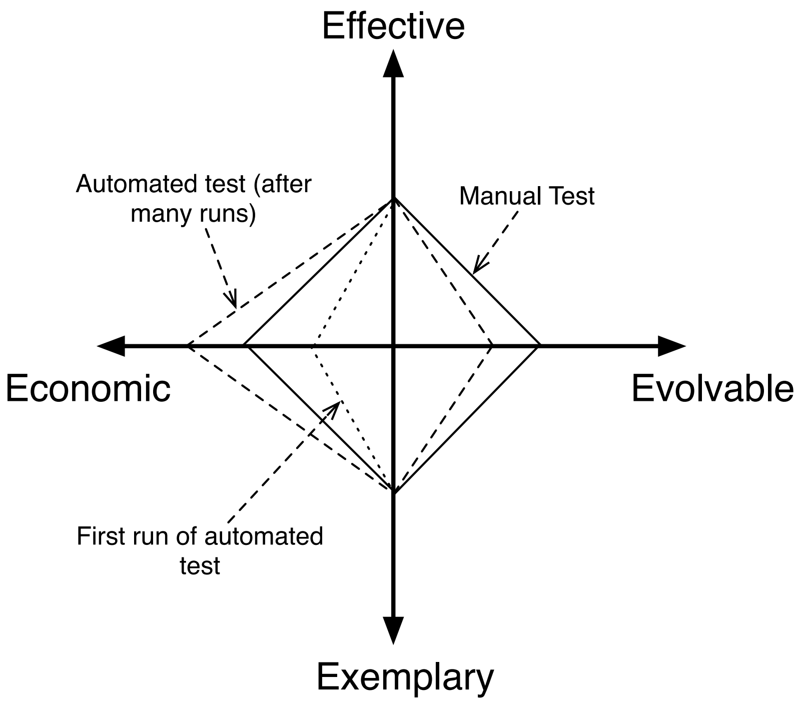


Figure 12: Attributes of good tests (Graham & Fewster, 2000)

2.1.3 Automated Testing

One goal of MTCC is the automation of acceptance tests. In the following we discuss in detail why we argue that such automation is beneficial and which activities of testing are suitable for automation. We regard test automation as advantageous with respect to both the effectiveness and the efficiency of testing, in particular in the context of regression tests. We argue that the subject knowledge and understanding for the requirements of a system that domain experts can provide is better suited for the testing of digital libraries than the complete automation of the test design phase. We therefore provide an editor to support the construction of tests.

Reasons for Test Automation

Since we regard testing as an activity that allows the assessment of the quality of a system during the whole software life cycle, it follows that testing and in particular test execution are activities that have to be supported and thus consume resources over the whole length of a product development and maintenance cycle. Test execution can be very frequent, continuous integration (Duvall *et al.*, 2007) approaches, for example, aim to execute each test at least daily.

Test execution with such frequency cannot be achieved manually in practice for a non-trivial number of tests, the automatization of test execution therefore is imperative. Test automation is advantageous for a number of reasons even when tests are less frequently executed (Graham & Fewster, 2000):

- Automated tests have economic advantages compared to manually executed tests. Manual testing requires human involvement and is thus costly while the execution of test cases will generally consume only minimal resources. For this reasons, the costs for the implementation of automatic testing will usually be lower than those for manual testing when tests are repeatedly executed.

- The execution of automated tests does not conflict with the day-to-day tasks of software engineers or domain experts. This focuses on the fact that the resources for manual test execution need to be available in all organizations and all phases of the software development process. If, for instance, manual test execution is done by software developers, test execution and software development will both compete for the time of this person. While the implementation of test automation software will also require resources from a software developer, this is a one-time effort.

- Automated tests are potentially more effective than manual tests. Automated tests support the execution of more tests more often. Such tests are also more reproducible than manual tests since no variances are introduced into the test execution process by human testers.

The above claim that a higher frequency of test execution, a greater number of tests and better reproducibility result in an overall increase in test effectiveness requires further consideration. We argue that such an increase occurs because of the following points:

- The number of tests and the frequency with which they can be executed by automated test software is not achievable by manual test execution. The number of automated tests that can be executed in a given amount of time is limited only by the available computation resources. While the number of executed tests alone has little influence on the ability of a test regime to find faults, the ability to frequently execute all tests for a given system greatly increases the effectiveness of regression testing.

- Automated tests are reproducible (Binder, 1999). Using test automation it is possible to exactly specify and repeat the inputs send to test the testee during test execution. This allows for the exact manipulation of the state of the testee and supports the execution of repeatable tests.

- Automated tests support the verification of quality attributes that can not be tested by manual tests in practice. Load tests that require a large number of requests to be send to the testee within a short period of time can not be realized without test automation.

We conclude that test automation has high potential to increase the effectiveness and efficiency of testing. Binder goes so far as to call automation the central aspect of testing:*"(Testing) is the design and implementation of a special kind of software system: one that exercises another software system with the intent of finding bugs"* (Binder, 1999). We advance the view that test automation primarily benefits the efficiency and effectiveness of test execution but that explorative testing (Kaner *et al.*, 2001) is necessary in addition in order to identify new tests for automation. It is therefore one purpose of MTCC to ease the automation of test cases identified by explorative testing by providing an environment for test modeling.

Phases of Test Automation

Our discussion of test automation so far was limited on the automation of test execution. This is justified by the fact that test execution is the phase of testing that is most often automated. Phases like test design and test implementation are far less frequently automated in practice.

Fewster and Graham (Graham & Fewster, 2000) partitions the automated testing process in five phases: The identification of test scenarios, the design of tests, the implementation of test automation, the execution of tests and the comparison of actual and expected results. Figure 13 illustrates these activities and their potential for automation:

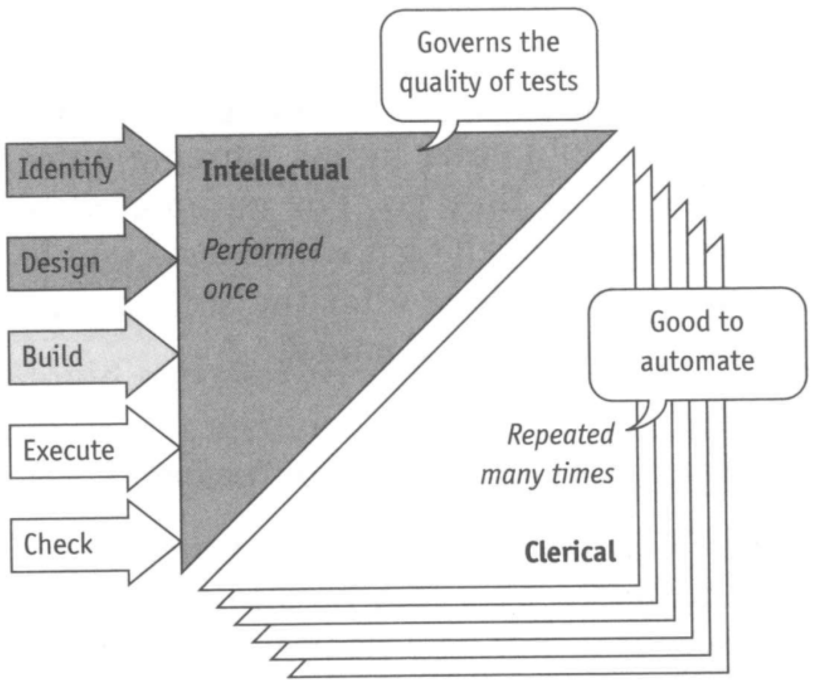


Figure 13: The five phases of testing (Graham & Fewster, 2000)

- The purpose of the Identify phase is the identification of functionality that can be tested.
- The goal of the Design phase is to produce concrete tests for the functionality identified during the Identify phase. Input values and expected results are specified for each test.
- During the Build or implementation phase, the tests designed in the previous phase are implemented in a notation suitable for automatic execution, for instance in a scripting language.
- The Execute phase sees the execution of the previously implemented tests on the system under test.
- The purpose of the Check phase is the evaluation of the success of the test based on a comparison of the expected and the actual behavior of the SUT.

As Figure 13 illustrates, the various phases of testing vary in their suitability for test automation. The automation of the test execution phase and the comparison of actual and expected results can generally be automated without significant problems, the automation of this phases also results in large practical benefits since both phases are frequently executed in the testing process.

MTCC automates the test execution and the Comparison phase. The build phase is also automated once the Test Adapter and the necessary templates are implemented. The design phase is not automated but is supported by the editor.

Automatic and Automated Testing

In order to distinguish MTCC from other approaches for test automation, in particular from Model-Driven testing (discussed in Section 3.3), we differentiate between automated testing and automatic testing.

The purpose of automated testing is the automation of test execution and comparison and in some cases the build or test implementation phase. Automatic testing also aims at the automation of the test design phase.

We consider automated testing not as a partial implementation of automatic testing but as a separate field. Since test design in automated testing is done by humans, the support of intellectual test design, not its automation, is the focus of the effort. Specifically, automatic and automated testing differ in the following points:

- Automatic testing requires a formal model of the testee (Bertolino *et al.*, 2004) that represents all states of the testee as well as events that can influence this state, automatic testing is therefore always also Model-Driven testing.

  Automated testing in contrast is based on the knowledge of domain experts; the approach does not require models at the level of detail necessary for automatic testing. We argue that automated testing is better suited for situations where testing is mostly based on the requirements of users, not formal specifications.

- Automated and automatic testing do not differ in test execution. Both approaches use an automated test runner to execute a test case without human involvement.

- A relevant difference between automated and automatic testing is the approach used to determine the success of a test. In automated testing, the conditions for the success of a test are defined by a human during test design, automatic testing utilizes an oracle (Baresi & Young, 2001) for the decision if a test was successful or not. Depending on the testee and the conditions to be tested, the complexity of an oracle can vary widely. An oracle that verifies that the testee did not crash during test execution is far less complex than an oracle that verifies complex business processes (Memon *et al.*, 2003). If an alternative implementation of the system under test exists, it can serve as an oracle in the context of differential testing (McKeeman, 1998; Evans & Savoia, 2007). The fact that automated testing can rely on human intellect to describe the success criteria of tests allows for the testing in situations where the automatic determination of test success is not possible, for instance in the context of Digital Libraries.

We argue that automatic and automated testing supplement each other. We further argue that the inclusion of humans in the testing process leads to advantages that cannot be achieved by automatic testing: While automatic testing is dependent on the correctness and completeness of the specification upon which the models used for testing and the oracle are built, automated testing allows for the discovery of new requirements in the testing process.

2.2 Acceptance Testing

Given that the intent of testing as a verification activity is to ensure that the implementation of a system conforms to its specification, and that a specification is meant to represent the requirements of the users of a system, it appears to be beneficial to the quality of software to directly involve domain experts in testing and thus ensure that the tests executed on the system represent the real requirements to the system.

Domain experts are assumed to have detailed knowledge of the subject matter and the functional requirements for a system, but not to have any skills in formal modeling, software development, or software testing (Andersson & Bache, 2004; Andrea, 2004a; Mugridge & Cunningham, 2005b).

2.2.1 Testing by Domain Experts

Tests that directly involve domain experts in system testing are called acceptance tests (Mugridge & Cunningham, 2005b). Domain experts can participate in acceptance testing in a number of different ways, They can manually design and execute tests, design tests that are automated by software developers or use specialized tools for acceptance testing such as FIT or Selenium. One purpose of the MTCC approach presented in this thesis is to provide a tool that allows the specification of tests by domain experts in the context of system families.

We consider acceptance testing an activity in parallel to the software development process that starts as soon as a first testable prototype of the system is available. The advantage of such an approach over acceptance testing only at the end of the software development process is that the tests can facilitate discussions over the quality of the system and help discover previously unknown requirements.

Compared to the evaluation of the degree to which a system meets its requirements purely based on a specification, testing has a number of advantages: Since testing is always specific for a concrete, existing system and since tests need to be detailed enough for automated execution, the preparation of tests supports the discovery of omissions or errors in the requirements better than a informal specification.

The inclusion of domain experts or customers in the software development process is explicitly demanded in agile methodologies like Extreme Programming (Beck & Andres, 2004), often in a testing role. The inclusion of customers in the testing is generally considered desirable (Armbrust *et al.*, 2004).

2.2.2 Types of Involvement

Since in most cases domain experts will not be professional testers nor software developers, domain experts cannot automate the execution of acceptance testing themselves by expressing their tests as test cases. They can instead participate in one of the following ways:

- Acceptance tests can be specified by domain experts in a informal way and are then implemented by software developers. The approach has the disadvantage that domain experts are not directly involved in the testing process. In order to ensure that tests represent the intentions of the domain experts, the test specification must be detailed enough to prevent misunderstandings. Such a specification must be as formal as the test it represents, it merely omits implementations details. As a consequence, test and specification are redundant.

- Acceptance tests are specified using a detailed but informal test plan, the execution of tests is done by professional testers. While there is no redundancy between a test implementation and the test specification — there is no implementation — this approach suffers from the disadvantages of manual test execution.

- Domain experts can test the SUT themselves, either by explorative testing or following a simple test plan. The main disadvantage of this approach is the increased work load on the domain experts, especially considering the fact that test will be executed frequently.

- Domain experts construct tests suitable for automatic execution themselves, using a tool usable without further training. This approach combines the advantages of automated testing and the involvement of domain experts in the testing process. This approach is followed by MTCC.

2.3 Quality of Test Automation

So far we have introduced our notion of quality for software and discussed criteria to evaluate the quality of tests, but we have not yet applied our criteria for quality to testing software itself.

In consideration of the fact that test automation software is software and thus can be judged in terms of its functional and non-functional qualities, we now discuss what determines the quality of automated acceptance tests. From the result of this discussion we derive the requirement of the MTCC approach.

2.3.1 Quality of Requirements and Quality of Tests

As always in software engineering, the quality that can be reached in any phase of the development of a system for testing is limited by the results of the proceeding phase. Without thorough requirements engineering, it is not possible to design the right system and without design, the results of the implementation are more than in question. This applies for incremental process models as well as the traditional V-Model.

The consequence for test software is, that while domain experts must be present at all phases of the development of test software, their presence is particularly important in the early phases of the development of test software when the functionality of a testee that can be tested and the basic capabilities of the test software are determined. The development of test software needs the cooperation of experts on the subject matter to be tested and on the implementation of the testee as described, for instance, in the concept of Domain Driven Design (Evans, 2004).

In the context of MTCC, this means that domain experts are not only involved in the construction of tests but also in the analysis of the application domain/system family and the individual systems within the system family.

2.3.2 Software Quality Attributes Applied to Test Software

The six quality attributes defined by the ISO standard 9126 can be applied to automated tests as well as to regular software. Non-functional quality attributes, such as the maintainability and usability, have a direct impact on the fitness of test software to verify the quality of a testee. In the context of software product-lines, it is of special importance that test software can be adapted to the specifics of different testees. In the particular case of MTCC, the usability of those aspects of the test software that are intended for use by domain experts are highly relevant to the ability of the system to test the members of the product family.

While the quality attributes for automated test systems are the same as those for regular software, special priorities apply. We argue that a system that aims to involve domain experts in the testing process must meet the following requirements:

- The functionality of the system must include the capability to cover all fault scenarios for the system family under consideration. The test automation system must include functionality to take a representation of a test in some formal notation, execute the test on the testee and decide on the success or failure of the test.

- In terms of usability, a system for the specification of automated acceptance tests for domain experts must consider a number of different aspects, the most important of which is the language or the editor that domain experts use for the specification of tests.

- The efficiency of an automated test system is defined by its ability to execute a large volume of tests with limited resources. The efficiency of the execution of an automated test depends on the resources necessary for the execution of the testee since mocking (Meszaros, 2007) is not possible in a system test context.
- Since test software needs to be adapted to changes of the testee that affect the testable interface of the testee, maintainability and adaptability are important attributes for such systems. Maintainability and adaptability are of special importance in a system family context when a sustainable test process must be established not only for a single system but for multiple different implementations. Beside the ability to follow changes in the testee, the test automation software must also be maintainable in the traditional sense, for example in order to support refactoring (Deursen *et al.*, 2001; Meszaros, 2007).
- An important quality attribute for a test software in a system family context is the ability to reuse tests. Reuse means that a test that was specified for one system can be transfered to and executed on another system if this system has the necessary functionality for the test to be executed.

We argue that all functional and non-function quality attributes are of high relevance for test automation software and that non-functional quality attributes have a significant impact on the overall effectiveness of such a system. For instance, we consider it highly unlikely that the specification of acceptance tests is practical if the user interface used for the specification is not usable. MTCC therefore aims to provide an editor that facilitates the construction of test cases.

2.4 Chapter Summary

Testing is one of the methods of quality assurance for software. It differs from other methods, for example Static Analysis, in the fact that it executes the code of the SUT and compares the encountered behavior of the SUT with its expected behavior. The quality of a test is defined by a number of factors. A prerequisite for a high-quality test is that it represents the requirements for the SUT and can be used to verify these requirements. A test must also be efficient — it must be possible to execute a test with minimal resources. One way to minimize the resources necessary for test execution is test automation.

Acceptance Tests involve domain experts in the testing process. Because domain experts are not professional software developers or testers, special tool support is necessary to support domain experts in the specification of automated acceptance tests.

As software systems, automated tests are subject to all software quality attributes defined in the ISO 9126. In addition to these quality attributes, some specific requirements apply to a test automation system in a system family background. Such a system must support tests that are reusable for multiple systems and can be adapted to changes in the SUT.

Chapter 3

Use of Models in Software Engineering

In this chapter we discuss the use of models in software engineering. We examine the relation of MTCC to the fields of Model-Driven software development and Model-Driven testing.

Based on the definition of Stachowiak (Stachowiak, 1973) of models as abstractions with a purpose and a representation, the use of models is a well established practice in software engineering as well as in computer science as a whole. Models are not only used in approaches that are explicitly Model-Driven but are employed in a number of different forms and for a multitude of tasks. We consider an approach Model-Driven if models are the primary artifacts in a process from which the different implementation artifacts are generated. Models are used to abstract data and control flow in programming languages (Scott, 2005). The design of a suitable domain model (Fowler, 2003) is a central part in the development of complex software systems and the elicitation of a suitable fault model (Binder, 1999) is a central part of the testing process. Section 3.1 discusses the definition of models we employ for MTCC and examines the notions of meta models and model transformations.

Current Model-Driven processes and technologies are characterized by the utilization of formal models on a level of abstraction above the specific implementation platform and by the use of model transformation, generally for the purpose of code generation, in order to bridge the implementation gap (France & Rumpe, 2007) between an abstract model and a specific platform.

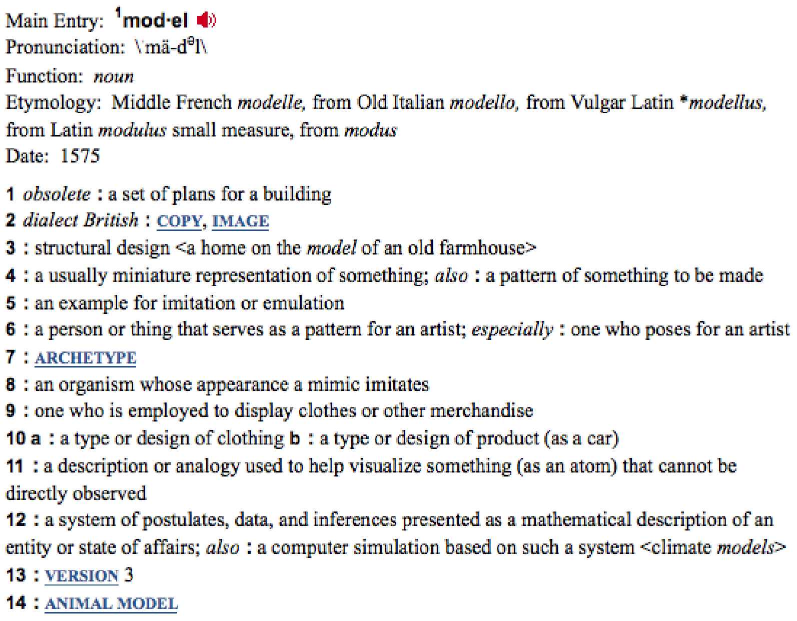
A number of different approaches for Model-Driven development exist that influence the MTCC approach to different degrees. MTCC is closest to the generative programming approach regarding the used models and the high degree to which the details of the implementation domain are abstracted. We introduce these approaches in Section 3.2 and compare them with MTCC.

One concrete use of models is Model-Driven testing, the verification of a software system based on a model of the system. Depending on the respective approach, the tested aspects of a system and its operating environment are represented by models that support the automatic generation of test cases and the allow an automated assessment of the success of the each generated tests. We discuss Model-Driven testing in Section 3.3 and relate it to MTCC.

3.1 Roles and Properties of Models

The term *model* has a number of different meanings, depending on the context in which its is used. Figure 14 gives an overview of 14 different definitions for the term model taken from the online version of the Merriam-Webster Dictionary[1].

For software engineering, the definitions of model as *structural design* and *a system of postulates, data, and inferences presented as a mathematical description of an entity or state of affairs; also : a computer simulation based on such a system* are of particular relevance as they allude to the prescriptive and descriptive aspects of modeling (Bézivin, 2005).



Main Entry: **¹mod·el**
Pronunciation: \ˈmä-dᵊl\
Function: *noun*
Etymology: Middle French *modelle,* from Old Italian *modello,* from Vulgar Latin **modellus,* from Latin *modulus* small measure, from *modus*
Date: 1575

1 *obsolete* : a set of plans for a building
2 *dialect British* : COPY, IMAGE
3 : structural design <a home on the *model* of an old farmhouse>
4 : a usually miniature representation of something; *also* : a pattern of something to be made
5 : an example for imitation or emulation
6 : a person or thing that serves as a pattern for an artist; *especially* : one who poses for an artist
7 : ARCHETYPE
8 : an organism whose appearance a mimic imitates
9 : one who is employed to display clothes or other merchandise
10 a : a type or design of clothing **b** : a type or design of product (as a car)
11 : a description or analogy used to help visualize something (as an atom) that cannot be directly observed
12 : a system of postulates, data, and inferences presented as a mathematical description of an entity or state of affairs; *also* : a computer simulation based on such a system <climate *models*>
13 : VERSION 3
14 : ANIMAL MODEL

Figure 14: Definition of *model* in the Merriam-Webster Dictionary

3.1.1 Attributes of Models

Greenfield & Short (2004) defines a model in the context of software development as follows: *"... a model is an abstract description of software that hides information about some aspects of the software to present a simpler description of others..."*. Evans (2004) gives the following definition: *"A system of abstractions that describes selected aspects of a domain and can be used to solve problems related to that domain"* where domain is defined as follows: *"A sphere of knowledge, influence, or activity."*. Stahl & Völter (2006) define the meaning of a model such: *"A model is a abstract representation of structure, function or behavior of a system."*

According to Stachowiak (1973), a model is characterized by three properties (translation by the author):

- **Representation property** *Models are reproductions or delineation of natural or artificial objects*
- **Abstraction property** *Models do not capture all attributes of represented original but only those that appear relevant to the creators and/or users of the model*
- **Pragmatic property** *Models are not always unambiguously assigned to their original. They fulfill a substitution role a) for certain subjects, b) for certain periods of time and c) under constraints on certain mental or physical operations*

[1]`http://www.merriam-webster.com/dictionary/model`

3.1.2 Types of Models

A common aspect of all definitions is the concept of abstraction. For this thesis we define models as abstractions with a purpose (Ludewig, 2003). In the following, we further differentiate models by two characteristics:

- Models may be formal or informal. We consider a model formal when it is represented in a way that allows the automatic processing or transformation of the model for its intended purpose. The structure and semantics of a formal model are defined by its meta model.

  Modeling languages like the UML can be used both formally and informally. When used formally in the context of the MDA (Brown, 2004), the MOF (Alanen *et al.*, 2005) serves as a meta model for the UML. Besides its use for formal modeling, the UML can also be used as a sketch in order to outline the structure of systems.

  Whether a modeling language is formal or informal is independent from the question whether it uses a graphical or textual representation (Harel & Rumpe, 2004).

- Models can be used either prescriptively or descriptively. Prescriptive models are used for the design of systems that are yet to be constructed, descriptive models represent an system that already exists (Bézivin, 2005).

From the definition of a model as an abstraction with a purpose follows that the subject represented by the model and the goal of modeling process must be considered in every modeling situation. The subject and the purpose of a model define the domain that is covered by the model. This domain determines the criteria that stipulate which aspects of reality have to be representable by the model at what level of detail.

Meta Model

A meta model defines the structure of a model as well as its semantics. The semantics that can be expressed are specific for a domain, therefore, the meta model is a representation of that domain. Figure 15 examines the relationship between formal models, their domain and the various concepts that define a meta model. Models are expressed in a domain specific language (DSL). A DSL defines a concrete syntax or formal notation used to express concrete models. The abstract syntax defines the basis concepts available for model construction, it is determined by static semantics of the meta model.

Notwithstanding the existence of a multitude of standards and approaches for meta modeling (Karagiannis & Kühn, 2002; Budinsky *et al.*, 2003; Alanen *et al.*, 2005), meta models and their use are an active field of research. This is also true for the integration of different meta models (Emerson & Sztipanovits, 2006; Baudry *et al.*, 2006) and the representation of constraints on meta models (Cabot & Teniente, 2006). We argue that neither research nor practice have yet succeeded in establishing an universal meta modeling standard.

Meta models are related to ontologies (Saeki & Kaiya, 2006). While ontologies aim at the presentation of knowledge, the purpose of meta models is the definition of a formal language.

Figure 15 illustrates one important property of models: The separation of the representation of a model, its concrete syntax, from its semantics. This property allows the use of multiple representations for one meta model, for instance the textual representation of UML (Spinellis, 2003) or — in the case of MTCC — the representation of the feature model through a graphical editor.

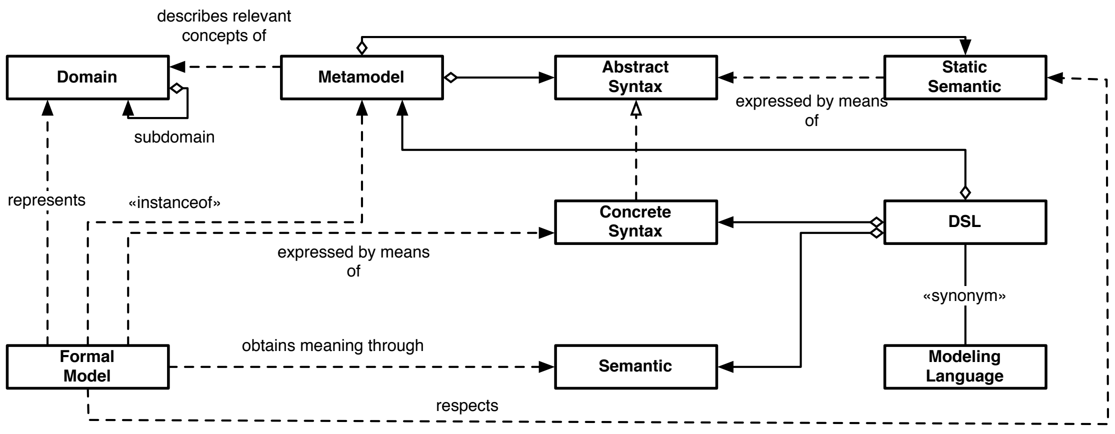


Figure 15: Concepts of Model-Driven software development (Stahl & Völter, 2006)

3.2 Model-Driven Software Development

The purpose of Model-Driven software development is to close the gap between the problem domain and the solution domain (Stahl & Völter, 2006). Models abstract from the technical details of an implementation and facilitate the use of concepts from the subject domain not only in requirements engineering and design, but also during the implementation of a system.

France & Rumpe (2007) express this goal as follows: *"current research in the area of model driven engineering (MDE) is primarily concerned with reducing the gap between problem and software implementation domains through the use of technologies that support systematic transformation of problem-level abstractions to software implementations."*

The stated goals and claims associated with Model-Driven software development, in particular the MDA approach, have been the subject of critique (McNeile, 2003; Thomas, 2004). We argue that potential deficits of specific approaches do not challenge the effectiveness of Model-Driven software development in general but rather that the methods and tools used for any software project must consider the individual requirements of each project.

In the following, we introduce multiple approaches to Model-Driven software development and relate them to the MTCC. We discuss the transformation of models into implementations. Furthermore we explain the relationship of Model-Driven development and programming languages with a particular focus on domain specific languages (van Deursen *et al.*, 2000). We present a summary of relevant approaches to Model-Driven development, a more detailed comparison can be found in (Völter, 2006a).

Model Driven Architecture

The Model Driven Architecture (MDA) is a standard for Model-Driven development propagated by the Object Management Group (Mellor *et al.*, 2004; Uhl, 2006).

One defining aspect of the MDA is the successive refinement of models (Stahl & Völter, 2006). A Platform Independent Model (PIM) represents functional requirements in a formal notation. The PIM is used as the basis for one or more Platform Specific Models (PSM). Each PSM is an abstract representation specific for one aspect of the implementation, for instance, the database layer or the deployment architecture. In order to create an implementation, each PSM undergoes one or more transformation steps, turning the PSM into successively more detailed models and finally implementation artifacts. The MDA emphasizes model-to-model transformations in the software construction process, as the description above illustrates.

We argue that the MTCC approach does not benefit from the advantages offered by the

MDA approach because the increase in complexity that the adoption of the MDA would cause is not worthwhile.

Architecture-Centric Model-Driven Software Development

Architecture-Centric Model-Driven Software Development (AC-MDSD) is an approach introduced by Stahl and Voelter (Stahl & Völter, 2006). The purpose of AC-MDSD is the *"holistic automation during the development of infrastructure code respectively the minimization of redundant, technical code in software development"*. A concept central to the approach is the platform, the total of all parts of the infrastructure that are used for the implementation. Compared to the MDA, the focus of AC-MDSD is more on the abstraction of technical detail than on the complete representation and implementation of a system with models. We argue that this approach is more pragmatic than the vision offered by the MDA. MTCC shares the goal of AC-MDSD to abstract from infrastructure code; in the case of MTCC, test code and code used to interface with the testee.

Generative Programming

With Generative Programming, Czarnecki & Eisenecker (2000) introduced an approach that aims at creating complete implementations of software systems, optimized for specific requirements, based on a library of implementation artifacts and a formal representation of available configurations. Czarnecki describes the approach as follows: *"Generative software development is a system-family approach, which focuses on automating the creation of system-family members: a given system can be automatically generated from a specification written in one or more textual or graphical domain-specific languages"* (Czarnecki, 2005). Like Generative Programming, MTCC considers families of systems, more specific families of tests.

3.2.1 Generation of Implementations

In order to facilitate the creation of implementation artifacts from models, a transformation step is necessary (Cordy *et al.*, 2006). With the execution of one or more transformation steps, implementation details and models are integrated and successively more detailed models are created. A transformation step can either be a Model2Code transformation or a Model2Model transformation (Stahl & Völter, 2006). While a Model2Model transformation generally starts with an input model and generates another more detailed, model, Model2Code transformations generate an implementation artifact, usually program code. A number of different model transformation approaches exist that address different goals (Czarnecki & Helsen, 2003) and vary in the complexity and suitability for complex transformation scenarios. The test cases generated in the MTCC exhibit only minimal complexity. Accordingly, we limit our consideration of possible transformation strategies to template-based (Herrington, 2003; Voelter, 2003) code generation.

3.2.2 Models and Programming Languages

Model-Driven development shares a significant number of concepts with programming languages and compiler technology. Considerable parallels exist between model transformation and compilation — both processes create implementations from abstract representations. The concepts used by programming languages to describe both data and control flow (Scott, 2005) and the rules that govern their use and possible combinations in a program correspond to a meta model.

Domain specific languages (van Deursen *et al.*, 2000; Tolvanen & Kelly, 2004) exist for a wide variety of application domains (Mernik *et al.*, 2005; Hudak, 1996). Such languages have the same purpose as models in MTCC expressing the concepts of a specific application domain — they implement problem-specific abstractions (Grönniger *et al.*, 2006). Implementation techniques for DSLs are similar to those used in Model-Driven development (Völter, 2007, 2006c), especially for approaches that use code generation.

Another approach that serves to illustrate the close relation between Model-Driven development and programming languages is Languages Oriented Programming. Fowler gives an overview of one such approach (Fowler, 2005c,b,a), the MPS environment (Dmitriev, 2004). MPS aims to facilitate the integrated development of a meta model, an editor for modeling, and transformations for implementing DSLs. The intentional software approach is similar in its goals (Simonyi, 1995; Simonyi *et al.*, 2006).

MTCC uses a graphical editor to present models to domain experts. The XML-based representation of MTCC models can be considered a domain specific language for system and test description.

3.3 Model-Driven Testing

Testing is always Model-Driven in the sense that it is based on a fault model. A fault model considers a system with regards to its likely faults. Binder gives the following definition of a fault model *"(A Fault Model) identifies relationships and components of the system under test that are most likely to have faults. It may be based on common sense, experience, suspicion, analysis, or experiment..."* (Binder, 1999).

Similar to models used in the construction of systems, a fault model does not have to be formal to be useful for testing, it may take the form of a list containing the functionality of a system to be tested. Formal models that are suitable as a basis for automatic testing are called testable (Binder, 1999). The use of testable models for the automatic verification of software is the subject of Model-Driven testing.

Figure 16 displays an example of a Model-Driven testing process and the concepts that are relevant during the phases of the process.

- In a first step, a model that can be utilized for the automatic generation of tests is realized based on the requirements of the examined system. The model represents those aspects of the behavior and state of the testee that are relevant for testing. For any possible input to the system, it provides the outputs expected from the system. The models serve as an oracle.
- In the Test Case Specification phase, criteria for the selection of tests are defined and formalized. Criteria include the test coverage by different metrics as well as random selection of tests and the use of previously recorded user interactions with the system. The Test Case Specification is a formalization of this criteria that facilitates the automatic selection of tests cases.
- Tests are generated and executed on the SUT. The adaptor in Figure 16 corresponds to the concept of the test runner in MTCC. The results of all executed tests are recorded.

3.3.1 Types of Model-Driven Testing

As in Model-Driven development, many approaches to Model-Driven testing exist. A criterion by which different approaches can be differentiated from MTCC is *Redundancy* (Utting *et al.*, 2005). Redundancy differentiates between approaches that reuse models created for the construction of a system from those that use specialized models created only for testing. Bertolino uses the term test-based modeling (Bertolino, 2007) for the latter group.

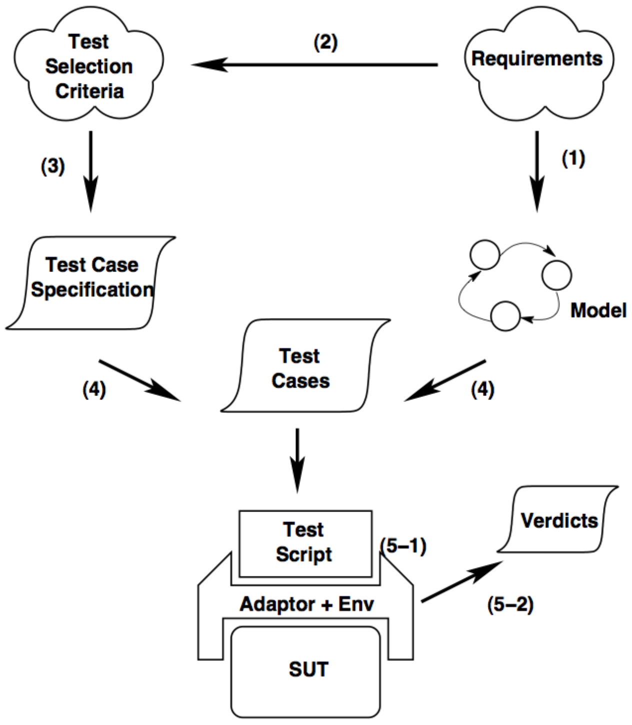


Figure 16: A process for Model-Driven testing (Utting *et al.*, 2005)

Approaches that aim at reusing or extending models or modeling tools created for the construction of software are limited by the modeling languages used in this process, often the UML. As a consequence, there is a great number of approaches to Model-Driven testing that are based on the UML (Cavarra *et al.*, 2004; Briand *et al.*, 2005; Seifert & Souquieres, 2008; Linzhang *et al.*, 2004; Kim *et al.*, 2005; Hartmann *et al.*, 2004).

One application of the UML for testing is the UML2 Testing profile (Schieferdecker *et al.*, 2003; Baker *et al.*, 2003; Dai, 2004). The purpose of the U2TP is to provide an UML profile that adds test-relevant concepts to the UML2 core language and thus allows the modeling of tests using UML specific tooling. The U2TP can be regarded as Model-Driven development applied to the testing domain. By envisioning intellectual test specification, the U2TP bears similarities to MTCC, but differs in the modeling language and the support for modeling by domain experts.

Test-based modeling approaches are not limited to any particular modeling language. Utting *et al.* (2005) present an overview over approaches to Model-Driven testing and the models used in these approaches. Finite State Machines (El-Far & Whittaker, 2001; Andrews *et al.*, 2005), decision trees and decision tables are frequently used as models for testing purposes (Binder, 1999).

3.3.2 Relation to MTCC

Figure 17 displays a taxonomy of Model-Driven testing that classifies approaches to Model-Driven testing along seven dimensions. Not all dimensions defined by the taxonomy can be applied to MTCC, in the following we only discuss the relevant classification criteria.

Subject MTCC models represent a testee from a system family of testees. The test-relevant Services of a testee as well as the Test Steps that exercise and verify these Services are modeled.

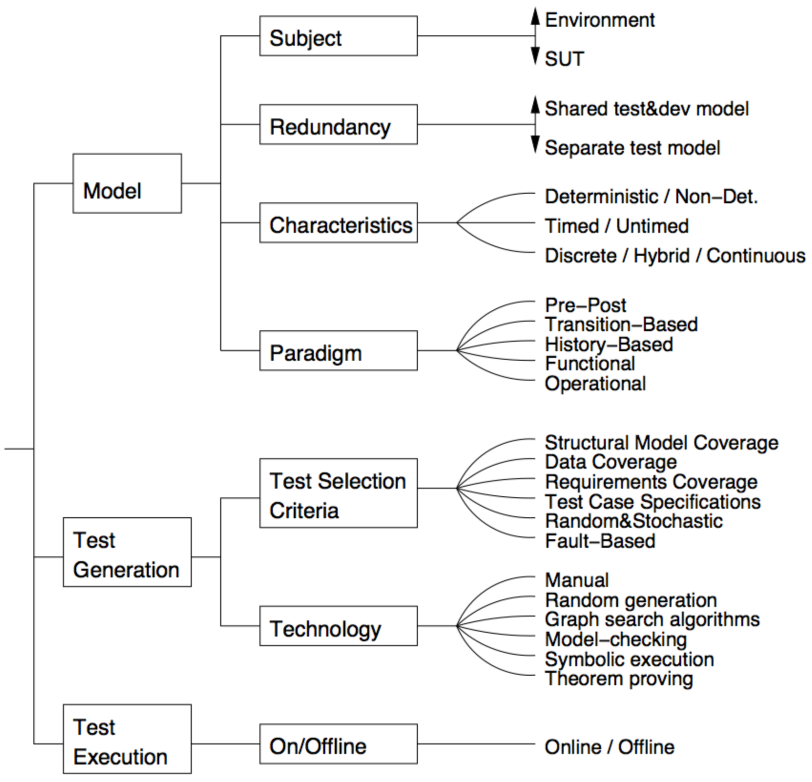


Figure 17: A taxonomy of Model-Driven testing (Utting *et al.*, 2005)

Redundancy The models constructed for the MTCC are only used for testing.

Test Selection Criteria MTCC envision test case construction by domain experts. Domain experts select and implement those test that they consider relevant based on their knowledge of the requirements of the system.

On/Offline MTCC generates and executes test cases offline.

3.4 Chapter Summary

A number of different meanings exist for the term *model*. In this thesis, a model is a formal abstraction that serves a specific purpose. A model is considered formal if its structure and semantics conform to an implicit or explicit meta-model.

Model-Driven Software Development attempts to solve the *Problem Implementation Gap* by using formal models that represent concept of the application domain. Applications are modeled and the resulting models are used, optionally after further refinements, to generate implementation artifacts. MTCC is a Model-Driven approach that uses models to represent SUTs and tests that exercise a SUT.

Model-Driven Testing uses formal models of a SUT and optionally the environment of the SUT to generate tests and to assess the success or failure of these tests. MTCC is closely related to Model-Driven testing approaches but differs in that tests are not generated but constructed by domain experts.

Chapter 4

System Families and Variability

In this chapter we discuss system families (Parnas, 1976) and their relation to software product-lines (Weiss & Lai, 1999). We introduce feature models (Czarnecki *et al.*, 2005a) and examine to which degree approaches to Software Product Line Testing (Tevanlinna *et al.*, 2004; Pohl & Metzger, 2006) can be applied to MTCC.

Section 4.1 discusses the properties of system families as the object of investigation in MTCC. . A system family is a group of programs who are better discussed in terms of their commonalities than their differences (Parnas, 1976). The members of a system family can be programs within the same application domain that fulfill the same tasks but were developed in separation. Alternatively, the members of a system family can be variants or versions of a common product. A Software Product Line (Weiss & Lai, 1999; Clements & Northrop, 2001) is a number of related products which are derived from a common set of core assets. The focus of software product-line engineering is on the organized reuse and adaption of implementation artifacts (Meister, 2006a) and the systematic management of variability (Bachmann & Clements, 2005).

One means to formally express the variability in a product-line are feature models (Kang *et al.*, 1990) as introduced in Section 4.2. MTCC utilizes feature models both to represent the functionality and properties of a testee and to describe the possible parameterization of Test Steps. Feature models facilitate the representation of the supported features and combinations of features of the members of a system family in an formal, hierarchical way. Feature models define operations that allow derivations of a feature model that represents a more constrained variability. Specialization operations perform local transformations on the feature model that result in a more specific feature model. A fully specialized feature model is a configuration (Czarnecki *et al.*, 2005b,a).

From the definition of a system family follows that the requirement exhibited by its members overlap to a certain degree. This leads to the question how tests for the common requirements of systems can be shared among systems. MTCC's primarily means of support for test reuse are abstract test models that can be transfered between systems. Section 4.3 discusses the testing of system families and product-lines and discusses the role of MTCC in this context.

4.1 Variability in Software Engineering

The increasing complexity of software systems and the associated increase in cost necessitates the reuse of implementation artifacts. One context for reuse that is of particular interest to this thesis are program families or system families[1] (Parnas, 1976). Parnas defines program

[1]We use the term system family instead of the term program family used by Parnas since we take the position that it better reflects current usage of the terms system and program.

families as follows: *Program families are defined as sets of programs whose common properties are so extensive that it is advantageous to study the common properties of the programs before analyzing individual members*

Different kinds of system families exist, depending on the relationship of the members of the product-line:

- System variants are members of a system family that were adapted for different markets or customers. Variants may differ in their provided features or may be adapted to the individual requirements of customers.
- Product versions are systems where a newer version of a software builds on an older version. Since it cannot be assumed that every customer of a software system can always use the most recent version, different versions have to be maintained in parallel.
- An existing system and its replacement are members of a software family when both systems share no parts of their implementation but are used for the same tasks and are subject to the same requirements.

Despite the existence of various approaches to facilitate reuse in software families, the degree of reuse in software development is still unsatisfactory, reuse in practice is often opportunistic (Meister, 2006a).

In order to support reuse within system families, a number of requirements must be meet:

- The variability (Gurp *et al.*, 2001; Bachmann & Clements, 2005) in the system family, meaning the range of common and different features that can exist for the members of a product-line, has to be captured and formalized. Domain analysis (Arango, 1989, 1994) is employed in order to identify and capture the relevant concepts of the domain that are subsequently formally expressed as feature models (Kang *et al.*, 1990; Czarnecki *et al.*, 2005a). We call the set of all systems that are covered by the variability of the system family the scope of the system family.
- Processes and methods must be established that, based on requirements for a system to be build, support a statement on whether the scope of the product-line covers these requirements. If the system is within the scope of the product-line, a configuration must be created that describes the particular features of the systems in terms of the variability of the system family.
- The realization of all configurations that can be expressed within the scope of the system family has to be possible. Tooling and core assets must exist that allow the fully automatic implementation of a product from the previously build abstract configuration.

A system family for which all the above requirements are fulfilled, is called a Software Product Line (SPL) (Clements & Northrop, 2001; Weiss & Lai, 1999). Meister (Meister, 2006a) defines a SPL as follows: *"A product family is called an SPL if it was explicitly designed for the reuse of common elements and the tractable reuse realization of varying elements."*

In Software Product Line Engineering, work on concrete products is done in parallel with work on the artifacts of the product-line infrastructure. Figure 18 illustrates this point.

- In Product-engineering, the requirements for a specific system are captured. Based on these requirements a product configuration is prepared and as many features of the product based on the core assets of the Product Line as possible are realized. Features that cannot be automatically realized, are implemented by hand and form the basis for new requirements to the SPL.

- SPL engineering addresses the product-line infrastructure itself. SPL-scoping defines the scope of the SPL, the domain considered by the product-line and the set of products that can be realized by the product-line. Once the scope of a SPL is defined, the set of *Core Assets* (Bachmann & Clements, 2005) used to realize members of the product-line are first established (Bachmann & Clements, 2005). core assets for product realization can be implemented based on a number of different approaches. Potential implementations include techniques from Model-Driven development (Voelter & Groher, 2007; Loughran *et al.*, 2006; Czarnecki & Antkiewicz, 2005), the use of component technology (Kettemann *et al.*, 2003), approaches based on programming languages (Patzke & Muthig, 2002) and frames (Patzke & Muthig, 2003).

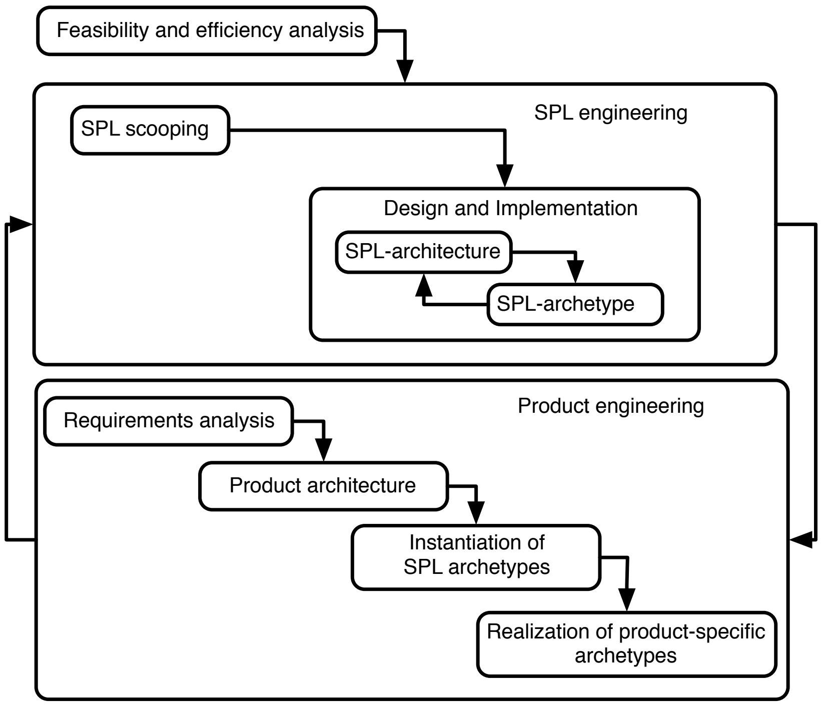

Feasibility and efficiency analysis
SPL engineering
SPL scooping
Design and Implementation
SPL-architecture
SPL-archetype
Product engineering
Requirements analysis
Product architecture
Instantiation of SPL archetypes
Realization of product-specific archetypes

Figure 18: SPL development process (Meister, 2006a)

4.1.1 Analysis and Scoping of a Product Line

SPL-Scoping, the part of SPL-analysis that addresses the definition of the extent of the domain covered and the concepts within the domain, is a prerequisite for the establishment of a SPL. SPL-scoping is an adaption of the earlier concept of domain analysis that differs in focusing not on the properties of an abstract domain but on the construction of concrete products in the domain (Meister, 2006b; DeBaud & Schmid, 1999).

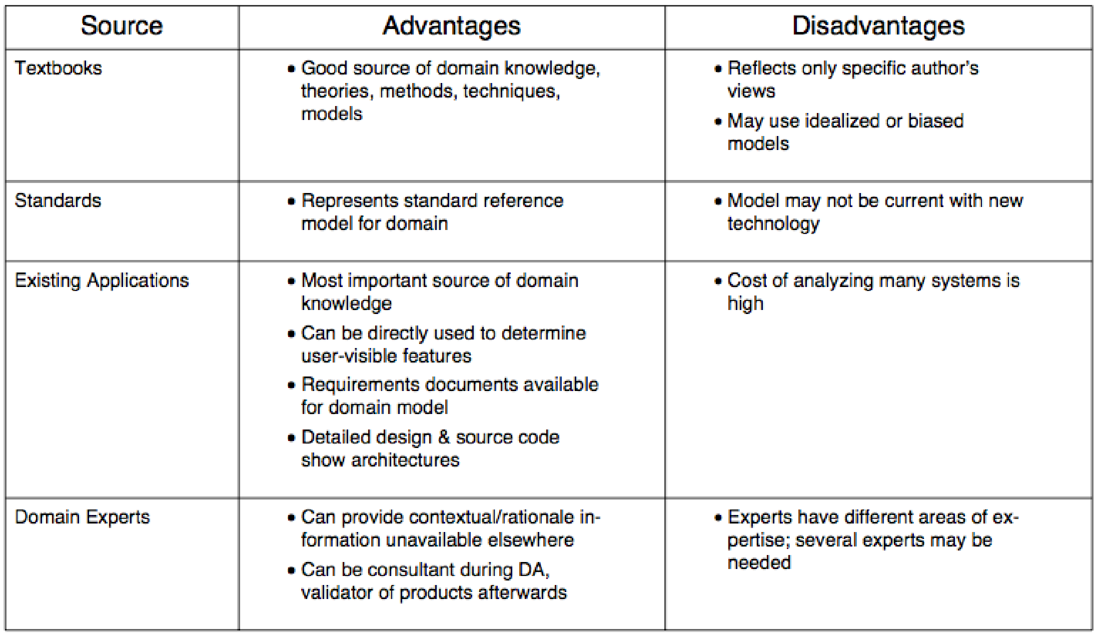
An important aspect of product-line engineering is the identification of the relevant features. Czarnecki & Eisenecker (2000) give two conflicting definitions of the term feature:

- *An end-user-visible characteristic of a system.*
- *A distinguishable characteristic of a concept (e.g., system, component, and so on) that is relevant to some stakeholder of the concept.*

For this thesis we use the definition that originates from the FODA (Kang *et al.*, 1990) approach of feature as *'user-visible aspects or characteristics of the domain'* We argue that this view on features better represents the external view of a system taken in acceptance testing.

Multiple approaches for domain analysis and SPL-analysis are defined software engineering (Bayer *et al.*, 1999; Bachmann & Clements, 2005; Czarnecki & Eisenecker, 2000; Neighbors, 1989; Clements & Northrop, 2001; Weiss & Lai, 1999; Matinlassi, 2004). Each approach defines information sources used in the analysis process and defines the methods and notations used to formalize the knowledge gained in the analysis. Figure 19 gives an overview of possible sources of information for domain analysis. In order to identify test-relevant functionality, MTCC considers the systems to be tested in the system family as well as the test regime in use and the requirements formulated by the domain experts. The latter is of particular importance in situations where no test for some functionality of the testees exists.

With the exception of feature models, the details of domain analysis, such as the process employed and the artifacts used, are not discussed in detail. MTCC does not assume the utilization of any particular approach for the capture and formalization of the concepts and features of a domain. The exclusion of these subjects from MTCC is due to two reasons: First, the contribution does not lie in the field of analyzing but in the application of domain knowledge, and second, the analysis necessary for MTCC as it is implemented for this thesis is simplified by the fact that the system family and all systems to be tested already exists. The domain analysis process is therefore not subject of this work.



Source	Advantages	Disadvantages
Textbooks	• Good source of domain knowledge, theories, methods, techniques, models	• Reflects only specific author's views • May use idealized or biased models
Standards	• Represents standard reference model for domain	• Model may not be current with new technology
Existing Applications	• Most important source of domain knowledge • Can be directly used to determine user-visible features • Requirements documents available for domain model • Detailed design & source code show architectures	• Cost of analyzing many systems is high
Domain Experts	• Can provide contextual/rationale information unavailable elsewhere • Can be consultant during DA, validator of products afterwards	• Experts have different areas of expertise; several experts may be needed

Figure 19: Information sources for domain analysis (Kang *et al.*, 1990)

Feature-Products Maps (DeBaud & Schmid, 1999) provide an overview, which products of a system family support which features. Every feature is correlated with every product in the system family. The features themselves and thus the variability of the system family are represented by feature models.

4.1.2 Variability and System Family Testing

We consider two aspects of the variability of a system family of particular importance for MTCC:

- MTCC does not assume that the testees considered are members of a software product line. Instead, MTCC is based on the premise that the testees form a system family defined by a common set of test-relevant features. MTCC has the characteristics of a product-line approach for the construction of tests. MTCC Core Assets are the models used to represent systems and tests as well as the editor and the subsystems responsible for model transformation and test code generation.
- In this thesis, we only apply MTCC to an already existing system family of Digital Libraries. This constraint allows for simplifications in the domain analysis phase that cannot be assumed if the systems in the system family are still in the development process.

4.2 Feature Modeling

The purpose of feature models is the representation of the variability in a system family. A feature model organizes the features of a system and the dependencies of the features in a tree structure. Specialization steps are operations that support the refinement of the variability and thus the specification of concrete systems in terms of their features. MTCC uses feature models for the representation of the Services provided by testees as well as for the representation of the parameterization of tests.

4.2.1 Structure of Feature Models

A feature model is a tree structure that facilitates the hierarchical description of the features of a system. Features within a feature model can represent system-subsystem relationships, choices among different functions that a system can provide or abstract, non-functional requirements.

Figure 20 displays a simple feature model in the notation introduced by the FODA (Kang *et al.*, 1990) approach. The feature model in the example represents the features of a car as follows:

- Every feature is the composition of its children. The feature Car is defined by the lower-level features Transmission, Horsepower and Air conditioning.
- Features can be mandatory or optional. While every car must have a transmission, Air Conditioning is optional and can be de-selected.
- Features can be alternatives. The Car represented in the feature model either has an automatic or a manual transmission, but never both.
- Features can hold numerical values, for example, the feature Horsepower.

Alternatives like the one between automatic and manual transmission in Figure 20 are a special case of a feature group. Besides alternatives that allow only one of the features from the feature group to be selected, FODA also defines feature groups that allow the selection of an arbitrary number of features.

Not all relevant aspects for the configuration can be represented by the hierarchical structure of the feature model alone, rationales and composition rules are employed to add extra information to a feature model. A rational gives a human-readable reason that can serve as a basis for the selection or de-selection of a feature. Composition rules describe under which conditions certain features can be selected or combined. A feature diagram is a set of feature models together with their rationales and composition rules. MTCC does not use rationales

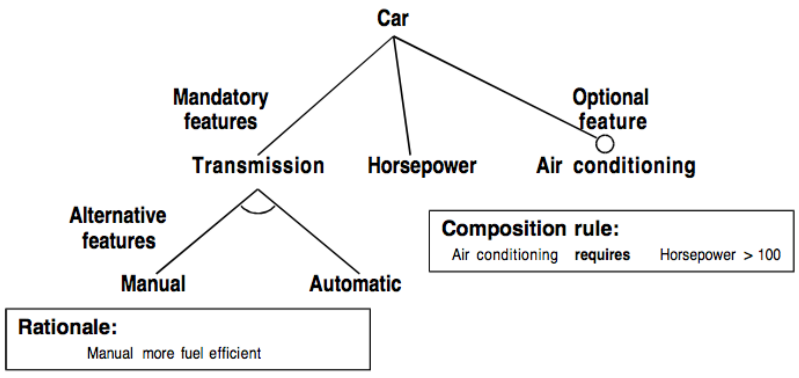


Figure 20: Example for a FODA Feature Diagram (Kang *et al.*, 1990)

or composition rules. We argue that the complexity that these rules add to feature models is not justified in the context of MTCC, primarily because of the descriptive, fine grained nature of the feature models used in MTCC.

An important property of feature models is the abstraction of implementation details. While automatic approaches exist that map features to implementation artifacts (Czarnecki & Antkiewicz, 2005), the overall level of abstraction provided by feature diagrams is higher than is the case for modeling approaches that represent high-level implementation artifacts such as UML class diagrams.

Based on the notation and the semantics for feature models defined for FODA, a number of different extensions were developed (Heymans *et al.*, 2007). The feature models used in MTCC are based on feature models with cardinalities (Czarnecki *et al.*, 2005b,a). Compared to the feature models defined by FODA, feature models with cardinalities exhibit the following extensions.

- **Cardinalities for Features** The cardinality of a feature node describes how often the feature and its children can be included in the feature model. Cardinalities are expressed as an interval [n,m] where n describes the minimal occurrence of a feature and m its maximal occurrence. Mandatory features have a cardinality of [1,1], the cardinality of an optional feature is [0,1].
- **Attributes** Attributes are nodes in a feature model that can hold numerical or textual information.
- **References** Reference nodes support the inclusion of a referenced feature model or a subtree of a feature model in place of the reference node.

4.2.2 Configuration and Specialization of Feature Models

A feature model describes the variability within a system family in terms of all its potential configurations. A configuration is a feature model without any variability, the feature model is completely specialized.

Czarnecki *et al.* (2005a) describe the relation of a feature model and its configuration as follows: *'The relationship between a feature diagram and a configuration is comparable to the one between a class and its instance in object-oriented programming'*. The transformation of a feature model in a completely specialized form is called the configuration process (Czarnecki *et al.*, 2005a). The configuration process is defined by the application of a series of specialization steps. Each specialization step creates a new feature model with less variability than the initial feature model.

Seven specialization steps are defined for feature models with cardinalities. Figure 21 illustrates their application on feature models:

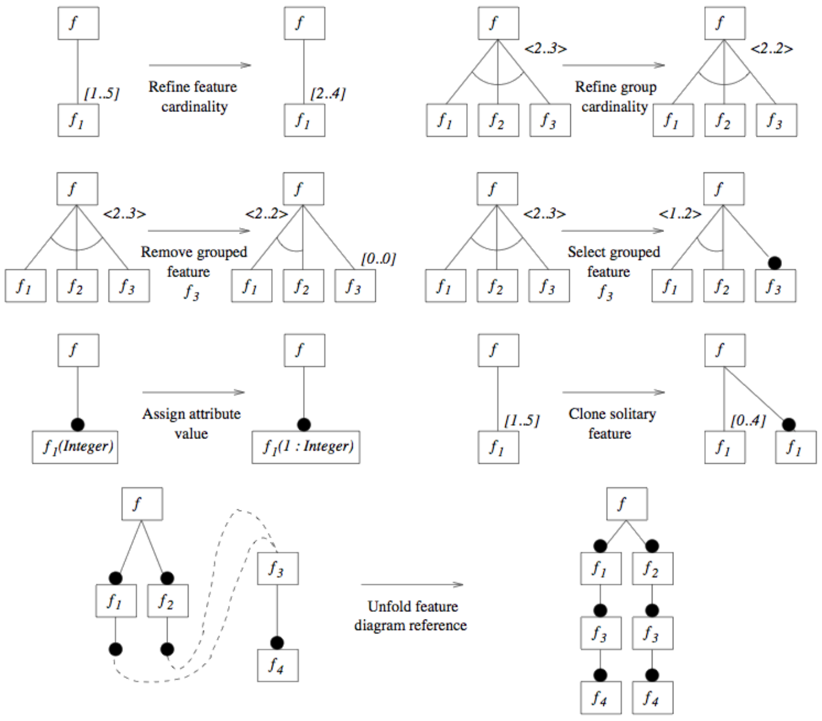


Figure 21: Specialization steps (Czarnecki *et al.*, 2005b)

Refinement of feature cardinalities The refinement of feature cardinalities adjusts either the minimal or the maximal occurrence of a feature in such a way that the range between the minimum and and maximum is smaller than in the initial feature model.

Refinement of group cardinalities This specialization step narrows the interval that describes the minimal and the maximal number of selectable features in a feature group.

De-selection of features from a feature group This specialization step sets the cardinality of one child of a feature group to [0,0], the node can no longer be included in a configuration. By de-selecting the node, the number of maximal children of the feature group is lowered by one.

Selection of one feature from a feature group This specialization step sets the cardinality of one child node of a feature group to the interval [1,1]. By applying the specialization step, the minimal as well as the maximal number of children for the feature group are lowered by one.

Assignment of attribute values This specialization step sets the value of an attribute. If a type system for attributes is used, as is the case in the situation illustrated in Figure 21, the legal values are constrained by the type system.

Cloning of solitary feature nodes As the name suggests, the cloning specialization step duplicates a feature node which have maximum cardinality greater than 2. The cardinality of the cloned feature is [1,1], the minimum and maximum occurrence of the cloned feature node are lowered by one.

Unfolding of feature reference nodes When a feature reference is unfolded, the referenced features are included in place of the feature reference.

As described by Czarnecki et al (Czarnecki *et al.*, 2005b) for the Staged Configuration Process, the configuration of feature models in MTCC is done in a series of incremental phases. First, a representation of a single system is derived from a description at the system family level, then references from Test Step models to the system model are unfolded, and finally the feature models of Test Steps are configured.

Some restrictions apply to the implementation used in MTCC for feature models with cardinalities. Solitary features are limited to the cardinalities [0,1], [0,0] and [1,1], refinements of cardinality and cloning of features are not supported.

4.3 Testing of System Families

A number of approaches address testing in the context of system families or software product-lines. Compared to the testing of solitary systems, such methods face a number of specific challenges:

- Ideally, the development of tests and the infrastructure needed for test execution should be based on a set of core assets. The testing of single systems corresponds to the product engineering activity (Pohl & Metzger, 2006) in the context of a product-line while work on common test assets is related to Software Product Line Engineering.
- The features that are available for different systems under test within a product-line or system family have to be considered in test selection. This requirement has two aspects: Each system can only be exercised when it exhibits all features necessary for the execution of a test and each feature of a system should be covered by a test (Geppert *et al.*, 2004b; McGregor *et al.*, 2004). We call this requirement the test selection challenge.
- Test runners and interfaces must exist that support the execution of tests for all systems within the system family. In a SPL approach, the implementation artifact must support testing (Kolb & Muthig, 2006).

Software Product Line Testing (SPLT) is an area of intensive research (Tevanlinna *et al.*, 2004). A significant difference between SPLT approaches and MTCC is that MTCC does not assume that the testees are members of a SPL and are built from a common set of core assets.

A significant difference between testing in the context of software product lines and software product families that are not based on a common set of core assets is that no commonalities in the implementation or the interfaces of the member of the system family can be assumed. Furthermore, no formalization of the variability of the different members of the system family will be available for the members of the system family — the members of the system family were not configured.

As we already pointed out, MTCC does not assume that testees are members of a software product-line. As a consequence, neither the existence of core assets nor a formal representation of the variability of the product-line can be assumed. Since reuse of existing assets is not possible, the identification and capture of variability in the context of MTCC is done for the first time and only for the purpose of testing. We argue that while the reuse of product line assets for testing would be desirable, the use of models dedicated to testing simplifies the modeling process, thus the construction of the MTCC test models is far less costly than the creation of models for software construction.

MTCC does not aim at the fully automatic testing of product-lines. Since domain experts select the systems of the system family to be tested as well as the features of a system to be tested, one aspect of the test selection challenge introduced above does not apply. The other aspect of the test selection challenge, to ensure that only tests supported by the features of system are specified for that system, is addressed by the MTCC editor and its approach to test reuse.

MTCC addresses the remaining requirements and particularities of testing in a system family context as follows:

- The core assets of the MTCC approach, the infrastructure necessary for testing all members of a system family, are strictly separated from artifacts that are specific to the testing of a single system. Specifically, the editor, the Test Generator and the models used to represent the system family at the domain level are used for all systems tested with MTCC.
- The MTCC testing process is organized in two parallel activities: The derivation of system-specific tests and the work on domain models and testing core assets.
- Tests constructed in MTCC, in the form of Test Configurations for one specific member of the system family, are potentially reusable for multiple members of the system family. A test is reusable for all systems which features support the execution of the test.

While we do not address this aspect in this thesis, the formalization of the variability within a system family and the representation of system family members as configurations facilitates other means of quality assurance than automated acceptance testing. For instance, with a suitable detailed description of the features of individual systems and an infrastructure that adapts the different interface of testees, Differential Testing (McKeeman, 1998; Evans & Savoia, 2007) can use the behavior of one system as an oracle for other systems under test.

4.4 Chapter Summary

MTCC is a testing approach for system families. A system family is a set of systems that are defined by their shared properties, not their differences. While Software Product Lines are based on the systematic reuse and adaption of a common set of Core Assets, system families are generally not the product of systematic reuse. MTCC addresses the testing of system families and therefore does not assume the existence of common infrastructure or artifacts.

A central aspect in working with system families and product lines is the systematic treatment and representation of the variability. Feature models are a means to represent the features that exist in a system family and their relationships. MTCC uses feature models with cardinalities to represent the test-relevant features in a system family and the tests that exercise these features.

Chapter 5

Information Retrieval and Digital Libraries

In this thesis, we apply MTCC to a system family of Digital Libraries, specifically to web portals for scientific information. Digital Libraries are a class of Information Retrieval systems; the functional quality of such systems is defined by their ability to fulfill the information need of a user and present him or her with relevant documents for a search.

The vagueness inherent to the Information Retrieval task makes the participation of domain experts in the testing process particularly important. Section 5.1 introduces the concepts and models relevant to Information Retrieval, special focus is given to evaluation measures for Information Retrieval and the notions of quality in IR.

Section 5.2 gives an overview of the application domain of Digital Libraries. We introduce basic usage scenarios for this class of systems and discuss the meaning of quality for a Digital Library. We also outline the treatment of structural and semantic heterogeneity as a Service that, while it supports users in searching, also raises the complexity of Digital Libraries and thus necessitates testing.

5.1 Information Retrieval

The field of Information Retrieval (IR) addresses the question how the information need of a user concerning a collection of documents can be captured, operationalized, and satisfied. To this purpose, a number of different retrieval models were designed and implemented, for instance the Boolean, the Vector or the Probabilistic Model. These models define representations for information needs as queries and documents as document surrogates. They further provide operations that allow the calculation of the similarity of queries and document surrogates and thus support a statement about the relevance of a document.

Information Retrieval is an empirical field, a number of measures and evaluation methods exists to support the validation of retrieval models as well as retrieval systems.

5.1.1 Purpose of Information Retrieval

The purpose of Information Retrieval can be well illustrated by contrasting IR with data retrieval (Baeza-Yates & Ribeiro-Neto, 1999). Data retrieval deals with formal queries that support for each document a definite, automatic, unchanging decision whether the document satisfies the criteria defined by the query. The subject of Information Retrieval in contrast is not a formal query but a vague information need. In addition, if documents in data retrieval are highly structured records, for example, taken from a relational data management system, documents in Information Retrieval are either unstructured texts, semi-structured (Abiteboul *et al.*, 2000) or sparse.

Manning *et al.* (2008) define IR as follow: *"Information Retrieval is finding material (usually documents) of an unstructured nature (usually text) that satisfies an information need from within large collections (usually stored on computers)"*.

Depending on the information need of a user, the document collection and the nature of the retrieval task, and in absence of a decidable model of relevance, IR bears resemblance to the proverbial search for a needle in a haystack. In reality, this comparison does not illustrate the full breadth of different scenarios covered by IR. Beyond the search for a specific needle in a well-defined haystack, Koll (2000) lists the following scenarios for IR:

- *a known needle in a known haystack*
- *a known needle in an unknown haystack*
- *a unknown needle in an unknown haystack*
- *any needle in a haystack*
- *the sharpest needle in a haystack*
- *most of the sharpest needles in a haystack*
- *all the needles in a haystack*
- *affirmation of no needles in the haystack*
- *things like needles in any haystack*
- *let me know whenever a new needle shows up*
- *where are the haystacks; and*
- *needles, haystacks — whatever.*

An information need can address specific documents or all documents that have specific attributes. The needed information can be clearly defined — at least in the mind of searcher — or it can be only vaguely outlined. The information can be needed in the present or sometimes in the future. From the user perspective, a document is considered relevant if it satisfies the information need of a user. In order for an IR system to estimate the relevance of a document, a model of the information need must be compared with models of the documents (Cooper, 1973). Figure 22 illustrates the relation of an information need and its representation as a query and documents, represented by document surrogates.

From the technical perspective of an IR system, a document is relevant when the document surrogate and the query are similar.

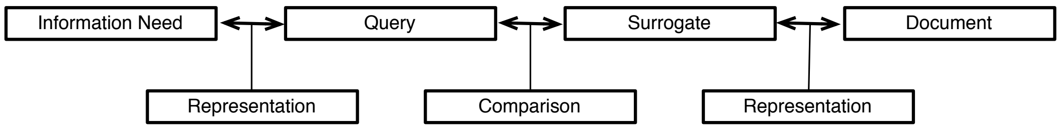


Figure 22: Concepts of Information Retrieval (Belkin & Croft, 1987)

The design, implementation and evaluation of retrieval models for the representation and comparison of information needs and documents is a focus of research for Information Retrieval. One aspect of Information Retrieval that is of significance, but can rarely be represented in IR models, is the dynamic nature of an information need. Rather than answering isolated queries, the IR process is an interaction between a user and the IR system in which an information need is refined or changes.

When IR is regarded as a dialogue between a user and a IR system, the support that the IR system can provide for different users, their skills, preferences and backgrounds becomes more important. To meet the requirements that arise from a dialog-centric or user-centric view on IR, the field has developed a number of different interaction models, user interfaces and methods to tackle the use and enhancement of queries and results:

- In ad-hoc retrieval, users formulate queries to a collection of documents. In filtering, documents are added to a collection and are tested for their relevance on a set of predefined queries.
- In search, a user expresses a specific information need in a query and sends this query to the IR system, the systems returns the subset of all documents that are likely relevant to the query. In browsing, the user does not necessarily have an explicit information need, rather, a collection of documents is explored.

Information Retrieval as a field has produced various approaches to support the formulation of information needs and the visualization of results. An overview is presented by Baeza-Yates and Ribeiro-Neto (Baeza-Yates & Ribeiro-Neto, 1999). Graphical user interfaces exist for the treatment of structural and semantic heterogeneity (Stempfhuber, 2003) or for the combined treatment of the usability and design (Eibl, 2000).

Methods that facilitate the transformation or enhancement of initial queries have a great impact on the ability of an IR system to return documents that are relevant to the information need of a user. Such methods facilitate the integrated retrieval over collections that use different indexing vocabularies (Mayr & Petras, 2008) or support the consideration of specialized discourse domains in search (Petras, 2006).

Approaches to clustering and automatic classification (Chakrabarti, 2003; Weiss, 2005) support the enrichment of a result set of documents by further information that support the user in refining his or her information need.

5.1.2 Concepts and Methods of Information Retrieval

The relevance of a document regarding an information need is represented by the similarity of a document surrogate and a query in the context of a retrieval model.

Figure 23 gives an overview of retrieval models in IR. Since MTCC takes an external, responsibility-based view of testing, details regarding the theory and implementation of the retrieval model used by the testee are neither available to MTCC nor of interest to acceptance tests. We therefore limit our discussion to the differences between the Boolean retrieval model and retrieval models that support the ranking of results by relevance.

The subjects of retrieval models are not an information need and documents but rather queries and document surrogates.

A common format for document surrogates that is both used by the Boolean and the vector model is the representation as a multi set or bag of words that are included in or relevant to the document.

Depending on the specific transformations that a document undergoes during indexing, the terms in the document surrogate can be identical to those in the document. The terms in the surrogate may also form a subset of the terms in the document, for instance, when stop words are removed from the document; or the surrogate may contain a superset of the original terms, for instance, when synonyms are added to the document. In addition to the removal or addition of terms, transformation processes at the term level such as stemming and lemmatization replace individual terms in documents.

If a document is structured into different fields, for instance the title, author and abstract of a bibliographic record, a document can be represented by a surrogate that contains a multi-set for each field of the document.

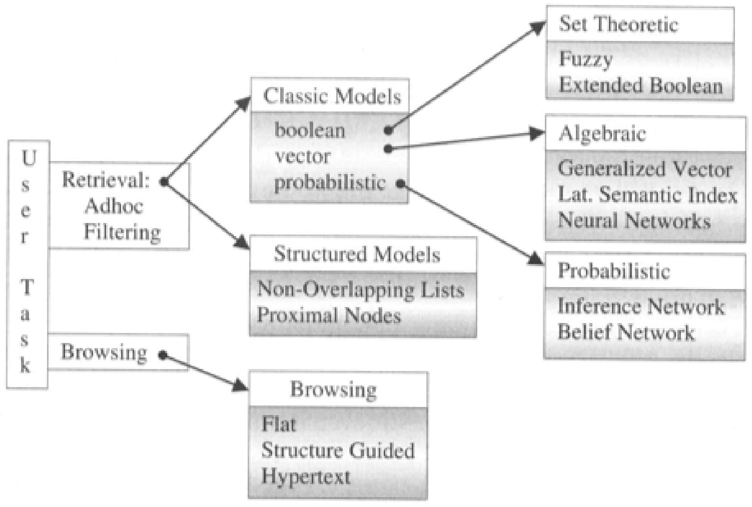


Figure 23: Overview of retrieval models (Baeza-Yates & Ribeiro-Neto, 1999)

Two types of retrieval models can be distinguished: 1) Models that are based on a purely binary model of relevance and 2) vague or graded retrieval models that support the ranking of the documents according to their likely relevance. When a retrieval model with a binary model of relevance is used for search, each document of an IR system is assigned to one of two sets, one containing documents that are relevant to the query and the other containing documents that are not relevant. The result that a system with a binary IR model returns for a query contains all documents from the set of relevant documents without further distinction. Systems based on a graded retrieval model return results sorted by their decreasing probability of relevance, they assign a relevance value to each document.

The Boolean model is based on Boolean algebra: Queries are represented as Boolean expressions. Queries are answered as follows: For each term in a query a set of all documents is built that contains this term, then, unions and intersections of the individual sets are built based on the Boolean operations used in the query. The Boolean retrieval model supports the formulation of complex and precise queries by experts for a retrieval system but is problematic for less experienced users. Not optimally formulated queries can either lead to result sets that are empty or contain so many elements that the user is overwhelmed. A problem inherent to the Boolean retrieval model (Eibl, 2000) is that it cannot differentiate between documents that nearly match a query: If a document matches all but one of the terms specified in a conjunctive query, it has the same negative relevance as a document that contains no term at all.

The limitations of the Boolean model were motivating factors in the research of graded retrieval models like the vector model or probabilistic model. The vector model represents both queries and documents as vectors in the vector space whose dimensions are defined by the set of all terms in the information retrieval system. The value of each component of the vector represents the weight of the corresponding term to the document or query.

The approach used to determine the weight of a term has significant impact on the calculation of relevance values by the IR system (Salton & Buckley, 1988). Basic concepts to calculate term weights is the use of term frequency (tf) and inverse document frequency (idf).

Term frequency is the number of occurrences of a term in a document. The inverse document frequency is defined as the logarithm of the quotient of the number of documents in the IR system and the number of documents in which the term is included.

$$idf_t = \log \frac{N}{df_t} \tag{5.1}$$

Based on the tf, idf and a normalization factor, the weights of all terms for all documents are calculated.

The similarity between a query and a document surrogate is done on the basis of a document vector and a query vector, a frequently used measure is the cosine of the angle between the two vectors:

$$sim(d_1, d_2) = \frac{\vec{V}(d_1)\vec{V}(d_2)}{|\vec{V}(d_1)||\vec{V}(d_2)|} \tag{5.2}$$

The vector model assigns a graded relevance value to each document held by the retrieval system. It is a vague retrieval model in the respect that for queries containing multiple terms, it can also return documents that do not contain all terms. The vector model is better suited for inexperienced users who lack the skill or inclination to formulate complex queries based on the Boolean model. At the same time these properties of the vector model limit its applicability for the formulation of complex and precise queries.

In addition to the Boolean and the vector models, another relevant retrieval model is the probabilistic model (Robertson *et al.*, 1980) that applies probabilistic theory to the Information Retrieval task and provides graded relevance values for queries. The extended Boolean model (Salton *et al.*, 1982) combines the support of the Boolean model for formulation of complex queries with the ability to return documents graded by relevance.

The combined support for complex queries and graded relevance ranking is a feature of most current IR systems, independent from the internally used retrieval model.

5.1.3 Evaluation for Information Retrieval

Since the effectiveness of an Information Retrieval system is defined as its ability to identify documents relevant to a user's information need and as the relevance of a document with regards to an information need cannot be formally proven, the effectiveness of an IR system must be assessed by empirical means. In order to evaluate an IR system and to compare the effectiveness of different systems, a standardized collection of information needs, documents and in some cases relevance assessments for the documents regarding the information needs are used. We call such a collection an evaluation collection. Using a collection, the evaluation of an IR system is done as follows:

- Every information need included in the corpus is formulated as a query that is then executed on the IR system.
- The documents returned for the query as well as their ranking are recorded.
- The (ranked) list of documents for each query is compared with the relevance assessments included in the corpus. If the corpus does not include assessments for all documents, result documents are intellectually assessed.
- Metrics for the effectiveness of the IR system are calculated based on the relevance assessments. The MTCC approach supports the calculation of Relevance and Precision as explained below

A number of metrics exist for the evaluation of IR systems. The two most basic measures in use are Recall and Precision.

$$Recall = \frac{Number\,of\,relevant\,documents\,in\,the\,result\,set}{Number\,of\,relevant\,documents\,in\,the\,IR\,system} \tag{5.3}$$

$$Precision = \frac{Number\,of\,relevant\,documents\,in\,the\,result\,set}{Number\,of\,documents\,in\,the\,result\,set} \tag{5.4}$$

Recall describes the completeness of a result set in relation to all relevant documents in the IR-system, precision the ratio of relevant documents and documents that were falsely returned as relevant. Since optimizations of an IR system for either recall or precision will generally have a negative impact on the other measure, recall and precision values for a system must always be considered in combination (Manning *et al.*, 2008).

Evaluation of Graded Retrieval Models

As described above, recall and precision are best suited for the evaluation of systems with a binary concept of relevance. Graded retrieval models return a ranked list of documents, for such systems and for large result sets it is not only important that relevant documents are part of the result set but also that they are ranked correctly, since most users will only inspect the first, most highly ranked documents (Blair, 1980; Cooper, 1973). Because recall and precision are based on sets and thus ignore the order of result documents, other metrics are needed for the evaluation of graded IR systems.

One means for the evaluation of ranked result is a Precision-Recall chart. Such a chart displays the precision of a result set at different levels of Recall. Figure 24 displays a variant of such a chart, an averaged 11-point precision / recall graph. An advantage of such a representation is that it eases the comparison of results for different evaluations. As illustrated by Figure 24, precision falls with increasing recall, the first results returned by the system are therefore the most likely to be relevant by the judgement of the system.

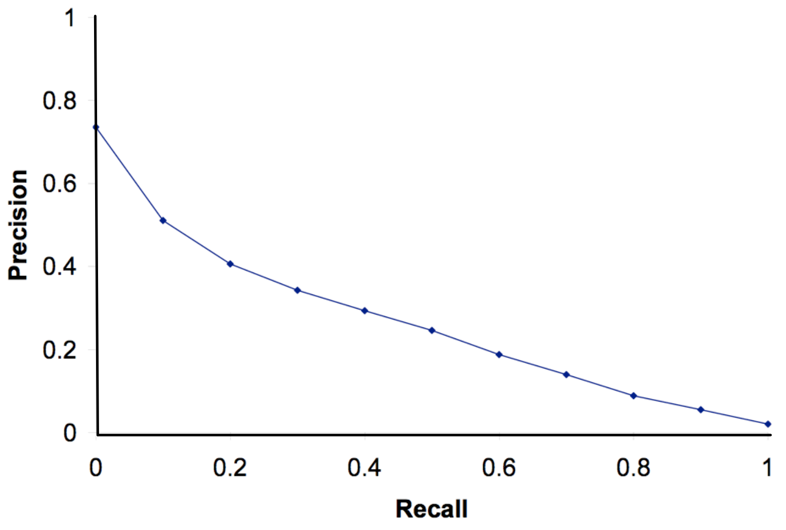


Figure 24: Averaged 11-point precision / recall graph (Manning *et al.*, 2008)

For a number of situations, for instance, when it is likely that a user will only inspect a limited number of documents, regardless of their relevance, the precision value after the first k documents is better suited to evaluate an IR system than precision at a specific recall level. Precision at k and R-Precision are measures applicable for such situations.

As the name implies, precision at k measures the precision of the first k documents of a result list. R-Precision considers the precision after the first k documents known to be relevant of been returned, compared to precision at k, this allows for better comparability between queries with large difference in the number of returned results.

Beyond the measures presented here, a number of specialized measures exist, for example for web-retrieval (Lewandowski, 2007). Mandl (Mandl, 2008) discusses the quality of websites for Information Retrieval with respect to automatic assessment of quality.

Limitations of the Notion of Relevance

The evaluation process used in Information Retrieval and the measures used in that process are based on a number of simplifications that have to be considered in the interpretation of results. One limitation of the evaluation processes outlined above is the use of binary relevance assessments for the documents in the corpus. While this is a simplification of reality, it does not put evaluations done with this measure under question (Manning *et al.*, 2008).

A more fundamental simplification is the assumption that the concept of relevance is static for every user at each moment of time. This ignores the fact that, for instance, the relevance of a document changes once this document is known to the user.

A final weakness of the evaluation process is that it is assumed that the information need of a user is fully formed and can be expressed in a single query. This ignores the fact that the Information Retrieval process is a dialog and that the information need of a user is evolving (Schamber *et al.*, 1990).

The evaluation of Information Retrieval as a dialog is not possible with the system-centric evaluations discussed above. A user-centric (Voorhees, 2002) approach that includes aspects like the usability of a system is needed instead. Because MTCC abstracts from the interface that a user employs to interact with the SUT, is not intended to serve as a tool for a user-centric evaluation.

5.2 Digital Libraries

Digital Libraries are specialized Information Retrieval systems. The digital libraries discussed in this thesis are topic-specific portals that provide scientific information (Mayr *et al.*, 2005) for specific fields of science. This thesis investigates a family of on metadata-driven Digital Libraries, primarily for the social sciences. We argue that Digital Libraries outside this system family, for example the ACM[1] or the IEEE Digital Library[2], can also be represented and tested using the MTCC approach.

We use the term portal to indicate that these particular Digital Libraries integrate various different databases but use the term Digital Library for the remainder of this text.

Figure 25 illustrates the relations of portals, topic-specific portals and databases.

The fact that the Digital Libraries discussed here integrate several databases raises the complexity of each system considerably. One reason for this increase in complexity is the presence of data from a number of different sources. The Digital Libraries discussed here can include the following types of information (Mayr, 2006):

OPACs Online Public Access Catalogs contain information about the books, periodicals, and other media available in a library. The records of an OPAC usually contain bibli-

[1]`http://portal.acm.org/`

[2]`http://http://www2.computer.org/portal/`

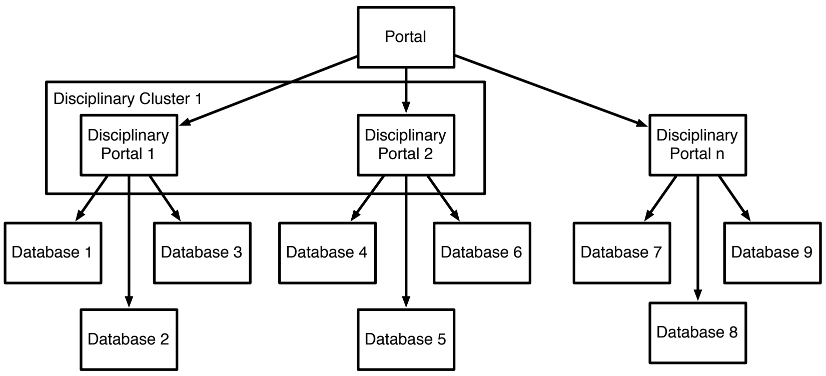


Figure 25: Portals, disciplinary portals and databases (Mayr *et al.*, 2005)

ographic data and a subject description in an indexing vocabulary, but do not have an abstract.

Full-text Electronic full-text may be either available at costs or, for instance in the case of Open Access Repositories, for free. While the cost of full-texts is not directly relevant to IR, the treatment of free and at cost resources raises the complexity of a Digital Library.

Topic Guides Topic guides (Fachinformationsführer) describe Internet resources of relevance for a particular field of science, for instance, the homepages of institutions.

Topic Database Topic databases (Fachdatenbanken) are collections of information that supply scientists with research-relevant information. A important aspect of topic databases are bibliographic records, often containing an abstract as well as subject descriptions from a indexing vocabulary.

In addition to these information types, Heinz and Stempfhuber (Heinz & Stempfhuber, 2007) list 14 different types of information, called modules, among them expert data bases and event calendars.

The integration of potentially structurally as well as semantically heterogeneous databases in a Digital Library gives rise to a number of challenges concerning the implementation technology, information science, and organizational / legal issues.

One approach to the management of different information types is the Shell Model defined by Krause (Krause, 1996, 2006a). As illustrated in Figure 26, the Shell Model places different kinds of information in concentric layers around a core of high-quality information. The shell model thus supports the organization of different types and qualities of information by their likely relevance to the user of a digital library.

5.2.1 Structural and Semantic Heterogeneity

Structural and semantic heterogeneity (Stempfhuber, 2003) are a consequence of the integration of multiple databases within one Digital Library.

Databases are structurally heterogeneous when the schema used to describe information differs. Differences can include different field names, incompatible data types and ranges of legal values and different granularity used to describe concepts. Structural heterogeneity is a relevant problem outside the domain of Digital Libraries (Hasselbring, 1997, 2002b; Pedersen & Hasselbring, 2004). Rahm and Bernstein (Rahm & Bernstein, 2001) give an overview of the challenges imposed by structural heterogeneity and outline some approaches to their solution.

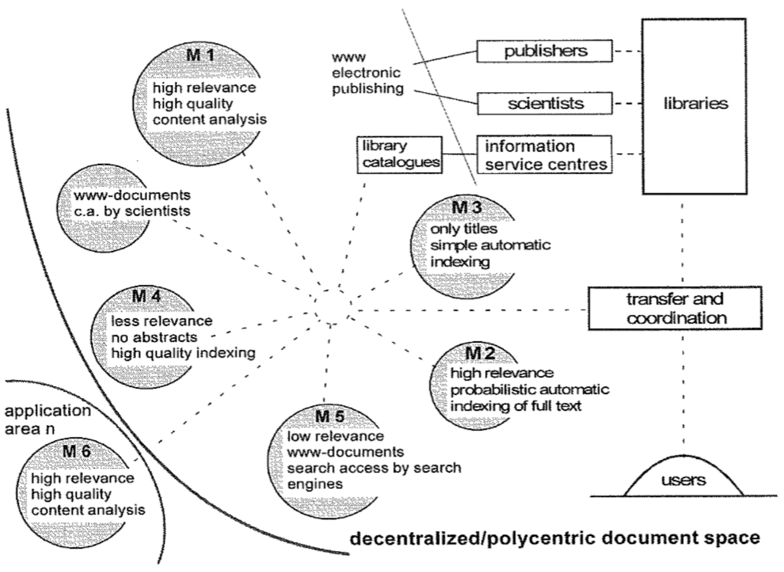


Figure 26: Shell Model(Krause, 2006a)

Semantic heterogeneity is caused by the use of different Knowledge Organization Systems (KOS) like thesauri, classifications and subject lists for the different databases. Since a user of a Digital Library cannot be expected to be familiar with multiple KOSs, it is important to make the different KOSs used in a Digital Library interoperable (Zeng & Chan, 2004). One means to provide such an operation are cross-concordances (Mayr & Petras, 2008).

Structural and semantic heterogeneity as well as use of methods for treatment of such heterogeneity like cross-concordances increases the complexity of a Digital Library as a technical system. This rise in complexity increases the probability of defects and thus necessitates more testing.

5.2.2 User-oriented Quality of Digital Libraries

If we define the quality of a Digital Library as the degree to which it satisfies the needs of a user, it becomes obvious that the quality of a Digital Library may be different to each user. Surveys concerning the factors that define the quality of a Digital Library to a user (Thurmeier, 2007) place emphasis both on the importance for a Digital Library to support complex queries and on the usability of the system. In more detail, users have the following requirements for a system (Heinold, 2007) (translation by the author):

- *High coverage of potential information sources*
- *Ease of the search process:*
  - *quickly accessible search forms for a simple search*
  - *an uncomplicated way to modify the results of a search that allows the reuse of previous searches.*
- *Simple assessment of the relevance of results:*
  - *by the preview of a document in the list of results*
  - *by sorting the result by different criteria*

- *Simple access to the needed information*
    - *links to document sources*
    - *direct and uncomplicated access to electronic texts*

The significant role of usability is confirmed by Lewandowski (Lewandowski & Hoechstoetter, 2007) in the context of web-retrieval. It was also found that advanced functionality for such systems, for example, Boolean operators, were rarely used. While it is unclear, how far these findings apply to the users of Digital Libraries, we argue that differences between available functionality and used functionality are likely to exist for digital libraries as well. In combination with the survey results outlined above we argue that the assessment of the quality of a Digital Library must involve users.

5.2.3 Testing of Digital Libraries with MTCC

In order to test a Digital Library in practice, some simplifications have to be accepted. The following are relevant to testing with MTCC:

- The ability of a Digital Library to provide information relevant to an information need can only be tested using the system-centric evaluation approaches outlined in the previous section. Since MTCC abstracts from the specifics of the graphical user interface, testing of IR as an interactive process involving the information need of a user at a specific point in time is limited.
- Tests can only represent a static information need, the expected behavior of a test executed multiple times cannot change.

Similar to the evaluations of IR effectiveness, testing IR systems is based on simplifications. Figure 27 puts the previously discussed concepts of Information Retrieval in relation to the abstractions used in MTCC. A query is a representation of an information need, tests are representations of the requirements for a system. As an IR system can only judge the relevance of a document, tests in MTCC can only verify the testable aspects of a system that are represented by the system model.

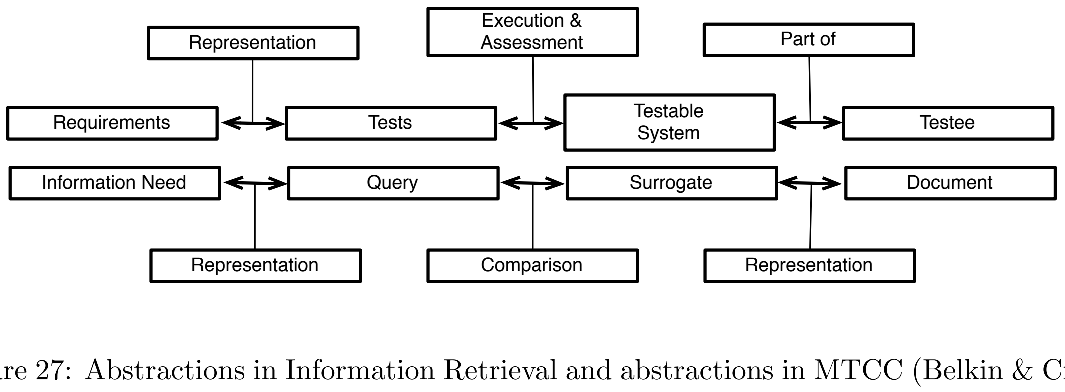


Figure 27: Abstractions in Information Retrieval and abstractions in MTCC (Belkin & Croft, 1987)

As discussed in Chapter 2, testing can only improve the quality of a system by finding faults. Applied to the testing of Digital Libraries with MTCC, this means that MTCC must be able to model the Test Steps and Services needed to fulfill Information Retrieval tasks with MTCC. In particular, MTCC must be able to compare the expected and the actual results for a query.

The fact that MTCC is a Model-Driven approach and thus abstracts from some aspects of tested systems has consequences for the aspects of digital libraries that can be tested with MTCC. The graphical user interface of a system determines what interactions between a user and a IR system are possible and is an important aspect of the user-centric quality of a system. Since MTCC does abstract from all aspects of the user interface, the user interface aspects of a IR system cannot be tested in MTCC. More generally, MTCC does not support testing of any usability-related aspects of a system.

MTCC does support the testing of the functional aspects of a IR-system within the constraints of the Cranfield paradigm (Voorhees, 2002) as well as the testing of the technical functionality and correctness of a system.

5.3 Chapter Summary

The goal of Information Retrieval is to fulfill the information need of a user on a collection of document by identifying those documents that are relevant to the user. Because an information need is an inherently vague concept, Information Retrieval systems are based on a retrieval model that describes how informations needs can be encoded as queries and how a relevance assessment for a query and a document representation can be derived.

Digital Libraries are Information Retrieval systems that store digital documents or metadata. This thesis applies MTCC to a system family of three Digital Libraries for scientific information, primarily bibliographic metadata.

Two types of Information Retrieval evaluation can be distinguished, system-centric evaluation and user-centric evaluation. In a system-centric evaluation, the quality of an Information Retrieval system is assessed based on the results returned for isolated queries derived from a number of standardized information needs. The degree to which the returned documents fulfill the information need for the query is established based on relevance assessments by domain experts. Metrics like MAP, Recall and Precision describe the effectiveness of an Information Retrieval system. A user-centered evaluation of Information Retrieval does not only consider individual queries but treats Information Retrieval as an interactive process. In order to conduct a user-centered evaluation, the usability of a IR system and therefore its user interface must be taken into account.

MTCC models abstract from the user interface of a SUT, MTCC therefore does not support user-centric evaluations of Information Retrieval systems. MTCC does support the verification of Recall and Precision values for queries defined by domain experts for a collection with existing relevance assessments.

Part II

Model-Driven Test Case Construction

Chapter 6

The MTCC Approach

This chapter gives a detailed overview of the test construction process with MTCC, the roles involved in the process and the models and artifacts used.

The MTCC testing process consists of five steps: (1) the analysis and modeling of the system family under test, (2) the modeling of testees from the system family, (3) the composition of a system-specific test model, representing the available tests for a system, and finally (4) the construction of tests represented as sequences of Test Step configurations and (5) the transformation of these abstract representations into executable Test Steps.

The MTCC process distinguishes two different roles for participants in the test construction process, domain experts and domain engineers. Domain experts provide knowledge about the functional requirements of a system, domain engineers contribute implementation knowledge and modeling skills. Section 6.1 describes the phases of the MTCC test construction process and the responsibilities of the different roles.

MTCC uses a number of different models. Models address (1) different levels of specificity — for instance at the system family level and system level — and (2) different entities such as the Services of a system and the Test Steps that exercise these Services and (3) different aspects of a system, specifically its features and its dynamic behavior. Section 6.2 introduces the different types of models used in MTCC.

6.1 The MTCC Process

The testing process based on MTCC consists of three phases followed by an external test execution phase. Figure 6 on page 18 illustrates these phases and their respective activities, Figure 28 illustrates the relation of the MTCC Phases to the phases of the system family engineering approach as presented by Czarnecki (Czarnecki, 2005).

In the *Domain Engineering* phase, a system family is analyzed with respect to its test relevant Services, the features of these Services and the Test Steps needed to exercise these. Domain Engineering in MTCC is comparable to Domain Analysis (Czarnecki & Eisenecker, 2000; Arango, 1989, 1994) applied to system families, it is also analogous to Software Product Line Engineering in that core assets for the testing of the system family under test are identified and implemented.

In *Application Engineering*, the results of domain engineering are refined. Models describing the system family or domain are specialized for the representation of specific systems. The focus of *Test Engineering* is the utilization of the test editor that supports domain experts in modeling tests based on the models instantiated in the previous phases of MTCC. Test modeling with the editor is done by the selection and configuration of Test Steps, we call this Test Step instantiation. A Test Step instance represents a concrete interaction with a Service

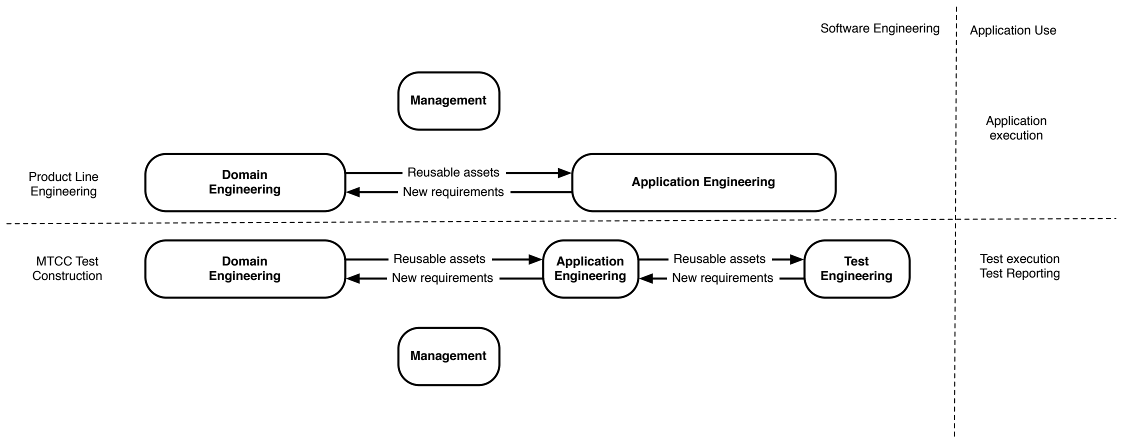


Figure 28: Phases of MTCC and system family engineering(Czarnecki, 2005)

instance of the testee. The feature model of a Test Step instance is a specialization of the Test Step feature model that represents the configurations for the Test Step instance, taking into account the specific features of the Service addressed by the Test Step. The parameterization of a Test Step is expressed as the configuration of the feature model representing the Test Step instance. The use of feature models to model the inputs and expected results of a test case leads to a more fine grained use of feature models than in product configuration, where features often represent the general capabilities of a system (Wagelaar & Straeten, 2006). In the test code generation activity, Test Step configuration serves as the base for the configuration of executable test cases.

The last phase of Testing with MTCC is not part of the core MTCC process. How the activities of this last phase, test execution and test reporting, are conducted is determined by the test runner for which tests are generated and by the general testing regime in place.

While each of the phases of the MTCC process builds on the results of the previous phase, MTCC does not assume a sequential execution of the different phases. We assume that domain engineering, Application Engineering and test engineering are executed in iterations.

MTCC can be considered a product-line approach for the construction of tests. Figure 29 illustates the relationship of the work on applications within a domain and the domain infrastructure in Product Line Engineering. Applied to the testing process of MTCC, three processes corresponding to the phases of the MTCC process. In test engineering, tests are constructed based on the assets implemented in domain engineering and Application Engineering. Requirements for the models and testing infrastructure flow back to Application Engineering, requirements for new classes of Services and Test Steps serve as input to domain engineering. Additions to the MTCC domain models from domain engineering are used to model new aspects of systems in Application Engineering. The core assets of the MTCC testing process are the domain models, the editor, the routines for model composition and the infrastructure for test generation. Figure 28 illustrates the relationship of the three MTCC phases to Application Engineering and Domain Engineering in System Family Engineering (Czarnecki, 2005). The concept of System Family Engineering as used in the figure is comparable to our concept of Product Line Engineering.

The different roles involved in the overall testing process with MTCC are expected to be in constant communication. Domain experts construct tests and inform domain engineers about test-relevant, but not yet testable functionality. Domain engineers extend and maintain the models and tools used in the test construction process and oversee the generation and execution of test cases.

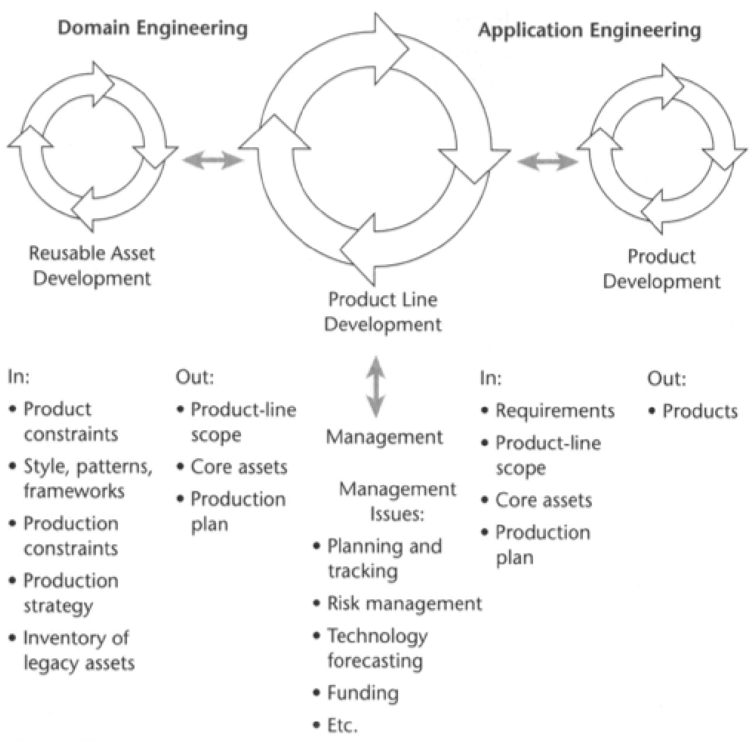


Figure 29: Product Line Practice as defined by the SEI Framework (Czarnecki & Eisenecker, 2000)

6.1.1 Roles and Relation to Software Development

Two roles are distinguished in the MTCC testing process, domain experts and domain engineers:

- Domain experts have knowledge about the functional requirements of the application domain or at least one system in the application domain, but do not know how systems in the domain are implemented.
- Domain engineers have knowledge about the implementation of the systems within the system family and have the necessary skills to implement and maintain the artifacts and models of the MTCC approach, but do not have sufficient knowledge about the functional requirements of the system to implement tests.

Domain engineers and domain experts cooperate in the application of the MTCC processes. MTCC assumes that the system under test is available at least as a prototype and that the execution of tests generated from MTCC models on the testee is possible.

Roles in the MTCC Process

Roles in MTCC are defined by their knowledge about the functional requirements or technical implementation of the system under test.

It is one purpose of MTCC to support the capture and formalization of domain experts' knowledge and operationalize it as information for the verification of systems.

The capture and formalization of requirements is facilitated by the MTCC editor, a tool that allows the construction of tests for specific systems without requiring that the user possesses skills in programming or formal modeling.

The design and realization of the MTCC models as well as the implementation of the MTCC infrastructure and its maintenance are the responsibility of the domain engineer. He or she must be able to represent the test-relevant aspects of the domain identified by domain experts in the appropriate models.

We argue that the cooperation of domain experts and domain engineers is a prerequisite for the implementation of high-quality software that meets the requirements of its users. We see this notion explicitly reflected in a number of software development methods like Domain Driven Development (Evans, 2004). We argue that the need for cooperation applies to testing software as well.

In addition to the differentiation of roles by the respective knowledge of their holders, it is also possible to distinguish roles based on the MTCC phase they are active in. For instance, domain engineers working on the MTCC core assets need different skills than system engineers that model the Services of one particular system and maintain the Test Adapter necessary for test execution. It could be argued that domain engineers and system engineers as well as domain experts and system experts (knowing the functional requirements for one system) constitute different roles. For this thesis, we only use the roles of the domain engineer and the domain expert. While good reasons exist for the definition of additional roles (Krahn *et al.*, 2006), we argue that for our prototype implementation of MTCC and its evaluation, such roles are not needed.

Relationship of MTCC to Product Development

MTCC defines two preconditions that the software development process whose products are to be tested must fulfill: The software development process must have incremental elements and the testable version of the SUT must be available when MTCC is used.

MTCC can only be used within the context of software development processes that have iterative or incremental elements (Larman & Basili, 2003). The naive use of the waterfall-model (Royce, 1987) in a strictly sequential manner as well as the classical V-Model (Broy & Rausch, 2005) and all other models that do not allow for requirements discovered in testing to feed back into the design and implementation process are not well-suited for MTCC, since it is not possible for tests to influence the development process in a form of executable specification.

Another reason for the use of MTCC in iterative processes is the need for a testable system, either a prototype or a version of the final system, that supports the execution of tests and thus the decision whether the requirements defined by domain experts are meet by the system.

Since the roles in MTCC are compatible with intensive cooperation of software developers and customers in agile approaches to software development (Beck & Andres, 2004), we argue that MTCC can be used in such processes. MTCC tests can be an executable alternative to user stories in situations where MTCC models for the functionality to be implemented already exist and the implementation in the context of the system is planned next.

MTCC makes no assumption about the use of any particular method for the identification of the test-relevant features of a system and about the process used by the domain experts to determine relevant tests for these features. Methods from the field of requirements analysis (Mylopoulos, 2005; van Lamsweerde, 2001; Kulak & Guiney, 2003) can be used as well approaches that are explicitly designed to be used in the testing context (Maletic *et al.*, 2000).

Relationship of MTCC to the Testing Process

MTCC is an approach to acceptance testing. The MTCC approach is a supplement to other testing techniques. MTCC does not replace approaches such as developer testing or Model-Driven system testing, nor does it compete with approaches to SPL testing.

6.1.2 Modeling of the Domain and Individual Systems

The first phase of the MTCC approach is domain engineering. From the modeling perspective, domain engineering in MTCC is an analysis process with the purpose of identifying test-relevant functionality of the testees, followed by the formalization of the identified functionality as models.

In the Application Engineering phase, concrete instances of systems are modeled as specializations (Czarnecki *et al.*, 2005b) of the domain models realized in the domain modeling phase. The dynamic behavior of the system is modeled as a finite state machine.

Domain Engineering

The purpose of domain engineering is the analysis of all systems within the system family for test-relevant Services and Test Steps. The results of this analysis process are formalized in models. The Domain Feature Model represents all test-relevant Services in the domain as feature models. The Domain Test Model represents the Test Steps that serve to exercise and verify these Services.

The first step of domain engineering is domain analysis (Arango, 1989; Czarnecki & Eisenecker, 2000). The subject of analysis are all features (Kang *et al.*, 1990) of the members of the considered system family that are test-relevant. A feature is test-relevant when it is needed for the specification of acceptance tests by domain experts. In MTCC, test-relevant features are grouped into Services. A Service is a combination of features that represent some aspect of a functionality relevant for the system family. For instance, the functionality needed to start a search or a list of results for a query are both Services for the system family of Digital Libraries. Within a system family and frequently within a system, multiple different instances of a Service will exist, for example, different forms used to start a search. The variability between this different instances is expressed in the feature model of a Service

In addition to Services, domain engineering identifies the Test Steps used to exercise a Service Each Test Step is either an action that manipulates the Service or an assertion that compares some aspect of the state of the Service with an expectation. Every Test Step is specific to one of the Services of a system family. Each Service has multiple Test Steps assigned to it. An overview of all Services and Test Steps for the system family of digital libraries considered in this thesis can be found in Section 9.4. The MTCC activities of Service modeling and Test Step modeling build on the results of the analysis activities and formalize them in two models, the Domain Feature Model and the Domain Test Model.

The Domain Feature Model describes each identified Service as a feature model. The structure of the feature model represents the various features of a Service the variability (Czarnecki *et al.*, 2006b; Coplien *et al.*, 1998) of the feature model represents the difference that can exist between multiple instances of the same Service For the search form mentioned above, different instances can vary in the number of fields available to formulate a search as well as in the advanced search options that are supported. Each Service Domain Feature Model is expressed as a feature model with cardinalities as introduced in Section 4.2. The Domain Feature Model is a catalog for Services that exist for the considered system family. It does not describe how Services are combined to describe a system, the workflow of a system or any aspect of the dynamic behavior of the system.

The Domain Test Model represents all Test Steps that exist for the Services described in the Domain Feature Model. Every Test Step is specific to one Service it represents either an action that executes the Service and thus manipulates its state or an assertion that either verifies if some expectation about the state of the Service is true or reads some aspect of the Service state and saves it for later use. Submitting a search form and thereby starting a search is an example of an action, reading the number of results from a list of documents is an example of an assertion.

Test Steps are represented as feature models that contain references to the feature model of the Service exercised, these references are unfolded in the Model Composition activity of Test Engineering and allow the inclusion of those features of the feature model of a Service instance that are relevant to test construction.

Application Engineering

In the Application Engineering phase, models for one concrete system from the considered system family are realized.

The Application Feature Model describes a system in terms of its Service instances. Each Service instance is a specialization of one of the Services described in the Domain Feature Model. It is possible and common that multiple different instances of one Service exist for a system, for instance, a simple search form for unexperienced users and a more complex search form for experts.

Details about the structure of the feature models included in the Application Feature Model and the information contained in these models can be found in Section 7.2.

The Application State Model describes the dynamic aspects of a specific system. The model is represented as a non-deterministic finite state machine that represents possible traces through the system.

The finite state machine of the Application State Model distinguishes two kinds of states: Context states and Service states. Only Context states can be reached from the initial start state of the FSM. From a Context state, only Service states can be reached. The transition from a Context state to a Service state corresponds to the selection of one Service from the set of all Services defined in a given Context. From a Service state, only Context states can be reached, the state transitions from a Service to a Context state represent the execution of a Service and the resulting change of the current Context. For example, the execution of a search Service can result in a transition to a Context representing a page that displays the result of a search or, in cases where no documents were found, a transition to a Context representing the ”no-results found” page. Section 7.1 discusses the dynamic models used in detail.

Figure 30 illustrates the relevant concepts represented by the MTCC model at the system family and system level as well as the relations of the these concepts. A system in MTCC is represented by a number of Contexts. Every Context contains a number of of Service instances. While a Context has an unique name, it is primarily defined by the Service instances it contains. For web-applications like the Digital Libraries, a Context will typically correspond to a page of the application. A concrete example for a Context in such a system would be the page that displays the results of a search. This Context would include Service instances representing one or more lists of result documents, links to other pages of the application, and controls to manipulate a list of the results, for example by changing its sorting.

Figure 31 displays one page of a Digital Library that corresponds to a Context. The context includes the following seven Services: (1) Information about the search that was sent to the digital library, (2) controls to browse the results of the search, (3) feedback about the number of documents found, (4) controls to change the sorting of a list of results, (5) facets

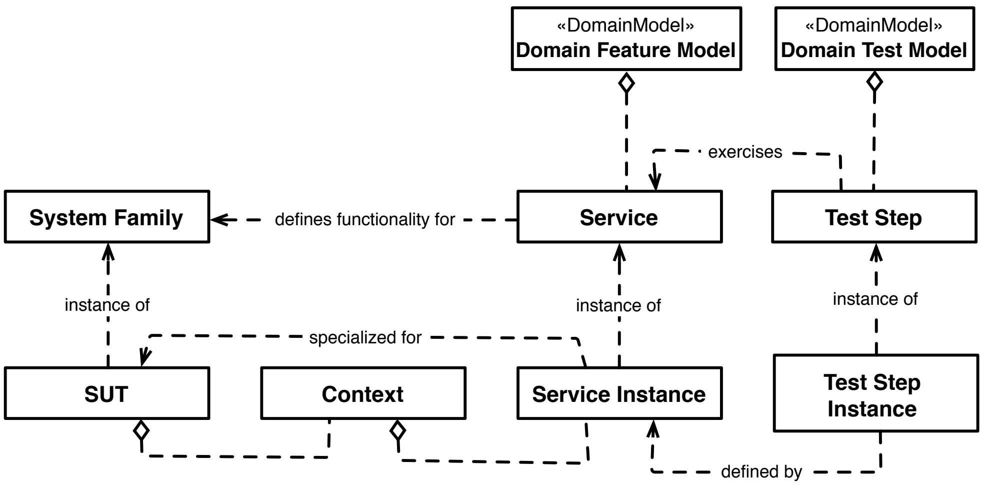
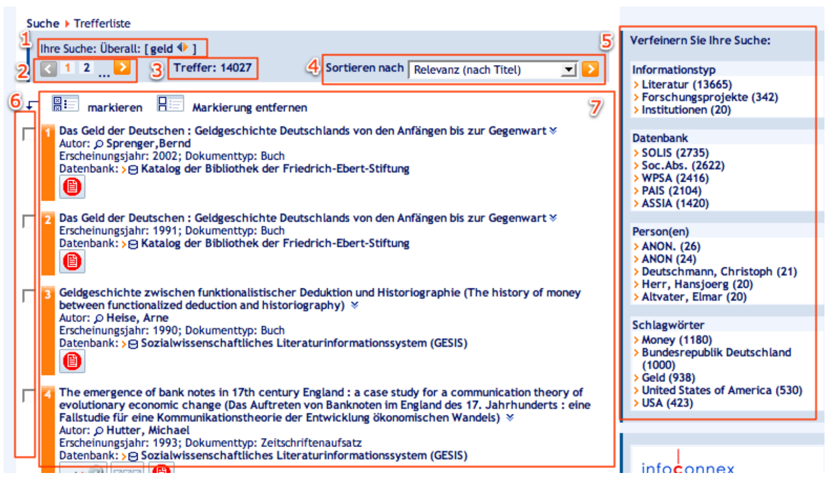


Figure 30: Relevant concepts of MTCC on the system and system family level

to further restrict the results of the search, (6) controls to mark result documents for experts, and (7) a list of result documents for a search.



1 Das Geld der Deutschen : Geldgeschichte Deutschlands von den Anfängen bis zur Gegenwart
Autor: Sprenger,Bernd
Erscheinungsjahr: 2002; Dokumenttyp: Buch
Datenbank: Katalog der Bibliothek der Friedrich-Ebert-Stiftung

2 Das Geld der Deutschen : Geldgeschichte Deutschlands von den Anfängen bis zur Gegenwart
Erscheinungsjahr: 1991; Dokumenttyp: Buch
Datenbank: Katalog der Bibliothek der Friedrich-Ebert-Stiftung

3 Geldgeschichte zwischen funktionalistischer Deduktion und Historiographie (The history of money between functionalized deduction and historiography)
Autor: Heise, Arne
Erscheinungsjahr: 1990; Dokumenttyp: Buch
Datenbank: Sozialwissenschaftliches Literaturinformationssystem (GESIS)

4 The emergence of bank notes in 17th century England : a case study for a communication theory of evolutionary economic change (Das Auftreten von Banknoten im England des 17. Jahrhunderts : eine Fallstudie für eine Kommunikationstheorie der Entwicklung ökonomischen Wandels)
Autor: Hutter, Michael
Erscheinungsjahr: 1993; Dokumenttyp: Zeitschriftenaufsatz
Datenbank: Sozialwissenschaftliches Literaturinformationssystem (GESIS)

Verfeinern Sie Ihre Suche:

Informationstyp
- Literatur (13665)
- Forschungsprojekte (342)
- Institutionen (20)

Datenbank
- SOLIS (2735)
- Soc.Abs. (2622)
- WPSA (2416)
- PAIS (2104)
- ASSIA (1420)

Person(en)
- ANON. (26)
- ANON (24)
- Deutschmann, Christoph (21)
- Herr, Hansjoerg (20)
- Altvater, Elmar (20)

Schlagwörter
- Money (1180)
- Bundesrepublik Deutschland (1000)
- Geld (938)
- United States of America (530)
- USA (423)

Figure 31: Service instances in a web page

Implementation of Infrastructure Code

After the Application Feature Model and the Application State Model for a testee are realized, the last activity of the Application Engineering phase is the implementation of infrastructure that is necessary for the execution of tests. Two artifacts have to be implemented in MTCC to allow the generation and execution of test cases.

Templates allow the generation of test cases for one specific test runner. The Test Adapter is a library that provides functionality needed for the execution of the previously generated test cases. The Test Adapter is specific to the system tested and to the test runner used to execute the tests.

6.1.3 Composition of Models

The MTCC editor supports the construction of tests for a member of a software family based on the information contained in the Application Feature Model, Application State Model and Domain Test Model. The use of these models enables the editor to guide a user in the test construction process by allowing only the construction of tests that are supported by the system represented by the model.

In order to reduce the complexity of the editor, the editor does not use the Application Feature Model, Application State Model and Domain Test Model as separate models. Instead it uses an integrated model, the Application Test Model. The Application Test Model is a composition of the three initial models, it includes both information about Services and their associated Test Steps and information about the possible sequences of Test Steps that can be executed on a system.

While the Application State Model and the Application Test Model represent a system in terms of its Services, the representation used by the Application Test Model is based on the Test Steps that can be executed on a system, the configurations that are possible for the Test Steps and the possible sequences of Test Steps.

Reasons for the Composition of Models

In order to discuss the reasons for the composition of the Application Feature Model, Application State Model and Domain Test Model into the Application Test Model, we first discuss our reasons for separating these models. The decision to use three different models and only to integrate the information contained in these models in the Application Test Model immediately before the instantiation of the editor has conceptual as well as practical reasons:

- By representing Service instances in the Application Feature Model and Test Steps in the Domain Test Model, Test Steps only need to be defined once at the system family level. Service instances have to be defined at the system level.
- The definition of Test Steps at the system family level helps to avoid the redundancy that would result from the inclusion of all Test Steps for all referenced Service instances. It thus has positive influence on the maintenance of the models. For instance, in order to add a new Test Step to a Service the Test Step only needs to be added to the Domain Test Model, the Service instances exercised by the Service need not to be changed.

The information contained in the Application Feature Model and the Application State Model is best expressed with different types of models. While a feature model is by definition an excellent representation for the possible configurations of a Service instance in the Application Feature Model, a FSM is better suited to describe the possible sequences of Service invocations included as described by the Application State Model.

While the use of separate models eases the maintenance of the models, such a representation as an internal model for the editor is disadvantageous for two reasons:

- The editor needs to represent a system in terms of the Test Steps that can be executed on it, not in terms of its Services.
- The use of separate models eliminates redundancy in the models, but raises the complexity necessary to work with the models.

The Composition Process

The Application Test Model has to parts: a catalog of Test Step instances represented by feature models and a dynamic part represented by a FSM. We call the catalog of Test Step instances the structural part.

The structural part is created from the Application Feature Model by replacing every Service instance with instances of those Test Steps that are assigned to it and can be executed on it. The Test Step instances are created by checking all Test Steps defined in the Domain Test Model for compatibility with the Service instances defined in the Application Feature Model. If the Test Step and the Service instance are compatible, a Test Step instance is created by unfolding the feature references in the Test Step and is added to the catalog.

The dynamic part of the Application Test Model is created by rewriting the Application State Model in such a way that Service state is replaced by a number of Test Step states in the same way that Test Step instance replace Service instances. The resulting FSM describes all possible sequences of Test Steps that can the executed on a testee.

6.1.4 Modeling of Test Configurations

The editor supports the construction of test cases based on the dynamic and structural information in the Application Test Model.

From a technical point of view, a test is constructed by building a sequence of configured Test Step instances. The FSM in the dynamic part of the Application Test Model determines which sequences of Test Steps are available. The feature models of the structural part define how each Test Step instance can be configured.

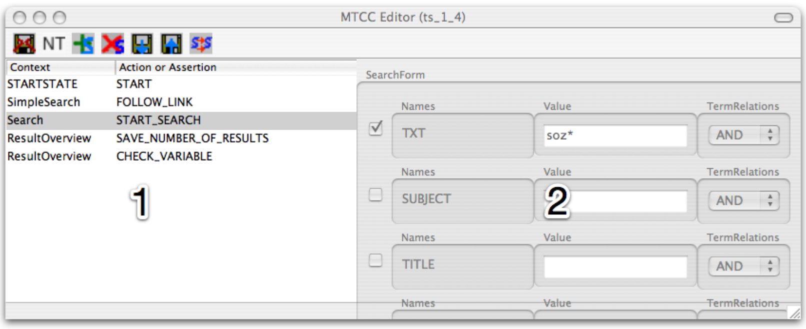
From the users point of view, test construction is defined by two activities: The selection of Test Steps that represent a usage scenario that is to be tested and the parameterization of each individual Test Step.

Like Action Word (Blackburn *et al.*, 2004) based approaches to testing or FIT (Mugridge & Cunningham, 2005b) workflow tests, MTCC represents a test as a sequence of individual steps that manipulate and verify the testee. The combined Test Steps represent one possible session of a user with a system. From a technical point of view, a MTCC test is a trace through the functionality of the system. One property specific to MTCC is that the selection and combination of Test Steps is guided by the editor. A user can only construct those sequences of Test Steps that are supported by the system under test as described by the finite state machine of the Application Test Model. For example, a Test Step to submit a query would only be available if such an action would be supported by the testee in its current state. The selection of Test Steps determines the functionality of a testee that will be exercised by the test.

The parameterization of the individual Test Steps determines exactly how the Service instances of the testee are exercised and verified. The possible parameterization or configuration of each Test Step instance is defined by feature models in the structural part of the Application Test Model. The editor represents the specialization steps that can be applied to the feature model in terms of the elements of a graphical user interface. As it is the case with the selection of Test Step instances, the editor uses information from the Application Test Model to guide the user in test construction. A test in MTCC is a sequence of configured Test Step instances for one system of the system family. We call such a test a Test Configuration. An in-depth discussion of the MTCC editor is given in Section 8.1, details on the specialization of feature models for the purpose of the configuration of Test Step instances are given in Section 8.2.

Representation of Tests in the Editor

The editor displays the sequence of Test Step instances that define the behavior of a test as a list on the left part of the GUI. The Test Step instance that is currently being configured is displayed in the right part of the editor. The graphical user interface that represents each Test Step is dynamically built by the editor based on both the structure of the feature model of the Test Step instance and the semantics of individual nodes in the feature model. Figure 32 shows the MTCC editor. The Test Steps for the current test are shown in the left part of the GUI (1), the right part of GUI (2) displays a representation of the current Test Step.



Context	Action or Assertion
STARTSTATE	START
SimpleSearch	FOLLOW_LINK
Search	START_SEARCH
ResultOverview	SAVE_NUMBER_OF_RESULTS
ResultOverview	CHECK_VARIABLE

SearchForm

	Names	Value	TermRelations
✓	TXT	soz*	AND
	SUBJECT		AND
	TITLE		AND

Figure 32: Representation of the available and current teststeps in the MTCC editor

In a breadth-first traversal of the feature model, each node is checked against a number of GuiBuilder objects. Each GuiBuilder object is responsible for the display of specific nodes or subtrees of a feature model. When a GuiBuilder object is responsible for the display of a subtree of a feature diagram, it either builds a complete representation of GUI elements for the feature model or a partial representation and delegates the rendition of some subtree to other GuiBuilder object, similar to the application of the *Chain of Responsibility Pattern* (Gamma *et al.*, 1995).

The configuration of a Test Step instance done via the editor is represented by configuration nodes in the feature model. Configuration nodes are attached to the nodes of the feature model that are configured by the user. For instance, when an optional feature is de-selected by clicking the checkbox that represents that feature in the GUI, a configuration node representing refinement of the feature cardinality from [0,1] to [0,0] is attached to the feature node.

The representation of the specialization steps used to configure a Test Step instance as nodes of the feature model instead of directly applying them to the model has advantages for the representation of feature models in a GUI. Since the specialization steps used during configuration do not directly manipulate the feature model but are represented as a separate object, it is easy to undo and change the configuration of a feature model.

Reuse of Tests

One purpose of MTCC is the reuse of tests, more precisely of Test Configurations for multiple systems. Since a Test Configuration is constructed from the Test Step instances for one particular system as represented by the Application Test Model of that system, a test can be reused for another system if its Application Test Model supports the same sequence of Test Steps as the initial system and each individual Test Step instance supports the same configuration as the corresponding Test Step instance. Section 8.3 discusses test reuse in detail.

6.1.5 Test Script Generation and Execution

In order to use the abstract Test Configuration for testing, MTCC employs code generation. MTCC generates test cases for COTS or test runner. It is assumed that test scripts for such a test runner are compatible with the structure of xUnit (Meszaros, 2007) tests. The test cases generated by MTCC do not exercise the testee directly but use a manually implemented wrapper library, the Test Adapter. The Test Adapter provides implementations for each Test Step instance in the system family for a specific pair of system and test runner.

With the use of code generation and the Test Adapter, MTCC facilitates the decoupling of the semantics of the functionality to be tested and its implementation. Since MTCC does not assume the use of any particular test runner, the integration of MTCC in an existing testing process is also supported.

Generation of Test Code

Compared to other applications of code generation, the code generation approach in MTCC is relatively simple. Since test cases neither include complex data structures nor complicated control flow, a template-based approach to code generation is sufficient.

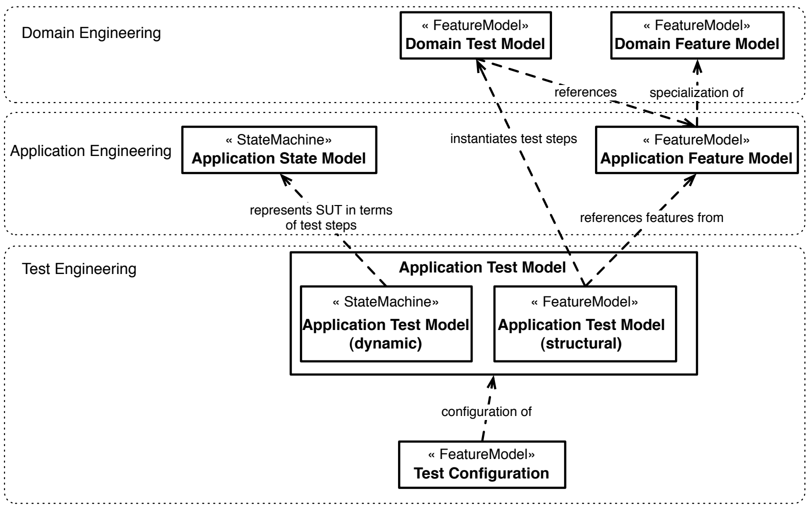
Every Test Configurationobject is represented by one test case. The test case is a sequence of method calls for methods defined in the Test Adapter. Each method corresponds to one configured Test Step instance in the Test Configuration.

The generation of test case code is done in two phases, in the first phase, the feature model of each configured Test Step instance is transformed into a simplified representation as a set of key-value pairs. This presentation is called the parameter object. In the second phase of the code generation process, method calls for each Test Step instance in the test are looked up from the Test Adapter library and code for the invocation of the method with the parameter object as its parameters are generated. Section 8.4 discusses the MTCC approach to test code generation. The execution of the generated tests as well the logging and reporting of test results are not part of the MTCC process.

6.2 Overview of the MTCC Models

The models used in the MTCC are directly related to the phases of the MTCC testing process. In Domain Engineering, Test Steps and Services relevant to the system family as a whole are expressed in the Domain Test Model and the Domain Feature Model respectively. Application Engineering derives the specific Services instances for one system by specializing the Domain Feature Model into the Application Feature Model. The Application Test Model describes the possible sequences of Service invocations for a system. In test engineering, the Domain Test Model, the Application Feature Model and the Application Test Model are merged into one composed model that describes a test in terms of the Test Step instances that can be executed on it. The editor uses the Application Test Model to support the construction of Test Configurations, sequences of configured Test Step instances that are an abstract description of a test.

Figure 33 illustrates the relations of the models used in the MTCC process and their associated phase of the MTCC testing process. Chapter 7 discusses the information contained in the models and their implementation in detail.



Domain Engineering
« FeatureModel» Domain Test Model
« FeatureModel» Domain Feature Model
references
specialization of
Application Engineering
« StateMachine» Application State Model
instantiates test steps
« FeatureModel» Application Feature Model
represents SUT in terms of test steps
references features from
Test Engineering
Application Test Model
« StateMachine» Application Test Model (dynamic)
« FeatureModel» Application Test Model (structural)
configuration of
« FeatureModel» Test Configuration

Figure 33: Models used in MTCC and their relationships

6.2.1 Domain Level Models

The MTCC domain level models represent the test-relevant aspects of all members in a system family modeled by MTCC. MTCC uses two models for the representation of test relevant models, the Domain Feature Model and the Domain Test Model.

The Domain Feature Model represents the commonalities and variability for each type of Service present in the system family. It is usually implemented as a feature model with cardinalities[1]. The Service instances of the specific members of the system family are specializations of the Services defined in the Domain Feature Model. By deriving all Service instances from a common model, the comparability of individual Service instances is ensured. This is an important factor when the features provided by one Service instance are compared with those of another Service instance in order to check if a Test Configuration can be transfered from the first Service to the second.

The Domain Test Model is a catalog of Test Steps. Each Test Step is specific to one class of Service Test Steps do not include information about the structure or variability of the Services. Instead, they contain references to the aspects of a Service instance that is relevant for the execution of the Test Step. MTCC models at the domain level do not have information about the dynamic behavior of the testee.

6.2.2 System-Level Models

The MTCC System-Level models created in Application Engineering describe one system in terms of its Service instances and the possible interactions with these Service instances.

The Application Feature Model represents Service instances of one particular system within a system family. Service instances are feature models that are specializations of the Service classes expressed in the Domain Feature Model. Multiple instances of a Service can exist within one Application Feature Model. The relationship of a Service and a Service instance is analogous to the relationship of a class and an object in object oriented approaches.

To give an example, the representation of a search form Service in the Domain Feature Model would describe the kinds of fields available for search and the options to influence the search that can exist for a system family. The Service instance in the Application Feature

[1]For tooling reasons, we implemented the Domain Feature Model used for the evaluation as a class library that instantiates Application Feature Model models directly.

Model describes one concrete search form for one system.

The feature models of Service instances are not fully specialized. The remaining variability for representing the parameters that can be used for the invocation of a Service The Application State Model describes all possible interactions between a user and the SUT as possible sequences of Service invocations. The Application State Model is implemented as a finite state machine. The Test Steps defined in the Domain Test Model are not refined on the system level.

6.2.3 Test-level Models

MTCC uses two models at the test engineering level: The Application Test Model serves as input to the editor, the Test Configuration represents tests on a system.

The Application Test Model is a composition of the Application State Model, the Application Feature Model, and the Domain Test Model. It does not include any information or refinements not already included in one of its input models. Instead, the Application Test Model is an alternative representation of the information included in the other models that is explicitly designed for the purpose of facilitating test construction using the MTCC Editor.

As can be seen from Figure 33, the Application Test Model consists of a structural part and a dynamic part. The dynamic part is derived from the Application State Model. It is a finite state machine that describes all possible sequences of Test Steps that can be applied to a SUT.

The structural part is instantiated by unfolding the references from the Test Steps defined in the Domain Test Model to the Service instances defined in the Application Feature Model. This unfolding of references creates instances of Test Steps whose features are a combination of the feature model of the Test Step and the feature model of a Services instance.

We use the name Application Test Model for both the dynamic and structural aspects of the model to emphasize that the Application Test Model and all its parts are created in one process and only for the purpose of test construction. The composition of the Application Test Model is a fully automated process that occurs immediately before test construction in the editor. The Application Test Model is a transient model, it is only created for the purpose of test construction and is never persisted

Test Configuration Models are abstract descriptions of tests for one system. Conceptually, these models represent a test as a sequence of configured Test Step instances. The feature models that represent the Test Step instances are fully specialized.

6.2.4 Relationship of MTCC Models to the MDA

Since MTCC uses models of the different levels and steps of refinement and thus exhibits parallels to concepts used in the MDA, the question arises how the hierarchy of CIM, PIM and different levels of PSM (Mellor *et al.*, 2004) relate to the domain- and system-level models of MTCC.

One significant difference is that the different models used in the MDA approach represent different levels of abstraction with respect to an implementation. The MTCC models at any level, in contrast, are agnostic to the final implementation of the tests. In MTCC, different levels of modeling differ not in their details with regards to the implementation, but in the scope of the modeled entity like system family, system and test.

6.3 Chapter Summary

The MTCC process is in many aspects similar to Software Product Line Engineering. Where the Software Product Line Process is defined by two parallel activities, Domain Engineering and Application Engineering, MTCC consists of three phases, Domain Engineering, Application Engineering and Test Engineering. The focus of Domain Engineering is the analysis and modeling of the test-relevant features of the system family and the implementation and maintenance of the MTCC Core Assets. The Application Engineering phase considers a specific SUT. The models created in Domain Engineering are specialized to represent the features available for the SUT and the artifacts needed for text execution are implemented. The products of the MTCC approach are automated acceptance Tests. These tests are created in the Test Engineering phase. In Test Engineering, domain experts use the MTCC editor to create an abstract test representation. These abstract representations are then used to create executable tests.

The models used in MTCC can be distinguished by the phase in which they are used and by the aspect of the tested system that is represented. The features of a SUT, the tests used to exercise these features and parts of the dynamic behavior of the SUT are presented by feature models and finite state machines. Domain-level models represent the Services and Test Steps that are relevant for all members of the system family. System-level models represent the behavior of a system and the instances of Services that are available for the system. Test-level models combine the information included in the system and domain-level models and represent tests.

MTCC partitions a system into basic units of functionality called Services. Services are grouped in Contexts. Contexts and the Services defined in Contexts define the testable interface of a testee. The actions that are available to exercise and verify the SUT are represented as Test Steps.

Chapter 7

Models for System and Test Representation

This chapter discusses the models used in MTCC in detail. While the proceeding chapter focused on the utilization of models in the MTCC testing process and the different models used at different levels of abstraction, this chapter focuses on the implementation of the models.

Section 7.1 describes the Application State Model, the model used by MTCC to represent the behavior of a system. MTCC employes a finite state machine for the Application State Model, specifically a UML Protocol State Machine (Rumbaugh *et al.*, 2004), to represent the possible interaction of a user or test with the system under test.

In section 7.2, we discuss the use of feature models (Czarnecki *et al.*, 2005a; Kim & Czarnecki, 2005) in MTCC to represent both the Services of systems under test and the Test Steps that are applied to these systems. The Services of a system that are to be tested are represented by the Domain Feature Model and the Application Feature Model. A Service in MTCC is the feature model representation of some functionality recurring for multiple systems in the system family under test and testable in isolation. The representation of a Service instance at the system level includes details about the functionality provided by this particular Service especially as they concern the interface of the Service

The Domain Test Model is a domain-level structural model that describes the Test Steps that can be applied to the Service instances defined in the Application Feature Model.

Section 7.3 discusses the composition process that merges the Application Feature Model, Domain Test Model, and Application State Model into the Application Test Model. The Application Test Model is a transient model that represents the available tests for a member of a system family. The model is optimized to support test case construction using the editor. The Application Test Model includes a dynamic part derived from the combination of the Application State Model and the Domain Test Model as well as an dynamic part that is created by unfolding the references from the Test Steps in the Domain Test Model to the Services instances in the Application Feature Model.

7.1 Models for the Behavior of a Testee

The Application State Model and the dynamic part of the Application Test Model describe the behavior of the SUT. Both models are represented as finite state machines where states represent the selection and execution of Services or Test Steps respectively. The Application State Model and the Application Test Model are related as follows:

- The Application State Model is a System-level model. It describes a system as a number of Contexts, each of which contains a number of Services. The transitions between Context and Services states represents the selection and execution of Service The Application State Model is modeled during the Application Engineering phase. It does not include any information about the actions or assertions available to test the Service

- The dynamic part of the Application Test Model is a test-level model that describes all sequences of the Test Steps that are available for the manipulation and verification of a testee. This model is a combination of information from the Application Test Model and the Test Steps definitions from the Domain Test Model.

MTCC does not define any models of the behavior of systems at the system family level. We argue that the differences of systems within a system family regarding their supported interactions with Services, are so significant that domain-level behavior models are not useful in practice.

The reason for the use of both the Application State Model and the dynamic part of the Application Test Model to represent the behavior of a testee is the intent to separate the representation of a system and tests for the system and thus ease the creation and maintenance of the models. By limiting the Application State Model to the representation of models, redundancies are avoided. The Application State Model is thus the canonical model for the behavior of a system, while the dynamic part of the Application Test Model is a representation derived from the Application State Model for the purpose of Test Step selection during test construction.

As discussed in the overview of the MTCC models in Chapter 6, the dynamic and the structural part of Application Test Model are transient models created in a fully automated process immediately before the use of the editor for test case construction. Since both the dynamic and the structural part of the Application Test Model serve the same purpose and are created in the same process, we treat the Application Test Model as one model. From a technical and conceptional point of view, the dynamic and structural parts of the Application Test Model are different models, this section discusses only the dynamic part, an examination of the structural part follows in Section 7.2.3.

7.1.1 Structure of the MTCC Behavior Models

As already discussed in Section 6.2, the models used in MTCC for the representation of the behavior are finite state machines. We use the UML2 Protocol State Machines to illustrate the structure of FSMs in this thesis where appropriate, but do not assume that the exact semantics of Protocol State Machines are implemented in MTCC.

From a conceptional point of view, MTCC differentiates between three kinds of states: Context states, Service instance states and Test Step instance states. The Application State Model contains Context and Service instance states. The dynamic part of the Application Test Model replaces each of the Service instance states from the Application State Model with one or more Test Step instance states.

Figure 34 illustrates the relations of the different concepts relevant for the dynamic models used in MTCC. A system is represented as an aggregation of Contexts. Each Context is in turn an aggregation of Service instances or Test Step instances. In the FSM, Contexts are presented as states from which the aggregated Service Test Step instances can be reached.

In the following, we discuss the structure of the FSM that represents a testee in the Application State Model as well as in the dynamic part of the Application Test Model.

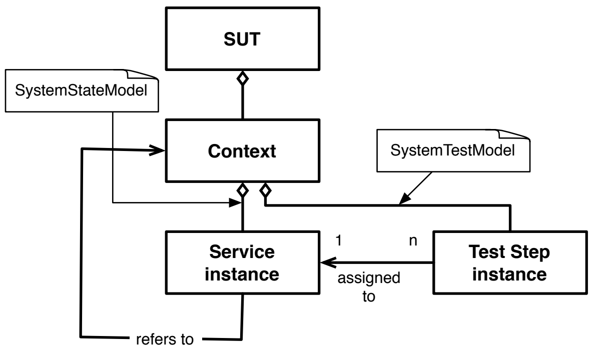


Figure 34: Relationship of the concepts in the dynamic MTCC models

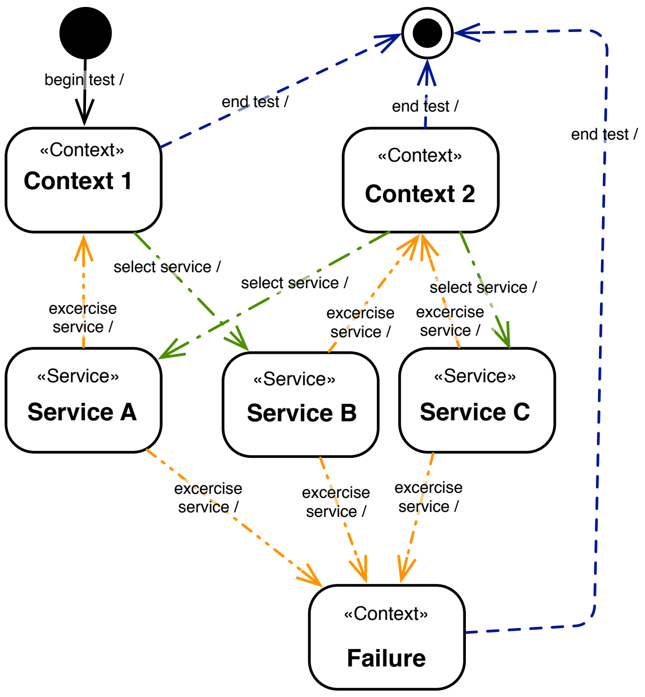


Figure 35: Representation of the Application State Model as a Protocol State Machine

7.1.2 Concepts and Structure of the Application State Model

The Application State Model represents all possible interactions with a testee as a sequence of Service invocations. Each invoked Service instance is selected from all Service instances defined for a Context. A testee consists of a number of Contexts, from which most Service instances can be reached. Each Service instance in the Application State Model can be reached from exactly one Context. The invocation of a Service instance leads to the transition of the system under test into a new Context. The new Context can either be the initial Context, from which the invoked Service instance was selected or a new Context. A Service instance can have multiple different Contexts that can be reached as the consequence of an invocation. For instance, one Context that represents the state of the system after a successful Service execution and another that represents the state of the testee after a failure occurred.

Figure 35 displays an example Protocol State Machine for a simple system that consists of three Contexts and three Service instances. Three types of states and three types of transitions are used in the FSM. The Context and Service states represent Context and Services as described above. The transitions are named after the event that triggers the state transitions.

The three types of transitions that are used in the Application State Model in addition to the transition from the start state are select Test Step, exercise service, and end test:

- The transitions triggered by the select Test Step event cause the change of the current state of the FSM from a Context state to a Service state. These transactions represent the selection of one Service instance, for example, a search form or an link, but not yet the execution of this Service instance.

- The transition exercise service changes the current state of the FSM from a Service state to a Context state. This transition represents the execution of a Service

  Starting from a Service state, multiple different Context states can be reached by an exercise service transition, the FSM is thus non-deterministic. Contexts reached by different exercise service transitions represents different outcomes of Service execution. The different outcomes include, but are not limited to, the successful and the failed execution of a Service

  In the example given in Figure 35, the successful execution of each of the three Services leads to a transition to either state Context 1 or Context 2. A failed Service execution leads to the transition to the state Failure.

- The end-state of the FSM is reached by one of the end test transactions. An end test transaction is available from every Context state in the FSM, it represents the end of a testing session.

Only some of the Context states of the Application State Model can be reached from the start state of the FSM. The states that are reachable from the start state represent those Contexts of a system in which a testing session can start, for instance, the home page of a Digital Library. In Figure 35, the only Context state that can be reached from the start state is Context 1. All tests must thus start with one of the Service instances available from this state.

The Application State Model is only a limited representation of the behavior of a system. It only includes enough information to represent the available sequences of Service invocations for a system. Information about the Service instances themselves beyond their possible successor Contexts are not part of the model. Particularly, the Application State Model does not model the internal state of Service instances.

Figure 36 display an excerpt from the Application State Model of the Sowiport system, rendered by GraphViz[1] exporter. This representation displays Context Nodes with thick borders

7.1.3 Test-level Behavioral Models

While the Application State Model represents possible sequences of Service invocations for a system, the dynamic part of the Application Test Model describes available sequences of Test Step instances. The dynamic part of the Application Test Model is created by the composition process discussed in Section 7.3. Each Service instance is replaced by a set of Test Step instances. The Test Steps defined in the Domain Test Model are tested for compatibility with each of the Service instances defined in the Application Feature Model. If a Test Step is compatible with a Service instance, the Test Step is instantiated, meaning that the references of this Test Step to the Service instance are unfolded and the referenced feature nodes are replace the references in the feature model (Czarnecki *et al.*, 2005a).

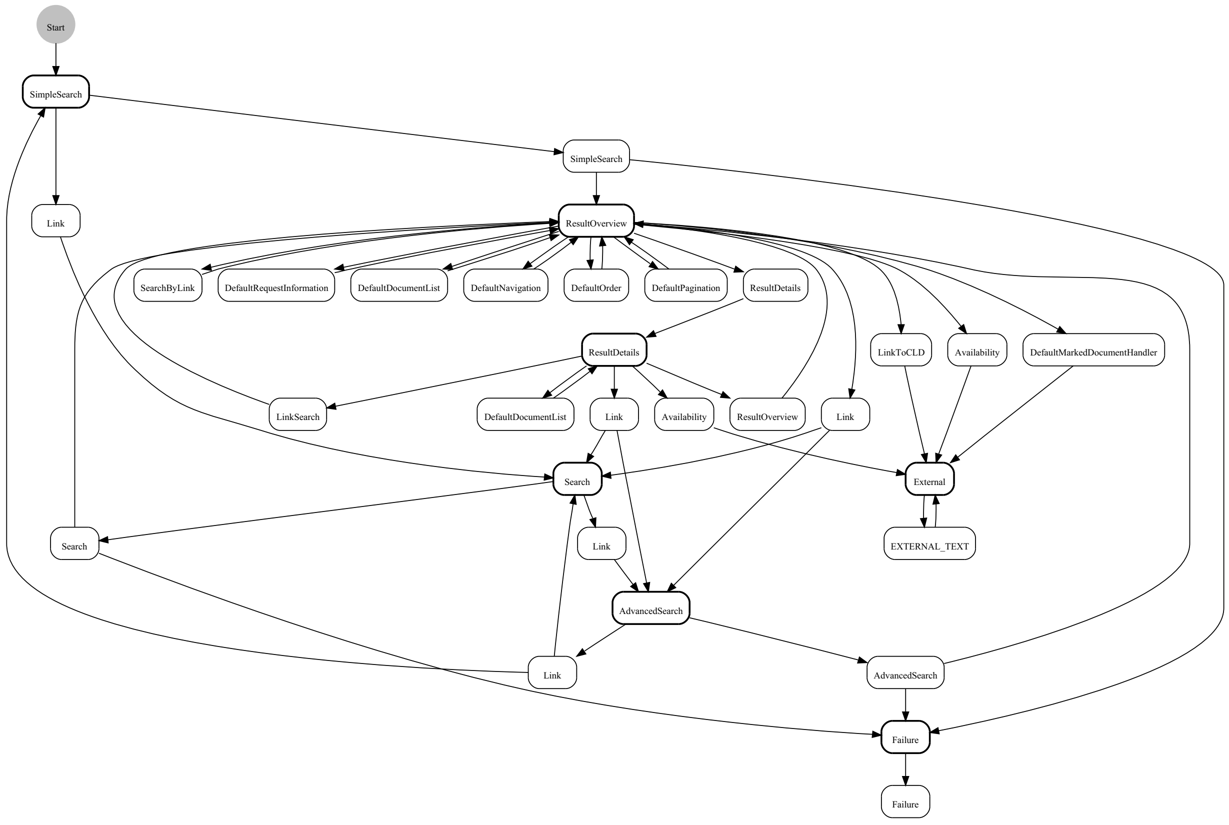


Figure 36: Excerpt from the Application State Model of the Sowiport System

Figure 37 gives an example for the FSM of the dynamic part of the Application Test Model. The system in the example consists of three Contexts and six Test Step instances.

The transition types of the dynamic part of the Application Test Model are similar to those in the Application State Model. Since Service instances in the FSM are replaced by Test Step instances, the names of the transitions change from select service to select Test Step and from execute service to execute Test Step. MTCC uses three types of Test Step instances in the dynamic part of the Application Test Model: Actions, SameStateActions, and Assertions.

- A Test Step instance of the type Action interacts with the testee and manipulates its internal state. The execution of this Test Step will generally result in the transition to a different Context than the one from which the Test Step instance was selected. Test Steps of the type action represent actions of a user to change the currently active page of a web application. A Test Step instance that sends a query to a Digital Library for instance can change the current Context to the result overview.

- A SameStateAction represents an interaction between a user and the testee that cannot result in a change of the current Context. After the execution of a SameStateAction, the system always returns to the initial Context. An example for a SameStateAction is the selection of filter values in a search form. While this interaction changes the state of the testee, it does not result in a change of the currently available Services.

- The Test Step Assertion represents an action that does not manipulate the state of the testee but either compares the state of the testee with an expectation or saves some

[1]`http://www.graphviz.org`

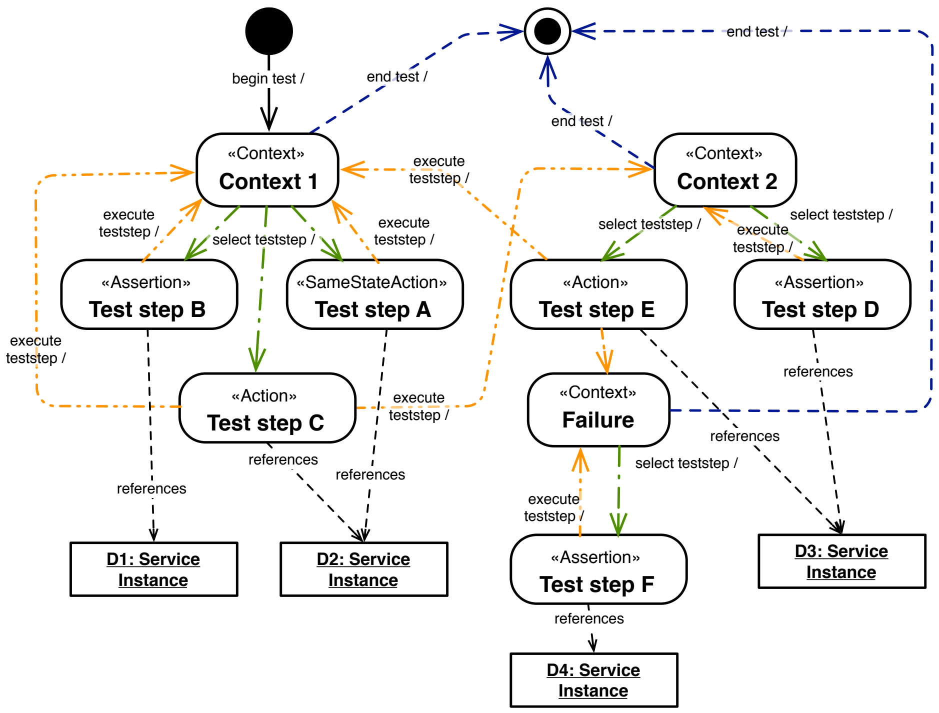


Figure 37: Protocol State Machine of all possible sequences of Test Step instances in the Application Test Model

aspect of the state of the testee in a variable. The execution of an assertion always returns the FSM to the Context state that was active before the execution of the test.

Similar to the Application State Model, the dynamic part of the Application Test Model only describes possible sequences of Test Step instances.

The testee and the execution test runner contain internal state information. For example, the test runner used in MTCC stores aspects of the externally visible state of the test runner in variables that can be used to compare the actual and expected behavior and state of a system. Such state information is not represented in the dynamic part of the Application Test Model. MTCC is a Black Box (Beizer, 1995) approach to testing. It does not interact with or models the internal state of a testee, thus the FSMs used in the Application State Model and the dynamic part of the Application Test Model do not represent this information.

7.2 Feature Models for System and Test Representation

The purpose of the structural MTCC models is the representation of the test-relevant aspects of a system and the Test Steps that exercise these aspects. The structural models are expressed as feature models with cardinalities. The domain-level Domain Feature Model and the derived, system-level Application Feature Model represent the testable functionality of a system as the number of Services or Service instances. The Domain Test Model contains models for all Test Steps that can be executed on these Services.

7.2.1 Feature Modeling in MTCC

In MTCC, feature models are used to describe the Services that provide the test-relevant functionality of a system. MTCC takes an external view on the Services that focus on the external interfaces provided by the Service as it is seen by a user or an automated test. The feature models therefore represent both high-level features like the availability of specific operators in search and low-level details about the different filter values that can be used to constrain the results of a search.

Test Steps are represented as feature models that reference the feature models of Services. Test Steps themselves are system family level models that are not specialized for any system or Service of a system. System-specific information is included in feature models by unfolding the references to the Service instances in the Application Feature Model.

The feature models used in MTCC are a variant of feature models with cardinalities as introduced by Czarnecki (Czarnecki *et al.*, 2005a). This kind of feature model is an extension of simpler forms of feature models (Czarnecki & Eisenecker, 2000) which in turn are based on feature models defined in the FODA (Kang *et al.*, 1990) context.

Compared to FODA feature models, feature models with cardinalities have the following extensions (Czarnecki *et al.*, 2006a) relevant in the context of MTCC:

- Feature groups are treated as independent nodes in a feature model. Feature groups support the selection of a subset of all children of a feature group.
- Attributes provide a way to save textual or numeric information in feature models without being limited to predefined values. One application of attributes is to assign names to cloned features or to assign numbers from a potentially unlimited domain. MTCC uses attributes to save test entries by users in feature models and thus allows, for example, the pre-filling of search forms in Test Step instances that submit a query. Attributes in MTCC are untyped and can save a textual value.
- References support the linking and inclusion of subtrees from one feature model in another. References therefore help to avoid redundancies. Where one part of a feature model would have to be replicated without references, with references it can simply be referenced from multiple nodes within the feature model. In MTCC, attributes are used by Test Steps to refer to the features of Service instances that are relevant for the parameterization of the Test Steps.
- Cloning supports the repeated replication of a subtree of the feature model. The clones of a feature are not different from other features in a feature model and can be configured independently from each other. MTCC does not use cloning in test modeling: Cloning is not implemented in the MTCC prototype used for evaluation in the context of this work, but cloning would be useful when features are replicated during the derivation of a Application Feature Model from the Domain Feature Model.

In addition to the concepts above, MTCC extends feature models with cardinalities by the addition of configuration nodes. These nodes represent specialization steps that are appended to a feature node when the node is specialized. Configuration nodes are discussed in detail in Section 8.2.

The particular implementation of feature models with cardinalities that we implemented for this thesis has a number of limitations. One limitation concerns the possible cardinalities for feature nodes. Cardinalities other than [1,1] and [0,1] are currently not supported. References are implemented as attributes that contain the path of the referenced nodes.

On the implementation level, MTCC uses XML to represent feature models. XML is used for the serialization of feature models. A XML object model is used as the superclass for the class hierarchy of the MTCC feature nodes. Figure 38 displays the classes used by MTCC to represent a feature model and the attributes of these classes.

Listing 7.1 gives an example for the representation of a feature model as XML.

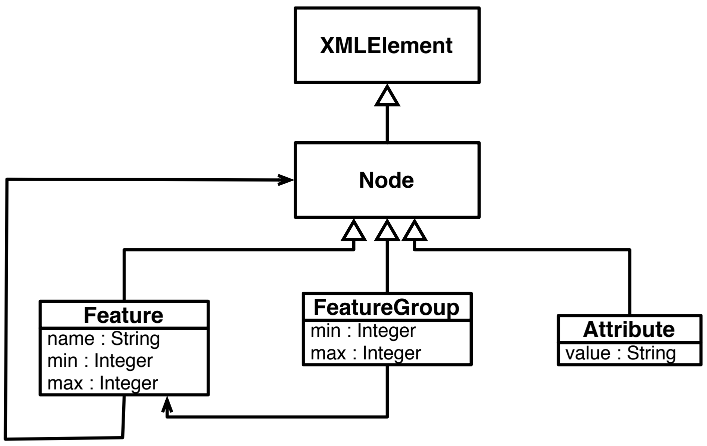


Figure 38: Hierarchy of the MTCC feature classes

```
<Feature max="1" name="Variable" min="1">
  <FeatureGroup max="1" min="1">
    <Attribute value="Wert␣1" />
    <Attribute value="Wert␣2" />
  </FeatureGroup>
</Feature>
```

Listing 7.1: XML representation of a feature model

In contrast to other approaches (Cechticky *et al.*, 2004) that use XML to implement feature models, MTCC does not use schema languages to describe the possible structure of a feature model and the specialization steps that can be applied to it. Instead, we implement methods for the members of our class hierarchy that allow the individual nodes to report their available configurations.

Since the methods that implement the application of specialization steps to the MTCC models include checks that ensure that only valid transformations can occur, we argue that schema validation is not needed.

In relation to other notations used for the representation of XML feature models, like the textual notation introduced by Deursen (van Deursen & Klint, 2002), we argue that while XML is more verbose than specialized notations, the tool support available for XML together with the fact that the XML models are not intended to be edited by humans makes a XML-based approach better suited in the context of MTCC.

The utilization of feature models in MTCC is invisible to the domain experts that are involved in test construction. MTCC does not make any assumptions about the tools used by the domain engineers. While MTCC does not include an editor for feature models like the one presented by Antkiewicz (Antkiewicz & Czarnecki, 2004) that would allow the direct manipulation of feature models, the utilization of such an editor to support the realization of the Domain Feature Model and the derivation specialized Application Feature Model should be beneficial.

7.2.2 Modeling of Test-Relevant Services

By using feature models, an initial general description of the test-relevant aspects at the domain-level can be refined into models for system description and finally into models that facilitate the construction of tests for a system. On the system-family level, the overall features of Services are represented as Services in the Domain Feature Model. Following an analysis of a system from the system family, Service instances are realized in the Application Feature Model by applying specialization steps to the feature models of Services. The Test Step instances defined in the structural part of the Application Test Model duplicate those subtrees of the Application Feature Model Service instance feature models that are relevant for the Test Step. In the test modeling, the feature models that represent these aspects of Service instances as well as the feature models that define the interface of a Test Step are configured to a concrete Test Step without any remaining variability. A sequence of such configured Test Step instances, a Test Configuration, represents a test.

With that said, the modeling of tests with MTCC is partly an application of the staged configuration process (Kim & Czarnecki, 2005; Czarnecki *et al.*, 2005b) for system construction to the domain of test construction.

In addition to the representation of the test-relevant features of a model, MTCC also uses feature models to represent the overall structure of a testee. A SUT in MTCC is described as a hierarchy: The system contains of a number of Contexts, which in turn consist of a number of Services. Figure 39 illustrates this hierarchy and the classes and attributes at the different levels of this hierarchy as a UML class diagram. Figure 40 gives an example for the overall structure of a system as a feature model. Each Service instance is represented by a feature model that includes both attributes like the unique name and the type of the instance of well as the test-relevant features that determine the testable interface of the functionality represented by the Service instance. The information about the testable features is defined in the sub-nodes of the interface node of the feature model. The features outside the interface node describe the Service instance and are not variable after Service instance instantiation.

As Figure 39 displays, SUTs, Contexts and Services each have an unique name. MTCC uses this name to differentiate and to reference different Contexts and different instances of one Service within a Context. The type attribute of a Service instance informs about the Service that this instance was derived from. The type of a Service instance determines whether a Test Step defined in the Domain Test Model can be applied to the Service each Test Step is specific to one type of Service The MTCC models include documentation for each type of Service that is defined for a system family.

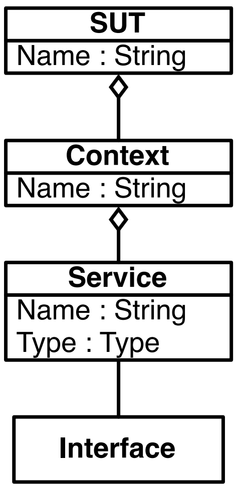


Figure 39: Entities used in MTCC to represent the structure of a system represented as a class hierarchy

The structure of the interface-nodes of the feature model is specific for each type of Service The different Services for the system family of Digital Libraries considered for this thesis are discussed in detail in Chapter 9.

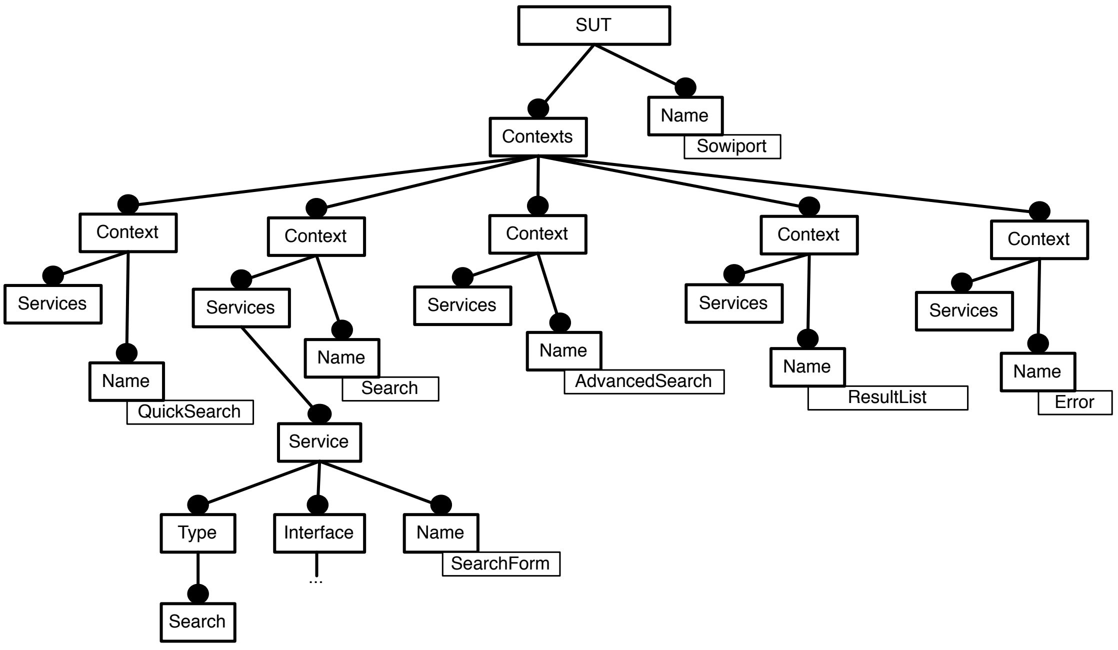


Figure 40: Example feature model for a system in MTCC

Figure 41 displays a part of the feature model of a Service instance of the SEARCH_FORM Service Compared to the feature model of the SEARCH_FORM Service defined in the Domain Feature Model, this feature model is specialized to a certain degree. For instance, except for one feature, all children of FieldType and FieldCombination feature groups are de-selected.

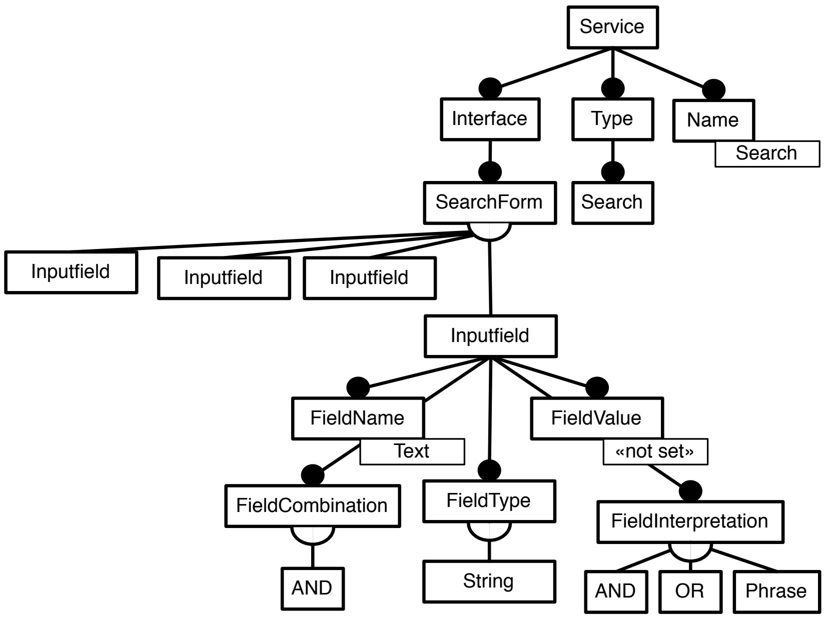


Figure 41: Part of the feature model of a Service instance

7.2.3 Representation of Test Steps

Test Steps represent actions that can be used to manipulate the state of a testee as well as assertions that either compare an aspect of the state of the testee with some expected value or save such an aspect of the state in a variable for later use. All Test Steps for a particular system family are defined in the Domain Test Model. Every Test Step is specific to one type of Service During composition, all Test Steps are checked for compatibility with all Service instances that are of the type required by the Test Step. Details on the composition process are given in Section 7.3.

In addition to the type of Service instance, Test Steps can define a number of additional criteria called checks in MTCC, that can be used to test whether a Test Step can be applied to a particular Service instance.

If a Test Step and a Service instance are compatible, a number of zero or more references that each step defines for parts of the feature model of a Service instance are unfolded. The referenced subtrees are thus copied into the feature model of the Test Step. We call this unfolding of references the instantiation of the Test Step. The resulting model is called the Test Step instance.

The structural part of the Application Test Model contains all Test Steps instances that can be instantiated for a given system.

From the perspective of automated testing, Test Steps correspond to action words (Graham & Fewster, 2000) that represent particular functionality that can be executed on a Service Test Step instances thus can be seen as specialized action words that support the execution of one particular Service and which define an interface that represents the testable features of this Service instance.

Test Steps in MTCC contain the following information:

- A unique name that is used in the editor to describe the functionality and executed by the Test Step in the editor. When the Test Step is instantiated, this unique name together with the name of its exercised Service instance defines the Test Step instance.
- A number of checks that are executed against a Service instance and on which a decision on the compatibility of the Test Step and the Service instance can be determined. The most basic of these checks that are defined by all Test Steps in MTCC is the check for the type of a Service instance.
- Features that define the interface of the Test Step. This includes both feature models that represent choices specific to the Test Step, such as the choice between different variables that can be used for the comparison of a value or an attribute that allows the specification of a value, and references that include parts of the feature model of a Service instance into the feature model and hence the interface definition of a Test Step instance.

From a purely conceptional point of view, the use of feature models for the representation of Test Steps is not necessary since Test Steps are not specialized but rather include specialized, system-level information. We argue that the use of feature models for the representation of Test Steps is justified by the advantages that result from the use of feature models as a common metamodel for both Services and Test Steps. The use of a common metamodel allows the reuse of existing tooling and usually creates less complexity than would result from the utilization of different metamodels. We further argue that feature models are well suited to represent the parameterization of Test Steps.

Figure 42 illustrates the different concepts included in a Test Step as a UML class diagram.

- Checks are used to test a Service instance regarding its compatibility with the Test Step. Test Steps in MTCC support three different classes of checks:

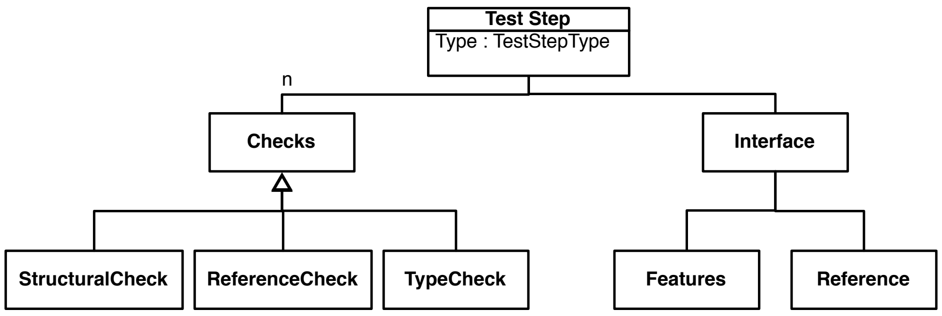


Figure 42: Class diagram of the relevant concepts of a MTCC Test Step

- An instance of the class TypeCheck is used by every Test Step. This check tests whether the type of a Service instance is equal to the StepType attribute of the Test Step. Only when that condition is fulfilled, are the types of Test Step and Service instance compatible and any remaining checks of the Test Step need to be tested.
- Instances of the class ReferenceCheck ensure that a reference to the feature model of a Service instance can be unfolded. When the referenced features are not present in the Service instance, the check fails and the Test Step cannot be instantiated for this Service instance. In contrast to the references defined in the interface part of the Test Step feature model, no copy of the referenced feature is made when the references defined in ReferenceCheck are unfolded.

  In the implementation of MTCC, the references used as ReferenceChecks are expressed as XPath statements.
- Instances of the class StructuralCheck support the test of Test Steps against Service instances based on the structure of the feature model of the Service instance as defined by a schema language. As is the case with the ReferenceChecks discussed above, StructuralChecks in the current implementation address the underlying XML representation of the MTCC feature models. Each StructuralCheck contains a reference to a RelaxNG[2] schema that is tested against a node.

- The Interface part of a Test Step defines the feature model that represents the possible configurations of the Test Step instance that can be instantiated for the Test Step. As already described, the feature model both includes feature specific to the Test Step and defines references to Service instances.

Figure 43 displays a feature model of a Test Step before checks are executed and the references are unfolded. The name of this Test Step is START_SEARCH. The Test Step represents the submission of a query to an IR system using a search form. This Test Step is of the type Action, meaning that it manipulates the state of a testee and causes it to change into a different context.

Two checks are defined for the Test Step: The TypeCheck specifies that instances of the Test Step can only be instantiated with instances of the SEARCH_FORM Service The ReferenceCheck checks for the existence of a feature with a specific name. The XPath used by the ReferenceCheck is truncated in the figure.

The Interface feature defines the interface that is available for this Test Step and that is represented as GUI elements by the editor. The Test Step in Figure 43 includes three elements:

[2]`http://relaxng.org`

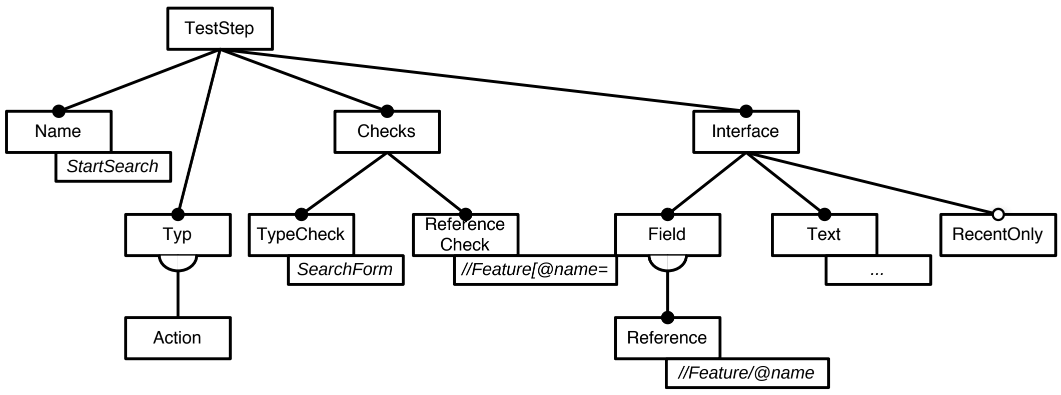


Figure 43: Feature model of the Interface elements of a Test Step

- The feature group under the Field node defines a reference to the feature model of a Service instance. When this reference is unfolded, the subtree of the feature specified by the reference is copied into the feature model of the Test Step, replacing the reference.
- The Text feature is a part of the interface of the Test Step that allows the specification of textual information, represented by an attribute node.
- The RecentOnly feature represents a choice that can be made for this Test Step. The RecentOnly option can either be selected or de-selected for the Test Step.

7.3 Composition of the Application Test Model

The Application Test Model is a composition of the Application Feature Model, the Application State Model, and the Domain Test Model. The purpose of the model is to provide a representation of the SUT that is optimized as a basis for test modeling, that is the creation of Test Configurationinstances, using the MTCC editor.

7.3.1 Purpose of the Composition Process and the Application Test Model

The purpose of the composition process is the realization of the Application Test Model. It is the instantiation of a model whose dynamic parts describe the sequences of Test Steps that can be used to exercise and verify a testee while the structural parts of the Application Test Model describe the interface provided by each Test Step instance.

In contrast to the models from which it is created, the Application Test Model is a transient and automatically created model with the sole purpose to serve as an information source to the editor during test modeling. As a consequence, the Application Test Model is optimized to provide the editor with information in the least complex way feasible, but does not aim for good maintainability or the avoidance of redundancy.

7.3.2 The Composition Process

In the following we discuss the composition process in general as well as the derivation process used to realize the dynamic part of the Application Test Model.

As in the composition process used to derive the structural part of the Application Test Model, all Test Steps defined in the Domain Test Model are tested against all Service instances defined in the Application State Model. Figure 44 gives an overview of the composition process.

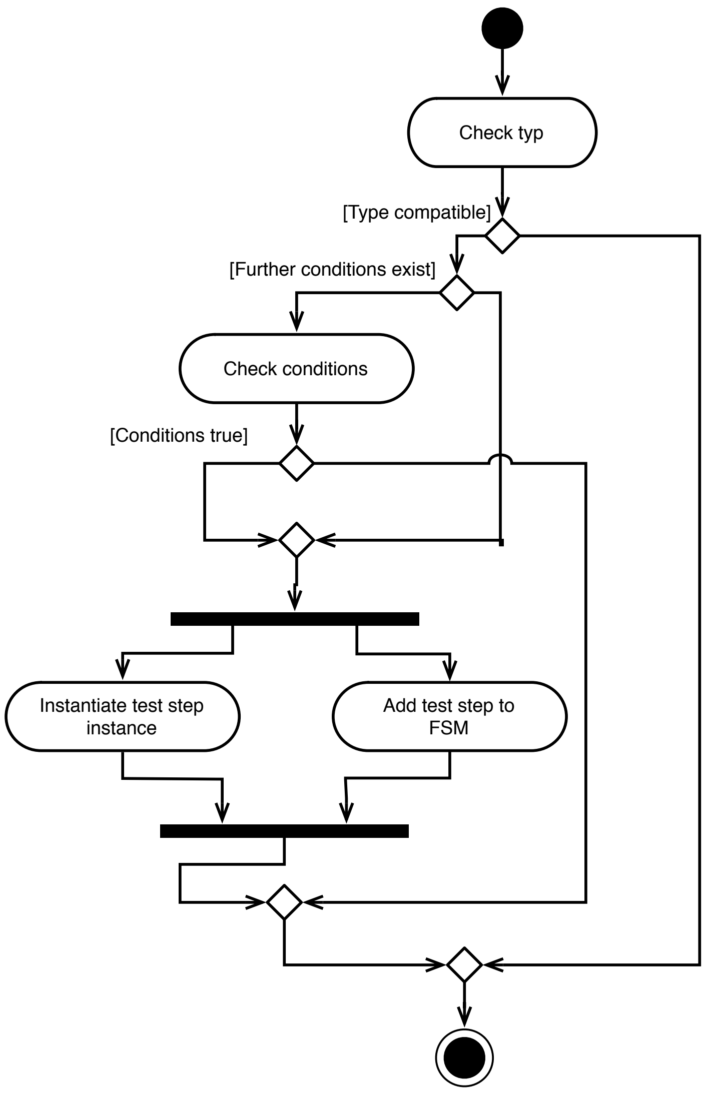


Figure 44: Flowchart of the MTCC Composition

If a Service instance and a Test Step are compatible, a state representing the Test Step instance is created and added to the FSM. Initially, the dynamic part of the Application Test Model is a graph that consists only of Context states: A start state and an end state. During the composition process, all Test Step instances that can be executed from a Context are added to the FSM. For each added Test Step instance, the following transitions are added to the FSM:

1. A select Test Step transition from the context of the Service instance referenced by the Service to the Test Step instance is added to the FSM.
2. A execute Test Step transition is added to test the FSM for every Context state that represents a possible state of the testee after the execution of the Test Step instance is added to the FSM.

The resulting FSM is similar to a Application State Model where every Service state is replaced by one or more Test Step instance states. The number of states in the dynamic part of the Application Test Model is therefore higher than the number of states in the Application State Model.

The Context states that can be reached from the state of a Test Step instance are determined by the type of the Test Step instance: For Test Step instances that are either Assertions or SameStateActions, a transition to the initial Context from which the Test Step instance was selected is added to the FSM. For Test Step instances that have the type Action, transitions are added to all Contexts that are reachable from the Service instance of the Test Step in the Application State Model.

A number of Test Step instances, for example the comparison of variables, are not dependent on the Services of the system under test but represent functionality of the test system itself. In order to represent such Test Step instances in the same way as the Test Step instances that exercise or otherwise interact with the testee, MTCC uses a virtual Service instance, the DefaultService. The DefaultService is automatically included in every Context of a system. Every Test Step instance that must be available regardless of the Services provided in the current Context thus references an instance of the DefaultService. All instances of the DefaultService are identical, it does not define any interface.

7.4 Chapter Summary

MTCC uses feature models to represent the test-relevant functionality of a system and finite state machines to represent the behavior of the system.

The behavior of a system is described by a system-level model, the Application State Model. In order to represent the dynamic behavior of a system, the system is partitioned into Contexts. Each context is defined by a number of Services that are available for that Context. Contexts and Services are logically represented as states in a finite state machine. Transitions represent the invocation of Services and the resulting Context changes.

Feature models are used by MTCC both on the domain-level and on the system-level. The system-level feature models included in the Application Feature Model are specializations of the feature models defined in Domain Engineering that are included in the Domain Feature Model. Each feature model in the Application Feature Model represents an instance of a Service in one Context of the represented system.

In MTCC model composition, the Application Test Model, the Application Feature Model and the domain-level Domain Test Model are combined into a test-level model, the Application Test Model. The Application Test Model describes which Test Steps can be executed in the different Contexts of a system. In order to instantiate the Application Test Model, the Test Steps defined in the Domain Test Model are tested for compatibility with the Service instances defined in the Application Feature Model. If a Service instance and a Test Step are compatible, a new Test Step instance is created. The static part of the Application Test Model describes the supported parameterization for Test Steps, the dynamic part describes the possible sequences in which Test Steps can be invoked to test a system.

Chapter 8

Application of MTCC Models

This chapter examines the use of the Application Test Model for test construction. In order to facilitate the construction of tests by domain experts, the MTCC editor is used to allow the representation and configuration of the Application Test Model in a graphical user interface. MTCC supports the reuse of existing Test Configurations for multiple testees in a system family based on the transfer of Test Configuration instances in order to execute an abstract Test Configuration. To test, MTCC uses code generation to generate a concrete test script.

Section 8.1 examines the representation of the dynamic and structural parts of the Application Test Model in the MTCC editor. The editor represents tests as sequences of configurable Test Steps. Domain experts create tests by selecting and configuring the Test Step instances that represent the actions that are intended to exercise the testee. Test Step instance selection and configuration are done in the editor. The editor is a graphical user interface that represents specific features and subtrees of features by standard elements of graphical user interfaces like lists of values, test input fields or check boxes.

One purpose of the editor is the configuration of the Test Step instances defined in the Application Test Model and thus the removal of all variable elements of the feature model. While MTCC does not require or implement all specialization steps defined for feature models with cardinalities (Czarnecki *et al.*, 2005a), it is important that specializations applied to Test Step instances are reversible to allow for the correction of mistakes. In order to support the reversal of specialization steps, MTCC introduces configuration nodes that support the explicit representation and manipulation of selection steps in MTCC feature models. Section 8.2 discusses the realization of the configuration activities for feature models in MTCC.

MTCC considers testees in the context of software system families and aims to support the reuse of Test Configuration models created based on the Application Test Model of one testee for other testees. Section 8.3 discusses the requirements for such reuse in MTCC and the techniques used to support the transfer of tests systems.

In order for tests to be transfered from one system to another, it is necessary that the target system supports the execution of all Test Steps used in the test in the same order in which they are used in the test.

Additionally, the Test Steps for the initial system and the target system must be compatible in the sense that the feature model of each Test Step instance in the target systems supports the configuration that was applied to the feature model of the corresponding Test Step instance for the initial system.

If more than one sequence of Test Step instances for the target system is compatible with the Test Step instances in the initial system, an automatic decision on the sequence that best represents the semantics of the original test is not possible. It is also not possible for MTCC to decide whether a test that can be technically transfered by MTCC really represents the intention of the test on the target system. For these reasons, test reuse in MTCC is a semiautomatic process.

The Test Configurations created in the test modeling process do not contain information about the technical details required for test execution. In order to exercise a system based on the tests, these models must either be interpreted by a runtime system or must serve as the basis for test code generation. Section 8.4 discusses the approach to test code generation used in MTCC. MTCC uses a template-based code generator that generates test cases for xUnit test runners. MTCC test cases depend on a library that contains implementations for all Test Step instances of a given system under test.

8.1 Representation of Test Steps in the Editor

The purpose of the MTCC editor is to support domain experts' formal modeling or programing skills in the specification of tests.

The test models created by the domain experts have to be formal in the sense that they support the generation of scripts without the need for further refinement or rework by domain engineers. In order to meet this requirement, the editor needs to fulfill the following conditions:

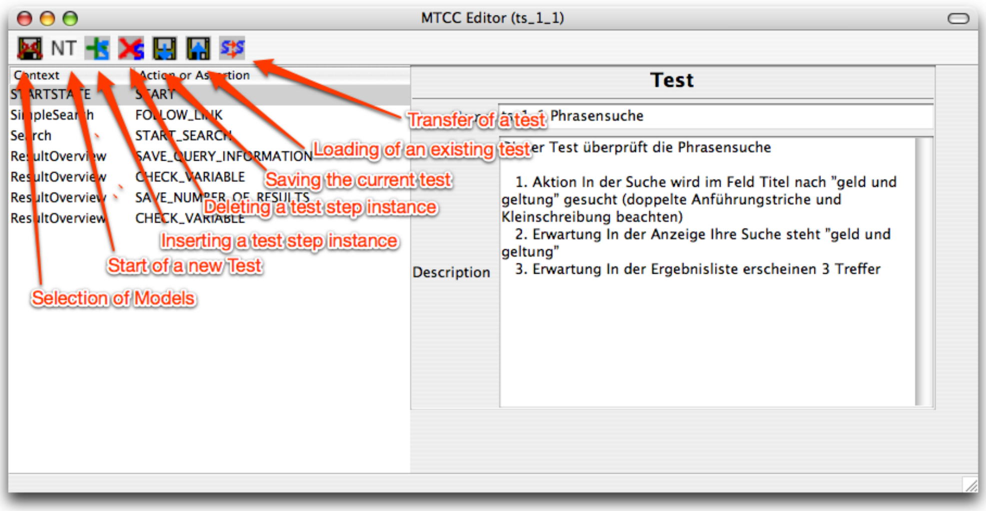
- The editor has to be able to represent the information contained in the Application Test Model in a form that allows a domain expert to express a testing intent in a MTCC test. The possible configurations of the MTCC editor have to be represented by GUI elements in an understandable way.
- The editor has to be able to translate the interactions of a user with the GUI into specialization steps that configure the feature model of a Test Step instance. The specialization steps have to be expressed in a way that allows for changes when the configuration of a Test Step instance is changed in the editor.
- The editor must be able to save tests in a format suitable of code generation.

In the following, we introduce the MTCC editor from a user's point of view. We examine the test modeling process as the selection and configuration of Test Step instances. Based on the external view on the editor, we examine the representation of feature models as GUI elements in Section 8.1.3.

8.1.1 Overview of the MTCC Editor

Figure 45 displays a screen shot of the editor. The user interface consists of three separate areas: A toolbar that gives access to the functionality provided by the editor is located at the top of the editor window. A list of the Test Step instances that are currently selected for a test can be seen in the left half of the editor, the right half of the editor consists of the GUI representation of the currently selected Test Step and a text area that explains the functionality of this test. The overall structure of the editor — a list that allows the selection of elements in the left part of the GUI and a detailed representation of the selected element in the right part — has been chosen for MTCC because it is frequently used in programs in other domains, for example, in email clients. It is therefore likely that users are familiar with this representation.

The toolbar gives access to the functions of MTCC such as loading and saving of tests and the insertion and the deletion of Test Step instances. All functions of MTCC that can be selected with the toolbar are also available in a menu not displayed in Figure 45. Starting from the left, the following functions or actions can be invoked from the toolbar:



Context	Action or Assertion
STARTSTATE	START
SimpleSearch	FOLLOW_LINK
Search	START_SEARCH
ResultOverview	SAVE_QUERY_INFORMATION
ResultOverview	CHECK_VARIABLE
ResultOverview	SAVE_NUMBER_OF_RESULTS
ResultOverview	CHECK_VARIABLE

ts_1_1 Phrasensuche

Description: Dieser Test überprüft die Phrasensuche

1. Aktion In der Suche wird im Feld Titel nach "geld und geltung" gesucht (doppelte Anführungstriche und Kleinschreibung beachten)
2. Erwartung In der Anzeige Ihre Suche steht "geld und geltung"
3. Erwartung In der Ergebnisliste erscheinen 3 Treffer

Figure 45: Screenshot of the start page of the MTCC editor

Selection of models This button opens a dialog that allows the selection of the currently used Application State Model, Application Feature Model and Domain Test Model. All three models must be available as XML files in the file system. Once the models are loaded by the editor, a new Application Test Model is created in the composition process discussed in Section 7.3.

Start of a new test This action starts a new test for the currently selected Application Test Model. The Test Step instances that are currently loaded into the editor are lost.

Inserting a Test Step instance This action inserts a new Test Step instance into the currently loaded test. A dialog allows the selection of a Test Step instance from a list, the Test Step instance is then inserted after the Test Step instance that is currently selected. Which Test Step instances can be inserted, is determined by the Application Test Model.

Deleting a Test Step instance This action removes the currently selected Test Step instance from the list of Test Step instances. The first Test Step instance in a test can never be removed.

Saving the current test This action opens a dialog that allows the selection of a file in which the current test can be saved as a Test Configuration in a XML format.

Loading of an existing test This action loads an existing test in the editor and changes the currently used Application Test Model by loading the Application State Model, Application Feature Model, and Domain Test Model that were used for the construction of the test and composes a new Application Test Model based on these models.

Transfer of a test This action starts the transfer process of the current test to another system, the target system of the test transfer is selected from a list of all modeled systems for the current system family.

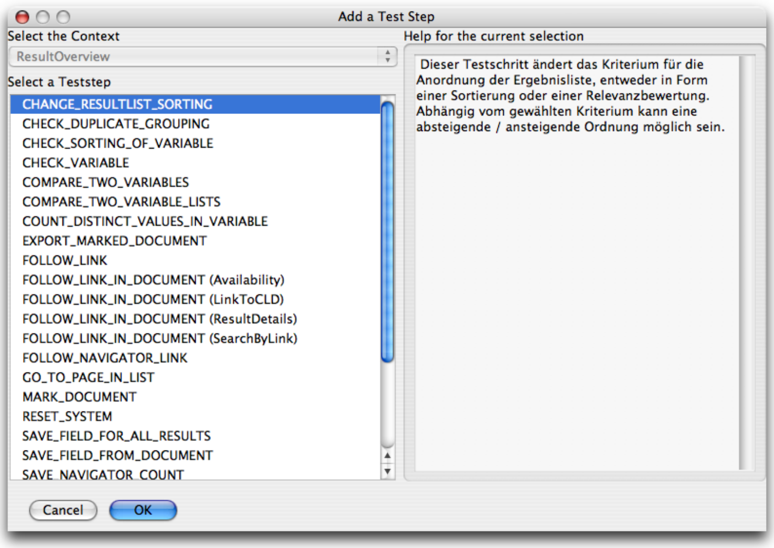
The list of Test Step instances includes all Test Step instances that are part of the current test in the order in which they will be applied to the testee. Each Test Step instance is represented by the type of the Test Step and the name of the Context of Test Step. If more than one instance of a Service exist for the Context, the name of the Service instance that the Test Steps invoke is displayed in parentheses.

In Figure 45, the currently selected Test Step is the START Test Step. This Test Step instance is part of every MTCC test. It represents the start state of the FSM and cannot be removed from the test. This Test Step instance does not represent an interaction with the testee but allows the definition of metadata for the test, specifically a description and a title.

8.1.2 Construction of Tests from Test Step Instances

A test in MTCC is a sequence of configured Test Step instances. One part of test modeling is therefore the selection and combination of those Test Step instances that represent the interaction between a user and the testee that is to be tested.

Figure 46 illustrates the dialog used for the selection of Test Steps. The Test Steps presented for selection are those that are available for the current Context based on the information included in the Application Test Model.



CHANGE_RESULTLIST_SORTING
CHECK_DUPLICATE_GROUPING
CHECK_SORTING_OF_VARIABLE
CHECK_VARIABLE
COMPARE_TWO_VARIABLES
COMPARE_TWO_VARIABLE_LISTS
COUNT_DISTINCT_VALUES_IN_VARIABLE
EXPORT_MARKED_DOCUMENT
FOLLOW_LINK
FOLLOW_LINK_IN_DOCUMENT (Availability)
FOLLOW_LINK_IN_DOCUMENT (LinkToCLD)
FOLLOW_LINK_IN_DOCUMENT (ResultDetails)
FOLLOW_LINK_IN_DOCUMENT (SearchByLink)
FOLLOW_NAVIGATOR_LINK
GO_TO_PAGE_IN_LIST
MARK_DOCUMENT
RESET_SYSTEM
SAVE_FIELD_FOR_ALL_RESULTS
SAVE_FIELD_FROM_DOCUMENT
SAVE_NAVIGATOR_COUNT

Dieser Testschritt ändert das Kriterium für die Anordnung der Ergebnisliste, entweder in Form einer Sortierung oder einer Relevanzbewertung. Abhängig vom gewählten Kriterium kann eine absteigende / ansteigende Ordnung möglich sein.

Figure 46: Dialog for the selection of Test Step Instances

The graphical user interface of the dialog used for Test Step instance selection is partioned in two parts. The left part of the dialog is a list of all Test Step instances available in the current Context. If the system can be in one of multiple Contexts as the result of the execution of the previous Test Step, the current Test Step can be selected from a Drop Down list. The right part of the dialog displays a help text for the currently selected Test Step. The selection of a Test Step instance is done in two steps: First a Context is selected, then a Test Step instance from the Context.

In Figure 46, only Test Step instances from one Context can be selected, the control for the selection of Contexts is therefore grayed out.

In general all Contexts that are potential successor contexts of the Service instance of the previous Test Step instances can be selected. Since the Application Test Model does not contain information about the semantic differences of the different Contexts, the selection of a Context is based only on the expectations of the user.

In the dialog displayed in Figure 46, a Test Step instance is selected from a list of all available Test Step instances sorted in alphabetic order by the name of the Test Step. The name of each Test Step serves as a minimal description of the action performed by the Test Step instances. We use a verb-noun combination for each of the Test Steps defined in MTCC.

We employ a textual representation of Test Step instances since it allows easier communication about Test Steps than an icon based representation. The selection of the right Test Step is supported by a short description of the currently selected Test Step in the right half of the dialog.

As Figure 46 illustrates, some Test Step instance are included more than once in the list of the Test Step instances. The reason for this is that multiple instances of the same Service are defined for this Context, this leads to multiple instances of the Test Steps exercising this Service For the system represented in the figure, multiple instances of the FOLLOW_LINK Service representing HTML Links between pages are defined. To support the selection of the correct Test Step instance, the name of the Service instance is appended to the name of the Test Step.

The type of a Test Step instance is not displayed in the list. We argue that information about the type of a Test Step instance does not support the user in test modeling, therefore we use the type information only internally.

After a Test Step instance was selected in the dialog, it is inserted into or appended to the sequence of Test Step instances of the test.

When a Test Step instance is added or removed from the test, the sequence of Test Steps can become invalid. For instance, when a Test Step instance that submits a query to a Digital Library is removed from the list of Test Steps, the following Test Step instances that rely on the fact that the search Test Step caused the system to change, cannot be executed. In order to detect situations where the sequence of Test Steps is not valid, MTCC checks the list for correctness after each addition or deletion. If the sequence is invalid, that is, if the sequence of Test Step instances cannot be reproduced with the dynamic part of the Application Test Model, the name of the first invalid Test Step instance in the list is displayed in red. The validity can be restored by adding or removing the correct Test Step instances.

8.1.3 Representation of a Test Step Instance by GUI Elements

The second requirement of the editor beyond the construction of valid Test Step instances is the representation of the feature models of these Test Step instances in a form that allows their configuration by domain experts. In MTCC, the mapping of feature models to GUI elements and thereby the representation of feature models in a suitable form for domain experts is done by the GUIBuilder class and a number of subclasses of the Buildlet class.

These classes select the GUI elements that represent a given feature or feature subtree, build the GUI representation and integrate it with the GUI for the other features of the interface of a Test Step instance.

In contrast to other approaches for the creation of GUIs for feature models, the approach used by MTCC is not solely based on the structure of the feature model but takes into account the concepts that are represented by the feature models and the conventions for the representation of these concepts in the respective application domain.

Figure 47 illustrates the mapping of the nodes of a feature model to GUI elements in the MTCC editor. The following aspects of the mapping are of special relevance:

1. The hierarchy of the feature model is mirrored by the hierarchy of the GUI elements. The SearchForm and SearchField nodes of the feature model, for example, correspond to the hierarchy of the GUI containers used to display them.

2. The optional children of the SearchForm feature group are represented by Check Boxes.

3. If a feature group has a maximal cardinality of 1 and thus allows only the selection of a single child, and if none of the child nodes has children, this feature group is represented

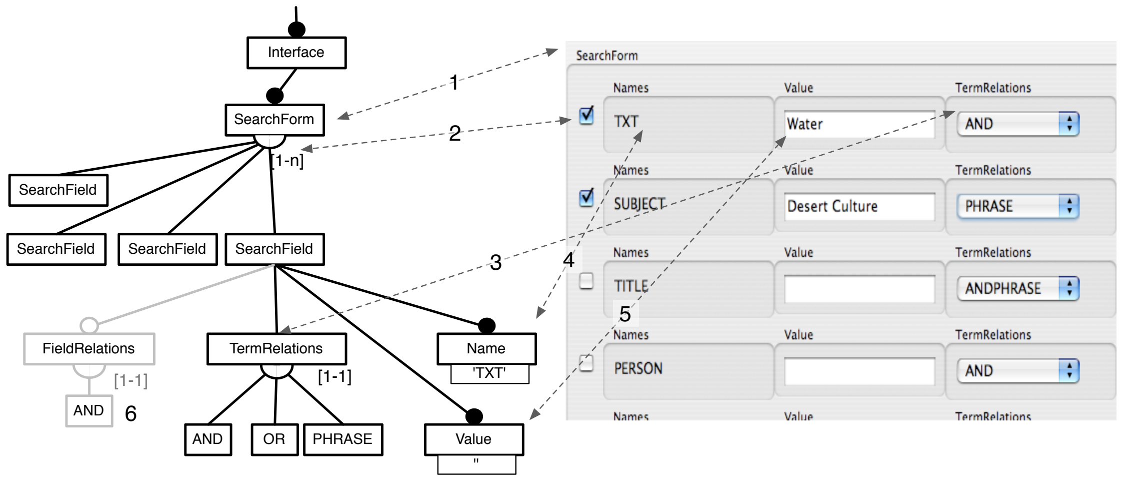


Figure 47: Mapping of features to GUI elements

by a Drop Down list.

4. An attribute that is already configured in the Application Test Model and thus has no remaining variability, is represented as a static test panel.

5. Attributes that are not configured in the Application Test Model are represented by a text input field.

6. Feature groups with only one child and a cardinality of [1,1] — and thus no variability — are not represented in the GUI.

Figure 48 illustrates the relationship of the classes GuiBilder and Buildlet and their subclasses. The GuiBuilder class coordinates the mapping of feature models to GUI elements and identifies the subclass of the Buildlet class that can build the GUI representation for a given feature. The instances of the subclasses of Buildlets are responsible for the actual instantiation of the GUI elements for a subtree of the feature model.

The behavior of a GuiBuilder instance and therefore the GUI representation of a feature model is primarily determined by its associated Buildlet objects.

MTCC differentiates generic from domain-specific Buildlets. Domain-specific Buildlets are used to represent concepts from the application domains under test that have to be represented in accordance with the conventions of the application domain under test.

An example from the application domain of Digital Libraries are complex search forms. Domain experts often expect that such concepts are represented by a specific arrangement of GUI widgets. In order to support such convention in MTCC, domain-specific Buildlets are used.

Generic Buildlet objects are not specific to the application domain and instead build a representation of feature models based on the structure of a feature model. Factors that determine the representation include the node type, the cardinality and in more complex cases the structure of a subtree of features.

The build method of the GuiBuilder class is called when a Test Step instance is selected in the editor. The following activities are performed in order to map a feature model to a tree of GUI elements:

- The list of all Buildlet objects registered with the GuiBuilder instance is traversed until an object is found that supports the mapping of the current feature node and potentially

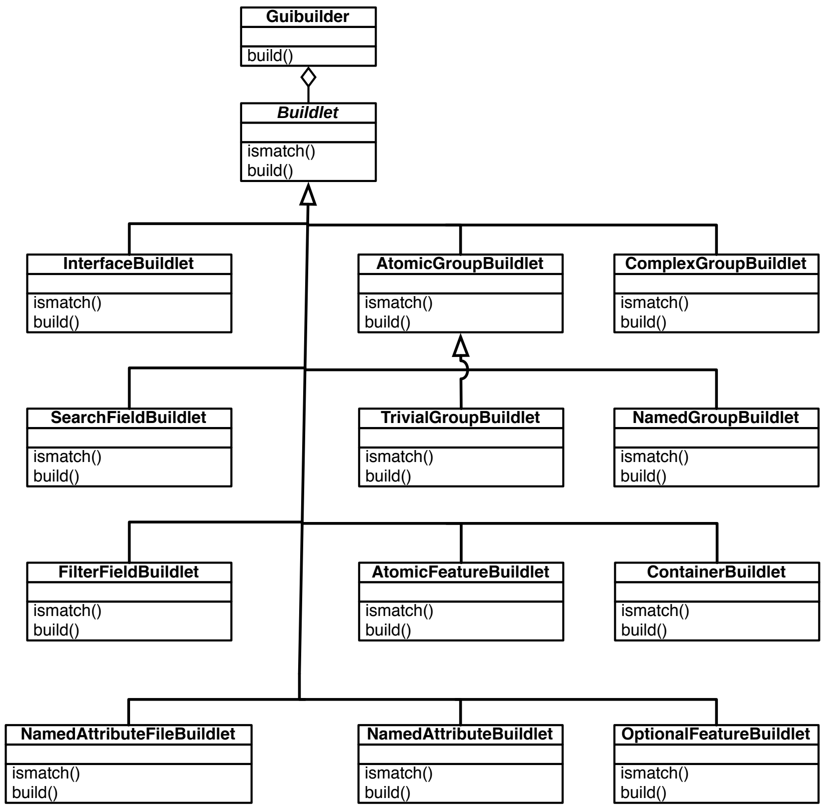


Figure 48: Classes used in the GUI representation of feature models

its sub nodes into GUI elements. In order to check the suitability of a Buildlet, its ismatch() method is called. Among the aspects considered by the ismatch() methods of different Buildlets are the name of the feature, its type, its cardinality and the type and cardinality of its children.

The List of Buildlet objects is traversed sequentially. The first Buildlet, for which the execution of ismatch() returns true, is used for the representation of the feature. Generally, the most specific Buildlets are tested first, while the more generic ones are considered last. We discuss the Buildlet objects used in the Context of this thesis and the sequence in which these objects were arranged in a later part of this section.

- After a suitable Buildlet instance has been found, its build() method is called with the current node of the feature model as its parameter. This method instantiates the GUI elements that represent the current node and potentially its children. In order to connect interactions with the user interface with the configuration of the feature model, callbacks that apply specialization steps to the feature model are registered.

  If a feature model has children that are not mapped to GUI elements by the currently active build() method, the Buildlet either calls the build() method of another Buildlet directly or delegates to the GUIBuilder that determines a suitable Buildlet by the process described above.

  The result of a call to the build() method is an object from the hierarchy of GUI classes used to implement the editor. This GUI object will often be a container class that contains the GUI elements of children that were mapped by other Buildlet instances.

The implementation of the MTCC approach prepared for this thesis uses four domain-specific and eight generic Buildlet classes. We first discuss the domain-specific Buildlet classes in the order in which they are tested against a feature model:

InterfaceBuildlet This Buildlet constructs the container object that wraps all GUI elements used for the representation of a Test Step instance interface.

SearchFieldBuildlet This Buildlet builds a representation of a single field of a search form. As Figure 49 illustrates, the representation of a search field is similar to the GUI elements used by the systems within the system family itself. Such a representation would not be possible on the basis of generic Buildlet objects alone.

FilterFieldBuildlet Analogous to the SearchFieldBuildlet, this object represents a filter used in a search form. Figure 49 displays the representation of a filter in the editor in comparison to the GUI used by the Sowiport Digital Library.

NamedAttributeFileBuildlet This Buildlet represents a single attribute node that is used to store file names as its value. While this Buildlet is not strictly specific to the application domain of Digital Libraries, it is domain-specific insofar in that it represents the attribute based on its intended usage, not its structure.

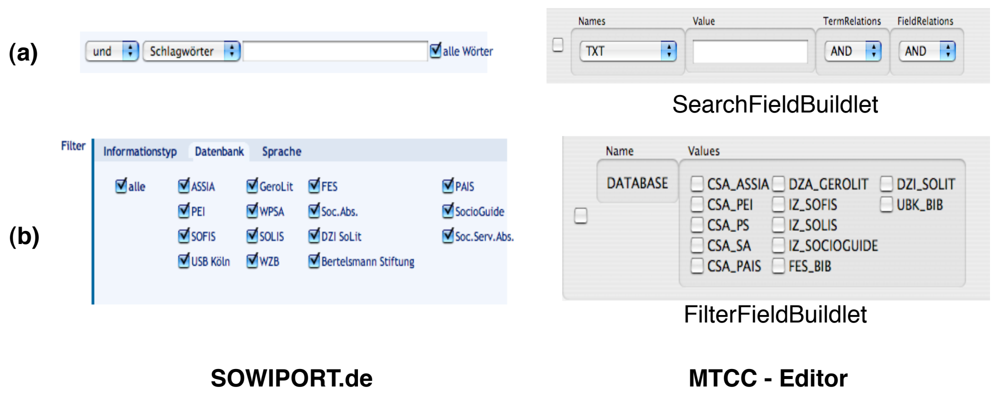


Figure 49: Representation of domain-specific features in the GUI and in the MTCC editor

The following generic Buildlet objects are used in MTCC. Figure 50 displays a part of the graphical user interface of the editor and relates the utilized GUI elements and their relation to the generic Buildlet responsible.

TrivialGroupBuildlet This Buildlet is used to represent feature groups with children that are leaves in the feature diagram. In cases where the cardinality of the feature group is [1,1], the feature group is represented by a GUI element that allows the selection of a single value from a list. Otherwise each child feature is represented by a Check Box.

AtomicGroupBuildlet This Buildlet is a subclass of the TrivialGroupBuildlet for feature groups with a single child node. When the cardinality of the feature group is [0,1], the feature group is represented by a Check Box with one option that represents the single child of the feature group. If the cardinality of the feature group is [1,1] and thus the child node is mandatory, it is selected automatically and the feature group is represented by a static text label.

ComplexGroupBuildlet This Buildlet is used for the representation of feature groups with non-trivial child nodes. Depending on the cardinality of the feature group, children are either represented by a Check Box for each child, or in cases where the cardinality is [1,1], by a radio box. The mapping of child nodes of the feature group to GUI elements is delegated to other Buildlets.

NamedAttributeBuildlet This Buildlet is used for the representation of feature nodes with an attribute node as its only child node. The feature is represented by a text input field. The name of the feature that the attributes belongs to is displayed in the border around the input field.

AtomicFeatureBuildlet This Buildlet is used for the display of a single feature without children and a cardinality of [1,1]. Since no further specialization is possible, the name of the feature is represented as a static text label.

ContainerBuildlet A ContainerBuildlet is used to display feature models where the root node has a cardinality of [1,1], but where the child nodes can be configured. Such features are represented by a container object of the GUI library used for the editor. The container object is not displayed in the GUI, it only serves as a holder for GUI elements.

OptionalFeatureBuildlet This Buildlet is used for the display of optional features with a cardinality of [0,1]. The representation of the children of this feature is delegated to another Buildlet.

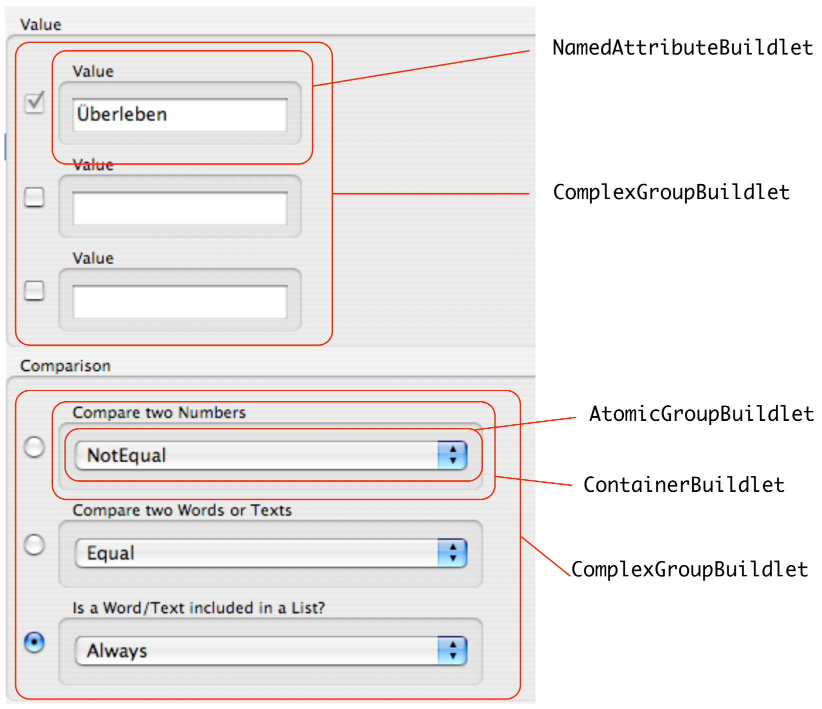


Figure 50: Part of the MTCC editor with the Buildlets used

8.2 Configuration of Test Models

The MTCC editor must not only support the display of Test Step instance feature models but also allow domain experts to interact with these models. Specifically, it has to translate user actions into specialization steps that configure the feature model.

The MTCC editor uses event handlers for every control that represents a part of the feature model of a Test Step instance. Each event handler holds a reference to the feature node represented by the GUI and calls methods on its interface to reflect the configuration of the Test Step instance as specified with and represented by the GUI.

On the level of the feature model, the configuration of a Test Step instance is represented by configuration nodes that are appended to the nodes of the feature model which are the subject of configuration. Each configuration node represents a single specialization step.

The representation of specialization steps has a number of advantages compared to the direct application of specialization steps and thus the manipulation of the feature model. For instance, it supports changes to already specialized nodes.

8.2.1 Representation of Specialization Steps as Configuration Nodes

Every action that a user takes on the GUI representation of a Test Step instance results in the application of one or more specialization steps that are applied to the feature model:

- The input of text into an input field is represented by setting the value of an attribute node.
- The selection of an element from a Drop-Down list or the selection of elements from a Check Box is represented by selecting and de-selecting features from a feature group.
- The selection or de-selection of a solitary features is represented by changes to the cardinality of the feature.

The obvious way to express actions on the GUI as specialization steps on the feature model of a Test Step is the direct manipulation of the feature model. Direct manipulation changes the feature model by the application of specialization steps with every action of the GUI and then builds a new GUI representation for the changed feature model.

A significant disadvantage of this approach is that it is not possible to undo already applied specialization steps. A specialized feature model does not contain information about its structure before the specialization occurred. For example, if a solitary feature with a cardinality of [0,1] is selected and thus the cardinality is changed to [1,1], it is not possible to discern its cardinality previous to configuration. For the representation of feature models in the MTCC editor, this loss of information is not acceptable. It is not possible to change the configuration of a Test Step instance once a specialization step was applied to the feature model, since the initial state of the feature model is lost.

Another disadvantage of the direct manipulation approach concerns the consistency of the GUI. Since each specialization step changes the feature model underlying the GUI, the GUI elements also change with every interaction with the GUI. For example, once a value has been selected from a Drop-Down list, the list itself would be removed from the GUI.

In order to support changes to the specialization steps that were applied to a feature model and to avoid changes to the overall structure of the GUI representations of Test Step instances, at least two different approaches are available: The utilization of multiple versions of a feature model or the representation of specialization steps as distinct entities, similar to the Command Pattern (Gamma *et al.*, 1995). We choose the latter approach for MTCC: Specialization steps are represented as nodes in the feature model.

8.2.2 Implementation of Configuration Nodes

Figure 51 displays the class diagram of the feature object used in MTCC together with the classes used to represent the specializations that can be applied to each node.

To support the representation of specialization steps as nodes in the feature model, two class hierarchies are used. One class hierarchy consists of *Mixin* (Bracha & Cook, 1990) classes used to extend the functionality of the original feature classes. A Mixin is an abstract class that defines methods that are added to the functionality of classes that inherit from the

Mixin. Mixins only serve to provide functionality to subclasses, an is-a relationship between a subclass and the Mixin does not exist.

The second class hierarchy includes classes representing the configuration nodes. The Mixin classes extend the functionality of the feature nodes with methods for the creation, removal, and querying of feature nodes as well as functionality that allows the configuration nodes to be applied to the feature model.

These methods are used by the event handlers registered for actions on the user interface. The decision to use Mixin classes as the basis of the implementation was made to decouple the configuration mechanism from the basic classes used for feature modeling. The subclasses of BaseConfigurationMixin do not have any class attributes. They only extend the available methods of features, feature groups and attributes.

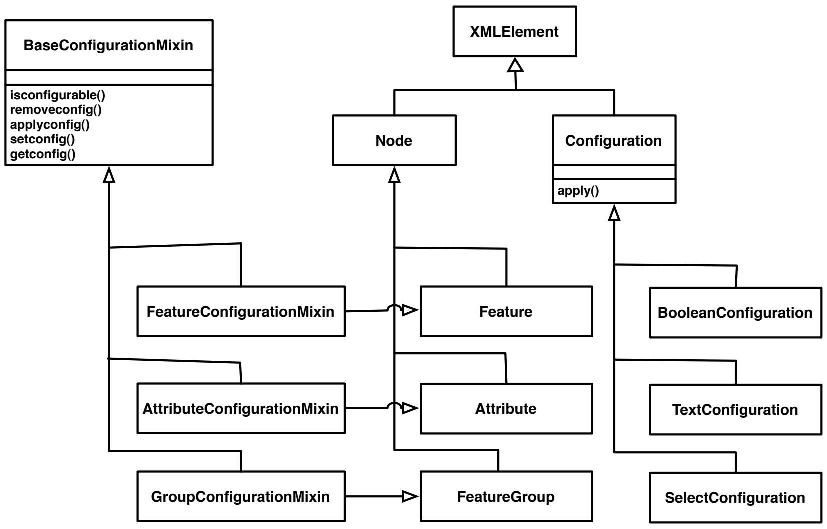


Figure 51: Class hierarchies of features and configuration nodes

The configuration nodes store the specialization steps that exist for a given node. Every configuration node is associated with one node of the feature model. Figure 52 gives an example for the relationship of feature nodes and configuration nodes. Not all specialization steps defined for feature models with cardinalities are implemented in MTCC, the cloning (Czarnecki *et al.*, 2005a,b) of features for instance is not supported.

When the feature model of a Test Step instance is loaded into the editor, a default specialization for each feature is instantiated and appended to the feature. This default configuration corresponds to the configuration of the GUI elements in the user interface.

MTCC implements the following configuration nodes:

BooleanConfiguration This configuration object is used for the selection and de-selection of features with a cardinality of [0,1]. Each instance of BooleanConfiguration stores a single Boolean value that represents the configuration of the node. A cardinality of [0,0] is represented by the Boolean value false. A configuration of [1,1] is represented by the value true. The default configuration of the value is false.

TextConfiguration This configuration node is used to store values that are assigned to attribute nodes. MTCC does not use a type system, the value currently stored by the attribute is therefore always represented as a string. The default value is an empty string.

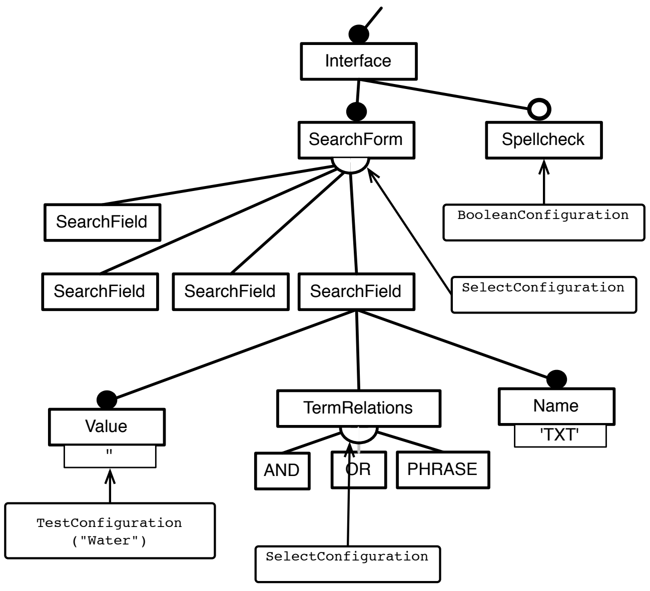


Figure 52: Feature model with configuration nodes

SelectConfiguration This configuration object is used to represent the selection of nodes from the available children of a feature group. All legal cardinalities for feature groups are supported. The configuration object stores the selection of values as a list of Boolean values. The default configuration for this node selects the minimal number of children that must be selected according to the cardinality of the feature group.

8.2.3 Application of Configuration Objects

When configuration nodes are used, the transformation of a feature model according to the specified specialization steps only takes place when the configuration nodes are applied to the feature model.

In addition to the transformation semantics for feature models with cardinalities as described in Section 4.2, MTCC also defines a set of transformation rules that conserve the position of nodes within the tree of the feature model. Figure 53 displays the feature model from Figure 52 after the configuration nodes have been applied. The absolute position of many nodes in the feature model has changed, for example, all fully specialized feature group nodes are no longer part of the model.

Figure 54 displays the feature model after the configuration was applied using the MTCC semantics. The absolute positions of all remaining nodes is unchanged. While MTCC preserves nodes that do not add information to the feature model and are thus redundant, the transformation displayed in Figure 54 facilitates the comparison of feature models before and after the application of a specialization step. MTCC uses this property when testing the reusability of test code.

8.3 Reuse of Tests

MTCC approaches the reuse of tests for different systems by providing functionality for the transfer of Test Configuration instances creates for one Application Test Model to a target Application Test Model. A target Application Test Model is compatible with a Test Configuration if it supports the Test Step sequence specified in the dynamic part of the Application Test Model

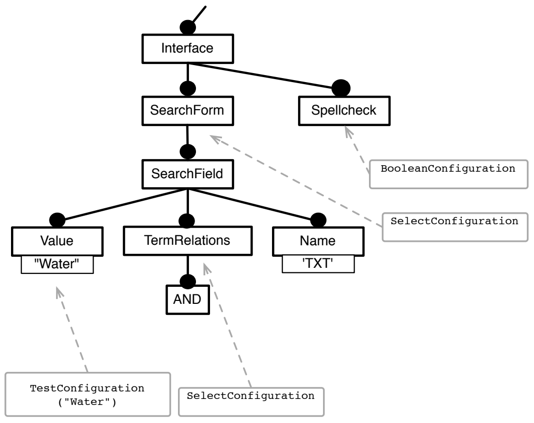


Figure 53: Feature model after the application of its configuration nodes

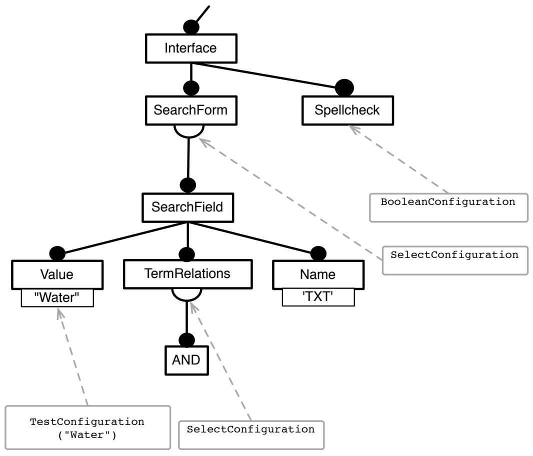


Figure 54: Feature model after the structure-preserving application of its configuration

and if the Test Step instance of the target Application Test Model can be configured like the Test Step instances of the test.

8.3.1 Transfer of Test Step Sequences

As described in Section 7.1, MTCC represents tests as sequences of configured Test Step instances. Every Test Step instance represents an action or a check for the specific interface of the system under test.

Which sequences of Test Step instances are available for a system, is determined by the dynamic part of the Application Test Model. A sequence of Test Steps can only be applied to a testee if the FSM can produce a sequence of Contexts and of Test Step instances of the appropriate type.

This condition can be formalized as follows: If only the types of Test Step instances are considered then a test can be transfered to another system when the FSM of the dynamic part of the Application Test Model for the target system can be transformed into an acceptor FSM whose states are the Test Step instances for the system and in which a transition between the Test Steps exists if a Test Step instance in the Application Test Model can be reached via any Context node. Figure 55 gives an example for a finite state machine in the acceptor representation. The FSM accepts a test that consists of the Test Steps $Start, A, C, D, E$. The test $Start, A, B, C$ is not accepted.

As we discussed in Section 7.1, since MTCC supports more than one instance of a Service in one Context, it is possible that more than one instance of a Test Step exists for one Context. MTCC also supports Test Step instances that are nondeterministic insofar that they can transfer the SUT into one of multiple Contexts. Both these properties lead to a situation where one sequence of Test Steps can be expressed by different Test Step instances.

As Figure 56 illustrates, an acceptor in MTCC is not deterministic for the reasons described above. The test $Start, A, C, D, A$ can be accepted by both the sequence $Start, A_1, C_1, D_1, A_1$ and $Start, A_1, C_2, D_1, A_2$. An unambiguous mapping of the Test Step instances of a given test to the Test Step instances of a target system is not available for all testees.

Because it is not possible for an automated system to decide which of the multiple sequences of Test Step instances best represents the intention of that test, MTCC uses an semi-automated approach to test reuse. MTCC builds a list of all candidate sequences that

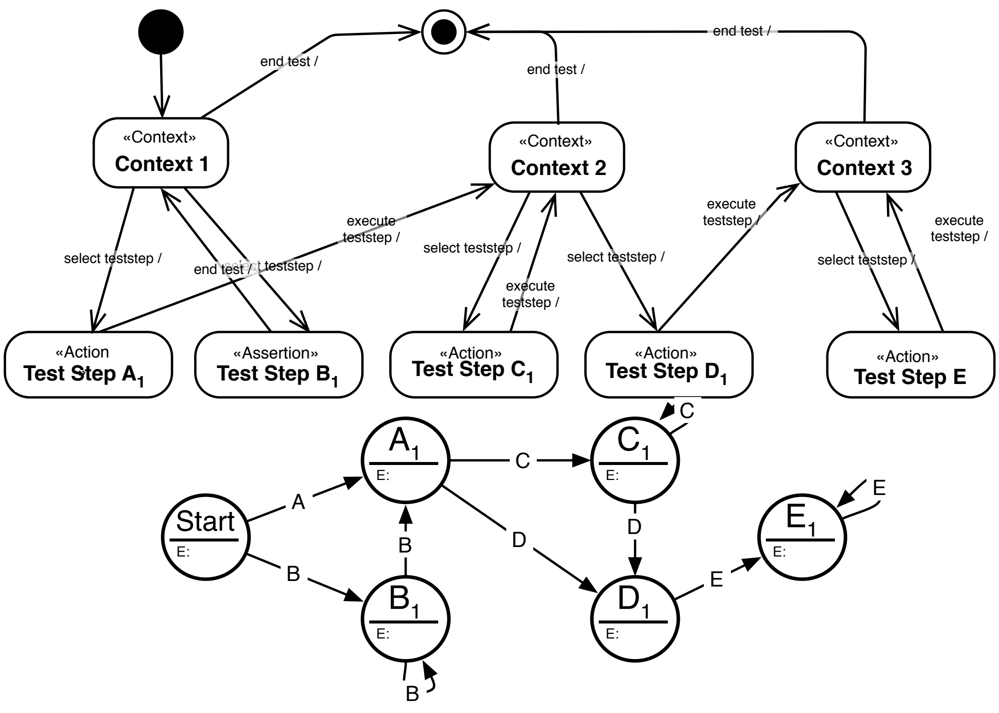


Figure 55: Acceptor for Test Step sequences based on the Application Test Model

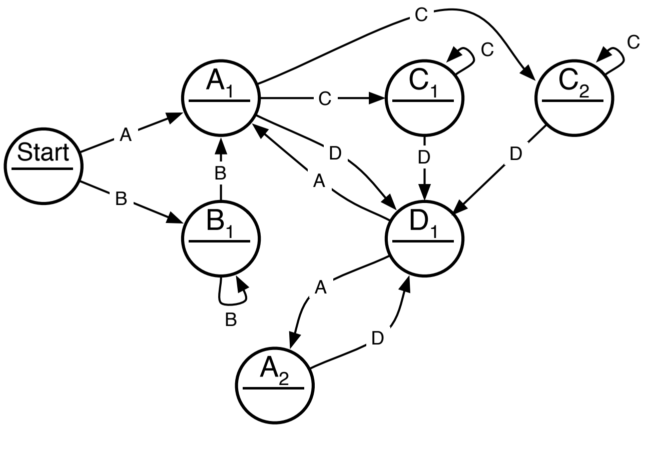


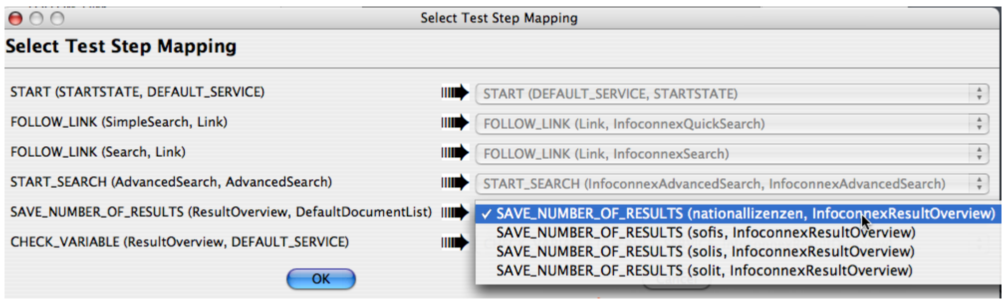
Figure 56: Non-deterministic acceptor in MTCC

can represent the test on the target system. When this list is build, the user selects the sequence that is the best representation of the intended behavior or, if none of the candidate sequences is suited, aborts the test transfer process.

MTCC represents the different candidate sequences as a tree of Test Step instances whose root is the start state of the dynamic part of the Application Test Model. Each traversal through the tree from the root to a leaf represents one trace through the Test Step instances of the test.

MTCC uses a dialog that displays the tree of candidate sequences to the user. Figure 57 displays an example of this dialog: For every Test Step instance in the test, a Drop-Down list of candidates is displayed. It displays for each Test Step instance the name of the Service instance exercised by the Test Step instance and its respective Context.

The dialog reflects the tree structure of possible traces through the system. The Test Step instance available for selection at each moment depends on the Test Step instance selected for the previous step in the sequence.



Select Test Step Mapping

Select Test Step Mapping

START (STARTSTATE, DEFAULT_SERVICE)	START (DEFAULT_SERVICE, STARTSTATE)
FOLLOW_LINK (SimpleSearch, Link)	FOLLOW_LINK (Link, InfoconnexQuickSearch)
FOLLOW_LINK (Search, Link)	FOLLOW_LINK (Link, InfoconnexSearch)
START_SEARCH (AdvancedSearch, AdvancedSearch)	START_SEARCH (InfoconnexAdvancedSearch, InfoconnexAdvancedSearch)
SAVE_NUMBER_OF_RESULTS (ResultOverview, DefaultDocumentList)	✓ SAVE_NUMBER_OF_RESULTS (nationallizenzen, InfoconnexResultOverview)
	SAVE_NUMBER_OF_RESULTS (sofis, InfoconnexResultOverview)
CHECK_VARIABLE (ResultOverview, DEFAULT_SERVICE)	SAVE_NUMBER_OF_RESULTS (solis, InfoconnexResultOverview)
	SAVE_NUMBER_OF_RESULTS (solit, InfoconnexResultOverview)

OK

Figure 57: Dialog for the selection of Test Step instance sequences

In order to transfer a test as described above, it is not sufficient that a sequence of Test Step instances of the appropriate type can be found. It is also necessary that the configuration of each Test Step instance can be transfered to the corresponding Test Step instance of the target.

8.3.2 Transfer of Test Step Configurations

In order to transfer a test from one system to another, it is necessary for the target system to have all features that are needed for the execution of the test. In MTCC, tests are transferable if all Test Step instance are transferable. A Test Step instance is transferable if the Service instance of the target system supports all features that are needed for the execution of the action or check that is represented by the Test Step instance.

On the implementation level, MTCC considers a Test Step instance to be transferable when the configuration of the Test Step instance can be reproduced for the Test Step instance of the target system. In more detail, MTCC considers the feature models of two Test Step instances compatible if the feature model of a configured Test Step instance after the application of the configuration nodes is a rooted subtree of the the feature model of the target Test Step instance. Figure 58 displays a feature model and a target feature model. The models are considered compatible since the left model is a subtree of the right model.

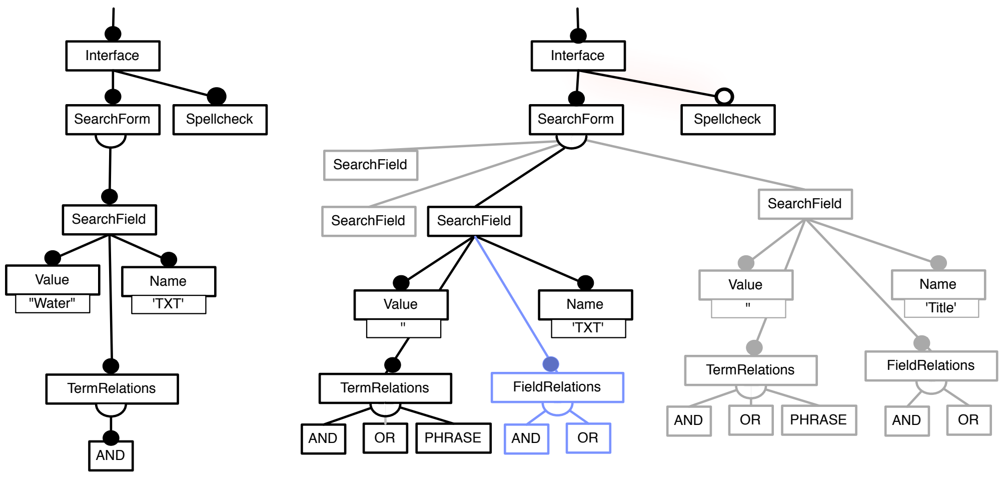


Figure 58: Two compatible feature models

In order to test the compatibility of two Test Step instances, the following activities are carried out:

- The Test Step instance to be transfered is copied. The configuration objects of the copy are applied as described in Section 8.2.3. The application of the configuration nodes removes all feature nodes that are neither selected by a user nor included in the feature model because the editor set a default value for the node.

- The feature models of both Test Step instances are canonicalized by recursively sorting all children for every node in the feature model. The sorting process considers the type of a node, its name and cardinality and in the case of attributes, its value. The purpose of the canonicalization is to facilitate the later comparison of feature models.

- The configured Test Step instance is tested against the target feature model to determine if it is a subtree. The Test Step instance is a subtree for every node present in the model there is an equal node in the model for the target system. Since both feature models are canonicalized, the upper bound of the number of comparisons necessary to check for a subtree relationship is equal to the number of nodes in the target model. Two nodes are considered equal if their type, name, cardinality and value are either equal or if a specialization action exists that can be applied to the target node that makes them equal.

When the user has selected a sequence of Test Steps from the set of candidate sequences that are compatible to the Test Step instances of the original test, the target Test Step instances are configured. The goal of this configuration is to apply specializations to the feature model of the target Test Step instance in such a way that the selected nodes of the target Test Step instance are equal to the selected nodes of the original Test Step instance feature model. We call this activity configuration transfer. In order to transfer a configured Test Step instance, the configuration nodes of the original Test Step are applied to the feature model. The resulting subtree of the feature model is then used as a template for the target feature model. The nodes of the target model that correspond to the nodes of the subtree are configured such that all subtree nodes are selected in the target model. If the number of children of the original and the target node differ, the cardinalities of feature groups are adapted as necessary.

One consequence of the method described above is that the target Test Step may include features that are not present in the original model. The FieldRelation node in Figure 58 is an example for a feature that is only present in the target model. Since MTCC cannot determine if such features change the semantics of a test, the results of the test transfer process must be verified by a domain expert familiar with the system.

8.3.3 Limitations of Test Reuse

Some limitations apply to the approach to test reuse discussed in this section. When MTCC builds the set of candidate sequences of Test Step instances, it only considers sequences as compatible that contain exactly the same Test Steps in exactly the same order as the original system. In practice, these requirements are too severe. The order, in which assertions are executed on a system, for instance, are not always relevant to the behavior of a test.

One potential weakness concerning the transfer of configurations is that features that are explicitly de-selected are not considered when the configuration is transfered. In practice, this has little consequence: Optional features are by default not selected in the MTCC editor. This means that a user will not often de-select features that he or she does not wish to be used in the test since most features are de-selected by default.

8.4 Test Execution based on Test Configuration Instances

The Test Configuration instances modeled by a domain expert are by themselves not sufficient for testing as they lack detailed information about the implementation of the testee and the test runner used for test execution. In order to test a system, MTCC uses a code generation approach that generates test cases from abstract models and templates.

The test cases generated by MTCC utilize the Test Adapter as a platform that provides functionality needed to exercise a SUT with a specific test runner. Figure 59 displays the artifacts relevant for test code generation in MTCC. In the following we examine these artifacts, their purpose and their interaction.

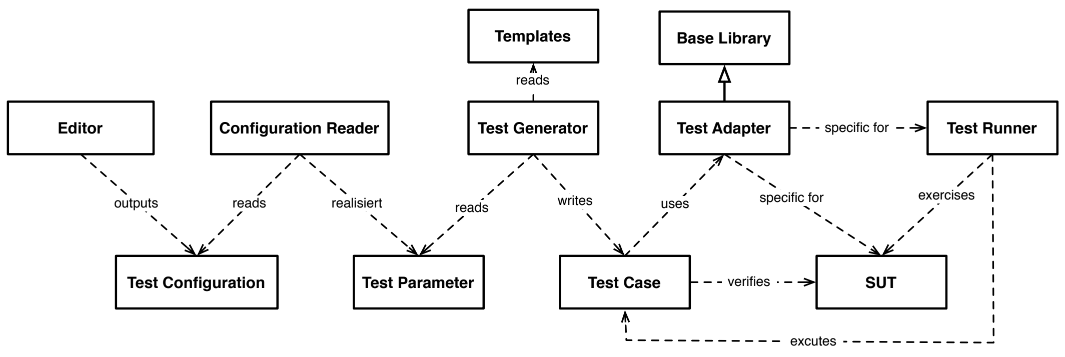


Figure 59: Artifacts used in test case generation

8.4.1 Transformation of Test Configuration Instances

Before test cases are generated from Test Configuration instances, each instance is transformed into an intermediate format. The purpose of the Configuration Reader is to transform a Test Configuration instance created with the MTCC editor into a simple representation that is used in the further generation of test code. The feature model of each configured Test Step instance is transformed into a Test Parameter object. Every Test Parameter object is implemented as an associative array with entries that represent the same information that is contained in the feature model of a configured Test Step.

Most entries in the Test Parameter object are strings, other possible data types for values include lists, numbers or other associative arrays. Listing 8.1 gives an example for a Test Parameter object in Python notation.

```
para = {
        'searchfields': [
                {
                        'term_relation': 'AND',
                        'field_relation': 'AND',
                        'name': 'TITLE',
                        'value': u'Frauen␣Maenner'
                },{
                        'term_relation': 'AND',
                        'field_relation': 'AND',
                        'name': 'YEAR',
                        'value': '1990-2005'
                }
        ]
}
```

Listing 8.1: Representation of a Test Parameter object as an associative array

One purpose of the Configuration Reader is the validation of Test Configuration instances, especially regarding the question if all necessary information needed for the Test Step instance is included in the model. The primary reason for the transformation into Test Parameter objects is the simplification of the data format used to express the semantics of the test. An associative array is used for the simplified data format since its availability can be assumed for any programming language.

This is relevant if the MTCC code generator and the test runner used for test execution are not written in a common language.

The actual transformation of the Test Configuration instances into Test Parameter objects is done by classes implemented at the system family level and is thus independent from the concrete system under test or the test runner used to execute tests.

8.4.2 Test Case Generation

MTCC assumes that test cases are generated for an xUnit implementation. Specifically, we assume that a test is represented as a test case that includes a series of method or function calls. It is assumed that a test fixture can be initialized by setUp methods/functions and that individual tests are organized into test suites which are automatically executed.

Assuming an object-oriented target language of test generation, every MTCC test is represented by a source code file with a single test class that contains a single test method. While this approach results in many individual files, it facilitates the selection of tests for execution.

The structure of a test method in MTCC is a sequence of method calls to the Test Adapter library that implements the functionality needed to exercise the system under test with the current test runner. Each method call represents the execution of a Test Step instance and is called with the Test Parameter object that represents the configuration for that Test Step and an object that stores information about the current state of test execution, for example, internal variables.

The Test Adapter contains an implementation of every Test Step instance for a specific combination of SUT and test runner. Listing 8.2 is an excerpt from a generated test case in Python. Some details of the code are omitted for clarity, the Test Parameter instances, for example, are not included in their entirety.

From the perspective of Model-Driven software development as described by Stahl and Voelter (Stahl & Völter, 2006) and illustrated in Figure 60, the Test Adapter is a platform for the generated code. For test scripts created by MTCC, this platform contains implementation details about the execution of test cases. In this context, the generated test cases are schematic code and the Test Configuration instances correspond to the application model.

MTCC does not require or assume the use of any specific approach to code generation. We argue nevertheless that the simple structure of the test code generated with MTCC are best generated with a simple approach like the *Templates and Filtering* (Stahl & Völter, 2006) method.

More complex approaches (Voelter, 2003; Völter, 2005, 2006b), while advantageous for complex code generation scenarios, are over-specialized for the generation of tests that are basically linear sequences of method-calls without any control flow.

We already discussed our motivation for the use of the Test Parameter object. Accordingly, we argue that the direct generation of program code from feature models is not well-suited for MTCC because the feature models used in the test engineering phase do not represent implementation artifacts on the level of components or subsystems but rather represent the parameterization of a single Test Step.

```
import sowiport_executor_selenium as runner

class PortalTest(unittest.TestCase):
    def test(self):
        context=Context()

        para = {...}
        self.failUnless(runner.start(context,para))

        para = {...}
        self.failUnless(runner.follow_link(context,para))

        para = {...}
        self.failUnless(runner.follow_link(context,para))

        para = {...}
        self.failUnless(runner.set_filter(context,para))

        para = {...}
        self.failUnless(runner.start_search_advanced(context,para))

        para = {...}
        self.failUnless(runner.save_navigator_count(context,para))

        para = {...}
        self.failUnless(runner.end(context,para))

if __name__ == "__main__":
    unittest.main()
```

Listing 8.2: Generated test for the programming language Python

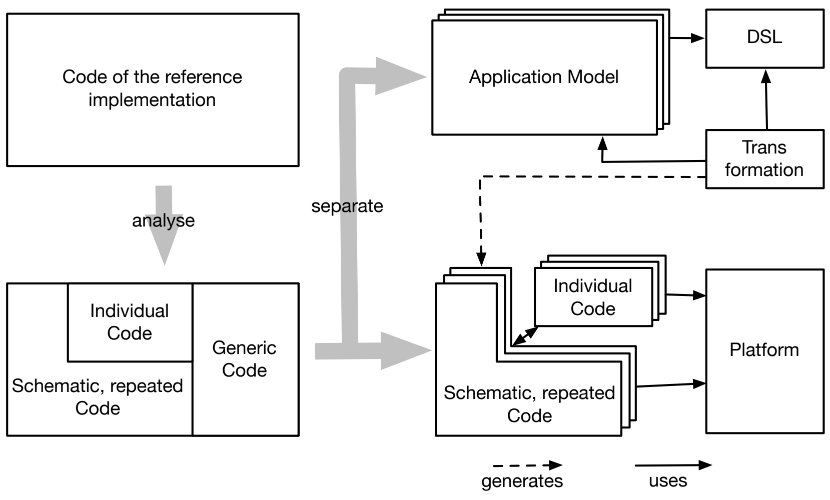


Figure 60: Concepts of Model-Driven software development (Stahl & Völter, 2006)

Figure 61 illustrates the concepts and the process of the *Templates and Filtering* code generation approach. The filtering phase mostly corresponds to the transformation of Test Configurations in Test Parameter objects. In the template phase, a template of a test class is used to generate program code for calling the methods in the Test Adapter that contain the implementation of a given Test Step instance.

Every method call is preceded by code that initializes the TestParamter-object that represents the configuration of the Test Step. Section 11.5 gives a concrete example of test code generation in MTCC based on the prototype implementation of MTCC used in the evaluation. Figure 79 on page 158 illustrates the main part of the MTCC implementation.

Figure 59 gives an overview of of the artifacts used in code generation. Figure 79 in Section 11.2 illustrates the architecture of the MTCC implementation used in the thesis.

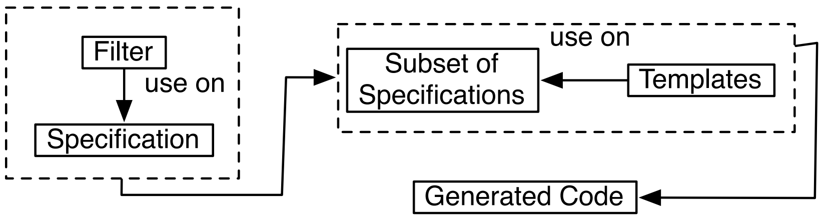


Figure 61: Concepts of the Template and Filtering approaches (Stahl & Völter, 2006)

8.5 Chapter Summary

In order to allow the construction of tests based on the information contained in the Application Test Model that represents a SUT, the model has to be represented to domain experts in a comprehensible way. Test Configurations have to be instantiated that represent the Test Steps taken on the testee and their parameterization.

The editor represents the available Test Step instances contained in the Application Test Model using GuiBuilder classes that map concepts from the feature models to generic or domain specific GUI representations. Interactions with these GUI representations are expressed as specialization steps for the feature model. MTCC expresses specialization steps as configuration nodes in the feature model to facilitate the editing of Test Configurations without directly affecting the structure of the underlying feature model.

A central aim of MTCC is the reuse of tests, represented by Test Configurations for different SUTs. MTCC supports the reuse of tests for a system if the sequence of Test Steps defined for the test can be expressed on this system and if each Test Step supports the configuration of the Test Step in the original test.

MTCC uses code generation techniques to generate test code from Test Configuration. Tests are generated in a Templates-and-Filters approach. Generated tests utilize a Test Adapters implementation that serve as a platform that encapsulates implementation specifics necessary to execute a test on a SUT.

Chapter 9

Application of MTCC to the Digital Library Domain

This chapter describes the application of MTCC for system families from the Digital Libraries domain.

Section 9.1 introduces the system family of Digital Libraries that MTCC is applied to in this thesis. We examine a system family of three Digital Libraries, the scientific portal Sowiport, the interdisciplinary information Service infoconnex and the IREON portal of the German Institute for International and Security Affairs, addressing international relations and area studies. In addition to the Digital Libraries, we investigate the Apache search server Solr.

Section 9.2 discusses the test-relevant Services for this system domain.

Beside the Test Steps that are specific to the requirements for a system family, MTCC also defines generic Test Steps, for instance for the comparison of variables that hold aspects of the state of a system family. These Test Steps are mostly independent from the systems, system family or application domain to which MTCC is applied. We examine these Test Steps in Section 9.3.

In Section 9.4, we discuss the system family-specific MTCC Services for the Digital Libraries considered in this thesis.

9.1 A System Family of Digital Libraries

The Digital Libraries Sowiport[1] and infoconnex[2] of GESIS as well as the IREON[3] portal of the German Institute for International and Security Affairs form the system family (Parnas, 1976) of testees to which MTCC is applied in this thesis.

MTCC regards the examined systems as Digital Libraries and therefore, as discussed in Chapter 5, as Information Retrieval systems. For the purpose of this thesis, we limit our analysis of Digital Libraries to the search process and the presentation of results. In addition to the digital libraries, MTCC is applied to the search server Solr[4]. We examine Solr in order to assess the applicability of MTCC to systems that are not members of the system family but rather part of their underlying infrastructure.

The three Digital Libraries considered have been designed and implemented at the German Social Science Infrastructure Services (GESIS) in cooperation with external partners.

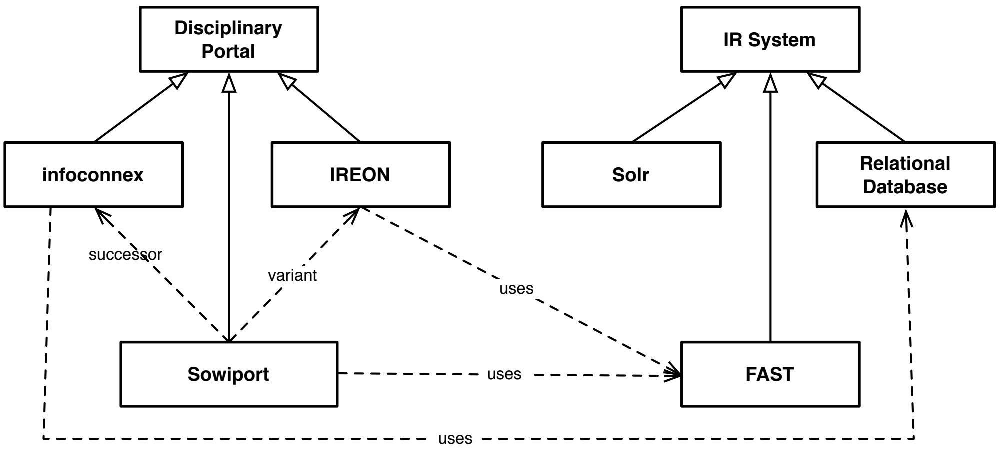


Figure 62: Relations of systems within system families

Figure 62 illustrates the relations of the systems as well as their commonalities.

The central use case that is relevant for all three Digital Libraries as well as Solr is search. A user formulates a query and submits it to the system, the system generates one or more lists of documents that are likely to be relevant to the query. The user surveys the documents in the result lists, selects documents according to her or his information need and further refines the query.

Sowiport, infoconnex and IREON differ in the functionality available for searching as well as in the types of interactions that are possible with an initial list of results. Solr does not support functionality for the refinement of queries.

All considered systems are web applications. A user interacts with such systems using a web browser and the HTTP protocol.

9.1.1 Sowiport

Sowiport is a disciplinary information portal for the social sciences. It integrates 15 databases and contains 2.5 million documents in total. The system is implemented in PHP and Java; a commercial IR system is used as its search back-end.

Figure 63 and Figure 64 display the simple and the advanced search form of Sowiport. The simple search form consists of seven text input fields that support queries for specific fields of documents. By using the Check Box *Alle Wörter*, the conjunction of the terms within one text entry can be switched between Boolean AND and OR. By selecting the Check Box in the lower part of the search form, the search can be restricted to documents that were published within the last four years.

The advanced search form displayed in Figure 63 extends the simple search by filters that restrict a search to a subset of all available documents and by providing support for Boolean operators that allow the formulation of more complex queries. The filter displayed in the figure supports the restriction of result documents by their information type. Other filters can be used to restrict a query to certain languages and databases.

The entry fields of the advanced search form allow the explicit selection of the fields that are to be searched as well as the choice of the Boolean operator that determines the

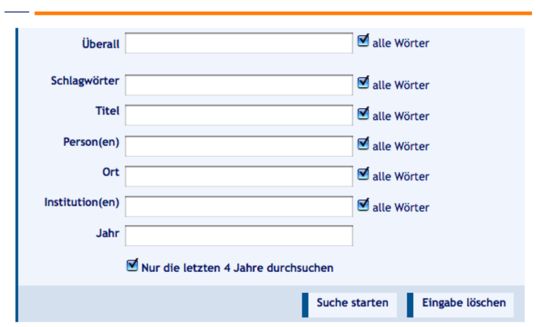


Figure 63: Simple search in Sowiport

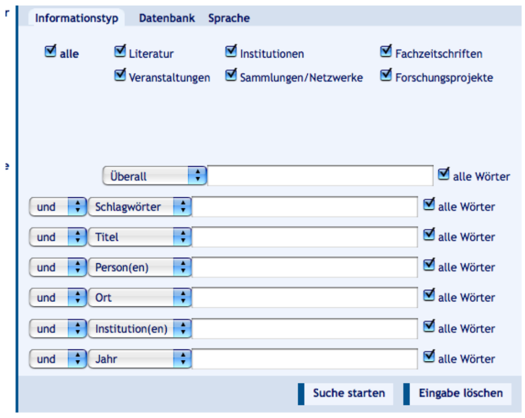


Figure 64: Advanced Search in Sowiprt

relationship of the multiple fields in search.

Figure 65 displays the result page that presents the documents that are returned for a query. The result page also includes Services to refine the query. A list with short representations of the documents that were found for a query is in the center of the result page. A representation of the query with all transformations that were applied to the terms of the query is displayed in the upper part of the page (1).

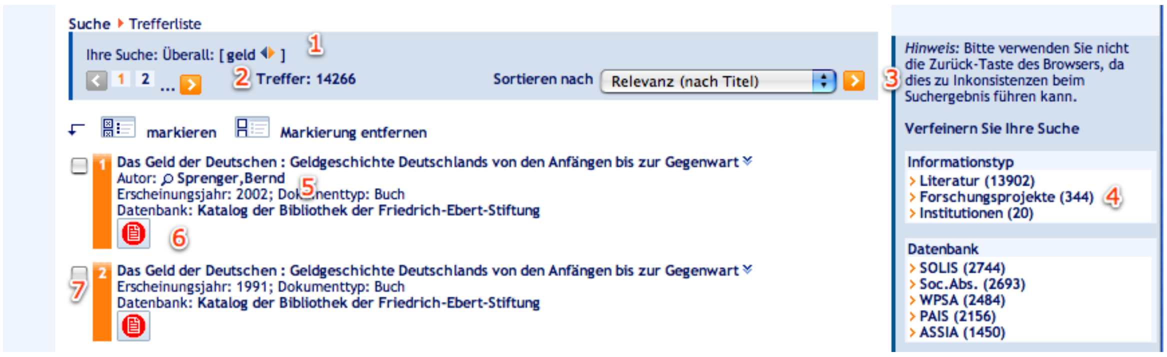
Controls exist for browsing the list (2) and for changing the order in which the documents are displayed (3). The result page only displays a short surrogate for each document that was returned. More information about the document is given in the detailed view of the document that is opened by clicking the title of the document (4). Some fields, for instance the author of a document (5), are links that send a new query to the Digital Library. In this case, a search for the author would be started. Other links allow users to work with reference management systems[5] or help to check whether the full text of a document is available (6).

By using a Check Box (7), it is possible to mark documents for export. The links in the right part of the page are navigators. Navigators allow the restriction of a result set by facets (Baeza-Yates & Ribeiro-Neto, 1999) and thus the refinement of the search. Every navigator displays the most frequently occurring values for a given field in the result set and their count. By clicking the link associated with a value, the result set of the query is restricted to all documents that contain this value for this field.

9.1.2 infoconnex

infoconnex is a portal that allows the integrated search of documents from the disciplines of social sciences, pedagogics and psychology. The portal is implemented in Java; the Information Retrieval component is based on a relational database.

Regarding the functionality and design of its search forms, infoconnex is similar to the simple and advanced search in Sowiport. Differences between Sowiport and infoconnex exist in the display of search results as displayed in Figure 66. infoconnex returns up to three result lists for a query. Each result list contains documents from one database. The currently


Suche ▸ Trefferliste

Ihre Suche: Überall: [geld] 1

1 2 ... 2 Treffer: 14266 Sortieren nach Relevanz (nach Titel) 3

markieren Markierung entfernen

1 Das Geld der Deutschen : Geldgeschichte Deutschlands von den Anfängen bis zur Gegenwart
Autor: Sprenger,Bernd 5
Erscheinungsjahr: 2002; Dokumenttyp: Buch
Datenbank: Katalog der Bibliothek der Friedrich-Ebert-Stiftung
6

7 2 Das Geld der Deutschen : Geldgeschichte Deutschlands von den Anfängen bis zur Gegenwart
Erscheinungsjahr: 1991; Dokumenttyp: Buch
Datenbank: Katalog der Bibliothek der Friedrich-Ebert-Stiftung

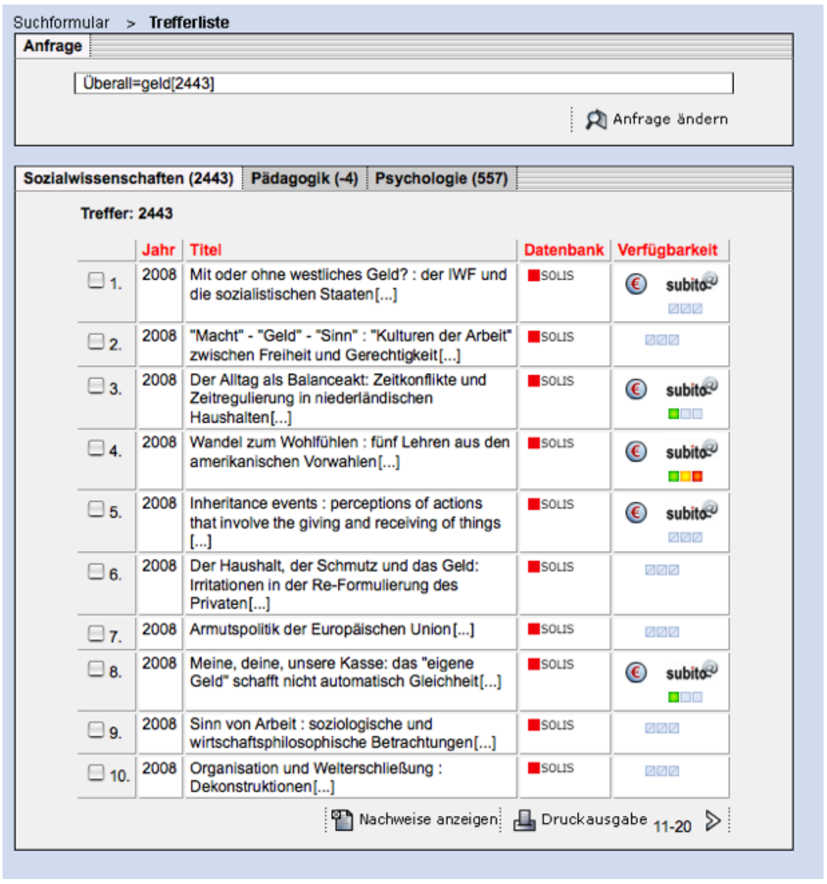
Hinweis: Bitte verwenden Sie nicht die Zurück-Taste des Browsers, da dies zu Inkonsistenzen beim Suchergebnis führen kann.

Verfeinern Sie Ihre Suche

Informationstyp
› Literatur (13902)
› Forschungsprojekte (344) 4
› Institutionen (20)

Datenbank
› SOLIS (2744)
› Soc.Abs. (2693)
› WPSA (2484)
› PAIS (2156)
› ASSIA (1450)

Figure 65: Result page in Sowiport


Suchformular > **Trefferliste**

Anfrage

Überall=geld[2443]

Anfrage ändern

Sozialwissenschaften (2443) | **Pädagogik (-4)** | **Psychologie (557)**

Treffer: 2443

	Jahr	Titel	Datenbank	Verfügbarkeit
1.	2008	Mit oder ohne westliches Geld? : der IWF und die sozialistischen Staaten[...]	SOLIS	subito
2.	2008	"Macht" - "Geld" - "Sinn" : "Kulturen der Arbeit" zwischen Freiheit und Gerechtigkeit[...]	SOLIS	
3.	2008	Der Alltag als Balanceakt: Zeitkonflikte und Zeitregulierung in niederländischen Haushalten[...]	SOLIS	subito
4.	2008	Wandel zum Wohlfühlen : fünf Lehren aus den amerikanischen Vorwahlen[...]	SOLIS	subito
5.	2008	Inheritance events : perceptions of actions that involve the giving and receiving of things [...]	SOLIS	subito
6.	2008	Der Haushalt, der Schmutz und das Geld: Irritationen in der Re-Formulierung des Privaten[...]	SOLIS	
7.	2008	Armutspolitik der Europäischen Union[...]	SOLIS	
8.	2008	Meine, deine, unsere Kasse: das "eigene Geld" schafft nicht automatisch Gleichheit[...]	SOLIS	subito
9.	2008	Sinn von Arbeit : soziologische und wirtschaftsphilosophische Betrachtungen[...]	SOLIS	
10.	2008	Organisation und Welterschließung : Dekonstruktionen[...]	SOLIS	

Nachweise anzeigen | Druckausgabe 11-20

Figure 66: Result page in infoconnex

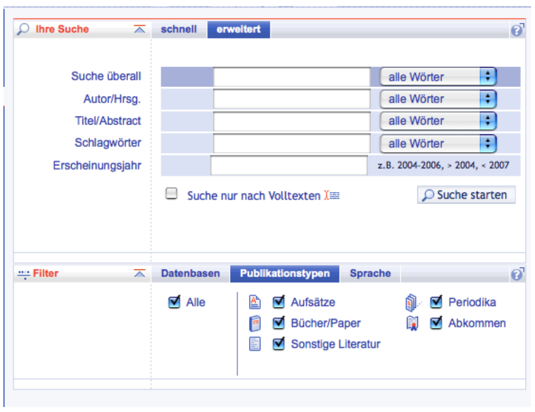
displayed list can be selected by clicking the tab at the top of the list.

In contrast to the other Digital Libraries, infoconnex does not support navigators or the ordering of documents by relevance based on the similarity of a query and a document. infoconnex does provide a relevance ranking based on the centrality of an author for documents from the social sciences (Mutschke, 2003).

[1]http://www.sowiport.de
[2]http://www.infoconnex.de
[3]http://www.ireon-portal.de
[4]http://lucene.apache.org/solr/
[5]http://www.refworks.com

9.1.3 IREON

The IREON portal provides information about international relations and area studies. The portal shares significant parts of its implementation with Sowiport, especially concerning the transformation of queries and underlying retrieval system. Sowiport and IREON also have some databases in common.



Ihre Suche | schnell | erweitert

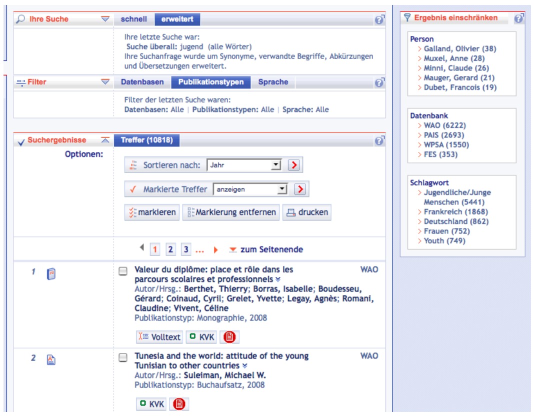
Suche überall		alle Wörter
Autor/Hrsg.		alle Wörter
Titel/Abstract		alle Wörter
Schlagwörter		alle Wörter
Erscheinungsjahr		z.B. 2004-2006, > 2004, < 2007

☐ Suche nur nach Volltexten | Suche starten

Filter | Datenbanken | Publikationstypen | Sprache

☑ Alle | ☑ Aufsätze | ☑ Periodika
☑ Bücher/Paper | ☑ Abkommen
☑ Sonstige Literatur

Figure 67: Advanced search in IREON



Ihre Suche | schnell | erweitert

Ihre letzte Suche war:
Suche überall: jugend (alle Wörter)
Ihre Suchanfrage wurde um Synonyme, verwandte Begriffe, Abkürzungen und Übersetzungen erweitert.

Filter | Datenbanken | Publikationstypen | Sprache

Filter der letzten Suche waren:
Datenbasen: Alle | Publikationstypen: Alle | Sprache: Alle

Suchergebnisse | Treffer (10818)

Optionen:
Sortieren nach: Jahr
Markierte Treffer: anzeigen
markieren | Markierung entfernen | drucken

1 2 3 ... zum Seitenende

1 Valeur du diplôme: place et rôle dans les parcours scolaires et professionnels — WAO
Autor/Hrsg.: Berthet, Thierry; Borras, Isabelle; Boudesseu, Gérard; Coinaud, Cyril; Grelet, Yvette; Legay, Agnès; Romani, Claudine; Vivent, Céline
Publikationstyp: Monographie, 2008
Volltext | KVK

2 Tunesia and the world: attitude of the young Tunisian to other countries — WAO
Autor/Hrsg.: Suleiman, Michael W.
Publikationstyp: Buchaufsatz, 2008
KVK

Ergebnis einschränken

Person
- Galland, Olivier (38)
- Muxel, Anne (28)
- Minni, Claude (26)
- Mauger, Gerard (21)
- Dubet, Francois (19)

Datenbank
- WAO (6222)
- PAIS (2693)
- WPSA (1550)
- FES (353)

Schlagwort
- Jugendliche/Junge Menschen (5441)
- Frankreich (1868)
- Deutschland (862)
- Frauen (752)
- Youth (749)

Figure 68: Result page in IREON

Figure 67 displays the advanced search for all databases in the IREON Portal. The functionality of search forms and filters is analogous to those in SOWIPORT. Besides the search forms displayed, IREON also provides a number of of additional search forms that are adapted for special subsets of the portal.

Figure 67 displays the result page in IREON. The functionality of the elements in the user interface is similar to Sowiport.

9.1.4 GESIS Solr Installation

In contrast to the Digital Libraries introduced so far, the Solr installation of the GESIS is a search server, not a Digital Library. The Solr search server only offers a minimal HTML-based user interface that supports the submitting of queries to a system and provides statistics about the operations of the system. The GESIS Solr installation includes the same document collections as the Sowiport and IREON portals. It is used for automated search tasks and complex queries for which the features of a Digital Library are not needed.

We include the Solr installation at the GESIS in our discussion since it represents a part of the low-level infrastructure that would be used by a Digital Library. By including Solr, we investigate whether MTCC can be used for the testing of such sub systems of Digital Libraries as well as for testing the libraries themselves.

9.2 Test-Relevant Services and Test Steps

In order to test the systems introduced in Section 9.1, they have to be described in terms of their Services and Test Steps. The Services and Test Steps relevant for the system family considered in this thesis are presented in Sections 9.3 and 9.4.

9.2.1 Manipulation of the SUT

We identify the test-relevant Services and Test Steps for the systems of the considered system family by examining the test sets discussed in Section 10.3.1. We argue that these Test Steps represent both current testing practice and the requirements of the domain experts to a testing system.

Each test set holds a number of tests or documented faults that can serve as the base of a test for one of the systems in the system family. The different test sets come from sources such as previously defined manual and automated tests. Manual tests are specified by domain experts, and documented, testable faults for a system.

Based on the test sets, the most important Services of a Digital Library are the search for documents and the display of those documents that were returned as the result of a search. In addition to these core functionalities, the various options that allow the refinements of queries are also of importance. MTCC must include models for Services and Test Steps that represent these functionalities and their interactions with a user. An additional requirement for MTCC is the ability to model retrieval tests.

An application of MTCC for Digital Libraries must include models that represent the functionality to formulate a query and submit it to the system. This model of the search Service must represent all options that the system provides for searching. Furthermore, MTCC must provide Services that represent a list of potentially relevant documents returned in the result page as well as the functionality provided to further refine this list or otherwise interact with it. MTCC must also include Test Steps that represent the interactions performed with a Service for example the entry of search terms into a search form, the selection of filter values or the interactions with document surrogates from a list of results.

The Services and Test Steps that support the refinement of a search are an important aspect of the Digital Libraries discussed here. They address the problem discussed in Chapter 5 that a user will often start a search with only a vague definition of his or her information need but will gradually refine it in the process of a search. Among the Services that serve this purpose are the following:

- The navigators in Sowiport and IREON support the restriction of a result set of documents to those documents that contain a specific value for a field.
- The order, in which the documents of a result set are presented to the user can be adapted to the preferences of the user. The systems discussed here support the arrangement of documents by different concepts of relevance or the sorting of documents based on the contents of fields.
- The link search in IREON and Sowiport allows the start of a new search based on one of the results of the previous search. For example, a click on the author of a document starts a search for all documents by this author.

MTCC represents each function above as an independent Service We argue that such a representation is advantageous compared to the extension of existing Services: The complexity of the individual Services is limited, making maintenance easier; and functionality can be added to the models of a system without changing existing Service An advantage of the MTCC modeling approach is that fine-grained Service models allow for a more precise description of the behavior of a system in the Application State Model.

9.2.2 Verification of the Testee

In order to test a system it is not only necessary to exercise the system and interact with its provided functionality, but it is also necessary to verify that the behavior of the system and its visible states conform to the expectations expressed in a test. The Services and Test Steps provided in MTCC for the verification of a system fall into two groups, those used to capture some visible aspect of the system state in a variable and those that compare the values of these variables with the expectations of the test:

- The aspects of the system states of a Digital Library that MTCC must be able to capture and store, include the number of documents in a result set, the fields included in the documents and the values of these fields. This also includes the values and counts with navigators. MTCC represents the inspection of the system state by Test Steps of the type assertion, meaning that the state of the testee itself is not manipulated, only captured. MTCC uses variables to store the captured aspect of the system state.
- Test Steps that verify the state of a system operate on the variables. Test Steps of this kind compare numeric or textual values, test the values in two lists for specific subset or superset relationships and determine Information Retrieval metrics like recall and precision.

As described above, MTCC uses variables to store an aspect of the system state for later comparison. The use of variables allows to separate Test Steps used for capturing the system state from those that test the system state against a condition. This separation helps to reduce the complexity of individual Test Steps and lowers the number of Test Steps that must be implemented for a system.

Variables are always expressed as lists of values. Single values are represented by a list with a single element. MTCC does not implement a type system, every entry in a variable list is implemented as a string. The interpretation of this string depends on the Test Step that uses a variable. Depending on the Test Step, textual or numeric interpretations are possible.

9.3 Generic Services

Generic Services and the Test Steps that refer to these Services are mostly independent from the system that is tested by MTCC.

Two types of generic Services exist in MTCC, system-independent Services and domain-independent Services. System-independent Services represent characteristic functionality for the application domain that MTCC is applied to, but Service instances do not differ between systems. The Test Steps of domain-independent Services represent functionality that is needed for testing with MTCC, but that is not specific for a particular system family or system. Most domain-independent Test Steps are assigned to the DEFAULT_SERVICE.

9.3.1 System-independent Services

The feature model of system-independent Services is identical for every instance in every Context and in every system. Since every instance of a Service is identical to every other instance, the Service instances do not contain any information and serve purely as markers for the availability of the Service in a particular Context.

The Service LINK marks Contexts that support the change to another Context without configuration of a Test Step interface. For the system family of web-based Digital Libraries considered in this thesis, the link Service represents the HTML links between different pages of the digital library. Information about the available targets of a LINK Service instance is included in the Application State Model and in the dynamic part of the Application Test Model. The feature model of an instance of the LINK Service only marks the existence of one or more links in a system.

The Test Step FOLLOW_LINK represents the following of a link. Since the activation of a link changes the current Context of the system, this Test Step is an action.

The Service EXTERNAL_TEXT is used for Contexts where test-relevant functionality is not or not completely represented by MTCC Service instances. It is used in cases where Contexts have to be tested that are not part of the system modeled with MTCC. For web applications, one example for such Contexts would be external pages, reachable by links from the modeled system. The Test Step SAVE_EXTERNAL_TEXT is a SameStateAction that stores all text that is displayed in a Context, for example, all textual content displayed on a web page.

9.3.2 Domain-independent Services

Domain-independent Services represent functionality that is specific to MTCC and mostly independent from the system or system family under test. The implementation of MTCC examined in this thesis only defines one domain-independent Service the DEFAULT_SERVICE. This Service is instantiated for every Context of a system. A number of Test Steps are assigned to the DEFAULT_SERVICE. All of these Test Steps with the exception of the RESET_SYSTEM Service are assertions, they do not change the current Context of the system or manipulate its state:

- The Test Step SET_VARIABLE assigns a user defined value to a variable. This variable can later be used in other Test Steps, for example, in comparisons. Two variants of this Test Step exist. The Test Step SET_VARIABLE_LONG_TEXT is represented in the editor by a multi-line text input field and allows the entry of long texts. The Test Step SET_VARIABLE_FROM_FILE is used to store values read from a file into a variable.

- The Test Steps COMPARE_TWO_VARIABLES and CHECK_VARIABLE are used to compare variables with expected values The Test Steps CHECK_VARIABLE requires the value to be compared against a value to be entered in the editor while COMPARE_TWO_VARIABLES is used to compare two previously defined variables.

  Since MTCC does not use a type system for the values in the variables, both Test Steps require the user to specify the data type used for the comparison. In addition, the user must select whether the values in the variables are expected to be equal, unequal or of another relation.

  The Test Step COMPARE_TWO_VARIABLE_LISTS is used to compare two variables that each hold multiple values. The values in the variables can either be treated as sets or ordered lists. They can be tested for equality and for is-part-of relationships.

- The Test Step CHECK_SORTING_OF_VARIABLE checks whether the values in a variable are sorted. Values can be interpreted as numbers or as strings.

- The Test Step RESET_SYSTEM is an action that return the system to its initial state before any Test Steps were executed on it. How this Test Step is implemented depends on the SUT. In the case of the digital libraries discussed here, the Test Step resets the session state of the SUT, emulating a user logging out.

- The Test Step COUNT_DISTINCT_VALUES_IN_VARIABLE calculates the cardinal number of the set of values in the variable and saves the result in another variable.

9.4 Domain-Specific Services

In the following we discuss the system family specific Services that represent the functionality of the Digital Libraries discussed in Section 9.1. For each Service we discuss its assigned Test

Steps. The Services and Test Steps were defined based on an analysis of the concepts required for the modeling of the test sets defined in Section 10.3.1

Every Service discussed in the following is represented by a feature model. All feature models have the following two feature nodes in common: The Type node that represents the class of the Service and the Name node that contains the unique name of this particular Service instance.

The feature models represents Services as defined in the Domain Feature Model. A Service instance would be a specialization of this model. We discuss both the system family and system-level models for complex Services.

9.4.1 The **SEARCH** Service

The SEARCH Service represented as a feature model in Figure 69, represents a search form. A search form in the model consists of multiple input fields, represented by an InputField node. For search forms that allow the selection of the queried field for each input field, the feature group FieldNames represents all selectable fields for a single InputField. The feature groups TermRelations and FieldRelations represent the relations of multiple terms within a single input field and of multiple input fields to each other. The FieldValue attribute is configured in test modeling and specifies the subject of the query for this input field.

The filters that are available for restricting a search are modeled as a FilterForm. Every FilterField represents a single filter with multiple selectable values. The feature FilterName describes the field that the filter refers to, for example, the language or the document type of a bibliographic record. Possible values for the filter are defined in the feature group FieldValues.

Figure 70 displays the feature model of a Service instance. In the Service instance, it is possible to search either in the field title or in the field person. The different input fields represented by the model are always in an AND relationship, the terms in an input field can be either in an AND or OR relation.

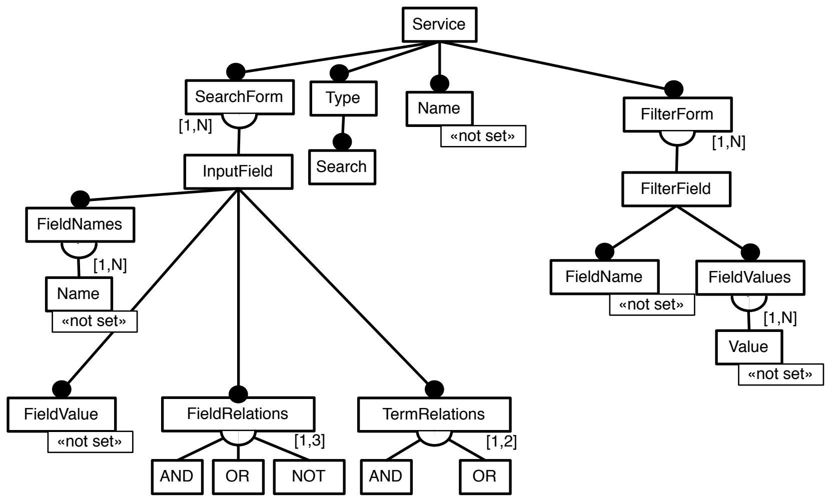


Figure 69: Feature model of the SEARCH Service at the domain level

Two Test Steps are defined for the SEARCH Service One Test Step is used to submit a search, the other to set the filters that limit a search to a subset of all documents in a database. The Test Step START_SEARCH is an action that represents the submission of a search. The Test Step SET_FILTER is used for the selection of filter values. Since the Test Step does not cause a change of Context, it is a SameStateAction.

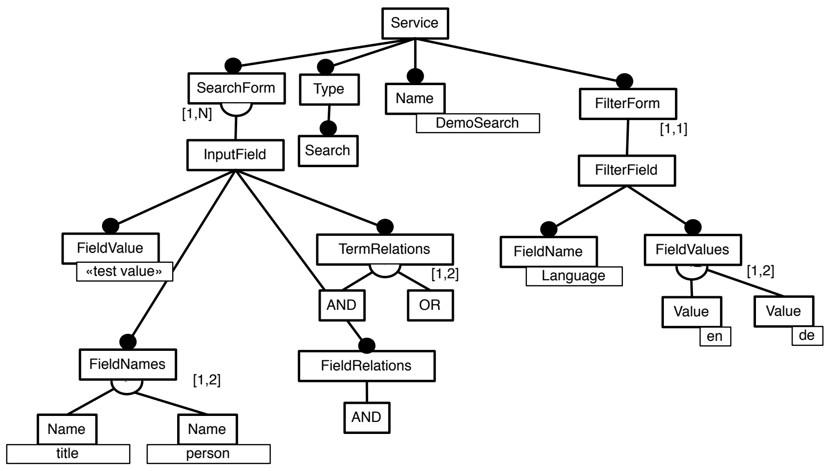


Figure 70: Instance of the SEARCH Service represented as a feature model

9.4.2 The **SEARCH_OPTIONS** Service

The SEARCH_OPTIONS Service displayed in Figure 71, represents options that are available to influence the processing of queries. The only search option available for the Digital Libraries considered here are cross-concordances. The SEARCH_OPTIONS Service is a separate Service from the SEARCH Service which allows the representation of a system where search options are set globally for all search forms.

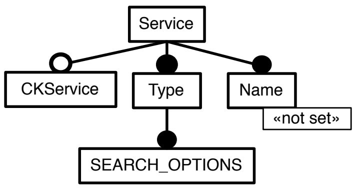


Figure 71: Feature model of the SEARCH_OPTIONS service

9.4.3 The **REQUEST_INFORMATION** Service

The Service REQUEST_INFORMATION, represented in Figure 72, models the presentation and explanation of a query after it was executed by the system. In the Digital Libraries considered here, this Service informs about the transformations that were applied to a query during query expansion. The Test Step SAVE_QUERY_INFORMATION is assigned to the Service This Test Step stores the information displayed by the Service in a variable.

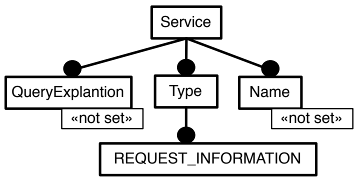


Figure 72: Feature model of the REQUEST_INFORMATION Service

9.4.4 The DOCUMENT_LIST Service

The DOCUMENT_LIST Service represents a list of documents returned for a query. In the feature model in Figure 73, this list is characterized by three feature nodes: The fields that each of the result documents can potentially contain are represented by the FieldNames node. This feature determines which fields of result documents can be checked against expectations and stored in variables.

The number of documents that can be simultaneously displayed is given by the feature PageLength. The SupportsSelection node reflects whether it is possible to mark documents, for example, in order to export them.

The DOCUMENT_LIST is not only used to present an overview of the results of a query, but also for the detailed display of documents. The difference between a DOCUMENT_LIST instance used to represent an overview of documents on the result page and an instance that represents one ore more documents in detail is that the latter has a PageLength of one and contains all FieldNames in the result overview. The feature GroupsDuplicates indicates if the result list supports the marking of duplicates in the result set.

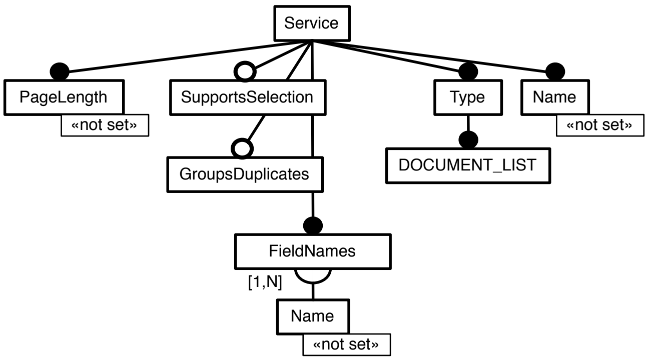


Figure 73: Feature model of the DOCUMENT_LIST service

The DOCUMENT_LIST Service has the greatest number of assigned Test Steps of all Services. This reflects the central role that the completeness and correctness of search results has in the Digital Library domain:

- The SameStateAction SAVE_NUMBER_OF_RESULTS stores the absolute number of documents that were returned as the result of a query.
- The SameStateAction SAVE_FIELD_FROM_DOCUMENT stores the value of one field for one document in a variable.
- The SameStateAction SAVE_FIELD_FOR_ALL_RESULTS stores the values of one field for all documents within a range of the result list.
- The Assertion Test Step CHECK_DUPLICATE_GROUPING verifies if two documents are marked as duplicates.
- The SameStateAction MARK_DOCUMENT marks a document on the current page of the result list.
- The Test Step CHECK_RECALL_AND_PRECISION_RESULTS is used to calculate recall and precision for a section of the result list and to store the result in a variable. A prerequisite for the invocation of this Test Step is that the IDs of all documents are available from the list of documents and that relevance assessments are available either for all documents or for a pool of documents.

9.4.5 The **MARKED_DOCUMENT_HANDLER** Service

The Service MARKED_DOCUMENT_HANDLER, displayed in Figure 74 represents the invocation of an action on some previously selected documents. Documents are selected with the Test Step MARK_DOCUMENT. An example for such an action is an export in the BibTex format. All available actions are described by the feature group ExportActions. Only the Action Test Step EXPORT_MARKED_DOCUMENT is assigned to the Service This Test Step allows the selection and execution of an action that is to be invoked on the documents by its name.

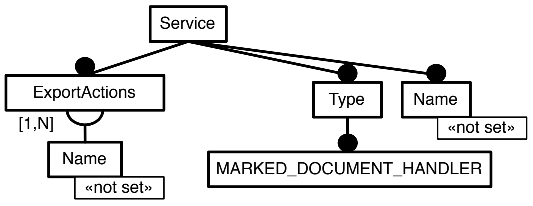


Figure 74: Feature model of the MARKED_DOCUMENT_HANDLER service

9.4.6 The **ORDERING** Service

The ordering Service describes the different criteria by which the documents in the result list can be ordered. Criteria include the fields by which the result list can be sorted, as well different concepts of relevance or ranking. The feature group OrderFields in Figure 75 specifies for each criterion whether it supports ascending and/or descending order. The Name attribute contains the name of the criterion. The Action CHANGE_RESULTLIST_SORTING is used to invoke the ORDERING Service

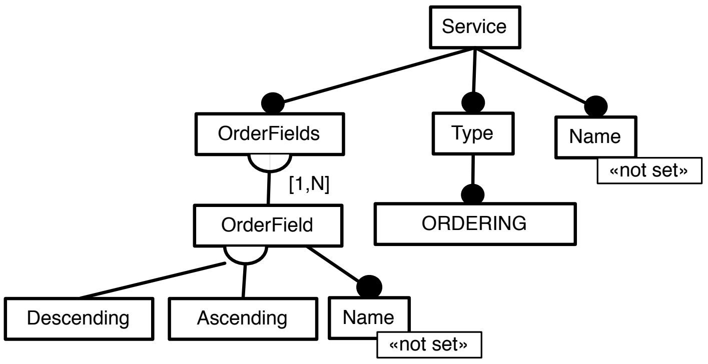


Figure 75: Feature model of the ORDERING service

9.4.7 The **DOCUMENT_LINKS** Service

The DOCUMENT_LINKS Service represents the functionality to start a *link search* based on values of a field in the result documents. In a link search, the search term is not entered by the user. Instead, the values for some fields of documents are represented as links. Following such a link starts a new search for the value of the link. Sowiport, for instance, supports a link search for all publications by an author by clicking on the author's name. The FieldNames feature group describes the fields of result documents that support a link search.

The Action Test Step FOLLOW_LINK_IN_DOCUMENT is assigned to the Service DOCUMENT_LINKS. This Test Step allows the selection of documents and fields within documents to be used for a link search.

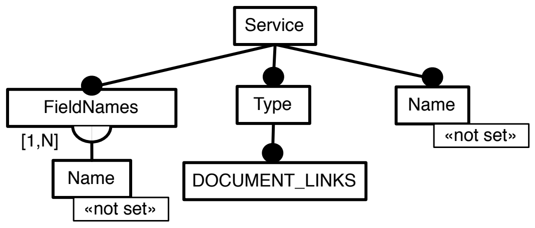


Figure 76: Feature model of the DOCUMENT_LINKS Service

9.4.8 The **PAGINATION** Service

The PAGINATION Service describes the different ways in which a user can browse the different pages of a result list for a search. Figure 77 displays the feature model of the Service The features Absolute and Relative describe whether jumps over sections of the result list are possible

The Test Step GO_TO_PAGE_IN_LIST, a SameStateAction, allows the change of the current page in the list of results.

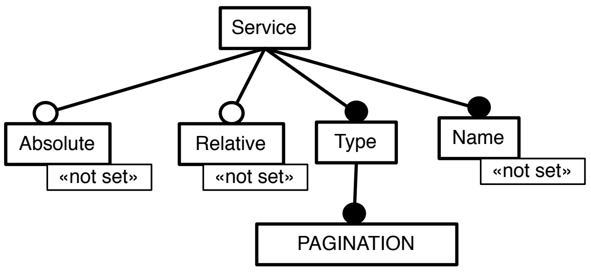


Figure 77: Feature model of the PAGINATION Service

9.4.9 The **NAVIGATOR_LIST** Service

The Service NAVIGATOR_LIST represents the navigators that allow the restriction of a result list to documents that have specific values for certain fields. Figure 78 illustrates the feature model of the Service The CompleteDisplay feature describes whether the navigator returns all values for a facet or, as is the case for the Digital Libraries considered here, only the most frequent values.

The NavigatorField node contains the name of the field that is represented by the navigator. The feature group Values includes the possible values for facets. Three Test Steps are assigned to the Service the SameStateActions SAVE_NAVIGATOR_COUNT, and SAVE_NAVIGATOR_VALUES, and the Action FOLLOW_NAVIGATOR_LINK:

- The Test Step SAVE_NAVIGATOR_COUNT stores the number of results for one value of a facet.

- The Test Step SAVE_NAVIGATOR_VALUES saves all values of a Navigator for a field in a variable.
- The FOLLOW_NAVIGATOR_LINK represents the selection of a navigator value and the restriction of the result list to documents that contain this value for the field. The Test Step is an Action.

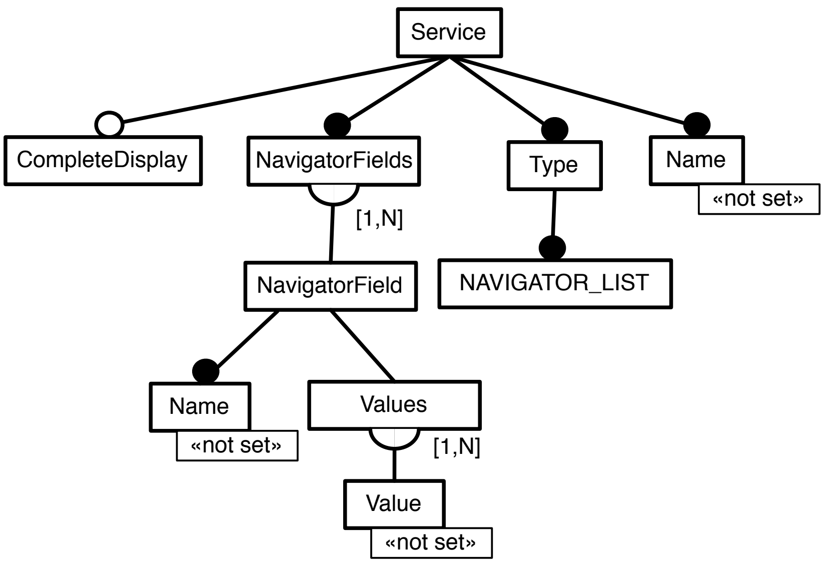


Figure 78: Feature model of the NAVIGATOR_LIST Service

9.5 Chapter Summary

MTCC is applied to a system family of three Digital Libraries and an installation of the Solr search server. These systems are analyzed for their test-relevant Services and the Test Steps that exercise these Services. Services are test-relevant if they represent features or behaviors of the SUT that is relevant for the domain experts to assess the quality of a system. MTCC distinguishes between domain-specific and generic Services. Domain-specific Services represent functionality that is characteristic for the system family to which MTCC is applied. Generic Services represent functionality of MTCC, for example the use of variables to store and verify aspects of the state of the SUT.

Test Steps are test-relevant if they either manipulate the state of a Service or serve to verify that the SUT is in a certain state. The Services for the system family of Digital Libraries considered in this thesis focus on the functionality of these system for searching and displaying results.

Part III

Validation

Chapter 10

Validation Goals and Design

In this chapter, we discuss the role of validation in software engineering. We apply the GQM approach to the validation of MTCC and formulate the goals and questions based on which MTCC is evaluated.

Section 10.1 examines the role of validation in software engineering. The need for a validation is motivated and different approaches to the validation of software are discussed.

Section 10.2 defines the goals of the MTCC validation. We investigate how different types of validation apply to MTCC.

Section 10.3 discusses the application of the GQM approach to the validation of MTCC in detail. Five validation questions are formulated in order to address the general feasibility of the approach as well as its practicability.

10.1 Validation in Software Engineering

A systematic validation is a prerequisite for proving the effectiveness and efficiency of a specific method or technology and for the verification of a hypothesis (Prechelt, 2001; Wohlin *et al.*, 2000). How a validation is conducted is determined by the available resources (Prechelt, 2001).

In this thesis, we employ the GQM approach in the context of a case study to validate both the feasibility of the MTCC and its practicality. The validation of the efficiency of the approach is the subject of future work.

10.1.1 Reasons for Validation

In software engineering, the validation of assumptions about the effectiveness and efficiency of a method or tool is necessary for the same reasons that the validation of hypotheses is necessary in science. Specifically, validation has two goals:

- The validation process assesses the general feasibility, practicality and efficiency of the object of the validation. Such an object might be a tool, a process or an algorithm. Here, the validation provides the basis for the future improvement of the object of validation (Basili, 1996).
- The discipline of software engineering in its entirety benefits from the experience of the conducted validations of past approaches. The experience and knowledge gained serve as the basis for further research.

Despite these facts, research in software engineering often ignores validation. The reasons come in part from the tradition of software engineering. Additionally, a number of practical reasons that are specific to software engineering hinder the conducting of robust validations for the discipline (Perry *et al.*, 2000).

In practice, the optimal validation of an approach or tool will not be possible in most cases since the resources necessary to conduct a perfect validation are prohibitively high (Hannay *et al.*, 2005; Tichy, 2000). We argue therefore that requirements for validation approaches must take the available resources into account. Based on this argument we adopt the three types of validation defined by Freiling (Freiling *et al.*, 2008). They address the feasibility, practicality and efficiency of an approach. The separate validation of the different properties allows the successive evaluation of a system and the evaluation of fundamental properties even with limited resources.

If software engineering is considered in the short term, then the purpose of the discipline is to provide methods that support the creation of high quality systems (see Section 1.1 for the meaning of high quality in this context). Considered for the long term, the objective of software engineering is the creation of theories that support a better understanding of the actors, processes and artifacts relevant to the field. These long-term and short-term views correspond to the roles of the practitioner and the researcher (Basili, 1996). Validation is an essential means to reach both the short term and long term goals of software engineering.

One aspect of validation is the assessment of a product or process with the goal to raise the quality of the system. This aspect is addressed by a number of approaches from manufacturing and quality management, especially methods like Continuous Improvement (Chase *et al.*, 2001). One concrete tool to assess the consequences of changes is the Plan-Do-Check-Act cycle, also known as the Deming Wheel (Chase *et al.*, 2001). The Plan phase of the approach develops hypotheses that are put into practice in the Do phase. The Check phase measures the effects of the changes; the Act phase makes changes to the production process based on the experiences of the proceeding phases and develops new hypotheses.

In contrast to manufacturing, the product of a software development process will usually only be created once (Basili, 1996). However, since software can be developed in increments or iterations, experience from one iteration can be used to make improvements to the process.

Software systems are defined by a large number of quality attributes (see Section 1.1). This leads to a situation where the development of measures that support an assessment of a software system is non-trivial. If the requirements for a system are known and the degree, to which these requirements are met can therefore be measured, empirical methods (Zelkowitz & Wallace, 1998) and the definition of the appropriate metrics (Kan, 2002) are essential means to assess trends in quality, thereby facilitating the design and implementation of higher quality systems.

The validation of assumptions is an essential aspect of the empirical method. Every hypothesis has to be provable or falsifiable by repeated verification (Perry *et al.*, 2000). For software engineering as an engineering science discipline it follows that a hypothesis or the predictions made for a process or software system must be compared with the behavior of the system. Empirical Software Engineering propagates the use of empirical methods in software engineering (Tichy *et al.*, 1995; Perry *et al.*, 2000). Such methods are also the basis of physics (Basili, 1996) or evidence-based medicine (Kitchenham *et al.*, 2004).

10.1.2 Challenges to Validation

A number of reasons exist for the relatively low adoption of empirical methods in software engineering. One aspect that hinders the application of such methods is the problem of comparability and reproducibility of results. Another factor is lack of knowledge about the required methods.

One central cause for the lack of reproducibility and for the difficulty in comparing different results is the high degree to which the software development process depends on the skills and abilities of the participants and their individual approach to software development (Basili, 1996). Beside differences in the participants, other factors also have a negative impact on comparability and reproducibility. These factors include differences in the software development process, the programming languages and tools used and the specifics of the organization in which a software system is developed.

A controlled experiment limits the influence of these external factors (Wohlin *et al.*, 2000), but in order to assess the consequences of an approach in practice, it is desirable to evaluate the approach in the field. An important factor in this context is that the control over the evaluation conditions is significantly lower in the field than in a controlled experiment.

In order to apply empirical methods to software engineering, it is necessary to involve human participants. Such involvement results in a considerable increase in the resources necessary to conduct the validation. Furthermore, the resources increase with the level of control that is needed over the validation setting. As a consequence of the above, only a small fraction of the validations done in software engineering are controlled experiments (Hannay *et al.*, 2005).

While the awareness of the need for empirical evaluation is rising, knowledge of the necessary methods and processes is still lacking in the community (Zelkowitz & Wallace, 1998).

Compromises about the scope and conducting of a validation cannot be avoided in practice. It is important not to accidently affect the validity of the validation. In practice, this means that only those aspects of a system can be validated for which hypotheses can be proven or falsified with the resources available for the validation.

In the following, we discuss the different types of validation and how they can be applied to MTCC with the resources available for this thesis.

10.1.3 Validation Methods

Different validation methods can be distinguished by the degree of control over both the object of the validation and the data acquisition process.

Following the definition by Prechelt (Prechelt, 2001), we distinguish the empirical validation methods experiment, case study, interview and literature survey. Each of these methods exerts less control over the validation setting than the preceding one. The above validation methods are mostly orthogonal to the validated aspect of a method or tool. These aspects are the feasibility, the practicality and the efficiency.

A validation must be conducted based on a systematic approach. One such approach is the Goal-Question-Metric approach (GQM) (Basili *et al.*, 1994). In the GQM, the goal for a tool or method is defined. Based on this goal, questions are formulated that assess whether a goal was reached. These questions in turn are based on metrics.

In the following we discuss the properties of the four methods of validation introduced above and examine their respective advantages and disadvantages.

- In a controlled experiment, the object of the validation and its environment are under the total control of the person conducting the experiment. Such a setting allows the exact control of the dependent variables of an experiment and therefore the precise assessment of the influence of changes on the object of the experiment. In order to limit the influence of the individual characteristics of the participants in the experiment, it is either necessary to conduct an experiment with a sufficiently large control group or to repeat the experiment with different groups (Martens, 2007; Prechelt, 2001).

  In addition to controlled experiments that are conducted in a laboratory, natural experiments are conducted in the field and investigate the object of the validation in its natural environment (Prechelt, 2001).

- A case study examines the concrete application of an approach for one example (Prechelt, 2001). A case study can either be conducted in the field or under laboratory conditions. Compared to an experiment, a case study offers less control over independent variables but also requires less resources than an experiment. The definition of a case study used here differs from the view of a case study as the observation of a project outside of a laboratory (Zelkowitz & Wallace, 1998). While the definition used above does include scenarios where a method or tool is applied with the express purpose to validate this tool, we argue that a case study differs from an *assertion* as defined by Zelkowitz. An assertion validates a method or tool solely based on criteria developed by the developer of the approach; a case study validates an approach based on external criteria.

- In a survey, a group of people is presented with a number of qualitative or quantitative questions. A survey can either be conducted as a personal interview or through a (online) questionnaire (Punter *et al.*, 2003). Regardless of the modus in which it is conducted, a survey does not have any control over the object that is validated but the collected data is under the control of the experimenter.

- Literature studies and meta-analyses examine prior empirical research for the subject area. Literature surveys or literature studies are historical methods (Zelkowitz & Wallace, 1998) that neither allow control over the object of validation nor over the data that is collected. The quality of a literature study is determined by the availability and selection of relevant sources.

Of the four validation methods described above, the controlled experiment is the best in terms of control over the validation process but also needs the most resources. Based on the available resources, the validation of MTCC is therefore conducted as a case study. This approach supports a validation of the essential hypotheses underlying the MTCC approach and serves as a basis for future validations.

Validation Types

We differentiate three types of validation (Freiling *et al.*, 2008; Koziolek, 2008; Becker, 2008) that consider consecutively more complex and harder to validate properties of a method or tool.

A type I validation assesses the feasibility of an approach. This type of validation examines whether the approach can be implemented with limited resources. A successful type I validation is a prerequisite for the conducting of a type II validation. In the case of MTCC, that means the ability to represent the members of a system family and test for these systems as models, to transfer tests between different systems and to generate and execute tests from the models.

A type II validation considers the practicality of a method or tool. An important aspect of the practicality of an approach is the degree to which it can be applied by its intended user. The conducting of a type III validation is only possible when the practicality of an approach is proven in the type II validation. For MTCC, the type II validation considers the question whether domain experts understand the concepts and implementation of the editor and are able to use the editor for test construction.

A type III validation evaluates the efficiency of an approach in terms of the resources necessary for its application and the associated benefits. The type III validation is the most resource intensive of the three validation types. To allow a robust assessment of the efficiency of an approach, a controlled experiment that compares the approach with current practices is a desirable but costly approach.

For this thesis, we conduct type I and type II validations of MTCC. The conducting of a full type III validation is the object of future work — only some aspects concerning the efficiency of MTCC are evaluated.

The Goal Question Metric Approach

One prerequisite for a validation is the exact definition of the aspects of a method or tool that are assessed and the derivation of metrics used for measurement. In this thesis, we use the GQM approach (Basili *et al.*, 1994, 1999) as the basis of our validation. In the GQM approach a validation is based on a goal. Questions are defined that allow the comparison of the goal and the implemented system, the questions in turn are based on metrics:

- The goal defines the relevant properties of the object under validation for the purpose of the validation. A goal is defined by the focus of the validation, the object of the validation, the issue of interest and the viewpoint from which the object is investigated.
- Multiple questions are defined for a goal, each of these questions addresses one aspect of the goal. Questions can be quantitative or qualitative in nature.
- Metrics are derived from the questions. Multiple metrics can be defined for each question. Metrics can be either objective of subjective. Objective metrics only depend on the object of the validation, subjective metrics also depend on the viewpoint of the validation.

GQM is a result-oriented approach (Zubrow, 1998). Metrics are selected for the purpose to answer specific questions about the goal of the system.

10.2 Goals and Questions of the MTCC Validation

The GQM validation of MTCC addresses five questions. The type I validation of MTCC is based on the following three questions that assess: (1) the ability of the MTCC models to represent systems and tests for the considered application domain, (2) the reusability of Test Configurations for multiple systems within the system family, and (3) the ability of MTCC to generate test cases from the models and execute them on the testee. The type II validation consists of one question. This question examines whether domain experts understand the concepts of the MTCC approach and are able to use the prototype implementation of the MTCC editor to construct tests. We only conduct a very limited type III validation in the context of this thesis. The corresponding question compares the resources needed for test execution with MTCC with those necessary for an existing, manual testing regime.

The basic hypothesis of this thesis is that the involvement of domain experts in the testing process in the context of system families is possible on the basis of models and that such a testing approach has a positive influence on the quality of a system.

Because MTCC is not a replacement of existing testing processes and since the effectiveness of MTCC in discovering faults is mostly determined by the knowledge and abilities of the domain experts that are involved in the testing process, the validation of MTCC differs from the validation of testing approaches that put less emphasis on the involvement of domain experts. As a consequence, we do not use the number of identified faults as a metric in the validation. The goal of the MTCC validation is to prove the suitability of the approach for the automated testing of Digital Libraries from the point of view of domain experts.

10.2.1 Hypothesis

The central hypothesis of the MTCC approach is that by constructing suitable models and by representing them in an editor, it is possible to support test construction by domain experts who lack formal modeling or programming skills. Furthermore, we advance the view that the abstract test models thus created can be used to test real systems and that these tests can be reused for different systems if they support the necessary features for test execution. We further argue that testing with MTCC is more efficient than manual test execution or other approaches to test automation, especially in a system family context.

10.2.2 Basic Assumptions of MTCC

One basic assumption of MTCC is that the involvement of domain experts in testing benefits the quality of a system. As discussed in Section 1.2, we argue that the knowledge of domain experts of the requirements for a system can be a form of executable specification (Fowler, 2006; Beck, 2002; Mugridge & Cunningham, 2005b). The usefulness of tests for quality assurance of software (Beizer, 1990; Myers *et al.*, 2004) and the advantages of test automation (Binder, 1999; Graham & Fewster, 2000) are accepted both in the software engineering community and in practice. The involvement of domain experts or customers in the software development and testing process is well established in agile software development approaches (Mugridge & Cunningham, 2005b; Andrea, 2004b; Andersson & Bache, 2004). The goal to involve domain experts in the testing process is also documented outside the agile community (Armbrust *et al.*, 2004).

The concept of system families is long established in software engineering (Parnas, 1976). From the definition of system families in Chapter 4 follows that the members of system families are likely to share requirements and therefore also tests that verify these requirements. It further follows that the reuse of tests and test infrastructure becomes more important.

Since the members of a system family are often not systematically derived from a common set of core assets but are rather results of an opportunistic reuse process, test descriptions for the members of such a system family must be based on an abstract model of the systems, not on their implementation.

We argue that MTCC addresses both the inclusion of domain experts in the testing process and the reuse of test assets for multiple systems.

10.2.3 Goal of the Validation

We define the following GQM goal for the validation of MTCC:

Goal Validate

Focus the suitability for automatic testing of a Digital Library system family of

Object the MTCC approach

Viewpoint from the perspective of domain experts.

We already discussed the goal of validation in Section 10.1. Because the validation is done from the viewpoint of domain experts, the usability of the MTCC editor and more generally the comprehensibility of the MTCC concepts are of special interest to the validation. The object of validation is the implementation of the MTCC approach for Digital Libraries and Solr prepared for this thesis. While a validation of MTCC independent from any concrete implementation would be optimal, the resources available for this thesis only allow one implementation to be evaluated. MTCC is a test automation approach. For the purpose of this thesis, it is applied to the digital library domain. The focus of the GQM goal is therefore on the applicability of MTCC to the Digital Library domain.

10.2.4 Validation of Testing Approaches

MTCC is a testing approach that depends on domain experts for the identification and construction of tests. This results in differences to other approaches to automatic testing. These differences have to be considered in a validation. As we discussed in Section 1.1.3, a number of different, sometimes conflicting, quality attributes can be applied to software for test automation (Kaner, 2003; Graham & Fewster, 2000). If the effectiveness and the efficiency of a testing approach are considered as its most important attributes (Kaner, 2003), then the quality of a test is defined by its ability to find faults with minimal resource usage.

In reality, only considering effectiveness and efficiency for a testing approach is an over-simplification. The individual quality attributes of each system have to be evaluated in the context of its goals and the domain (Do *et al.*, 2005; Ellims *et al.*, 2006) it is applied to. In the MTCC approach, the effectiveness of testing is determined by the ability of the domain experts to identify and design suitable tests for a system. A type III evaluation of MTCC would have to include domain experts and assess the effectiveness of their tests and the efficiency of the MTCC test automation over a long period of time and under realistic conditions. A validation of such scale is not possible in the scope of this thesis.

We focus our validation on the feasibility and practicality of the approach, especially on the ability of MTCC to express the necessary tests for the domain model, instantiate an editor instance for these models that allows the construction of tests configurations, and use the Test Configurations for testing the system.

10.3 GQM Application for MTCC

In the following, we define five questions related to the goal defined in the previous Section 10.2. The questions allow the assessment of the degree to which MTCC supports the goal that we derived from our hypothesis. An overview of all questions, subquestions and metrics is given in Table 10.1.

The feasibility of the GQM approach is addressed by the following three questions:

Q1 Can all relevant tests for the domain experts be represented by MTCC Test Configurations?

Q2 Is it possible to generate test cases from the abstract MTCC test configuration models and to execute these tests on the systems under test?

Q3 Can Test Configurations that were constructed for one system be reused for another system?

Question Q1 considers the basic feasibility for automated testing — MTCC can only automate a test that can be expressed in terms of MTCC models. Question Q2 builds on question Q1. If MTCC can represent a particular test in a model, it does not necessarily mean that the models can be used for the generation of test cases. Question Q3 also depends on question Q1 — given that a test can be expressed for one system, is it possible to reuse the test — unchanged after a transformation — for another system?

The fourth MTCC question considers the type II validation of MTCC. The question validates the practicality of the MTCC approach. We examine whether domain experts understand the concepts of the MTCC approach as the are presented by the MTCC editor.

Q4 Are the MTCC models represented by the editor suited for test construction by domain experts?

Only a limited type III validation is conducted for MTCC. Question 5 examines the time necessary for the manual execution of tests and compares this with the resources necessary to execute tests in MTCC.

Q5 What is the relationship between the resources necessary for the manual execution of tests compared to the resources needed to automate and execute the same tests with MTCC?

10.3.1 Systems and Test Sets

For the following discussion of the MTCC validation plan, it is necessary to define several sets, variables and functions.

Our evaluation considers the system family S of four systems $S = \{s_1, s_2, s_3, s_4\}$. The systems are discussed in detail in Chapter 9.

s_1 The search of the Sowiport portal.

s_2 The interdisciplinary portal infoconnex

s_3 The portal for international relations and area studies of the German Institute of International and Security Affairs

s_4 The search server Apache Solr

In order to assess the capability of MTCC to represent relevant tests for a domain, we use a number of different test sets. A test set is a collection of tests or testable faults for one of the members of the system family S. Section 10.3.2 discusses test sets in detail.

TS_1 This test set includes a number of tests that we defined as a baseline for the MTCC approach before the approach was implemented.

TS_2 This test set includes a number of test scenarios for quality attributes from the Information Retrieval domain that can be applied to the system s_4.

TS_{3a} A set of tests that a domain experts defined in a simple text format. This test set addresses the system s_1.

TS_{3b} A set of tests that was specified in the same text format as the tests in the test set TS_{3a}. In contrast to TS_{3a}, the test in the test set address system s_3 and were defined by domain experts for this portal.

TS_{4a} This test set includes a number of tests that are defined in a test plan for the manual testing of system s_2.

TS_{4b} This test set includes tests for system s_2. Like the tests in test set TS_{4a}, the tests were originally defined in a test plan for manual execution. Unlike the tests in TS_{4a}, the tests only address the search functionality of s_2.

T_5 This test set does not include tests as such but rather faults and TODO items that were reported for system s_1. We treat this faults as test scenarios if a regression test could be derived from the fault .

Each test set contains a different number of tests. The number of tests for each test set is stored in the variables $t_{x,y} x \in \{1, 2, 3a, 3b, 4a, 4b, 5\}, x \in \mathbb{N}$

10.3.2 Q1: Capability to Represent Tests

Question Q1 of the MTCC validation is: Can all tests that are relevant for the domain experts of the current system family be expressed by MTCC models? In order to answer this question we test to which degree the tests defined in the different test sets introduced in Section 10.3.1 can be represented by the MTCC models for the respective systems.

For every test $t \in T$ with $T \in T_{gesamt}, T_{gesamt} = \{TS_1, TS_2, TS_{3a}, TS_{3b}, TS_{4a}, TS_{4b}, TS_5\}$ the degree to which the test can be represented in MTCC is represented by the value gr, gr is defined as follows.

$$gr \in G, G = \{\mathsf{g}_{\text{yes}}, \mathsf{g}_{\text{can}}, \mathsf{g}_{\text{no}}, \mathsf{g}_{\text{extern}}\} \tag{10.1}$$

g_{yes} The test can be completely represented by the MTCC models used for the validation. This means that the Application Feature Model, the Application State Model, and the Domain Test Model are complete for the purpose of the test.

g_{can} The test cannot be represented with the models used for the validation, but the models could be extended to support the test if the necessary Services, Service instances or Test Steps are added to the Domain Feature Model, Application Feature Model, or Application State Model respectively.

g_{no} The test can neither be represented by the models used for the validation nor is the addition of Services, Service instances or Test Steps possible that would allow the representation of the test.

$\mathsf{g}_{\text{extern}}$ The test is not within the scope of MTCC. Examples for such tests are tests that are specific to the graphical user interface of a testee.

We consider a test t as supported by the MTCC approach if $gr(t) = \mathsf{g}_{\text{yes}}$. When a test set $TS \in TS_{gesamt}$ is considered in its entirety, then the support of MTCC for this test set is defined as

$$gr(T) = \frac{|\{t_1 : t_1 \in T \wedge g(t_1) = \mathsf{g}_{\text{yes}}\}| * 100}{|\{t_2 : t_2 \in T \wedge g(t_2) \in \{\mathsf{g}_{\text{yes}}, \mathsf{g}_{\text{can}} \ \mathsf{g}_{\text{no}}\}|} \tag{10.2}$$

Q1.1 Support of MTCC for Predefined Test Scenarios

The test set TS_1 has been defined prior to the design and implementation of the MTCC approach. The test set consists of 16 tests for system s_1. The test set was defined to serve as the basic for the development and validation of the MTCC models and was the result of a first informal analysis of the testable functionality of the testee. Each of the 16 tests covers a different functionality of the system and can therefore be considered as a representative of a test class. We define a test class as a trace through the functionality of the system for which many instances can exist. Searching for a term and comparing the number of result documents with an expectation would be a test class for which search for concrete terms would be instances or tests.

Each test in TS_1 is described as a test scenario in a structured test format that consists of actions taken on the interface of the testee and the expectation or assertions that must be true. An example of such a test description is given below. We translated it from the original German:

```
Test 4(bs) Test of the SOLIS database filter
```

```
This tests verifies the filters for databases for the extended search in Solis

  1. Action The filter for databases is selected in the Advanced Search
  2. Action The Check Box all is de-selected
  3. Action The Check Box Solis is selected
  4. Action The term 'social' is entered in the text field of the search form
  5. Assertion More than 17.000 documents are found.
  6. Assertion The person navigator contains 30 hits for the value Weber, Max
```

Every test scenario has a name and a short description. The test scenario is represented as a sequence of interactions that take place between the user and the system under test. Since user and system interact via the user interface, the actions refer to elements of the GUI. Actions and assertions in the test scenario correspond to the two action types and assertions in MTCC.

We choose the above format for the representation of test cases since it is precise enough to serve as a basis for the manual execution of tests and test modeling while it is not bound to a particular modeling language.

We argue that test set TS_1 is relevant for the validation of MTCC since it covers a wide range of functionality for the system s_1 and is independent of the MTCC approach. Metric M 1.1.1 describes $g(TS_1)$, the degree of support for the tests of the test set TS_1.

Q1.2 Capability to Express Information Retrieval Tests

The test set TS_2 consists of tests that assess the results of applying cross-concordances (Mayr & Petras, 2008) to the retrieval process. The test set holds the 7 tests below that verify various expectations about the use of cross-concordances.

$t_{2,1}$ Are more documents found for a query that was expanded with cross-concordances than for queries that were not expanded?

$t_{2,2}$ Do the 10 or 20 most relevant documents for a query change when cross-concordances are used?

$t_{2,3}$ Does the ranking of the most relevant 10 or 20 documents change?

$t_{2,4}$ Do cross-concordances result in documents form more databases to be included in the result set than would be the case without cross-concordances?

$t_{2,5}$ How do cross-concordances change the number of databases in the most relevant 10, 20 or 100 documents?

$t_{2,6}$ How does the recall change for 10, 20, and 100 documents when cross-concordances are used?

$t_{2,7}$ How do cross-concordances influence the precision for the first 10, 20, and 100 documents of a result list?

Each of the 7 tests above would be tested with a initial query, for example a query derived from a CLEF (Petras *et al.*, 2007) topic.

Question Q1.2 assesses of the ability of MTCC to express tests for the effectiveness of Information Retrieval systems. Since the ability of a system to return documents that are relevant to an information need cannot be completely automated, MTCC must support tests that calculate recall and precision passed on pooled relevance assessments and must be able to test hypotheses like those expressed in the seven tests above.

Metric M1.2.1 describes the degree to which the test set $g(TS_2)$ can be expressed with MTCC

Q1.3 Capability to Express Tests Defined by Domain Experts

In the preparation for this thesis, domain experts affiliated with the Digital Libraries s_1 and s_3 were asked to specify tests for their respective system in the format described in Section 10.3.2. The domain experts were informed that the tests were collected in order to use them in a test automation approach, but did not know about MTCC. Examples for the specified tests can be found in Section A and Section A in the appendix.

Compared to the tests in test set TS_1, the tests in the test sets TS_{3a} and TS_{3b} were not defined for the evaluation of the MTCC approach but represent actual requirements for either system s_1 or s_3. Since the tests in the test sets are defined by domain experts, we argue that they are likely to represent realistic requirements to a test automation system.

The metrics M1.3.1 and M1.3.2 describe $g(TS_{3a})$ and $TS_{3b})$, the degree to which MTCC supports the expression of tests for the respective test sets.

Q1.4 Representation of Existing Manual Tests

The test sets TS_{4a} and TS_{4b} consist of manual tests for system s_2 that execute monthly on the system. Section A in the appendix gives an example of the tests in the test sets. A particularity of the tests in these test sets is that they do not define the expected behavior of the system under test, only the actions to be taken on the system.

Test set TS_{4a} consists of tests for different aspects of system s_2, some of which address details of the GUI and are therefore not in the scope of MTCC. All tests in the test set TS_{4b} address the search functionality of the system. Compared to the tests described in TS_1, TS_{4b} does not cover much of the functionality of the system under test but instead tests the search functionality of system s_1 in detail. The fact that all tests in TS_{4b} are instances of only a few test classes is relevant for the validation since it is likely that MTCC either supports all tests of a test class or no tests from this test class.

We argue that the test set TS_{4b} is relevant for the validation because it represents real tests taken from practice.

Metric M1.4.2 describes $g(TS_{4b})$, the support of MTCC for the tests in test set TS_{4b}, metric M1.4.1 considers $g(TS_{4a})$.

Q1.5 Support for Tests derived from Identified Faults

Test set TS_5 does not include tests as such. Rather, the test set is an export of all entries of the issue tracking system for system s_2. Some of the entries in the issue tracker are feature requests and many of the remaining faults in the issue tracker address aspects of the system that are not within the scope of MTCC. These entries are assigned to the category $\mathsf{g}_{\text{extern}}$. The subset of entries in the issue tracker that are in the scope of MTCC are faults that where found by domain experts in ad-hoc testing. Since it is desirable to automate such ad-hoc tests with MTCC, we derive tests from the identified faults and assess if the tests can be expressed with MTCC.

Metric M1.5.1 describes $g(TS_5)$, the degree to which MTCC supports these tests.

10.3.3 Q2: Executability of MTCC Test Configurations

The second question of the type I validation examines the executability of the MTCC Test Configuration models. We consider a Test Configuration instance executable when it can be used — either in its original form or after a transformation — to exercise a testee. Because Test Configuration instances serve as the basis for test case generation, a model is executable if a test case can be created from the model.

The question Q2 has a quantitative aspect. It is not only of interest if test code can be generated from the models but also how much resources are necessary to establish a test code generation infrastructure and to use this infrastructure to generate test cases.

We call the infrastructure needed for the generation of test cases and for the execution of these test cases the execution system. The execution system consists of the Configuration Reader, the Test Generator and the Test Adapter.

With the consideration of the resources necessary for test code generation and execution, question Q2 seems to address the efficiency of MTCC and thereby the type III validation. We argue that Q2 is part of the type I validation because the feasibility of MTCC is only given if MTCC can be implemented with the resources available for test automation in a given project.

Question Q2 consists of 4 subquestions:

- To what degree is the complete implementation of the execution system possible for a given testee? In order to answer this question, we implement an execution system for the tests of test sets T_1 and T_{3a} on system s_1. The question examines how many resources are necessary for the implementation of an execution system that covers most of the functionality of a system.

- Is the implementation of an execution system feasible for all members of a system family? In order to examine this question we implement a limited execution system for all members of the system family. This execution system only supports the execution of the single test class t_{a1}.

- Is the implementation of an execution system for different test runners possible? We implement an execution system for the test class t_{a1} on system s_1 with the test runner tr_2. This question investigates whether the MTCC test models are specific to one particular test runner.

- Can the test cases generated by the MTCC execution system be integrated into an existing testing process?

In order to investigate the above questions, we implement the needed execution systems for the MTCC prototype. We document the lines of code for each implementation as a simple metric for the resources necessary to implement the system. As discussed in Section 11.7.1, we consider lines of code an imperfect but usable metric.

Q2.1 Near-complete Implementation for a Testee

This question examines the resources necessary for a nearly complete implementation of the execution system of MTCC for one system. We call the implementation near-complete since it implements all tests that are defined for a system, not all functionality of the system. We implement the test sets T_1 and T_{3a} for system s_1

Metric M2.1.1 describes the percentage of all tests from T_1 and T_{3a} that can be supported by the implemented execution system. Metric M2.1.1 investigates the lines of code that are used for the implementation of the parts of the execution system.

Q2.2 Executability on the Members of a System Family

This subquestion addresses the executability of MTCC Test Configurations on the different members of the considered system family. These questions aims to prove that MTCC Test Configurations can be executed on all members of a given system family. In order to answer the question, an execution system for the test t_{a1} is implemented for all system in S. t_{a1} includes the following steps:

```
1. Action A search using multiple fields in an advanced search form is started.
2. Action The number of result documents is stored in a variable.
3. Assertion The number in the variable is compared to some expected value.
```

Test t_{a1} has been chosen for the validation since all systems in the system family S support the functionality required by the test. We restrict the validation of this question to a single test because of the available resource for this validation and since the test class described by test t_{a1} is the most frequently used for this system family as can be seen from T_{4a}.

Metric M2.2.1 examines whether the execution system for t_{a1} can be implemented for all members of the system family. Metric M2.2.2 describes the lines of code necessary for these implementations.

Q2.3: Executability with Different Test Runners

Question Q2.3 addresses the executability of MTCC Test Configurations with different test runners. Two test runners are used for the validation.

The test framework Selenium (tr_1) is used for testing a system by automating a web browser. Twill (tr_2) is a HTTP Protocol Driver (Brown *et al.*, 2007) used for testing web applications.

A test execution system is implemented for test t_{a1} on the system s_1 and for both test runners. Metric M2.3.1 describes whether the implementation of the execution system succeeds for the two test runners. Metric M2.3.2 measures the lines of code necessary for the implementation.

Q2.4: Compatibility with Existing Testing Regimes

Question Q2.4 addresses the compatibility of the test cases generated for the questions Q2.1, Q2.2, and Q2.3 with existing testing tools and the degree to which MTCC test cases can be integrated in existing testing regimes. For this question we examine the infrastructure needed in order to execute MTCC tests. This is a qualitative question that is not based on any metrics.

10.3.4 Q3: Reuse of Test Configurations

Question Q3, the third question of the type I validation, examines to what extent Test Configuration instances can be reused for different systems. Specifically, we investigate whether a Test Configuration instance that is specific to one Application Test Model can be transfered to the Application Test Model of another system. If such a transfer of test models is possible and if execution systems exist for all testees, then tests are reusable for different systems.

Question Q3 consists of three sub questions each of which corresponds to one member of the system family S for which tests can be reused. For each of the systems and each test in the test sets T_1 and T_3a that is intellectually reusable we check whether a Test Configuration created for system $s1$ can be reused for the target system.

We call a test reusable by hand if a domain expert with knowledge of the original test can execute the scenario described by the test on another system, that is, if all features required by the test are present for both systems and the requirements expressed by the test can therefore be verified on both systems.

Q3.1: Reuse for System Variants

Question Q3.1 considers the reuse of Test Configuration instances for variants of a testee. Systems s_1 and s_3 were developed in parallel and are similar regarding their implementation, we therefore call these systems variants of each other.

For every test $t, t \in T_1$ the function $x = trans(s_1, s_3, t), x \in \{\mathsf{g}_{\text{yes}}\ , \mathsf{g}_{\text{no}}\ , \mathsf{g}_{\text{can}}\ , \mathsf{g}_{\text{false}}\ \}$ describes whether a Test Configuration can be transfered or not.

The value g_{no} is used for tests that are not reusable by hand for different systems. Tests with the value g_{can} are reusable by hand but cannot be reused by MTCC. The value g_{yes} is used for tests that are reusable and whose semantics are correctly represented for the systems for which the test is reused. The value $\mathsf{g}_{\text{false}}$ is used for tests that can be automatically transfered but whose semantics are changed by the transfer process — the test for the target system is not a correct representation of the test on the original system.

Metric M3.1.1 describes the percentage of the reusable tests that can be reused in MTCC.

$$M3.1.1 = \frac{|\{t_1 : t_1 \in T_1 \wedge trans(s_1, s_3, t_1) = \mathsf{g}_{\text{yes}}\ \}| * 100}{|\{t_2 : t_2 \in T_1 \wedge trans(s_1, s_3, t_2) \in \{\mathsf{g}_{\text{yes}}\ , \mathsf{g}_{\text{can}}\ \mathsf{g}_{\text{false}}\ \}|}. \tag{10.3}$$

Q3.2: Reuse for Systems Versions

Question Q3.2 considers the reuse of test models for a system and another system that is the precursor of the first system. System s_2 can be considered a precursor of system s_1 since s_2 was developed earlier and both systems offer the same basic functionality, even though system s_2 includes some functionality that is not part of s_1.

Metric M3.2.1 for question Q3.2 is analogous to metric M3.1.1 but considers the transfer of tests between the systems s_1 and s_2.

Q3.3: Reuse for System Layers

Question Q3.3 examines the reuse of Test Configurations that were constructed for a system and are transfered to a subsystem. Specifically, we test if tests for system s_1 can be reused for system s_4. Since system s_4 is a search server not a Digital Library, these tests assess the potential for reuse between different layers of a system.

Metric M3.3.1 is analogous to metrics M3.2.1 and M3.1.1, but considers system s_4.

10.3.5 Q4: Practicality and Understandability of MTCC

Question Q4 considers the practicality of MTCC. More precisely, it investigates whether domain experts understand the approach and can use the editor to build Test Configurations from given test scenarios. Question Q4 is part of the type II validation.

The question primarily investigates whether the MTCC is understandable by domain experts and can be applied by them. While the usability of the editor implementation used for the validation contributes to the practicality, it is not the focus of the validation.

We investigate the practicality of MTCC based on two tasks, the adaptation of existing Test Configurations and the construction of new Test Configurations from textual specification. We consider quantitative as well as qualitative aspects. Qualitative aspects include feedback given by the domain experts about the usability of the editor and the understandability of the MTCC approach as well as the most frequent faults and problems that occur in using the editor. The quantitative aspects measure if the domain experts were able to express the scenarios without help, with some help or with no help at all.

We use test sets with seven tests each for the validation of the practicality of the approach. One type serves as the basis of the construction of new tests, the other consists of tests used to validate the adaptation of existing tests.

The test sets TV_{2a} and TV_{2b} each contain seven tests that were randomly selected from the test sets T_1 and T_{3a}. The test sets hold test scenarios that are used to validate the construction of new tests by domain experts.

$$TV_{2a} = \{ts_{1,10}, ts_{1,11}, ts_{3a,0}, ts_{3a,10}, ts_{3a,15}, ts_{3a,28}, ts_{3a,5}\} \tag{10.4}$$

$$TV_{2b} = \{ts_{3a,11}, ts_{3a,12}, ts_{3a,14}, ts_{3a,17}, ts_{3a,21}, ts_{3a,23}, ts_{3a,8}\} \tag{10.5}$$

The test set TV_1 consists of five tests randomly selected from the test set TS_{4b}. The tests in the test set are all based on the same test class: The submission of a query and the calculation of the results of a query.

$$TV_1 = \{t_{4b,88}, t_{4b,89}, t_{4b,90}, t_{4b,103}, t_{4b,104}\} \tag{10.6}$$

Four domain experts were involved in the validation, we denote them by the variable names $de_{1...4}$.

Q4.1: Adaptation of Tests

Question Q4.1 investigates the adaptation of Test Configurations from test set TS_{4B} by domain experts. Both the time needed for the adaptation of the tests and potential problems during the adaptation are considered. We argue that the adaptation of existing tests and the creation of new instances for already existing test classes is a frequent activity in testing and that tests from the test set TS_{4B} are well suited to represent this activity.

Metric M4.1.1 measures the average time needed by the domain experts to construct the test $t_{4b,88}$ and to adapt this test to the scenarios described in the remaining entries of the test set.

Q4.2: Construction of Tests

Question Q4.2 investigates the degree to which domain experts can use the MTCC editor to construct tests from given test scenarios. The question considers 14 tests, subdivided into two subsets of 7 tests each. In order to validate the practicality of MTCC, four domain experts evaluated the MTCC editor in 2 sessions each. During the first session, the tests from test set TS_{2a} were investigated. In the second session, the domain experts were given the task to model the tests from TS_{2b}. Details are given in Section 11.6.1.

As a quantitative measure we determine the degree to which the domain experts can express the test with the MTCC editor. We differentiate three degrees of support that can be given to a domain experts during test construction. If a test is modeled without any help the test falls into the category *nohelp*.

If some help is needed, either with respect to the MTCC approach or the MTCC editor, the respective tests falls into the category *help*. Examples for the help given to the domain expert are answers to concrete questions about the user interface or the meaning of the parameters for specific Test Steps. Generally, all information that could also be learned from the manual is considered limited help. If more help is needed and if it is likely that the construction of the test would fail even with the use of the manual, the test is assigned to the category $fullhelp$.

Details about the categories can be found in Section 11.6.1 Metric M4.2.1 describes the distribution of the tests TV_{2a} and TV_{2b} on the categories *nohelp*, *help*, and $fullhelp$ for all four participating domain experts.

10.3.6 Q5 Efficiency of MTCC

Question Q5 considers the efficiency of MTCC in comparison to manual testing. The question is an entry point to a further type III validation at a later date. We consider manual testing since it is the current approach used for testing in validating organization. A comparison of MTCC to other approaches of automated testing is the subject of later work.

The purpose of question Q5 is to assess the resources necessary for manual testing and testing with MTCC. As discussed in Section 1.1.4, four phases of testing can be differentiated. For question Q5 we only consider the third phase of testing, the execution of tests.

We compare the time needed for the manual execution of the tests in the test set TS_{4b} on system s_2 with the resources necessary to implement the execution system for the tests and the system. We argue that the comparison of manual testing with implementation costs for the test generation and execution system provides a better measure than the comparison with the resources necessary for test case generation and test execution since the automatic test execution does not depend on human involvement but does depend on the potentially resource-intensive implementation of the execution system.

We use two measures to discuss the efficiency of MTCC, (1) the lines of program code necessary to implement the execution system and (2) the time needed for the manual execution of the tests in test set TS_{4b}. The manual execution of tests is done biweekly.

10.4 Chapter Summary

Software Engineering is an empirical science, MTCC must therefore be evaluated in a systematic manner. This thesis applies the Goal Question Metric (GQM) approach to validate the feasibility and practicality of MTCC. The MTCC validation is conducted as a case study. The validation of MTCC consists of a type I and type II validation. The type I validation addresses the basic feasibility of the approach, the type II validation investigates the practicability the the approach. The type III validation compares the efficiency of MTCC with the efficiency of other approaches.

The type I validation of MTCC uses three questions: The first question assesses the capability of MTCC to represent tests that are relevant to domain experts. The second question investigates how abstract Test Configurations can be used the exercise concrete systems. The third question of the type I validation investigates the effectiveness of the methods employed by MTCC to reuse tests. The type II validation consists of one question that investigates the understandability of the MTCC approach to domain experts. The type III validation investigates the necessary resources for testing with MTCC with the resources necessary for manual testing.

Question 1	Is MTCC capable to represent all relevant tests?
Subquestion 1.1	Does MTCC support predefined test scenarios?
Metric 1.1.1	Degree of support for test set TS_1
Subquestion 1.2	Is MTCC capable to express Information Retrieval tests?
Metric 1.2.1	Degree of support for test set TS_2
Subquestion 1.3	Can MTCC express tests defined by domain experts?
Metric 1.3.1	Degree of support for test set TS_{3a}
Metric 1.3.2	Degree of support for test set TS_{3b}
Subquestion 1.4	Can MTCC represent existing manual tests?
Metric 1.4.1	Degree of support for test set TS_{4a}
Metric 1.4.2	Degree of support for test set TS_{4b}
Subquestion 1.5	Does MTCC support tests derived from identified faults?
Metric 1.5.1	Degree of support for test set TS_5
Question 2	Can MTCC Test Configurations be executed as automated tests?
Subquestion 2.1	Is a near-complete implementation of the execution system for one testee possible?
Metric 2.1.1	Percentage of executable Test Steps for TS_1 and TS_{3a}
Metric 2.1.2	Lines of code needed to make TS_1 and TS_{3a} executable
Subquestion 2.2	Can MTCC Test Configurations be executed on all members of a system family?
Metric 2.2.1	Percentage of executable Test Steps for t_{a1} for the system family S
Metric 2.2.2	Lines of code needed to make t_{a1} executable on all systems in S
Subquestion 2.3	Can MTCC Test Configuration be executed with different test runners?
Metric 2.3.1	Percentage of executable Test Steps for test t_{a1} on system s_1 and the test runners tr_1 and tr_2
Metric 2.3.2	Lines of code needed to make t_{a1} executable on system s_1 and the test runners tr_1 and tr_2
Subquestion 2.4	Is MTCC compatible with existing testing regimes?
Question 3	Can MTCC Test Configurations be reused?
Subquestion 3.1	Are Test Configurations reusable for system variants?
Metric 3.1.1	Percentage of the reusable tests for system s_3 supported in MTCC
Subquestion 3.2	Are Test Configurations reusable for systems versions?
Metric 3.2.1	Percentage of the reusable tests for system s_2 supported in MTCC
Subquestion 3.3	Are Test Configurations reusable for system layers?
Metric 3.3.1	Percentage of the reusable tests for system s_4 supported in MTCC
Question 4	Is MTCC practical and understandable by domain experts?
Subquestion 4.1	Can domain experts model and adapt simple tests?
Metric 4.1.1	Time needed by the domain experts to model $t_{4b,88}$ and adapt the test to TV_1
Subquestion 4.2	Can domain experts construct new tests?
Metric 4.2.1	Help needed by the domain experts to model TV_{2a} and TV_{2b}
Question 5	Is MTCC efficient compared to manual testing?

Table 10.1: GQM questions and metrics used in the MTCC validation

Chapter 11

Results of the Validation

The validation of MTCC was conducted between March and May 2008 at the GESIS [1] in Bonn. The participants of the validation were employees of the GESIS who were either involved in the specification of the systems, in the design of tests for one of the systems or in the manual testing of the systems. The validation was conducted on systems $s_{1...4}$. With the exception of system s_4, all systems were original systems used in production. The usability of MTCC was validated with a prototype of the MTCC editor.

11.1 Participants in the MTCC Validation

Four domain experts were participating in the MTCC evaluation:

p_1 This domain expert had experience in test design for system s_1 and took part in the manual testing of this system.

p_2 This domain expert was a student employee who undertook the execution of the manual tests defined in the test sets T_{4a} and T_{4b} for system s_2 in the years 2006 and 2007.

p_3 Domain expert p_3 was the student employee who succeeded domain expert p_2 in the execution of the tests for system s_2 in 2008.

p_4 This domain expert defined the test set T_2 and was familiar with the different systems within the system family.

11.2 The Prototype of the MTCC Editor

The implementation of MTCC was done according to the principles and techniques discussed in Part II of this thesis. A class library for the construction and manipulation of feature models implemented for this thesis was used to instantiate the Domain Test Model and the Application Feature Model and the Application State Model for every SUT.

Figure 79 illustrates the overall architecture of the MTCC prototype. We distinguish five logical packages of functionality: The Domain & System package includes class libraries that support the specification of domain-level and system-level models. These class libraries build on a Feature Modeling Layer that provides an implementation of feature models with cardinalities. The feature models in turn are implemented as nodes in a Document Object Model. The User Interface package includes the Editor that is responsible for the overall lock and feel of the GUI used for test modeling and the GuiBuilder classes that represent individual Test Step instances. The Model Composer is responsible for the instantiation of the Application

[1] German Social Science Infrastructure Services

Test Model and the Reuse System that rewrites Test Configurations for different system models are also part of the Editor package.

The Generator package includes the Code Generator that transforms MTCC Test Configurations into executable test cases. Templates determine the overall structure of the test code. The Test Parameter Writer rewrites the Test Configurations into a simpler representation better suited for code generation.

The MTCC Runtime and the External Runtime packages are responsible for the execution of the generated Test Cases. The Test Adapter provides a platform for test execution the decouples from the implementation details of the Test Runner and the SUT. The Test Runner executes the Test Cases. The Test Runner is often supported by Support Libraries for test execution, this thesis uses Selenium or twill. The Language Runtime used for test execution can be different from the language in which MTCC is implemented.

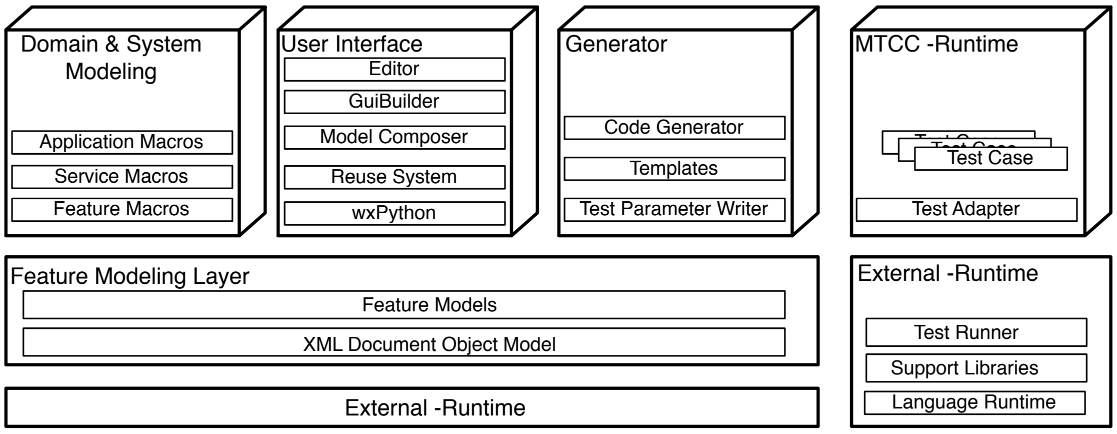


Figure 79: Logical architecture of the MTCC prototype

The prototype implementation of MTCC used for the validation was done in Python[2]. XML was used as a serialization format for the feature models and for the state machines used in the validation. wxPython[3] was used to implement the GUI. The implementation comprises 6600 Lines of Python.

Figure 80 displays a screenshot of the implementation of the editor as it was used for the validation.

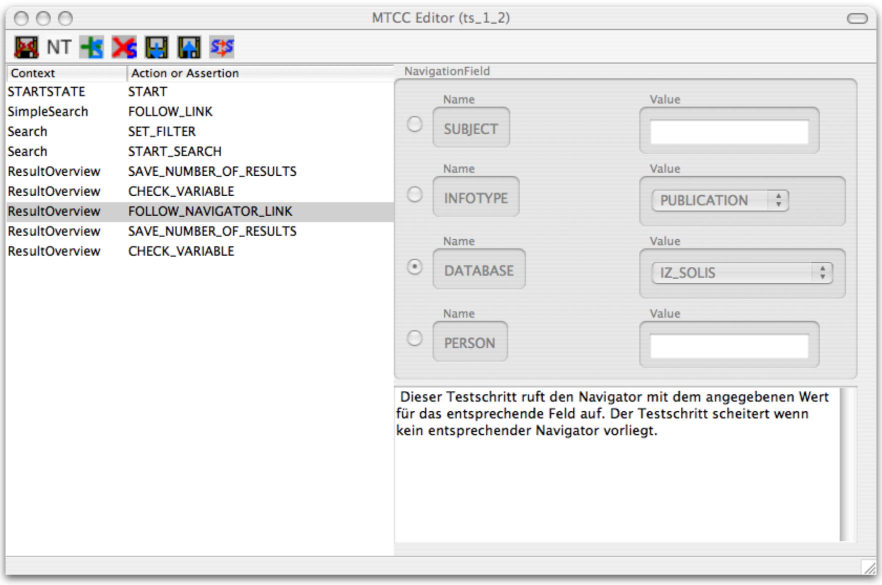
As described in Section 8.1, the left part of the GUI displays the Test Steps that the current test consists of; the right part of the editor represents the currently selected Test Step and allows its parameterization.

The validation prototype was tested on Mac OS X and MS Windows systems.

The dialog used for the selection of Test Step instances is displayed in Figure 81.

The prototype implements the techniques for test reuse discussed in Section 8.3. Figure 82 displayed a screen shot of the dialog used by MTCC to allow the selection of Test Step instances when a test is transfered between different systems. Because only one sequence of Test Step instances is available for the tests and systems used in Figure 82, the dialog is deactivated.

[2]http://python.org
[3]http://wxpython.org



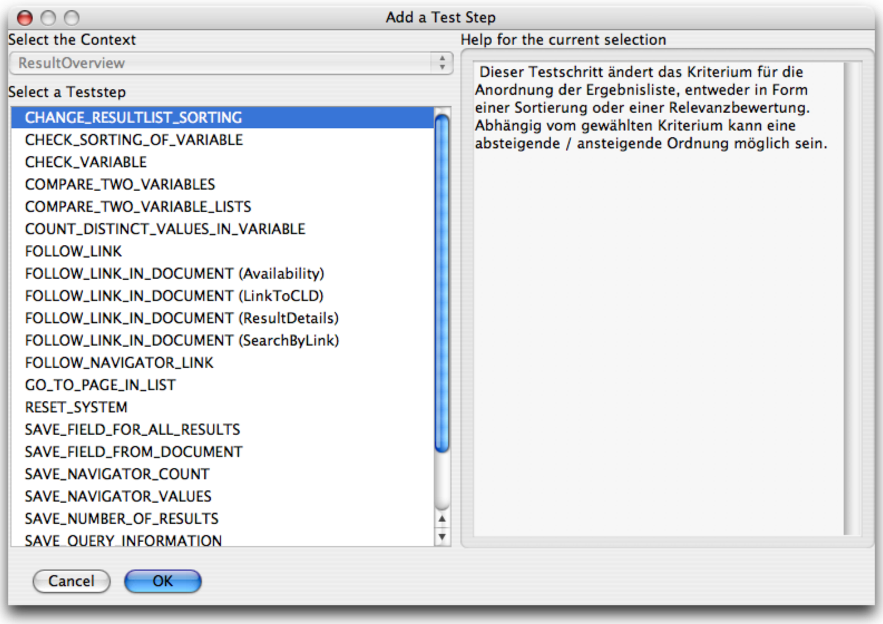
Context	Action or Assertion
STARTSTATE	START
SimpleSearch	FOLLOW_LINK
Search	SET_FILTER
Search	START_SEARCH
ResultOverview	SAVE_NUMBER_OF_RESULTS
ResultOverview	CHECK_VARIABLE
ResultOverview	FOLLOW_NAVIGATOR_LINK
ResultOverview	SAVE_NUMBER_OF_RESULTS
ResultOverview	CHECK_VARIABLE

NavigationField

	Name	Value
○	SUBJECT	
○	INFOTYPE	PUBLICATION
●	DATABASE	IZ_SOLIS
○	PERSON	

Dieser Testschritt ruft den Navigator mit dem angegebenen Wert für das entsprechende Feld auf. Der Testschritt scheitert wenn kein entsprechender Navigator vorliegt.

Figure 80: Screen shot of the editor prototype used for the validation



Select the Context
ResultOverview

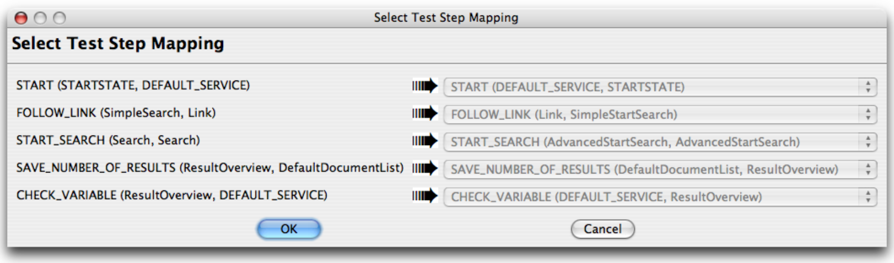
Select a Teststep
CHANGE_RESULTLIST_SORTING
CHECK_SORTING_OF_VARIABLE
CHECK_VARIABLE
COMPARE_TWO_VARIABLES
COMPARE_TWO_VARIABLE_LISTS
COUNT_DISTINCT_VALUES_IN_VARIABLE
FOLLOW_LINK
FOLLOW_LINK_IN_DOCUMENT (Availability)
FOLLOW_LINK_IN_DOCUMENT (LinkToCLD)
FOLLOW_LINK_IN_DOCUMENT (ResultDetails)
FOLLOW_LINK_IN_DOCUMENT (SearchByLink)
FOLLOW_NAVIGATOR_LINK
GO_TO_PAGE_IN_LIST
RESET_SYSTEM
SAVE_FIELD_FOR_ALL_RESULTS
SAVE_FIELD_FROM_DOCUMENT
SAVE_NAVIGATOR_COUNT
SAVE_NAVIGATOR_VALUES
SAVE_NUMBER_OF_RESULTS
SAVE_QUERY_INFORMATION

Help for the current selection
Dieser Testschritt ändert das Kriterium für die Anordnung der Ergebnisliste, entweder in Form einer Sortierung oder einer Relevanzbewertung. Abhängig vom gewählten Kriterium kann eine absteigende / ansteigende Ordnung möglich sein.

Figure 81: Selection of a Test Step instance in the MTCC prototype

11.3 Considered Systems

The validation considered the systems introduced in Section 10.3.1 on page 146. Every system was represented with a specific Application State Model and Application Feature Model; a common Domain Test Model was used for all models. All models were modeled to the extent which was needed for the validation — features that are not exercised by any test set were not included in the models We tested the executability of the systems based on their implementation as of May 2008, no changes for the purpose of the validation were made to the systems.



Select Test Step Mapping

START (STARTSTATE, DEFAULT_SERVICE)	START (DEFAULT_SERVICE, STARTSTATE)
FOLLOW_LINK (SimpleSearch, Link)	FOLLOW_LINK (Link, SimpleStartSearch)
START_SEARCH (Search, Search)	START_SEARCH (AdvancedStartSearch, AdvancedStartSearch)
SAVE_NUMBER_OF_RESULTS (ResultOverview, DefaultDocumentList)	SAVE_NUMBER_OF_RESULTS (DefaultDocumentList, ResultOverview)
CHECK_VARIABLE (ResultOverview, DEFAULT_SERVICE)	CHECK_VARIABLE (DEFAULT_SERVICE, ResultOverview)

Figure 82: Transfer of a Test Configuration to another system in the MTCC editor

11.4 Capability to Represent Tests

In order to validate MTCC's capability to represent tests for the application domain of Digital Libraries, we assessed for each entry t of every test set with $t \in T, T \in \{T_1, T_2, T_{3a}, T_{3b}, T_{4a}, T_{4b}, T_5\}$ into which category of support $g \in \{\mathsf{g}_{\text{yes}}, \mathsf{g}_{\text{no}}, \mathsf{g}_{\text{can}}, \mathsf{g}_{\text{extern}}\}$ the test falls. The total values of the different test sets and categories are displayed in Table 11.1. Table 11.3 displays the values for the metrics of question Q1 as defined in Section 10.3.2.

Test Set	System	Number of Tests	g_{yes}	g_{no}	g_{can}	$\mathsf{g}_{\text{extern}}$
TS_1	s_1	16	16	0	0	0
TS_2	s_4	7	7	0	0	0
TS_{3a}	s_1	30	26	3	0	1
TS_{3b}	s_3	50	41	2	3	4
TS_{4a}	s_2	12	8	0	0	4
TS_{4b}	s_2	137	137	0	0	0
TS_5	s_1	352	206	13	3	130
$TS_{1 \ldots 5}$		604	441	18	6	139

Table 11.1: Support of MTCC for the validation of test sets

The different test sets displayed in Table 11.1 differ considerable both in the number of entries and in the properties of the included tests. Before we discuss how the validation of MTCC was conducted and examine the results of the validation regarding the capability of MTCC to express tests for the application domain, we examine of the test sets $TS_{1 \ldots 5}$.

We differentiate three groups of test sets: (1) Test sets that were defined for the purpose of test automation, (2) test sets containing tests from current testing practice, and (3) test sets with entries that were derived from the entries of issue tracking systems.

- The test sets TS_1 and TS_2 were defined as basic requirements for a system for the automated execution of acceptance tests. The test sets TS_{3a} and TS_{3b} were defined as manual tests for systems s_1 and s_3. The test sets differ in the functionality of their respective testees and in the specificity of the tests.

- The test sets TS_{4a} and TS_{4b} hold tests that are executed biweekly on system s_2. Test set TS_{4a} consists of a number of different tests for a Digital Library, including a number of tests that are specific to the user interface of the system. The entries of test set TS_{4b} represent different queries for system s_2.

- Test set TS_5 differs from the other test sets in terms of the number of included tests as well as in the origin of the tests. With 352 entries, this test set includes more potential tests than all other test sets taken together. TS_5 is different from the other test sets as it contains faults and outstanding issues taken from the issue tracker of system s_1. Because the entries of the test set were not originally developed as tests, a large number of them are not in the scope of MTCC and fall in the category $\mathsf{g}_{\text{extern}}$.

In the context of this validation we decide for every potential test in every test set if the entry is a test for the functionality covered by MTCC. If this is not the case, we assign the entry to the category $\mathsf{g}_{\text{extern}}$. As discussed in Section 10.3.2 on page 147, we differentiate three categories of support for tests that are within the domain covered by MTCC.

We discuss and interpret the distribution of the entries in the different test sets for the categories of support in Section 11.7.2. In the following, we give an overview of the results of the validation and point out relevant aspects.

Test Set	g_{yes}	g_{no}	g_{can}	$\mathsf{g}_{\text{extern}}$
TS_1	100.00%	0.00%	0.00%	0.00%
TS_2	100.00%	0.00%	0.00%	0.00%
TS_{3a}	86.67%	10.00%	0.00%	3.33%
TS_{3b}	82.00%	4.00%	6.00%	8.00%
TS_{4a}	66.67%	0.00%	0.00%	33.33%
TS_{4b}	100.00%	0.00%	0.00%	0.00%
TS_5	58.52%	3.69%	0.85%	36.93%
$TS_{1...5}$	73.01%	2.98%	0.99%	23.01%
$TS_{1...5}$ without $\mathsf{g}_{\text{extern}}$	94.84%	3.87%	1.29%	

Table 11.2: Distribution of the entries of the validation test sets over the categories of support

Table 11.2 shows the distribution of the tests in the test set over the different categories of support.

MTCC supports 94,84% of all tests if all elements of of the test sets $TS_{1...5}$ that fall into the category $\mathsf{g}_{\text{extern}}$ are excluded from consideration. If the tests in $\mathsf{g}_{\text{extern}}$ are considered, 73.01% of all tests can be represented by MTCC.

All tests of the test sets TS_1, TS_2, and TS_{4b} can be represented by MTCC. The test sets TS_{3a} and TS_{3b} include tests that fall in the category g_{no}.

Of the three tests of test set TS_{3a} that cannot be represented with MTCC, test $t_{3a,16}$ is not supported because the representation of the test would require support for explicit control flow in Test Steps.

The tests $t_{3a,18}$ and $t_{3a,20}$ as well as $t_{3b,31}$ and $t_{3b,33}$ of test set TS_{3b} can be expressed with MTCC test model. Still, the tests fall in the category g_{no} because the execution of the tests would require access to external systems like the Refworks reference management system or email accounts, neither of which the MTCC execution system supports.

Test set TS_{3a} includes three tests of the category g_{can}. These tests address functionality of the system s_3 that is not represented in the Domain Feature Model. Specifically, the tests address an Service that allows the selection and use of terms from a controlled vocabulary for a search.

The test sets TS_{4a} and TS_5 each contain more than 30% tests that fall in the category $\mathsf{g}_{\text{extern}}$. For test set TS_{4a}, these tests primarily address aspects of the specific user interface of system s_2 that are abstracted in MTCC. The entries from test set TS_5 in category $\mathsf{g}_{\text{extern}}$ are mostly not tests but feature requests.

Metric	Test Set T	$gr(T)$
M1.1.1	TS_1	100,0%
M1.2.1	TS_2	100,0%
M1.3.1	TS_{3a}	89,66%
M1.3.2	TS_{3b}	89,13%
M1.4.1	TS_{4a}	100,0%
M1.4.2	TS_{4b}	100,0%
M1.5.1	TS_5	92,79%

Table 11.3: Metrics for question Q1 — percentage of the tests that can be represented in MTCC

Table 11.3 displays the metrics for all subquestions of question Q1. For four of the test sets, all examined tests can be expressed with MTCC.

11.5 Validation of the Executability

Question Q2 and its subquestions examine the executability of MTCC test configurations. Question Q2.1 investigates the resources necessary to implement an execution system for system s_1. Questions Q2.2 and Q2.3 examine a partial implementation of the execution system in order to prove the executability of Test Configurations for different testees and test runners. Question Q2.4 investigates to what degree the execution of MTCC test configurations is decoupled from the specifics of the MTCC approach and can be integrated into an existing testing regime.

The tables 11.4 and 11.5 display the metrics for question Q2. The execution of the Test Configurations was possible for all systems and all test runners. The number of lines of code considers only the parts of the execution system that were specifically implemented for the respective validation questions, common infrastructure code is ignored.

Metric	Percentage of executable Test Steps
M2.1.1	100%
M2.2.1	100%
M2.3.1	100%
M2.4.1	100%

Table 11.4: Metrics for question Q2 - percentage of executable Test Steps

Metric	Lines of codes
M2.1.2	530
M2.2.2	189
M2.3.2	66

Table 11.5: Metrics for question Q2 - use of resources

All questions consider quantitative as well as qualitative aspects, the qualitative results can be found in Section 10.3.3.

Details about the design and implementation of the MTCC execution system are given in Section 8.4, interesting aspects of individual implementations are discussed in the following.

Question Q2.1 examines if the complete implementation of an execution system for the tests TS_1 and TS_{3a} for system s_1 is feasible. We argue that the implementation of the test sets is sufficient to prove the feasibility of a complete implementation of the MTCC execution system since TS_1 and TS_{3a} cover all relevant test classes for system s_1 as defined by the domain experts.

The implementation of all Test Steps of the test sets TS_1 and TS_{3a} for system s_1 and the Selenium test runner was successful. Metric M2.1.1 therefore has a value of 100%, as is displayed in Table 11.4.

Metric M2.1.2 measures the lines of code and thereby the absolute resources necessary for the implementation of the execution system for TS_1 and TS_{3a}. Table 11.5 displays the results.

Table 11.6 displays the lines of code for the complete implementation of the MTCC execution system for all systems and test sets. The function ti returns the lines of code necessary for system s, the test runner tr and the test set ts. Test t_{a1} was defined for question Q2 in Section 10.3.3, it represents a search with a subsequent comparison of the actual and expected number of results.

Execution System	Number of Test Steps	Lines of Code
Configuration Reader	0	261
Test Generator	0	215
Types and Exceptions	0	120
Support Code	0	596
System Independent Test Steps	9	140
$ti(s_1, tr_1, TS_1 \cup TS_{3a})$	23	530
$ti(s_2, tr_1, \{t_{a1}\})$	3	69
$ti(s_3, tr_1, \{t_{a1}\})$	3	55
$ti(s_4, tr_3, \{t_{a1}\})$	5	65
$ti(s_1, tr_2, \{t_{a1}\})$	5	66
Sum	48	925

Table 11.6: Lines of code for the parts of the execution system

The execution system for question Q2.1 was implemented in 670 lines of Python code. 140 lines of this code implement system independent Test Steps that are used for all systems of the system family. 530 lines of code are specific to system s_1 and Selenium. 670 lines of code were necessary to implement the execution system, this is also the value for Metric Metric M2.1.3 measures 20 lines of code that are needed on average to implement each of the 32 Test Step instances needed for the tests defined in TS_1 and TS_{3a}.

The implementation of the MTCC execution system for the test tv_1 for different testees and different test runners as addressed in questions Q2.2 and Q2.3 was successful. Table 11.6 displays the results for metrics M2.2.1 and M2.3.1.

```
import unittest
import s1_executor_selenium as runner
from runner_shared import Bunch, Context

class PortalTest(unittest.TestCase):
  def test(self):
    context=Context()

    para = Bunch({})
    self.failUnless(runner.start(context, para))
```

```
        para = Bunch({'next_context': 'Search',
                'current_context': 'SimpleSearch'})
        self.failUnless(runner.follow_link(context,para))

        para = Bunch({'searchfields':
              [Bunch({'term_relation': 'AND', 'field_relation': 'AND',
                 'name': 'TXT', 'value': '20060111223'})]})
        self.failUnless(runner.start_search(context,para))

if __name__ == "__main__":
    unittest.main()
```

Listing 11.1: Excerpt from a generated test case

The generation of test cases from Test Configuration instances is done as described in Section 8.4. A test case is an instance of a xUnit test class, in Listing 11.1 a Python unittest class.

The generated test cases are dependent on the execution system for the system under test and the test runner, but are independent of the structure of the Test Configuration models used for the generation of test code.

In lines two and three of the listing, support classes and functions from the Test Adapter library are imported. With the exception of the Test Adapter for the tested system and test runner and some general support code, generated test cases are independent from the MTCC approach. The test cases can be executed like tests that were not generated by the MTCC approach and can therefore be integrated in an existing testing approach.

11.6 Validation of Reusability

The reusability of Test Configurations from one system to another was examined by modeling the tests of the test set TS_1 for the system s_1 and testing whether the resulting Test Configurations are transferable to systems s_2, s_3, and s_4.

The tests were not modified or selected with reuse for different systems in mind, a significant number of tests are therefore not reusable regardless of the test transfer methods implemented in MTCC. 50% of the tests in the test set can be reused for system s_4 by applying the techniques described in Section 8.3. 56.25% of the tests are transferable for systems s_2 and s_3. In the case of s_2, three of the transfered tests do not represent the intent of the original test.

Table 11.7 presents the transferability for every test of test set TS_1. As in the discussion of executability of tests, we distinguish different categories of support for the transferability of a test: g_{yes}, g_{no}, and g_{can}. A pair (t, s) with $t \in TS_1, s \in \{s_2, s_3, s_4\}$ falls into the category g_{yes} if a transfer of the test t from system s_1 to system s is possible and is automatically supported by MTCC. If a transfer of the test is possible in theory but is not supported by MTCC for technical reasons, the pair falls in the category g_{can}. The category g_{no} is used when a test t cannot be transfered, neither by automatic or by manual means.

The tests $t_{1,0}$, $t_{1,1}$ and $t_{1,4}$ for system s_2 fall under the category $\mathsf{g}_{\text{false}}$. The Test Configurations of these tests can be transfered to system s_2, but the semantics of the tests are not correctly represented by the transfered tests. System s_2 uses multiple instances of the DOCUMENT_LIST Service in one context. Each of these instances represents a subset of all results returned for a query. When one of the above tests is transfered in MTCC, the dialog for the selection of the Service instances addressed in the test is displayed. This approach fails for the tests $t_{1,0}$, $t_{1,1}$, and $t_{1,4}$ because the result list instances in systems s_1 and s_2 are not compatible.

Test	s_4	s_3	s_2
$t_{1,0}$	g_{yes}	g_{can}	g_{false}
$t_{1,1}$	g_{no}	g_{yes}	g_{false}
$t_{1,2}$	g_{can}	g_{can}	g_{can}
$t_{1,3}$	g_{yes}	g_{no}	g_{no}
$t_{1,4}$	g_{yes}	g_{yes}	g_{false}
$t_{1,5}$	g_{yes}	g_{no}	g_{no}
$t_{1,6}$	g_{no}	g_{yes}	g_{yes}
$t_{1,7}$	g_{yes}	g_{yes}	g_{no}
$t_{1,8}$	g_{no}	g_{yes}	g_{no}
$t_{1,9}$	g_{no}	g_{no}	g_{yes}
$t_{1,10}$	g_{yes}	g_{yes}	g_{yes}
$t_{1,11}$	g_{no}	g_{yes}	g_{yes}
$t_{1,12}$	g_{yes}	g_{no}	g_{no}
$t_{1,13}$	g_{yes}	g_{yes}	g_{yes}
$t_{1,14}$	g_{no}	g_{yes}	g_{yes}
$t_{1,15}$	g_{no}	g_{no}	g_{no}
Transferability	50%	56,25%	37,5%

Table 11.7: Reusability of tests

Table 11.8 displays the metrics for question Q3. 88.9% of the tests for system s_4 and 81.1% of the tests for s_3 can be correctly transfered. 60% of the tests for system s_2 can be transfered.

Metric	System	Value
M3.1.1	s_3	81.1%
M3.2.1	s_2	60.0%
M3.3.1	s_4	88.9%

Table 11.8: Metric for question Q3 - transferability of Test Configurations

11.6.1 Validation of the Usability

The usability of MTCC was examined in questions Q4.1 and Q4.2. All domain experts succeeded in the scenario defined by question Q4.1. They were able to construct an initial test from test set TS_{4b} for system s_2 and then use this test as a basis for the definition of the remaining tests from the test set.

Question Q4.2 examines the understandability of the MTCC approach and the usability of the MTCC editor based on 14 tests taken from the test sets TS_1 and TS_{3a}. 52% of all tests could be constructed without help, for further 38%, the domain experts asked for some help but were otherwise able to construct the test.

The validation of the usability and understandability was done with the participation of the domain experts described in Section 11.1 in the time between 2008/03/28 and 2008/05/07. Every domain expert took part in two sessions, in each of which seven tests were modeled. All domain experts worked on the same tests. These tests were selected randomly prior to the validation from the test sets TS_1 and TS_{3a}. The construction of the five tests in test set TS_{4b} examined for question Q4.1 was also conducted during the first session with each participant.

Every domain expert who took part in the validation was given a short introduction to the MTCC editor. The domain experts also had access to a manual describing the basic principles of the editor. This German manual can be found in Appendix B. Because the manual describes an earlier version of MTCC, it differs from the implementation described here. The first and the second session were always conducted on separate days, the time between the two sittings was between two days and two weeks.

Q4.1 Adaption of Test Configurations

Question Q4.1 examines if the domain experts are able to express the tests $t_{4b,88...90}$ and $t_{4b,103...105}$ using the MTCC editor.

All domain experts were successful in the construction of the tests, on average they took 7.4 Minutes. Table 11.9 presents the time that each domain expert took for the construction and adaptation of the tests.

Participant	Time in Minutes
p_1	5.50
p_2	7.00
p_3	9.00
p_4	8.00
Average	7.38

Table 11.9: Metric M4.1.1 time needed for the adaption of tests

Q4.2 Construction of Tests

In order to answer question Q4.2, 14 tests were randomly selected from test set TS_1 and TS_{3a}. For all participants, 51% percent of all tests fall in the category *nohelp*. For 6 tests or 11%, the domain experts were not able to model the tests, these tests were assigned the category $fullhelp$.

The remaining 38% of all tests could be modeled by the domain experts with minimal help, the tests therefore fall in the category *help*.

Table 11.10 informs about the degree to which the participants of the validation were able to model the tests. Some interesting aspects of the validation are pointed out in the following. We discuss and interpret the results in detail in Section 11.7.3.

The percentage of tests in the category *nohelp* is 67% for the second session, compared to 35% for the first session. While the tests selected for the second session cover less test classes than those in the first session, we argue that the experience gained in the first session contributed to the better results in the second session. An interesting point in that context is that all tests that could be modeled by all four participants without help are part of the second session.

Table 11.11 displays the help needed by the individual domain experts during the validation. All tests in the category $fullhelp$ were done by the domain expert p_2. Domain experts p_1 and p_3 did read the manual before each session, domain expert p_4 used the manual during the session, domain expert p_2 did not read the manual.

Session	Test	*help*	*fullhelp*	*nohelp*
1	$ts_{1,10}$	2	1	1
	$ts_{1,11}$	1	1	2
	$ts_{3a,0}$	2	0	2
	$ts_{3a,10}$	4	0	0
	$ts_{3a,15}$	2	0	2
	$ts_{3a,28}$	2	1	1
	$ts_{3a,5}$	1	1	2
1	Total	14	4	10
	Percentage	50.00%	14.29%	35.71%
2	$ts_{1,15}$	3	0	1
	$ts_{3a,11}$	0	0	4
	$ts_{3a,12}$	0	0	4
	$ts_{3a,14}$	1	0	3
	$ts_{3a,22}$	1	0	3
	$ts_{3a,23}$	2	1	1
	$ts_{3a,8}$	0	1	3
2	Total	7	2	19
	Percentage	25.00%	7.14%	67.86%
1,2	Total	21	6	29
	Percentage	37.50%	10.71%	51.79%

Table 11.10: Metric M4.2.1 Necessary support in test construction

Person	Category	Count	Percent
p_1	*help*	10	71.43%
	nohelp	4	28.57%
p_2	*help*	5	35.71%
	nohelp	9	64.29%
p_2	*fullhelp*	6	42.86%
	help	4	28.57%
	nohelp	4	28.57%
p_4	*help*	2	14.29%
	nohelp	12	85.71%

Table 11.11: Help needed by the domain experts

Problems and Relevant Mistakes in Test Modeling

In addition to the quantitative results discussed above, we also considered qualitative aspects in the validation of the usability. The most relevant cause of problems or faults during the validation was the use of the variables and the operations available for comparing the values of variables. A further source of misunderstandings were the names of Service instances and contexts.

Variables led to the following problems in the validation:

- Domain expert p_3 was repeatedly unsure how and when values were assigned to variables and how variables had to be used. Specifically, the domain expert did not assign values to variables before comparing them with an expected value. Also, often the wrong Test Step was used to assign a value to a variable.
- Domain experts had problems in deciding on the operations used to compare values

stored in variables. A particular problem in this context was the decision on the data types that were to be compared. We discuss this point in detail in Section 11.7.3.

- Variables were overwritten on some occasions before they could be used.

An additional problem unrelated to variables was the selection of the correct Test Step instance for a given task. This problem was in part caused by the use of internal variable names for Services and Test Step instances in the editor. The fact that the variable names were in English and were not always correct translations of the German names proved especially problematic.

The correct use of the Test Step FOLLOW_LINK had to be displayed to all domain experts, especially the correct selection of the target Context for a link was only understood after explanation. After such an explanation, all domain experts could use the Test Step successfully.

Comments from the Participants

The domain experts who took part in the validation made various remarks during the validation and gave suggestions for improvement. Concerning the MTCC approach itself we found that the representation of a test as a sequence of Test Steps was understood by all domain experts. In addition to comments about MTCC itself, the following feedback was given for the evaluated implementation of the MTCC editor.

- The domain experts liked the limited number of Test Steps and the naming schema for the Test Step instances.
- The domain experts missed the option to copy and paste Test Step instances.
- Changes to the GUI of the editor were suggested, for example, to the controls for adding and removing Test Steps and to the editor for Test Step instance selection.

11.6.2 Validation of the Efficiency

In order to investigate question Q5 of the type III validation as discussed in Section 10.3.6 on page 154, the resources necessary for test automation with MTCC are compared to the resources necessary for manual testing. The time needed for the execution of the manual tests are determined based on the experiences of the domain experts p_2 and p_3. Both participants estimated the time needed for the execution of all tests between 1.5 and 2 hours. Based on a biweekly execution of the tests, 39 to 52 hours a year are needed for test execution.

The execution system for the tests in test set TS_{4b} is 124 lines of code long.

11.7 Interpretation of the Validation

Based on the results of the validation, we argue that the MTCC approach as implemented for the purpose of this thesis is feasible as well as usable and understandable by domain experts.

MTCC models support the representation of 95% of all tests from the test sets $TS_{1...5}$. The MTCC editor is useable with minimal practice and use of the manual. Potential future improvements were identified concerning the usability of the editor and the reuse of Test Configurations for different systems within the system family. In the following, we discuss potential weaknesses in our validation plan and in the conducting of the validation. After this we interpret the results of the validation and derive questions for future work. We also discuss concrete improvement to the MTCC approach.

11.7.1 Robustness of the Validation

The validation of MTCC done for this thesis is primarily a type I and type II validation (Freiling *et al.*, 2008). Because we chose to do a case study for validation instead of a controlled experiment, the efficiency of MTCC could only be validated in part.

In the following, we discuss the plan and conducting of the MTCC validation. We focus the discussion on potential problems of the validation. We structure the discussion by the criteria defined by Prechelt (Prechelt, 2001): reliability, descriptiveness, reproducibility, and generalizability

Reliability

The reasons underlying the validation are given in Section 10.2.3. We argue that the validation questions listed are necessary for the validation. We argue that questions Q1, Q2, Q3, and Q4 together with their respective sub questions and metrics are sufficient to support the reliability of the type I and type II validations. As discussed in Section 14.1, the planning and conducting of a type III validation is the subject of future work. Consequently, we do not consider the results of question Q5 as reliable but nevertheless argue that these results are of interest insofar as the allow a first assessment of the efficiency in the absence of more reliable and complete data.

Based on the sufficiency of questions Q1, Q2, Q3, and Q4 concerning the type I and type II validation, we have to examine if the methods and data used to answer the questions are reliable in themselves.

We discussed the motivation for the subquestions and their metrics in Section 10.3.2 and Section 10.3.5. In the following we discuss the test sets and domain experts in terms of their reliability for the validation.

The test set $TS_{1...5}$ used in the validation would endanger the reliability of the validation if they were not representative for the full range of tests that MTCC needs to support. We argue that this is not the case for our validation.

As described in Section 10.3.2, the test sets considered for this validation were collected from various sources and represent requirements from different domain experts. We therefore argue that the test sets are a reliable and sufficient sample of the functional requirements for an automated acceptance test system.

The domain experts who took part in the evaluation and who are discussed in detail in Section 11.1 were all involved in the testing or test design for the member of the system family discussed here. We therefore argue that they form representative if small groups of users for the MTCC approach and specifically the MTCC editor. We consider it advantageous that some of the domain experts took part in the execution of manual tests for the system and thus had prior knowledge of testing. MTCC is an approach to support the automation of tests, not to teach the basics of testing. The fact that the domain experts had skills in manual testing does not endanger our claim that MTCC allows the construction of automated tests by domain experts with little modeling or programming skills.

We argue that four participants and 14 tests are sufficient to judge the basic understandability of MTCC. As Nielsen (Nielsen, 1994, 2001) points out, an heuristic evaluation of a system can be done with a limited number of users. The 14 tests used for the validation were selected at random but nevertheless cover the frequently used test classes for MTCC.

Optimizations of the usability of the editor and the evaluation of their corresponding implementations must be the subject of future work.

We emphasize that we do not consider validation conducted here sufficient to assess the efficiency of MTCC. As already discussed, such an validation would have to be conducted as a controlled experiment (Martens, 2007) with different control groups.

Descriptiveness and Reproducibility

We advance the view that the descriptiveness and reproducibility of the MTCC approach are proven.

We describe the derivation of the Domain Feature Model and Domain Test Model for the considered system family and the construction of the Application State Model and Application Feature Model for systems $s_{1...4}$ in Chapter 9.

Details about the design of MTCC can be found in Part II of this work. Excerpts from the test sets used for the validation can be found in Section A of the appendix.

Systems s_1, s_2 and s_3 are publicly accessible, only s_4 is an internal system. All the software used for the implementation is available as Open Source.

The validation of MTCC is described in detail, the details of the approach are available and can be used to replicate and verify the validation.

Generalizability

When discussing the generalizability of the validation of MTCC it is important to consider the fact that an application of MTCC is by definition specific to a system family and can only be validated in the context of this system family. It could therefore be argued that the results of this validation only apply to the system family of Digital Libraries discussed here.

We advance the view that the results of the validation can be generalized insofar as no reasons exist that would hinder the adaptation of MTCC to other dialog oriented systems besides Digital Libraries.

11.7.2 Feasibility of MTCC

We argue that the practicality of MTCC is proven by the results of the type I validation that was conducted based on the questions Q1, Q2, and Q3.

95% of all tests considered in the validation could be represented by MTCC. The respective Test Configurations could be used for the automatic generation of test code. Opportunistic reuse of Test Configurations for the different system models is possible for more then half of the Test Configurations in the validation.

Interpretation of the Expressiveness

Table 11.2 displays the percentage of tests from the test sets $TS_{1...5}$ supported by MTCC. If entries from the test sets that fall in the category $\mathsf{g}_{\text{extern}}$ are ignored, 94.84% of all tests are supported in MTCC.

The tests that are not supported are mostly from the test sets T_{4a} and T_5. Tests from these test sets that fall in the category g_{no} are mostly members of two groups of tests: (1) Tests whose execution is not supported by MTCC since external applications would have to be exercised and (2) tests that address specific aspects of the GUI of the testee that are abstracted in MTCC.

The following tests can be expressed by MTCC Test Configurations, but fall into the category g_{no} since they cannot be executed by MTCC:

- Test $t_{3a,18}$ requires access to the Refworks[4] application.
- Test $t_{5,184}$ verifies the correctness of the export of result documents by email.

[4]`http://refworks.com`

The following tests are examples for entries from test set TS_5 that cannot be expressed in MTCC since they are dependent on specifics of the GUI of the testee which is expressly abstracted away in MTCC.

- Test $t_{5,31}$ exercises the Drop Down list that is used to change the order in which documents are displayed in the result list. It checks whether the Drop Down List contains the correct values in the correct order. Since MTCC does not model the specifics of the user interface, this test cannot be represented.
- Test $t_{5,281}$ is adapted from a fault reported for the GUI. One of two instances of a control used for browsing a list of results was not operational. Again, MTCC is not able to represent the test since the specifics of the GUI are not represented in the MTCC models.

The metrics for question Q1 in Table 11.3 summarize the results of the validation for question Q1.

MTCC provided support of 89.13% for the test set TS_{3b}. Two of the tests in the test set are not supported because the execution of the tests would require the execution of external Services.

We argue that the feasibility of MTCC is not challenged by the fact that MTCC does not support such tests. The automation of tests that rely on the interaction with external Services is also problematic for other test automation approaches. The testing of GUI functionality is not the purpose of MTCC and can be better tested by dedicated approaches for GUI testing.

Execution of Tests

We consider the executability of MTCC to be proven by the results of the validation. The implementation of the execution system was possible with limited resources, generated tests could be executed by a COTS test system. As discussed in Section 11.5, the execution systems consist of 925 lines of code for all implemented Test Adapters and 596 lines of code for testing core assets. We argue that this resource usage is acceptable given the number of tests that can be expressed based on the implemented infrastructure, especially when the fact is considered that the code of the Test Adapters is structurally simple. The feasibly of the test code generation approach could be proven. The MTCC implementation used for this validation generates tests for the Python xUnit implementation.

We expect an implementation of an execution system for other programming languages to be unproblematic. The abstractions used by the Test Parameter objects provide common abstractions that are provided by most programming languages and no assumption about the execution of tests is made.

Reuse

The reuse of Test Configurations for different system models is possible for 50% of all tests of test set TS_1 and the target systems s_2, s_3, and s_4. The tests that are not reusable mostly fall in the category g_{no}. This category consists of tests that cannot be reused because of missing or otherwise incompatible features of systems. Reuse is therefore not possible regardless of the test reuse approach used.

As discussed in Section 11.6, the test $t_{1,0}$, $t_{1,1}$ and $t_{1,4}$ can be reused for system s_2, but do not correctly represent the intent of the test. We therefore consider these tests as not automatically reusable.

The test $t_{1,2}$ is in the category $\mathsf{g}_{\mathrm{can}}$for all systems in the system family S — the test is reusable in theory but cannot be transfered by the MTCC reuse system. The reason for this failure of the automatic reuse system is a lack of support for mappings between different names and representations for the same concept in the feature models. Specifically, a filter in s_1 could be represented as a semantically equivalent search field value in s_2, but an automated mapping of this concepts is not supported.

Between 60% and 90 % of all tests for a system can be automatically reused if only those tests are considered in the validation that do not fall in category $\mathbf{g}_{\text{no}}$. We argue therefore that the support for the reuse of tests that MTCC facilitates already allows for significant savings in testing costs. Based on this, we further argue that reuse in MTCC is successfully implemented.

11.7.3 Usability of MTCC

Based on the result of the validation of the usability of MTCC, we argue that the approach is understandable and usable given minimal training. While significant potential for improvement exists regarding the usability of the editor, the prototype implementation used in the evaluation can be successfully used for the construction of tests.

As Table 11.10 displays, 52% of the tests could be modeled without help, for 37% some help was needed and for 11% the tests could not be modeled. Because the same domain expert was responsible for all failed tests and because this domain expert did not read the manual, we argue that MTCC can be understood with minimal preparation. We also advance this view especially given the fact that the results of the second session — 68% of all tests could be modeled without help — were significantly better than the first session.

As we discussed in Section 10.3.5, one re-occurring problem in the use of the editor was the use of variables. We discuss potential solutions for this problem in Section 14.1.

As we pointed out in Section 11.6.1, the participants of the validation made a number of suggestions to improve the usability of MTCC. We argue that these improvements can be addressed in separation from the other issues discussed in the validation and that the MTCC approach is valid and usable and that the usability can be improved in the future even though the usability of the editor was not the focus of this thesis (Norman, 2006).

11.7.4 Efficiency of MTCC

Based on the limited validation of the efficiency of MTCC, the automation of the execution of test set TS_{4b}, we argue that the use of MTCC is beneficial from a resource point of view even over short periods of time. A robust validation of the efficiency, however, is the subject of future work.

As discussed in Section 11.6.2, 39 to 52 hours of time per year are spent on manual test automation. The implementation of the execution system for the tests in test set TS_{4b} consists of 124 lines of Python code.

If 4 hours are estimated for the implementation and maintenance of the execution system, which we argue is realistic based on 124 lines of code and the simple structure of the test code, and if 15 minutes are needed for each run of the automated test, then testing with MTCC requires 10.5 hours per year versus a minimum of 39 hours for manual testing. If we further consider the fact that the addition of new tests to the test set TS_{4b} would have no impact on the resources required by MTCC, then MTCC is significantly more efficient than manual testing, even for a limited number of tests and one system under test.

11.8 Chapter Summary

The validation of MTCC was conducted on three Digital Libraries and the installation of the Solr search server at the GESIS. Four domain experts were involved in the validation of MTCC. All domain experts had prior experience in the specification or execution of tests or information retrieval evaluation scenarios. Five different sets of tests were used for the validation of MTCC. These test sets include tests taken from the testing processes of the evaluated systems as well as tests that were contributed by domain experts for the validation and tests derived from entries in an issue tracking system.

The type I validation showed that 95% of all tests from the investigated test sets that were of interest to the MTCC approach could be expressed by MTCC test models. The execution of tests was successful for the validated test set and for all considered SUTs and test runners. The implementation of the execution system was done in 925 lines of Python code, an implementation of all 23 Test Steps needed for the execution of all tests in test sets TS_1 and TS_3b was done in 530 lines of code. Between 60% and 88% of all practically transferable tests for system s_1 defined in the test set TS_1 could also be transfered by the MTCC test reuse system.

The type II validation of MTCC was conducted in eight sessions. In each of these sessions, the four domain experts modeled seven tests. Out of the 56 tests modeled in total, the domain experts could represent 52% without any help, a further 37% of the tests could be expressed with minimal help that would also have been available from the MTCC manual. 11% of the tests could not be expressed.

We conclude that the MTCC approach is feasible, can be implemented with limited resources and can be understood by domain experts given minimal training.

Chapter 12

Related Work

This chapter examines approaches that address situations similar to those considered by MTCC, are faced with related challenges or use comparable methods to tackle the problem of abstracting testees that have to be exercised by automatic tests.

One basic difference between MTCC and approaches to Model-Driven testing (Blackburn *et al.*, 2004, 2002) is that MTCC is not based on the principle to automatically generate tests from a model. Specifically, MTCC does not aim to reuse models from the software construction process. Bertolino describes the motivation behind such reuse in the following: *"The leading idea is to use models defined in software construction to drive the testing process, in particular to automatically generate the test cases."* (Bertolino, 2007)

Models in MTCC, in contrast, are used to abstract implementation details and facilitate the systematic handling of the variability of a system family. The goal of MTCC is the construction of tests by domain experts, not the automatic creation of tests.

Given these fundamental differences — regarding both the goals and the methodology — between MTCC and automatic, Model-Driven testing, we do not discuss Model-Driven testing aimed at fully automatic test generation further.

This chapter discusses different fields of research related to MTCC and gives examples for concrete approaches. Section 12.1 investigates the use of use cases and examples for testing. Section 12.2 discusses approaches that employ Model-Driven software development techniques for the realization of test code. Section 12.3 introduces approaches that represent graphical user interfaces and web applications as models for testing. Section 12.4 examines approaches to software product-line testing, in particular an environment for test construction that is in some aspects comparable to the MTCC editor. Section 12.5 discusses approaches to involve users and domain experts in the programming or more general software development.

12.1 Testing Based on Usage Scenarios

We call approaches that aim to capture and formalize requirements and use them as a basis for testing and test generation requirements-based testing. Approaches to requirements-based testing frequently build on the notations and methods of the requirements engineering process and then extend and leverage them for testing.

The most significant difference between approaches to requirements-based testing and MTCC — beside the focus of the former on the testing of single systems and the aim to reuse and extend methods and artifacts from the software construction process — is test construction by domain experts as a central aspect of MTCC.

One way to express requirements are scenarios, defined by Ryser as follows: *'Descriptions of interaction sequences between two or more partners, most often between a system and its* *users'* (Ryser, 2003). A use case is an extension of a scenario that also describes alternate scenarios for special cases or failures.

A central aspect of requirements based approaches is the formalization of existing notations of requirements (Tsai *et al.*, 2001; Ryser, 2003; Poston, 1996; Nebut & Fleurey, 2006; Poston, 1998) in order to turn them into testable (Binder, 1999) models.

MTCC considers artifacts for requirements analysis and testing in separation. While MTCC Test Configurations serve as a limited, functional, executable specification (Hoffman & Strooper, 2000; Fowler, 2006), they are intended to supplement, not replace, existing specification.

One group of approaches to requirements-based testing is based on UML (Gutierrez *et al.*, 2006; Botaschanjan *et al.*, 2003), primarily on use case diagrams. Regarding their relationship to MTCC, these approaches are no different from the approaches discussed above.

12.1.1 The SCENT Approach

The SCENT approach developed in Ryser (2003) addresses the verification of requirements and the conducting of system-level tests based on scenarios expressed as state-charts. A particular focus of Ryser's work are dependencies between state-charts.

Domain experts are involved in the identification and design of scenarios and thereby contribute to the construction of high-quality software systems. SCENT differs from MTCC in the consideration of single systems and in the way in which domain experts are involved in the testing process.

The application of the SCENT approach is subdivided in five steps:

- Use cases for the considered system are captured as SCENT scenarios. The analysis of the testee for scenarios is done by software engineers and domain experts in cooperation. The goal of the approach is the description of scenarios in natural language.
- The scenarios are formalized as state-charts. Every scenario is modeled as a state-chart that represents every system event as a state and every user interaction as an event.
- The state-charts are enhanced with information that is later used for the generation of tests from the model. Among other information, pre- and postconditions are added to the states of the state-charts. Also, supported ranges for input values are added for the user events.
- The dependencies among the previously defined state-charts are analyzed and formalized. This is a prerequisite for the later composition of scenarios into tests. SCENT uses a graph-based notation to express sequences, alternatives and relations for scenarios. Figure 83 gives an overview of some of the relevant concepts.
- Tests are derived from the models of scenarios and their dependencies. This derivation will in most cases be done manually.

SCENT is similar to MTCC both concerning its realization as well as its goals. Both MTCC and SCENT envision the involvement of domain experts in testing and consider testing as an essential activity during the software lifecycle.

Basis differences between MTCC and SCENT exist in the concrete roles that domain experts have in both approaches, in the test construction process and in the treatment of system families in MTCC. These differences result in different requirements for the models used in the approaches.

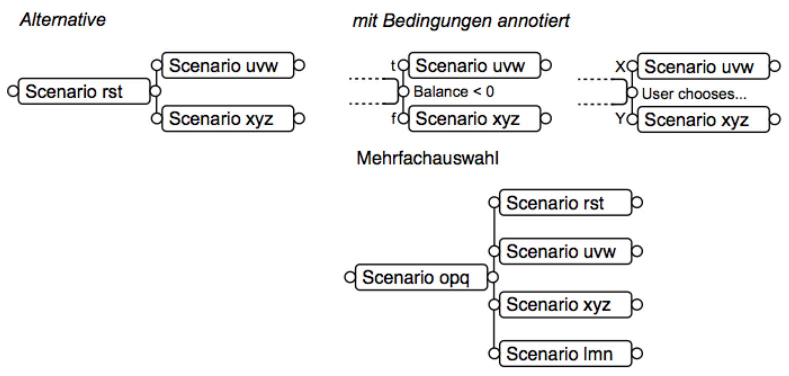


Figure 83: Notation to express dependencies of scenarios in SCENT (Ryser, 2003)

MTCC involves domain experts during the analysis of a system for its test relevant aspects as well as during the construction of tests. SCENT involves domain experts in requirements analysis and in the realization of test scenarios. These scenarios are not tests in themselves but serve as an input for test derivation.

The central artifacts in SCENT are not tests but the formalized scenarios and the dependencies between these scenarios. In contrast, the focus of MTCC is the construction of concrete tests. We argue that the MTCC approach is better suited to facilitate the maintenance of tests during the software development process. In MTCC, domain experts can construct tests at any given time without the need to create executable tests from scenarios. Since SCENT envisions the automatic generation of tests from the scenarios, these scenarios have to contain more information about the testee than needed in MTCC. While this causes an increase in the complexity of the models in SCENT and makes modeling more expensive, it can potentially result in a better test coverage than testing using the MTCC approach. Both approaches depend on the cooperation of domain experts and software engineers during requirements analysis.

MTCC and SCENT use state-charts or finite state machines to represent the behavior of systems under test, specifically to represent possible interaction sequences between users and the system under test. The approaches differ in the granularity in which systems are modeled — MTCC only includes high-level models of the behavior of a system.

One significant difference between MTCC and SCENT is the consideration of system families by MTCC.

12.1.2 The Eg Approach

The Eg approach developed by Gaelli (Gälli, 2006) uses scenarios, there called examples, for the testing of software as well as for documentation. This approach is related to MTCC insofar as Gaelli does not aim at the reuse of artifacts from design or requirements analysis.

Important differences between the Eg approach and MTCC exist both in the fact that Eg addresses software engineers and not domain experts and in the use of tests as examples, not as specification. Figure 84 illustrates this difference between MTCC and Eg: The acceptance tests constructed with MTCC correspond to story tests in the figure, Gaelli's approach addresses unit tests, the user of the Eg approach are developers. As MTCC, Eg does not restrict the role of tests on fault finding, but where MTCC considers tests as a executable specification, Eg considers them as examples that facilitate the understanding of software.

Figure 85 displays the Eg meta model. As the figure illustrates, Eg addresses a system under test on the class level. This fine-grained view is necessary to support the implementation-

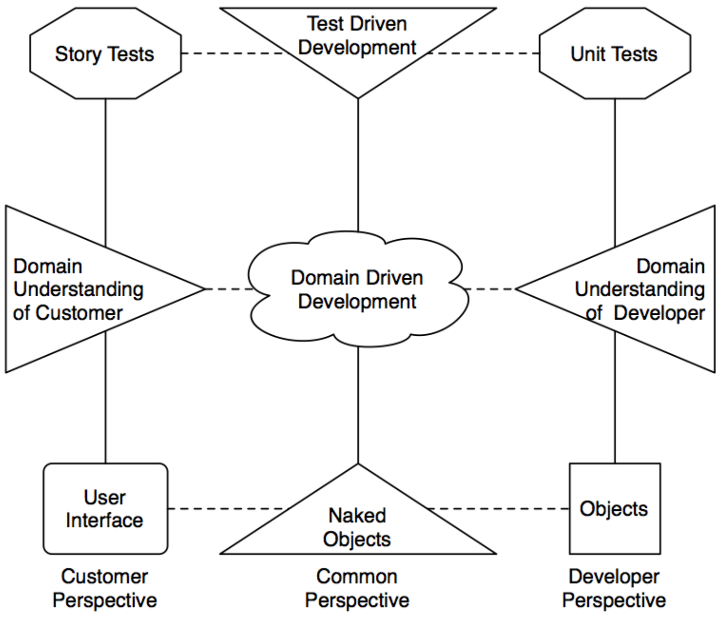


Figure 84: Viewpoints of domain experts (customers) and domain engineers (developers) (Gälli, 2006)

based tests envisioned by Eg, but is a basic difference to the abstract MTCC feature models that explicitly abstract the implementation.

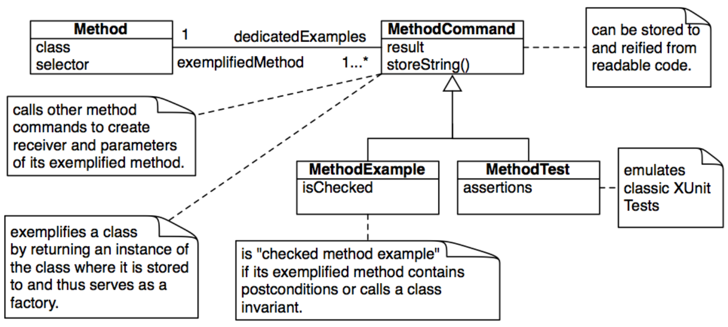

Method
class
selector
1
dedicatedExamples
exemplifiedMethod
1...*
MethodCommand
result
storeString()
can be stored to and reified from readable code.
calls other method commands to create receiver and parameters of its exemplified method.
exemplifies a class by returning an instance of the class where it is stored to and thus serves as a factory.
MethodExample
isChecked
MethodTest
assertions
emulates classic XUnit Tests
is "checked method example" if its exemplified method contains postconditions or calls a class invariant.

Figure 85: The Eg meta model (Gälli, 2006)

12.2 Modeling of Tests

Most approaches to test modeling do not aim to derive or generate tests from models but rather use a special notation to construct or model tests directly. The models or notations used in such approaches are dedicated to test construction, existing models from software design are not reused.

One difference between the approaches discussed here and MTCC is that the notation used for test description in the former is less abstract than the models used in MTCC. For example, tests are expressed in an XML notation or created using a spreadsheet application (Andrea, 2004b,a). One example for the construction of tests as tables is FIT (Mugridge

& Cunningham, 2005a,b; Mugridge, 2004). Like MTCC, FIT is an approach for the automation of acceptance tests. We discuss the relationship of FIT and MTCC in Section 12.5.

12.2.1 UML 2.0 Testing Profile

The UML 2.0 Testing Profile (Schieferdecker *et al.*, 2003; Baker *et al.*, 2003) adds concepts relevant for testing to the UML. The application of the UML2TP allows the modeling of tests. The automatic generation of test code from the models is envisioned (Dai, 2004). In contrast to MTCC, the U2TP is not a complete approach to testing but rather a specialized profile for test modeling.

A well established way to express tests independent from the specifics of a GUI are keywords and action words (Graham & Fewster, 2000). Like MTCC, approaches based on keywords or action words identify relevant actions on the system under test and abstract the specifics of the GUI (Andersson & Bache, 2004). Each keyword or action word represents a concrete step in a workflow or in the interaction between the testee and a user. Keywords and action words correspond to Test Steps in MTCC, but a difference exists in the fact that MTCC builds on this common concepts by providing a model based editor for domain experts and considering system families instead of single systems.

Test scripts are no different from other software in the fact that they can be generated in a Model-Driven software development process. One example for an approach that uses Model-Driven development in the testing domain can be found in (Sensler *et al.*, 2006).

12.2.2 Model-Driven Test Development

Approaches for Model-Driven software development can generate test code as well as application code. One approach that applies MDSD techniques to the testing domain is the work by Sensler et al (Sensler *et al.*, 2006). Building on an initial analysis, tests for a system are formalized in XML, this XML then serves as input for a code generation system. With the consideration of system families and the goal to include domain experts in the testing process, MTCC addresses goals beyond the separation from test models and specific system implementations addressed by such systems.

12.3 Abstract Representation of User-Interfaces

MTCC is an approach that abstracts user interfaces in order to allow the description of high-level function tests. Therefore, MTCC follows a different approach than approaches that seek to test the GUI itself.

12.3.1 Representation of GUIs

Memon (Memon *et al.*, 2001; Memon, 2001) uses a hierarchical model to describe the possible interactions of a user and a GUI. Tests are created automatically based on a test plan and the model of the GUI. Like in MTCC, structural and dynamic aspects of a testee are modeled, but differences exist in the granularity in which the GUI is represented. While MTCC only represents the GUI in terms of the available Test Steps, Memon's representation also includes GUI elements like menu items and other low-level artifacts.

Li et al (Li *et al.*, 2007) use a two-level model for a GUI. An interesting aspect of their work is the GUI component that roughly corresponds to a Service in MTCC.

A fundamental difference between MTCC and the approaches above is that the models used in MTCC do not represent the GUI, but the functionality provided by the GUI. This allows the use of a common domain model for systems with fundamentally different GUIs, for example the Digital Libraries and the Solr search server considered in the validation.

12.3.2 Modeling of Web Applications

Ricca (Ricca & Tonella, 2001; Ricca, 2003) developed the Reweb approach that models the structure of a web site as a graph structure. The representation of a web site as a graph structure is similar to the representation of a testee in the MTCC Application State Model.

A relevant difference between approaches like Reweb and MTCC is the fact that MTCC, while it is validated in the web application domain, is technology agnostic. Reweb in contrast is specific not only for the behavior and structure typical for web application but also for concrete implementation technologies as can be seen from the Reweb meta model displayed in Figure 86.

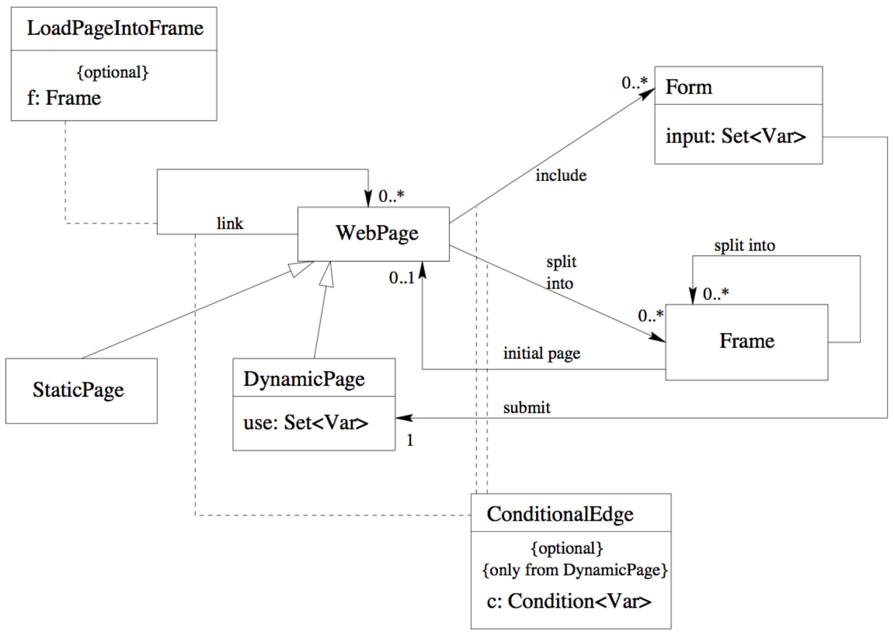


Figure 86: Meta model of a web application underlying the Reweb approach (Ricca & Tonella, 2001)

12.4 Testing of System Families

One central aspect of MTCC is the testing of system families, MTCC is therefore related to approaches for system product-line testing and to alternative approaches that address the testing of system families without systematic reuse.

Fundamental differences between MTCC and these approaches exist both in the underlying process and in the view of the respective testees.

12.4.1 Testing of Software Product Lines

A central question concerning the testing of products from software product lines (Tevanlinna *et al.*, 2004) is the selection of those software systems from all possible systems of the product-line that are to be tested (Geppert *et al.*, 2004b). Another, even more basic issue is the capture and treatment of the variability in the testing context (Bertolino & Gnesi, 2003b,a).

The problem of test selection is not relevant in the context of MTCC, because tests are selected by domain experts. The treatment of the variability of the system is not done on the level of individual tests but is done on the level of the system as expressed by the Application State Model and the Application Feature Model. In MTCC tests are constructed for one concrete member of a system family.

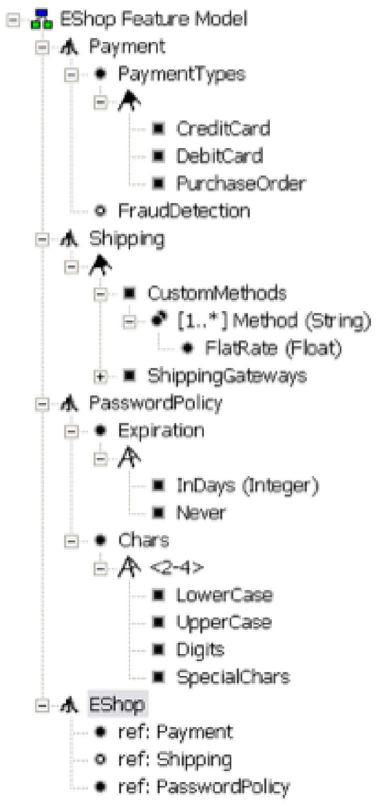


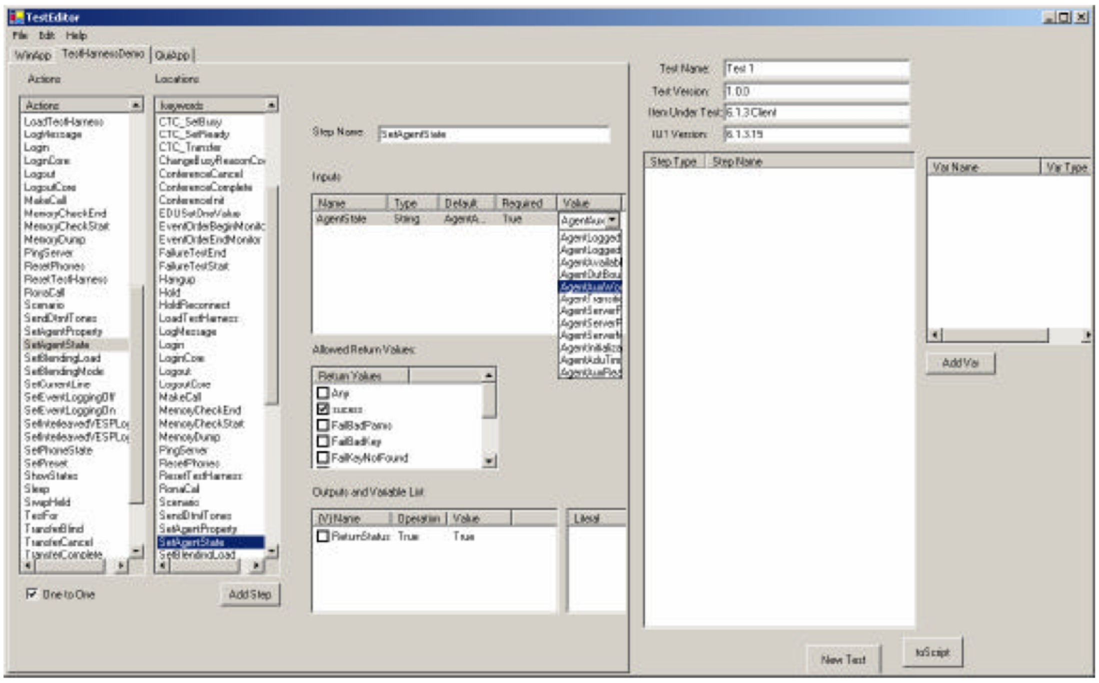
Figure 87: Representation of a feature model in a editor (Antkiewicz & Czarnecki, 2004)

The MTCC editor can be regarded as a tool for the manipulation of feature diagrams. It differs from other editing environments (Myllännistö & Soininen, 2005; Antkiewicz & Czarnecki, 2004) for feature models as displayed in Figure 87 insofar that its goal is not the representation of a feature model and supporting specializations on this feature model. Instead, the MTCC editor aims to offer a view of the feature model that allows domain experts the configuration of tests based on domain-specific GUI conventions.

12.4.2 Condron's TADE Approach

The TADE (Test Application Development Environment) (Condron, 2004) presented by Condron has a number of similarities to MTCC: The approach uses an editor to support the specification of tests for the members of a systen family. Figure 88 displays a screen shot of the TADE.

The most relevant differences between Condron's approach and MTCC are regarding the goals of the approach, the use of models and the reuse of test models. The goal of Condron's



Step Name: SetAgentState

Name	Type	Default	Required	Value
AgentState	String	AgentA...	True	AgentAux

Return Values
☐ Any
☑ success
☐ FailBadParms
☐ FailBadKey
☐ FailKeyNotFound

(V) Name	Operation	Value
☐ ReturnStatus	True	True

Literal

Test Name: Test 1
Test Version: 1.0.0
Item Under Test: 6.1.3 Client
IUT Version: 6.1.3.19

Figure 88: GUI of the TADE for test development (Condron, 2004)

approach is to decouple tests form implementation details of individual testees and reuse the testing infrastructure. MTCC aims at the reuse of tests and the involvement of domain experts. Regarding the use of models, the domain and systems models, their construction, specialization and composition are central aspects of the MTCC process. This use of models allows the systematic treatment of the common and variable test-relevant features of the member of a system family. The TADE presented by Condron is not model-based or Model-Driven in the sense used in the thesis. Finally, Condron does not address the reuse of test models for different variants or versions of a system.

12.5 Involvement of Domain Experts in Software Development

12.5.1 End User Programming

The automatic instantiation of graphical user interfaces for modeling systems usable by end users is an area of active research. We differentiate generic approaches without a specific application domain (Prinz *et al.*, 2006) from those that are designed for a specific purpose and for a concrete group of users, for example, for the development of component-oriented systems (Uflacker, 2005) or the programming of machine tools (Prähofer *et al.*, 2006). Examples for systems that are not specific for an application domain are meta case systems (Tolvanen & Rossi, 2003; Deufemia *et al.*, 2006). All the systems above aim to involve people without formal modeling skills, but differ in the fact that they do not address the testing domain.

An example for a commercial system that facilitates end-user programming is the Automator software included in Mac OS X[1]. Like MTCC, this program represents a program or test as a sequence of successive steps and supports the parameterization of each step. In contrast to MTCC, Automator is not based on a formal GUI-independent model.

[1]http://developer.apple.com/macosx/automator.html

An alternative to the use of a GUI like in MTCC is the use of simple, textual domain-specific language for end user programming (Spinellis, 2001; van Deursen *et al.*, 2000).

MTCC differs from the above approaches insofar as it is a model-based approach for testing. The models used to represent tests and testable systems are central to the approach, the concrete GUI representation used in the editor is decoupled from the models but could be adapted by changing the Buildlets used to construct the GUI. The relationship of the models and the GUI can be thought of in terms of the model-view-controller approach (Buschmann *et al.*, 1996).

12.5.2 FIT

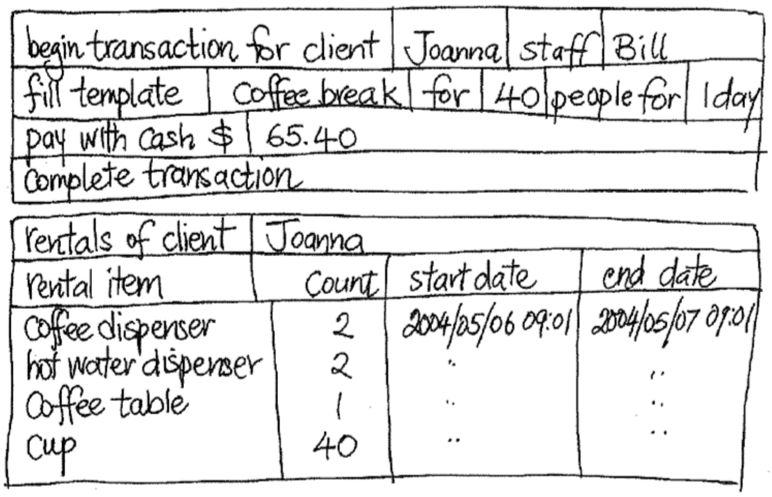
FIT, like MTCC, is an approach to the automation of acceptance tests. MTCC and FIT have a number of similarities regarding their respective goals as well as aspects of their realization.

A basic difference between FIT and MTCC is the consideration of system families in MTCC and the resulting differences in the underlying models that separate Test Configuration and test execution for MTCC. FIT does not address the systematic treatment of the variability within a family of testees.

Like MTCC, FIT considers the role of testing to be both in fault detection and in the identification and clarification of requirements. Tests are derived in story-driven test development, relevant use cases for testing are identified and serve as the basis for tests and the identification of the parameters needed for testing.

FIT supports different types of tests. Generally, tests either verify the correctness of calculations or exercise a testee based on a sequence of action words. The later type of tests is called a workflow test. All tests in FIT are expressed as tables. This representation was chosen because tables and spreadsheets as well as the tools used to interact with them are familiar to non-technical users.

Figure 89 displays a sketch of a table for a FIT workflow test. Such a sketch would be used in the analysis phase to identify functionality relevant to testing.



begin transaction for client	Joanna	staff	Bill		
fill template	Coffee break	for	40	people for	1day
pay with cash $	65.40				
complete transaction					

rentals of client	Joanna		
rental item	Count	start date	end date
Coffee dispenser	2	2004/05/06 09:01	2004/05/07 09:01
hot water dispenser	2	"	"
Coffee table	1	"	"
cup	40	"	"

Figure 89: Sketch of a FIT workflow test (Mugridge & Cunningham, 2005b)

FIT uses fixtures to execute a test described in a table on the testee. Fixtures in FIT broadly correspond to Test Adapters in MTCC — they provide an implementation for the actions specified in a test. Figure 90 illustrates the relationship between tables and fixtures.

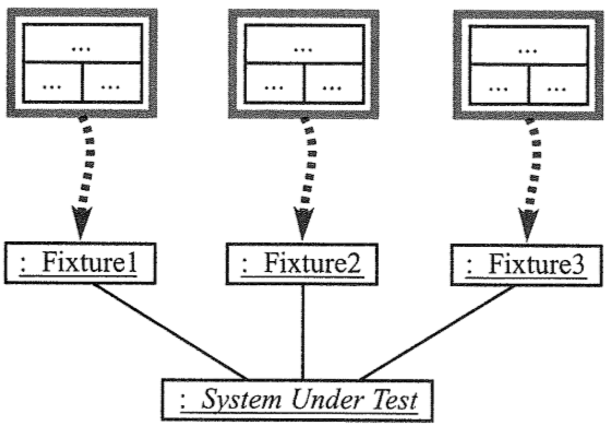


Figure 90: Relationship of testees, tests and fixtures in FIT (Mugridge & Cunningham, 2005b)

In contrast to MTCC, the GUI representation of a test and the underlying model of the test are not different concepts in FIT. The table used to represent the test is both the GUI and the model.

FIT does not address the testing of different systems in a system family. While it is possible to implement multiple fixtures to execute a test on different systems, such an application is not an explicit part of FIT.

12.6 Novelty of MTCC

MTCC is novel in its goal to facilitate the involvement of domain experts in model-based testing of software system families. In order to achieve this goal, MTCC combines the fields of acceptance testing, Model-Driven software development, feature modeling and software product-line engineering. The main attributes that distinguish MTCC from existing approaches are (1) the use of a hierarchy of models to support the intellectual specification of tests and facilitate the reuse and maintenance of these tests. (2) The use of an editor to represent this models to domain experts as a GUI based on the conventions of the domain.

12.7 Chapter Summary

MTCC is related to a number of different approaches and research directions in Software Engineering. MTCC differs from these approaches in its goals as well as in the research directions taken to address the various challenges and research questions.

Requirements-based testing reuses and enhances existing requirements representation as a basis for testing. MTCC does not assume that formal requirements exist and uses dedicated models to represent the SUT and test relevant for the SUT. The use of models specific to testing also puts MTCC in contrast to approaches like the U2TP that aim for the reuse of design models for testing.

MTCC is not an approach to Model-Driven GUI testing. The models used by MTCC represent the Services that are made available through the GUI, not the GUI itself. MTCC differs from approaches to software product line testing in that it does not assume the existence of a common set of Core Assets.

A central difference between MTCC and the above approaches is the strong focus that

MTCC puts on the involvement of domain experts in the testing process. MTCC differs from alternative approaches to automated acceptance testing in the use of models to systematically describe the available features in a system family and the Test Steps available to exercise these features. The combination of models dedicated to testing members of system families and the involvement of domain experts, facilitated by the MTCC editor, are novel contributions of MTCC.

Part IV

Conclusion

Chapter 13

Summary and Contributions

This thesis introduced and validated the MTCC approach to Model-Driven acceptance testing for system families. MTCC applies methods from software product line engineering to the testing domain and uses models to represent the test-relevant features of a system. Tests represented by models and serve as the basis for test code generation. MTCC makes several contributions to the domains of automated acceptance testing and to the testing of system families.

13.1 Summary

The MTCC approach introduced in this thesis supports the construction of acceptance tests for the members of a system family by domain experts. In this thesis, MTCC was applied to a Digital Library system family. The test-relevant functionality and workflow steps of the systems in the system family were captured in MTCC models. These MTCC models and an implementation of the MTCC core assets serve as the basis for the validation of MTCC.

The feasibility and practicality of MTCC as well as the usability of the MTCC editor were evaluated with good results.

The basic hypothesis of this work is that the use of models supports the construction of abstract tests by domain experts using an editor. We further advanced the hypothesis that the use of abstract models facilitates the reuse of tests in a system family context. We argued that abstract test models can be leveraged to exercise and test concrete systems. We introduced the MTCC process and its artifacts in Chapter 6 in order to demonstrate the validity of the approach.

13.1.1 The MTCC Approach

The MTCC process considers the representation of systems and Test Steps on three levels. On the domain or system family level (1), members of the family of testees are analyzed for commonalities and variability regarding their test-relevant functionality. The findings of this analysis are formalized as Services and Test Steps.

On the system level (2), a representation for a concrete system is built by specializing the feature models that represent Services at the domain level to concrete instance representations at the system family level. On the testing level (3), the models that represent the behavior, features, and tests of MTCC at the system and domain-level are integrated into a common model that is dedicated to test construction. This model represents a system in terms of the possible sequences of configurable Test Steps that can be executed on the system.

The actual construction of tests is done using the MTCC editor. The editor represents the MTCC models according to the conventions of the application domain and allows their configuration. Constructed tests — test configurations — are used to generate test cases that are executable with COTS test runners.

13.1.2 Applying MTCC to a System Family of Digital Libraries

We have applied MTCC to a system family of three Digital Libraries and on the search server Solr. The identification of relevant tests was done based on an already established testing regime, documented faults and requirements by domain experts. The identified tests include scenarios for the evaluation of Information Retrieval systems as well as functional tests.

13.1.3 Implementation

We implemented the MTCC core assets, specifically the editor, the domain model and the Test Generator for this MTCC evaluation. Test Adapters were implemented to different degrees for all four systems considered in the validation and for two COTS test runners.

13.1.4 Validation

A type I and type II validation for MTCC was done based on the GQM approach. The type I validation assessed the feasibility of the MTCC approach. The capability of MTCC to represent the relevant tests for the domain was validated. Additionally, the reusability of test models between different system models and the executability of MTCC models for the testing of system implementations were proven. A type II validation was conducted with the involvement of four domain experts. The validation lead to the following results:

- 94.84% of all tests considered in the validation could be expressed with MTCC. Tests that are not supported by MTCC are either dependent on specifics of the testee's GUI or exercise external Services.
- The Test Adapter instances needed for the validation could be implemented with few resources, specifically 900 lines of code.
- The reuse of Test Configurations for different members of the examined system family is possible for about 50% of all tests. The reuse techniques are limited by the fact that MTCC can not verify that tests that were transfered to a new system are semantically equivalent to the original tests.
- The MTCC approach is understandable and usable by domain experts given minimal training and preparation.

13.2 Contributions

We see the scientific contribution of MTCC primarily in the modeling concepts developed to express the testable features and tests for systems in the context of product-lines and in the concepts used to represent and manipulate the models as they are used in the editor. Specific results from the application of MTCC to the domain of Digital Libraries in this context are the domain-level models that capture the aspects of the domain relevant to acceptance tests and the system-level models that represent individual systems. The usage of feature models to describe tests in the context of a system family and the transfer of specializations that were applied to these feature models in order to facilitate test reuse are — to our knowledge — original contributions of MTCC.

Specifically, MTCC makes the following contributions to automated testing

- The process for test construction used by MTCC describes how domain analysis and software product-line methodologies can be applied to testing. With the identification of test-relevant functionality and actions, the formalization of this knowledge in domain-level models, and the specialization of these models to system-level models, MTCC defines a systematic process to realize a testing infrastructure for an existing system family.

- One novel aspect of MTCC is the use of feature models to represent the testable features of the members of a system family. MTCC employes feature models for the description of high-level Services as well as for the definition of parameters that are available for individual Test Steps. The separation of system-level Service instances and domain-level Test Steps, in combination with the use of references between feature models, reduces the redundancy in test modeling and thus supports maintenance and adaptation of test models.

- The systematic reuse of existing Test Configurations is an original contribution of MTCC. By checking the transferability of Test Step sequences both based on the features referenced by individual Test Step instances and on the sequence of Test Step types, MTCC enables the reuse of existing tests.

- On significant contribution of MTCC is the realization of an approach that allows the representation of feature models in an editor for tests as well as the manipulation of these feature models. The combination of Buildlets and configuration nodes allows for the representation of feature models according to the standards and customs of the application domain investigated by MTCC.

In addition to the points above, MTCC also made the following contributions:

- In the course of this thesis we applied MTCC to the Digital Library domain. This work resulted in MTCC domain and system level models for the test-relevant functionality of a system family of Digital Libraries. Because these models are based on the requirements of domain experts and on current testing practice for the considered systems, the test models represent the functionality that has to be covered by testing approaches within this domain.

- The validation of the MTCC approach outlines possible validation scenarios for other approaches with similar goals.

- The implementation of the MTCC execution system indicates the resources necessary to implement a test library for a web application from the Digital Library domain.

- The validation conducted as part of this thesis demonstrated that domain experts using the MTCC approach can contribute to automated testing given minimal training.

Chapter 14

Outlook and Conclusion

While the feasibility and the practicality of the MTCC approach were demonstrated by the MTCC validation, a number of interesting questions remain future work

14.1 Future Work

We differentiate two major areas of future research for the MTCC: The further validation of the approach and its extension and adaptation based on the results of this thesis.

14.1.1 Additional Validation

Three topics are relevant for future validations: The feasibility of MTCC for other application domains other than Digital Libraries, the assessment of MTCC in the field and the conducting of a type III validation:

- This thesis validates MTCC for a system family of digital libraries. In order to assess the feasibility and practicality of MTCC in other application domains, the capability of MTCC to represent the concepts of these domains must also be assessed. Beside the conceptual aspects, the ability of MTCC to exercise testees that are not web applications must also be verified.
- While MTCC was validated on production systems, the approach was not yet used in a production context. One question that a future validation could address is to what degree MTCC can replace or supplement an existing testing approach. An interesting question in this context is whether MTCC is primarily useful in the context of regression tests or if it also supports the identification of previously unknown faults.
- In the context of this thesis, a type III validation was only done to a very limited degree. In order to compare the efficiency of MTCC to other approaches to acceptance test automation, a controlled experiment that compares MTCC to FIT would be of interest.

14.1.2 Additions to the MTCC Approach

Two directions are possible for future work on the MTCC approach: The addition of new Test Steps and the improvement of existing tooling and changes to the fundamental aspects of the approach:

- The addition of new Test Steps or Services to the MTCC approach is the least complicated way to allow the testing of new functionality with MTCC.

- For this thesis we created the domain and system-level models used by MTCC programatically. The implementation of the tooling to create and edit MTCC models would facilitate the direct involvement of domain experts in the domain engineering and test engineering stages.
- MTCC does not provide a type system for attribute values. More generally, MTCC does not support constraints that would allow to distinguish valid and invalid feature configurations. The expansion of MTCC by a type system or constraints would make it possible to detect and display invalid Test Configurations in the editor.
- Test execution in MTCC is done offline, tests cannot be executed while they are constructed in the editor. If the capability of online test execution was added to MTCC, domain experts could verify that a test behaves as expected during test execution.
- In their present form, a sequence of Test Steps can only be transfered unchanged to a target system. Future work would have to investigate the transfer of tests if an existing sequence of Test Steps cannot be transferred but a semantically equivalent sequence exists.
- One way to guide domain experts in test construction would be this display of the test coverage for a specific Service of context of the SUT.
- A future subject of research could be the utilization of MTCC models to test the conformance of GUI with the WOB model (Krause, 2006b) by defining a suitable domain model that represents the attributes of the WOB model and than test the compatibility of systems with this model.

14.2 Concluding Remarks

MTCC demonstrates that Model-Driven automated acceptance testing can be realized for system families and that such testing is usable by domain experts. The MTCC approach builds on concepts from Model-Driven development and software product-line engineering to allow the involvement of domain experts in a testing process that facilitates the systematic treatment of the variability and commonalities in a system family. MTCC allows automated testing based on models for application domains that are characterized by vague, user-defined requirements which are hard to formalize for fully automatic Model-Driven testing approaches that do not involve domain experts.

In addition to further work on the MTCC approach and its validation, we envision the use of MTCC to test the systems presented in the validation and thereby both increase their quality and save human effort that would otherwise be necessary for test execution.

Appendix

Appendix A

Test Sets

Test Set TS_1

Listing A.1: Excerpt from TS_1 : Tests defined for the MTCC validation

```
<?oxygen RNGSchema="testsets.rnc" type="compact"?>
<out system="sowiport">
    <div category="yes" id="ts_1_0" scenario="ts_1_0.xml">
        <!-- Kein Titelfeld in  ü-->
        <reuse system="iblk" category="no">Kein Titelfeld im IBLK</reuse>
        <reuse system="solr" category="yes"/>
        <!-- Technisch übertragbar, semantisch problematisch wegen der
    verschiedene, getrennten Ergebnislisten-->
        <reuse system="infoconnex" category="yes"/>
        <h2>Test 1(bs) Suche mit Umlaut</h2>
        <p>Dieser Test überprüft die Suche mit dem Umlaut ü</p>
        <ol>
            <li>Aktion In der Suche wird im Feld Titel nach Überleben
          gesucht</li>
            <li>Erwartung In der Trefferliste enthalten alle Titel den
          Begriff Überleben</li>
        </ol>
    </div>
    <div category="yes" id="ts_1_1" scenario="ts_1_1.xml">
        <reuse system="iblk" category="yes"/>
        <!-- Problem mit den unterschiedlichen Listen -->
        <reuse system="infoconnex" category="yes"/>
        <!-- Solr hat keine Anzeige des Queries -->
        <reuse system="solr" category="no"/>
        <h2>Test 2(bs) Phrasensuche</h2>
        <p>Dieser Test überprüft die Phrasensuche</p>
        <ol>
            <li>Aktion In der Suche wird im Feld Titel nach ?geld und geltung?
          gesucht (doppelte
                   Anführungstriche und Kleinschreibung beachten)</li>
            <li>Erwartung In der Anzeige Ihre Suche steht ?geld und geltung?</li>
            <li>Erwartung In der Ergebnisliste erscheinen 3 Treffer</li>
        </ol>
    </div>
    <div category="yes" id="ts_1_2" scenario="ts_1_2.xml">
        <!-- Recent Years Filter ist nur für Sowiport vorhanden -->
        <reuse system="solr" category="can"/>
        <reuse system="infoconnex" category="can"/>
        <reuse system="iblk" category="can"/>
        <h2>Test 3(bs) Suche nach Soziologie im Zeitraum 1992-1997</h2>
        <p>Dieser Test überprüft die Suche in Zeiträumen und dient dem
          Abgleich der erzielten
            Treffermenge mit Erwartungswerten</p>
        <ol>
            <li>Aktion In der Suche wird die Checkbox Nur die letzten 4 Jahre
                                durchsuchen deaktiviert</li>
            <li>Aktion Im Feld Jahr wird die Zeichenfolge 1992−1997
                                eingetragen</li>
            <li>Aktion Im Feld Überall wird Soziologie eingetragen , die Suche
                                wird gestartet</li>
            <li>Erwartung Es werden mehr als 28000 Treffer gefunden</li>
            <li>Erwartung Es werden mehr als 4000 Treffer in Solis gefunden</li>
        </ol>
    </div>
</out>
```

Test Set TS_2

Listing A.2: Excerpt from TS_2 : Tests for an Information Retrieval evaluation

```
<?oxygen RNGSchema="testsets.rnc" type="compact"?>
<out system="fast">
    <div category="yes" id="ts_2_0" scenario="ts_2_0.xml">
        <h2>Anzahl Dokumentzahlen</h2>
        <p>Einsatz CK > Nichteinsatz > 25 Anfragen: mehr Dokumente
                        grundsätzlich gefunden? </p>
    </div>
    <div category="yes" id="ts_2_1" scenario="ts_2_1.xml">
        <h2>Top10 / Top 20 A?nderun</h2>
        <p> Ändern sich bei Einsatz der CK die Dokumente in den Top 10 oder
                        Top 20 gerankten? </p>
    </div>
    <div category="yes" id="ts_2_2" scenario="ts_2_2.xml">
        <h2>Top 10 / Top 20 Reihenfolge</h2>
        <p> Ändert sich bei Einsatz der CK die Reihenfolge von gerankten
                        Dokumenten in den Top 10 /
            Top 20? </p>
    </div>
    <div category="yes" id="ts_2_3" scenario="ts_2_3.xml">
        <h2>Anzahl Datenbanken: insgesamt </h2>
        <p> Werden bei Einsatz der CK mehr Datenbanken in der Suche
                        angesprochen , d.h. werden
            Dokumente aus mehr Datenbanken gefunden? </p>
    </div>
</out>
```

Test Set TS_{3a}

Listing A.3: Excerpt from TS_{3a} : Tests for s_1, defined by domain experts

```
<?oxygen RNGSchema="testsets.rnc" type="compact"?>
<out system="sowiport">
    <div category="yes" id="ts_3a_0" scenario="ts_3a_0.xml">
        <h2>Test 17 (ze) Test der Funktionalität des Suchfelds Jahr /
                        einfache Suche</h2>
        <p>Dieser Test prüft die Funktionalität des Suchfelds Jahr in der
                        einfachen Suche.</p>
        <ol>
            <li>Aktion: in der einfachen Suche Suche mit Titelstichwort:
                                Umweltschutz; Suche starten </li>
            <li>Erwartung: Treffermenge (T1) </li>
            <li>Aktion: in der einfachen Suche Suche mit Titelstichwort:
                                Umweltschutz</li>
            <li>Aktion: nur die letzten vier Jahre durchsuchen deaktivieren </li>
            <li>Aktion: Feld Jahr: 2004 2005 2006 2007 </li>
```

```
          <li>Erwartung: Treffermenge (T2), dabei T2= T1 </li>
          <li>Aktion: in der einfachen Suche Suche mit Titelstichwort:
                         Umweltschutz</li>
          <li>Aktion: nur die letzten vier Jahre durchsuchen deaktivieren</li>
          <li>Aktion: Feld Jahr: 2004-2007 </li>
          <li>Erwartung: Treffermenge (T3), dabei T3=T1=T2</li>
      </ol>
  </div>
  <div category="yes" id="ts_3a_1" scenario="ts_3a_1.xml">
      <h2>Test 18 (ze) Test Trunkierung Titelfeld</h2>
      <p>Dieser Test prüft die Trunkierung im Suchfeld Titel in der
                  einfachen Suche.</p>
      <ol>
          <li>Aktion: einfache Suche im Titelfeld: Drogen </li>
          <li>Erwartung: Treffermenge (T1) </li>
          <li>Aktion: einfache Suche im Titelfeld: Droge*</li>
          <li>Erwartung: Treffermenge (T2), T2&gt;T1</li>
      </ol>
  </div>
  <div category="yes" id="ts_3a_2" scenario="ts_3a_2.xml">
      <h2>Test 19a (ze) Test Verundung von Suchfeldern in der erweiterten
                  Suche / Literatur</h2>
      <p>Dieser Test prüft die Kombination von Suchfeldern
                  (Person und Überall) in der erweiterten
          Suche für den Informationstyp Literatur.</p>
      <ol>
          <li>Aktion: in der erweiterten Suche wird alle deaktiviert</li>
          <li>Aktion: in der erweiterten Suche wird Literatur ausgewählt</li>
          <li>Aktion: im Suchfeld Personen wird Mauss, Marcel eingegeben. </li>
          <li>Erwartung: Anzeige Ihre Suche zeigt Person: Mauss und Marcel;
                         Treffermenge (T1)</li>
          <li>Aktion: im Suchfeld Personen wird Mauss, Marcel eingegeben;
                         im Suchfeld Überall
              Soziologie</li>
          <li>Erwartung: Anzeige Ihre Suche zeigt Person: Mauss und Marcel
                         und Soziologie überall; Treffermenge (T2), dabei T2&lt;t1/></li
      </ol>
  </div>
</out>
```

Test Set TS_{3b}

Listing A.4: Excerpt from TS_{3b} : Tests for s_3, defined by domain experts

```
<?oxygen RNGSchema="testsets.rnc" type="compact"?>
<out system="iblk">
  <div category="yes" id="ts_3b_0" scenario="ts_generic_search.xml">
      <h2>Test 1a(gll) Suche mit Umlaut</h2>
      <p>Dieser Test überprüft die Auflösung des Umlauts ?ü? zu ?ue?</p>
      <ol>
          <li>Aktion In der erweiterten Suche in allen Datenbanken wird im
                         Feld Titel nach Überbevölkerung gesucht</li>
          <li>Erwartung Treffermenge T1</li>
          <li>Aktion In der erweiterten Suche in allen Datenbanken wird
                         im Feld Titel nach Ueberbevölkerung gesucht</li>
          <li>Erwartung Treffermenge T2, dabei T1=T2</li>
      </ol>
  </div>
  <div category="yes" id="ts_3b_1" scenario="ts_generic_search.xml">
      <h2>Test 1b(gll) Suche mit Diakritika</h2>
      <p>Dieser Test überprüft die Suche mit dem Akzent ´</p>
      <ol>
          <li>Aktion In der erweiterten Suche in allen Datenbanken wird
                         im Feld Autor nach Méchet gesucht</li>
          <li>Erwartung Treffermenge T1 (notieren)</li>
          <li>Aktion In der erweiterten Suche in allen Datenbanken wird
                         im Feld Autor nach Mechet gesucht</li>
          <li>Erwartung Treffermenge T2, dabei T1=T2</li>
      </ol>
  </div>
  <div category="yes" id="ts_3b_2" scenario="ts_generic_search.xml">
      <h2>Test 1c(gll) Suche mit Sonderzeichen</h2>
      <p>Dieser Test überprüft die Auflösung des ?ß? zu ?ss?</p>
      <ol>
          <li>Aktion In der erweiterten Suche in allen Datenbanken wird im
                         Feld Suche überall nach Aussenpolitik gesucht</li>
          <li>Erwartung Treffermenge T1 (notieren)</li>
          <li>Aktion In der erweiterten Suche in allen Datenbanken wird
                         im Feld Suche überall nach Außenpolitik gesucht</li>
          <li>Erwartung Treffermenge T2, dabei T1=T2</li>
      </ol>
  </div>
</out>
```

Test Set TS_{4a}

Listing A.5: Excerpt from TS_{4a} : Non-retrievals tests for system s_2

```
<?oxygen RNGSchema="testsets.rnc" type="compact"?>
<out system="infoconnex">
    <div category="external" id="ts_4a_0">
        <h2>Test der Browserfunktionalität</h2>
        <p>Läuft infoconnex auf allen Browsern und Benutzungsoberflächen?</p>
        <ol>
            <li>Internet Explorer</li>
            <li>Opera</li>
            <li>Mozilla Firefox</li>
            <li>Netscape</li>
        </ol>
    </div>


    <div category="external" id="ts_4a_2">
        <h2>Test der Darstellung</h2>
        <p>Stimmen die Style-Sheets? Bsp. Hinweisseite für Nutzer, die nicht
        über eine lizenzierte Bibliothek auf infoconnex zugreifen. "Es␣kann
␣␣␣␣␣␣␣␣nur␣eine␣Kurzinformation␣kostenlos␣angezeigt␣werden" (siehe
        Ausdruck)</p>
    </div>

    <div category="external" id="ts_4a_3">
        <h2>Test der Verfügbarkeitsrecherche</h2>
        <p>Ist die Verfügbarkeitsangabe richtig?</p>
        <ol>
            <li>Funktioniert der Link der Verfügbarkeitsinformation?</li>
            <li>Kann der Artikel wirklich bestellt werden?</li>
            <li>Kann man bei grüner Ampel den Artikel wirklich sofort
         herunterladen?</li>
            <li>Sind kostenlose Volltexte sofort verfügbar?</li>
            <li>Sind diese Volltexte lesbar (PDFs, Schrift,?)?</li>
        </ol>
    </div>

 </out>
```

Test Set TS_{4b}

Listing A.6: Excerpt from TS_{4b} : Retrieval tests for system s_2

```
<?oxygen RNGSchema="testsets.rnc" type="compact"?>
<out system="infoconnex">
    <div category="yes" id="ts_4b_3" scenario="ts_generic_search.xml">
        <h2>Anfrage 3</h2>
        <p>Jahr=1983:</p>
    </div>
    <div category="yes" id="ts_4b_59" scenario="ts_generic_search.xml">
        <h2>Anfrage 59</h2>
        <p>Überall="Soziale␣Ungleichheit"</p>
    </div>
    <div category="yes" id="ts_4b_136" scenario="ts_generic_filtered_search.xml">
        <h2>Anfrage 136</h2>
        <p>Filter: Datenquelle=SOLIS, FORIS; Sprache=englisch Suche:
        Titel=deprivation und Jahr=1993-2003</p>
    </div>
</out>
```

Test Set TS_5

Listing A.7: Excerpt from TS_5 : Tests derived from documented issues for s_1

```
<?oxygen RNGSchema="testsets.rnc" type="compact"?>
<out system="sowiport">
    <div category="external" id="ts_5_4" scenario="">
        <h2>Eintrag 4</h2>
        <p>Artikelübersichtsseite FQS unter Mac/Safari nicht schön (CSS)</p>
    </div>
    <div category="yes" id="ts_5_7" scenario="NOTMODELLED">
        <h2>Eintrag 7</h2>
        <p>in Suchanfragenanzeige Datenbank an sich prüfen kein endloses
                    Hinzufügen gleicher Informationstyp-Navigatoren</p>
    </div>
    <div category="yes" id="ts_5_13" scenario="NOTMODELLED">
        <h2>Eintrag 13</h2>
        <p>Zustandsanzeige: Fehlende Klammerung bei Suche: Schlagwort =
        (jugend or gewalt) and (ehe or familie) ergibt: Schlagwörter: Dasselbe
        Problem tritt auch bei Verwendung mehrerer Zeilen mit demselben
        Suchfeld auf.</p>
    </div>
    <div category="external" id="ts_5_14" scenario="">
        <h2>Eintrag 14</h2>
        <p>Schnellstartleiste: non-js-Version</p>
    </div>
</out>
```

Appendix B

MTCC Editor Handbuch

Dieser Text erläutert die Verwendung des Editors zur Konstruktion von Testfällen für Systemfamilien von Fachinformationssystemen durch Domänenexperten. Der Editor erlaubt die Spezifikation von Tests auf Basis von abstrakten Modellen, dieser Modelle können zeitlich getrennt von der eigentlichen Modellierung automatisch auf das zu testende Sysystem angewandt werden. Die Trennung von Testmodellierung und Testausführung ist wie folgt motiviert:

- Über das World Wide Web zugängliche Applikationen wie Fachinformationssysteme sind häufigen Änderungen unterworfen. Die Änderungen machen die häufige Durchführung von Tests notwendig.

- Die Notwendigkeit häufiger Testdurchführung macht die Automatisierung von Tests notwendig. Manuelles Testen ist mit zu grossem Ressourcenaufwand verbunden um ein System häufig und zugleich umfassend zu testen.

- Automatische Testverfahren müssen das zu testende System ansprechen können und sind damit abhängig von den Schnittstellen die das System bietet, Änderungen an den Schnittstellen machen somit Änderungen an der Testsoftware nötig für jeden betroffen Test nötig.

- Durch die Verwendung von abstrakten, explizit von der Testausführung entkoppelten Modellen kann die Semantik von Tests getrennt von deren Ausführung betrachtet werden.

Während die Verwendung von Modellen zur Testautomatisierung an sich die die Effizienz erhöht, dient die Einbindung von Domänenexperten der Steigerung der Effektivität, stellen also sicher das Tests die tatsächlich an das System gestellten Anforderungen erfüllt. Domänenexperten sind mit diesen Anforderungen vertraut. Es ist wünschenswert das Wissen um die Anforderungen für das Testen nutzbar zu machen. Diesem Ziel steht entgegen das Domänenexperten in der Regel nicht über die Ausbildung und die Zeit verfügen ihr Wissen in formalen Modellen verfügbar zu machen. Der im Folgenden beschriebene Editor verfolgt den Ansatz auch Domänenexperten ohne Modellierungskenntnisse die Konstruktion von Tests zu gestatten indem ausgehend von einer Beschreibung aller relevanten Tests auf einem System eine Oberfläche instanziert wird.

Im vorliegenden Abschnitt werden die Grundlagen des Editors dargestellt. Sowohl auf die Gestaltung der Oberfläche als auch auf die Darstellung eines Tests als Reihe von Aktionen auf dem Testling wird eingegangen. Der folgende Abschnitt ist eine aufgabenorientierte Darstellung der einzelnen Testschritte, den Abschluss dieses Textes bilden vier Beispielszenarien die die Anwendung des Editors illustrieren.

Grundlegende Konzepte

Ein Test im Sinne dieser Arbeit ist eine Abfolge von Testschritten die entweder den Zustand des zu testenden Systems, dem Testling, verändern oder prüfen ob eine Annahme über dessen Zustand zutrifft. Die Automatisierung von Test erlaubt die Ausführung einer großen Anzahl von Tests ohne menschliche Interaktion. Um automatisiert werden zu können muss eine Testbeschreibung alle Details enthalten die zur Ausführung des Tests auf den Schnittstellen des Testings nötig sind. Der hier vorgestellte Editor erlaubt die Konstruktion von automatisierbaren Tests als Folge von Testschritten auf der Grundlage einer formalen Beschreibung des zu testenden Systems.

Eindeutigkeit von Tests

Ein Test muss entscheidbar sein, nach der Durchführung des Test muss der Durchführende, sein es ein menschlicher Tester oder ein technisches System, über alle Informationen verfügen festzustellen ob der Test erfolgreich war oder gescheitert ist. Bei einem erfolgreichen Test ist die Anwendung aller Testschritte gelungen, es gab keine Fehler oder technischen Ausfälle und alle im Test explizit oder implizit enthaltenen Erwartungen wurden erfüllt.

Automatisierung von Tests

Ein Tests ist automatisiert wenn die Folge der durchzuführenden Testschritte und die Kriterien zur Entscheidung über das Gelingen oder Scheitern des Tests intellektuell festgelegt werden, die Ausführung des Tests jedoch automatisch erfolgt. Motivation für die Testautomatisierung ist der Wunsch, umfangreiche Bestände von Tests häufig und reproduzierbar auszuführen. Bei der Testautomatisierung sind die Definition eines Tests und dessen Ausführung zeitlich und technisch getrennt. Die Beschreibung eines automatisierten Tests kann entweder in Form eines Testprogramms in einer gängigen Programmiersprache erfolgen oder mittels einer aufgabenspezifischen Sprache für die Testbeschreibung. Die Verwendung einer aufgabenspezifischen Sprache hat den Vorteils dass die Ausführung von Tests von der Beschreibung der Tests noch weiter entkoppelt werden kann als dies bei der Verwendung eine Programmiersprache möglich ist.

Konzepte des Editors

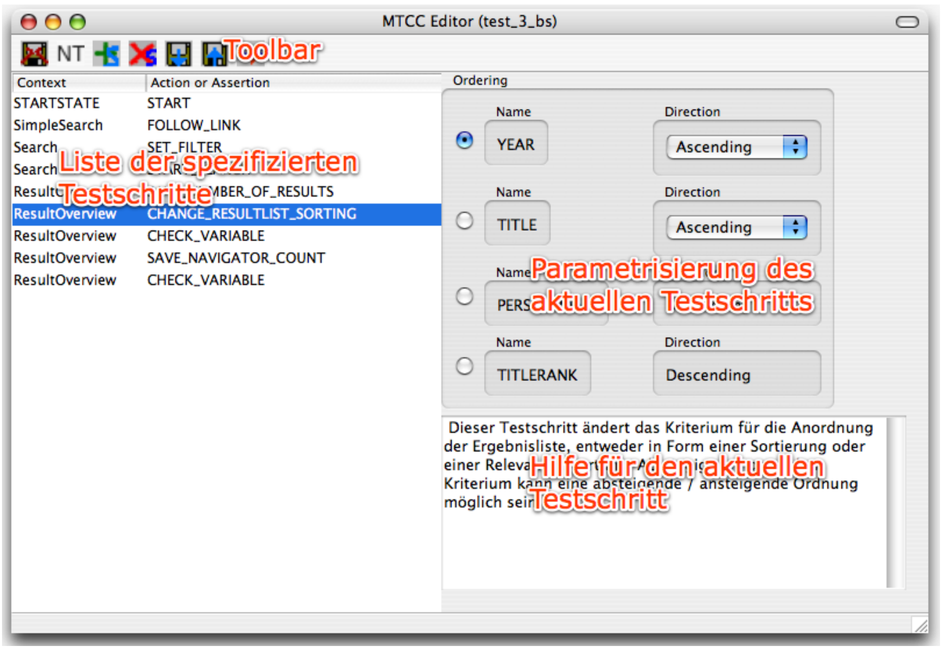
Der hier beschriebene Editor erlaubt eine Beschreibung von Tests als Folge von Testschritten auf Fachinformationssystemen. Die möglichen Reihenfolge von Testschritten wie auch die zur Verfügung stehende Parametrisierung der einzelnen Testschritte werden in einem Modell des Testlings beschrieben. Dieses Modell ist eine abstrakte Darstellung eines zu testenden Systems, beispielsweise des Fachinformationssystems SOWIPORT. Das Modell eines Testlings betrachtet diesen als eine Menge von Kontexten (Startseite, Erweiterte Suche, Ergebnisliste, Detailansicht, etc). Jeder Kontext ist definiert durch eine Menge von Aktionen oder Diensten (Absetzen einer Suche, Setzen von Filtern, Folgen eines Links)

Verwendung des Editors

Im Folgenden wird auf die Verwendung des Editors eingegangen. Die Verwendung des Editors gliedert sich in die folgenden Schritte: Die Ausformulierung eines Testziels, die Festlegung von Testschritten mittels derer sich dieses Testziel erreichen lässt, die Parametrisierung dieser Testschritte und schliesslich das Speichern des Tests. Wichtiger Bestandteil der Beschreibung von Tests ist die Kontrolle der beschriebenen Tests auf Vollständigkeit und Korrektheit. Bestehende Tests können wiederverwendet und angepasst werden.

Ablauf der Testkonstruktion

Abbildung 91 zeigt ein Bildschirmphoto des Editors. Die Benutzungsoberfläche gliedert sich in zwei Hauptbereiche: Eine Liste aller Testschritte des aktuellen Tests in der linken Hälfte und die Oberfläche zur Parametrisierung eines Tests in rechten Hälfte.


MTCC Editor (test_3_bs)
NT
Toolbar
Liste der spezifizierten Testschritte
Parametrisierung des aktuellen Testschritts
Hilfe für den aktuellen Testschritt

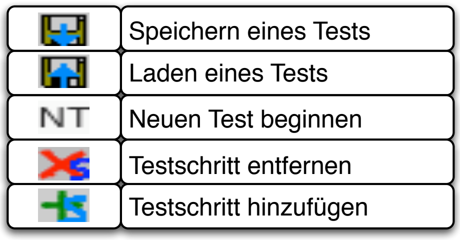
Context	Action or Assertion
STARTSTATE	START
SimpleSearch	FOLLOW_LINK
Search	SET_FILTER
Search	[illegible]
Result	[illegible]MBER_OF_RESULTS
ResultOverview	CHANGE_RESULTLIST_SORTING
ResultOverview	CHECK_VARIABLE
ResultOverview	SAVE_NAVIGATOR_COUNT
ResultOverview	CHECK_VARIABLE

Ordering

	Name	Direction
●	YEAR	Ascending
○	TITLE	Ascending
○	PERS[illegible]	[illegible]
○	TITLERANK	Descending

Dieser Testschritt ändert das Kriterium für die Anordnung der Ergebnisliste, entweder in Form einer Sortierung oder einer Releva[illegible] Kriterium kann eine absteigende / ansteigende Ordnung möglich sei[illegible]

Figure 91: Bildschirmphoto des Editors



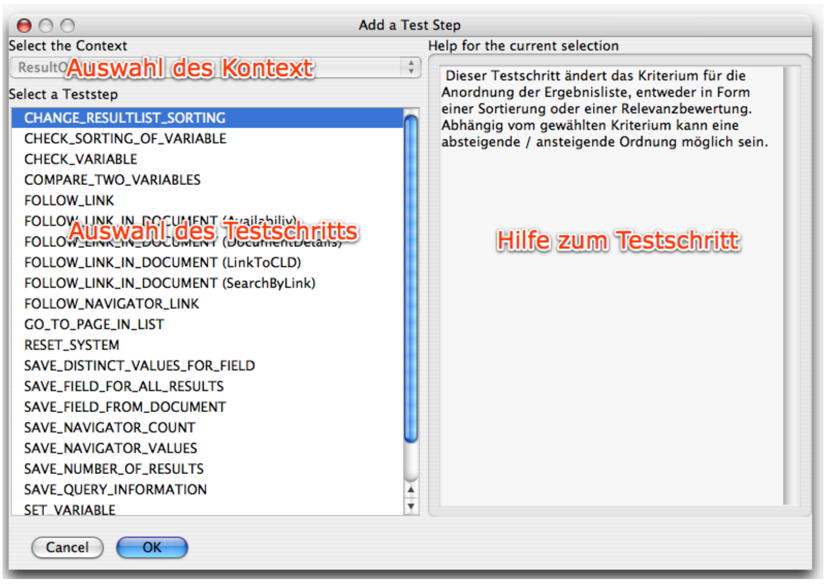
	Speichern eines Tests
	Laden eines Tests
NT	Neuen Test beginnen
	Testschritt entfernen
	Testschritt hinzufügen

Figure 92: Symbole in der Editor Toolbar

Die wichtigsten Kontrollen in der Toolbar des Editors sind in Abbildung 92 zu sehen. Nach dem Start des Editors ist in der Liste der Testschritte ausschliesslich der Startzustand enthalten. Dieser Startzustand dient lediglich der Vergabe eines Namens für den Test und einer kurzen Beschreibung. Durch Betätigen des grünen Plus Symbols wird ein Dialog geöffnet mit dem ein neuer Testschritt zur Liste der bestehenden Testschritte hinzugefügt werden kann. Dieser Dialog ist in Abbildung 93 zu sehen.

Die Auswahl eines Testschritts erfolgt in zwei Schritten, der optionalen Auswahl eines Kontexts der getestet werden soll und der Auswahl eines Testschritt der in diesem Kontext ausgeführt werden soll.

- Die Auswahl eines Kontexts ist dann notwendig wenn der vorangegangene Testschritt mehrere Zielkontexte haben kann, konkret ist es etwa möglich das SOWIPORT als



CHANGE_RESULTLIST_SORTING
CHECK_SORTING_OF_VARIABLE
CHECK_VARIABLE
COMPARE_TWO_VARIABLES
FOLLOW_LINK
FOLLOW_LINK_IN_DOCUMENT (Availabiliy)
FOLLOW_LINK_IN_DOCUMENT (DocumentDetails)
FOLLOW_LINK_IN_DOCUMENT (LinkToCLD)
FOLLOW_LINK_IN_DOCUMENT (SearchByLink)
FOLLOW_NAVIGATOR_LINK
GO_TO_PAGE_IN_LIST
RESET_SYSTEM
SAVE_DISTINCT_VALUES_FOR_FIELD
SAVE_FIELD_FOR_ALL_RESULTS
SAVE_FIELD_FROM_DOCUMENT
SAVE_NAVIGATOR_COUNT
SAVE_NAVIGATOR_VALUES
SAVE_NUMBER_OF_RESULTS
SAVE_QUERY_INFORMATION
SET_VARIABLE

Dieser Testschritt ändert das Kriterium für die Anordnung der Ergebnisliste, entweder in Form einer Sortierung oder einer Relevanzbewertung. Abhängig vom gewählten Kriterium kann eine absteigende / ansteigende Ordnung möglich sein.

Figure 93: Dialog zum Hinzufügen von Testschritten

Folge einer Suche eine Ergebnisliste oder, im Fall einer nicht erfolgreichen Suche, eine Fehlerseite anzeigt. Wenn der Testschritt FOLLOW_LINK aufgerufen wird stehen für den folgenden Testschritt alle Kontexte zur Verfügung die von der Ausgangsseite durch einen beliebigen Link erreicht werden können. Wenn nur ein Kontext für Tests zur Verfügung steht so ist die Auswahlliste für den Kontext im Dialog ausgegraut.

- Die für den Kontext zur Verfügung stehenden Testschritte werden in einer Liste unterhalb der Kontextauswahl angezeigt. Ein Testschritt wird durch Anklicken ausgewählt, für jeden ausgewählten Testschritt wird im linken Bereich des Dialogs eine Hilfe angezeigt.

Im Regelfall gibt es innerhalb eines Kontext nur eine Instanz eines Testschritts, einige Testschritte, insbesondere der Schritt FOLLOW_LINK_IN_DOCUMENT können jedoch mehrmals erscheinen, wobei die Art des verfolgte Links in Klammern hinter dem Namen des Testschritts erscheint.

Verwendung von Testschritten

Nachdem ein Testschritt ausgewählt ist muss dieser Testschritt parametrisiert werden, es müssen also die konkreten Werte festgelegt werden mit denen ein Test aufgerufen wird. Die für eine Parametrisierung zur Verfügung stehende Oberfläche hängt sowohl von der Art des Testschritts als auch von dem zu testenden System ab. Details zu den für jeden Testschritt zu vergebenden Werten wie auch zu Anwendungsszenarien für die Testschritte werden in Abschnitt B gegeben.

Testschritte lassen sich unterscheiden in Aktionen die den Zustand des zu testenden Systems beeinflussen und jene die der Überprüfung von Erwartungen dienen. Aktionen sind etwa das Absetzen einer Suche, das Spezifizieren von Filtern oder das Betätigen eines Links in einem Dokument. Zur Überprüfung von Erwartungen können die Werte von Feldern und die Anzahl von Dokumenten in in Variablen gespeichert und mit Erwartungswerten verglichen werden.

Wenn in der Liste der Testschritte ein Testschritt ausgewählt ist dem noch weitere Testschritte folgen so wird beim Hinzufügen eines neuen Testschritts dieser neue Testschritt an bestehenden Testschritt angehängt. Durch Anklicken des roten X kann ein bestehender, ausgewählter

Testschritt wieder entfernt werden. Wenn durch das Entfernen eines Testschritts Widersprüche entstehen, wenn beispielsweise versucht wird eine Ergebnisliste zu sortieren ohne das zuvor eine Suche abgesetzt wurde, so wird dies durch die Rotfärbung von Testschritten angezeigt deren Folgeschritt nicht ausgeführt werden kann.

Speichern und Laden von Tests

Wenn alle Testschritte ausgewählt und parametrisiert sind, kann ein Test durch das Betätigen des entsprechenden Icons in der Toolbar, der Diskette mit dem nach unten weisenden Pfeil, gespeichert werden. Es wird empfohlen sich bei der Auswahl eines Namens am Namen des Tests zu orientieren der auch in der Beschreibung des Tests verwandt wurde und das Kürzel des Autors des Tests in die Beschreibung mit aufzunehmen. Ebenso empfiehlt es sich zu prüfen ob der gewählte Name bereits vergeben ist; der Editor kontrolliert zur Zeit nicht ob bereits ein Test mit dem gewählten Namen existiert.

Durch Betätigen des Disketten-Symbols mit dem nach oben weisenden Pfeil wird ein bestehender Test geladen, das Symbol mit weissem Hintergrund und der Beschriftung NT legt einen neuen Tests an, der bestehende Tests wird verworfen.

Empfehlungen zur Testmodellierung

Ein Vorgehen zur Verwendung des Editors das nach unseren Erfahrungen den Arbeitsfluss bei der Testmodellierung erleichtert ist es zuerst eine Folge von Testschritten festzulegen um den grundsätzlichen Ablauf des Tests festzulegen und die Parametrisierung der Testschritte erst durchzuführen nachdem alle Testschritte ausgewählt sind. Wird dieser Ansatz gewählt so ist es besonders wichtig alle Testschritte vor dem Speichern des Tests auf ihre korrekte Parametrisierung hin zu prüfen.

Wenn eine Reihe von ähnlichen Tests modelliert werden sollen so kann es sich lohnen einen bereits bestehenden Test zu laden, anzupassen und zu speichern. Dies kann auch in der Form geschehen dass häufig verwandte Tests in Rohform, also noch nicht vollständig parametrisiert, gespeichert werden.

Verfügbare Testschritte

Nach der Vorstellung von Grundkonzepten im vorangegangenen Abschnitt werden nun die einzelnen Testschritte die MTCC zur Verfügung stellt erläutert. Die Testschritte werden in zwei Teilabschnitten behandelt. Im ersten Teilabschnitt werden Testschritte dargestellt die den Zustand des zu testenden Systems beeinflussen oder Ergebniswerte in Variablen speichern, der zweite geht im Detail auf die Verwendung von Variablen aus.

Aktionen auf dem zu Testenden System

Im Folgenden werden Aktionen vorgestellt die zur Manipulation eines zu testenden Systems im Rahmen von Tests verwendet werden order die dazu dienen Ergebnisse von Testaktionen zu speichern. Die Aktionen werden in der Reihenfolge dargestellt in der sie angetroffen werden würden wenn mit dem zu testenden System eine Suche ausgeführt werden soll. Als konkretes Beispiel wird SOWIPORT verwendet.

Statische Links zwischen Seiten

Statische Links zwischen Seiten oder Kontexten sind Links die nicht von den Ergebnissen einer Suche oder einer anderen Form von Interaktion zwischen dem Tester und dem zu testenden System abhängen. Beispiele in Sowiport wären die Links die von der Ergebnisliste zurück zur Suche und zur Erweiterten Suche führen.

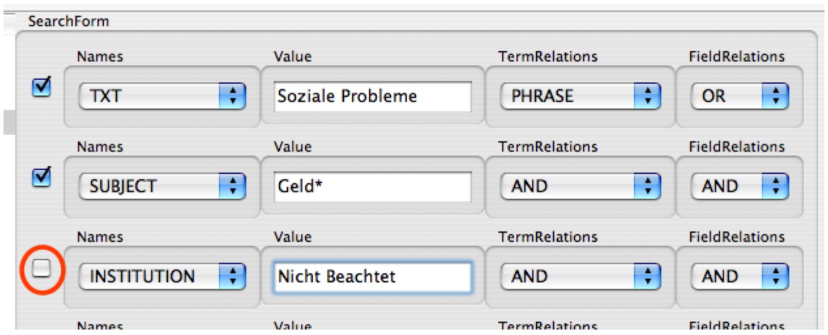
Im Editor werden solche Links mit dem Testschritt FOLLOW_LINK dargestellt. Unabhängig von der Anzahl der Links im aktuellen Kontext gibt es stets nur eine Instanz dieses Testschritts, die Verwendung entspricht somit weniger dem Folgen eines konkreten Links als der Aufforderungen, den aktuellen Kontext mit einem Link zu verlassen. Die Auswahl der Seite die angesprungen werden soll erfolgt indem im nächsten Testschritt ein Zielkontext ausgewählt wird.

Soll beispielsweise im Fachinformationssystem Sowiport der Link von der Ergebnisliste zurück zur Suche genommen werden so wird durch das Betätigen des grünen Plus-Symbols der Dialog zur Auswahl von Testschritten geöffnet und der Testschritt FOLLOW_LINK wird ausgewählt. Der Testschritt selbst hat keine Parametrisierung, die rechte Hälfte des Editors bleibt entsprechend bis auf die Anzeige der Hilfe leer. Nun wird abermals ein Testschritt hinzugefügt, im entsprechenden Dialog stehen die Kontexte Search und AdvancedSearch zur Verfügung.

Absetzen von Suchanfragen

Zum Auslösen von Suchen stehen drei Testschritte zur Verfügung. Die Schritte SET_SEARCH_OPTIONS und SET_FILTER legen die Optionen oder Filter fest die bei einer Suche verwandt werden, die Suche wird aber nicht abgesetzt. Der Schritt START_SEARCH dient sowohl dazu Suchbegriffe für eine Suche festzulegen als auch dazu die Suche abzuschicken

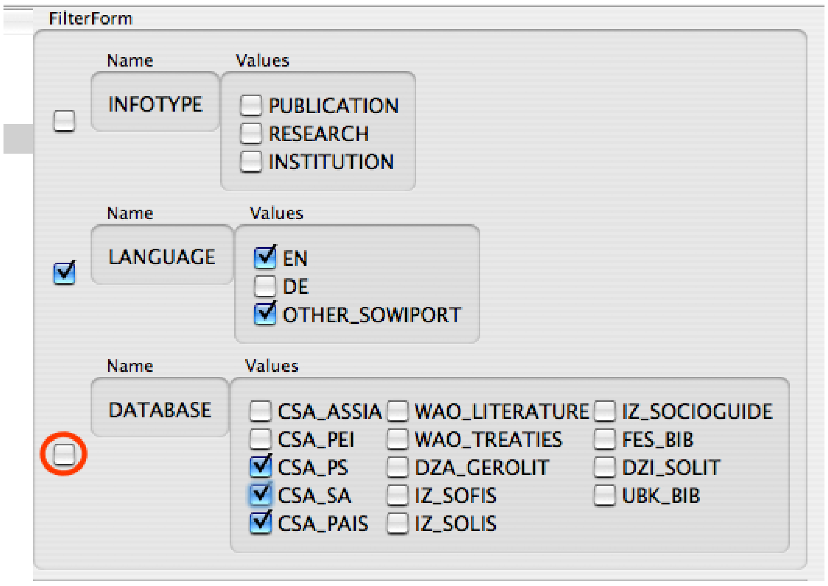
Eine Suchanfrage wird durch den Testschritt START_SEARCH abgeschickt, die mögliche Parametrisierung dieses Testschritt ist abhängig von der jeweils getesteten Suchmaske. Die Abbildung 94 zeigt die Darstellung dieses Testschritts für die Erweiterte Suche in SOWIPORT.


SearchForm

	Names	Value	TermRelations	FieldRelations
☑	TXT	Soziale Probleme	PHRASE	OR
☑	SUBJECT	Geld*	AND	AND
☐	INSTITUTION	Nicht Beachtet	AND	AND
	Names	Value	TermRelations	FieldRelations

Figure 94: Testschritt START_SEARCH der Erweiterten Suche in SOWIPORT.

In Abbildung 94 werden drei Felder des Testschritt gezeigt, im Feld TXT, der Überallsuche, wird nach *Soziale Probleme* gesucht. Die Interpretation der Worte innerhalb eines Suchfelds wird mittels der Kontrolle TermRelations festgelegt, im Beispiel wird eine Phrasensuche ausgeführt. Die Verknüpfung von Feldern untereinander wird mittels der Kontrolle FieldRelations festgelegt, im Beispiel werden das erste Feld und das zweite Feld miteinander verodert. Der Suchbegriff des zweiten Feldes ist *Geld**, es handelt sich um eine trunkierte Suche. Eine wichtige Eigenschaft des Testschritts ist dass nur solche Felder tatsächlich für die Suche verwandt werden die durch Auswahl aktiviert sind. Die ersten beiden Felder sind aktiviert. Das dritte Feld ist,wie in der Abbildung hervorgehoben, nicht aktiviert. Somit wird der Feldinhalt *Nicht Beachtet* bei der Suche ignoriert.



FilterForm

	Name	Values
☐	INFOTYPE	☐ PUBLICATION ☐ RESEARCH ☐ INSTITUTION
☑	LANGUAGE	☑ EN ☐ DE ☑ OTHER_SOWIPORT
☐	DATABASE	☐ CSA_ASSIA ☐ CSA_PEI ☑ CSA_PS ☑ CSA_SA ☑ CSA_PAIS ☐ WAO_LITERATURE ☐ WAO_TREATIES ☐ DZA_GEROLIT ☐ IZ_SOFIS ☐ IZ_SOLIS ☐ IZ_SOCIOGUIDE ☐ FES_BIB ☐ DZI_SOLIT ☐ UBK_BIB

Figure 95: Testschritt SET_FILTER in Sowiport

Mit dem Testschritt SET_FILTER werden die Filter einer Suchanfrage spezifiziert, die Suche wird aber nicht abgeschickt. Abbildung 95 zeigt den Testschritt für das Setzen von Filtern Filter der erweiterten Suche in Sowiport, ebenso wie bei Verwendung von START_SEARCH so muss auch bei diesem Testschritt jedes Filterfeld das für die Suche verwendet werden soll explizit ausgewählt werden.

Der Testschritt SET_SEARCH_OPTIONS dient der Festlegung von Optionen für die Suche, etwa der expliziten Aktivierung von Crosskonkordanzen. In Sowiport stehen keine Suchoptionen zur Verfügung.

Ändern der Sortierung

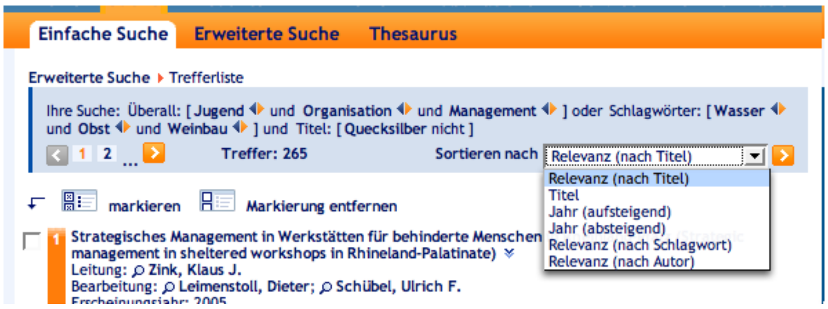
Die Ergebnisliste wird von einer Reihe von Testschritten angesprochen die in der Mehrheit dem Speichern von Daten aus der Liste dienen. Neben diesen Schritten besteht die Möglichkeit in der Ergebnisliste zu blättern und deren Sortierung zu ändern.

Test Testschritt SAVE_QUERY_INFORMATION speichert den Inhalt des Feld des Felds *Ihre Suche* wie dargestellt in der Abbildung 96 in einer Variablen. Die Variable kann in einem späteren Schritt verwandt werden um Annahmen zu überprüfen, etwa um sicherzustellen dass alle Suchbegriffe tatsächlich wieder aufgeführt werden.

Mittels des Testschritts SAVE_NUMBER_OF_RESULTS wird die Anzahl der Ergebnisdokumente für die aktuelle Suche in einer Variablen gespeichert. Grundlage der Ermittlung dieser Anzahl ist die Rückmeldung des getesteten Systems wie in Abbildung 96 zu sehen, eine Zahl der wirklich gelieferten Dokumente durch das Portal findet nicht statt.

Mit dem Testschritt GO_TO_PAGE_IN_LIST kann um eine vorgegebene Anzahl von Seiten in der Ergebnisliste geblättert werden. Dieser Testschritt erwartet als Parameter die Anzahl der zu blätternden Seiten und die Richtung in die zu blättern ist. Die Ausführung des Testschritts scheitert wenn nicht ausreichend viele Seiten zu Verfügung stehen.

Der Testschritt CHANGE_RESULTLIST_SORTING ermöglicht es die Reihenfolge festzule-



Erweiterte Suche ▸ Trefferliste

Ihre Suche: Überall: [Jugend und Organisation und Management] oder Schlagwörter: [Wasser und Obst und Weinbau] und Titel: [Quecksilber nicht]

1 2 ... Treffer: 265 Sortieren nach Relevanz (nach Titel)

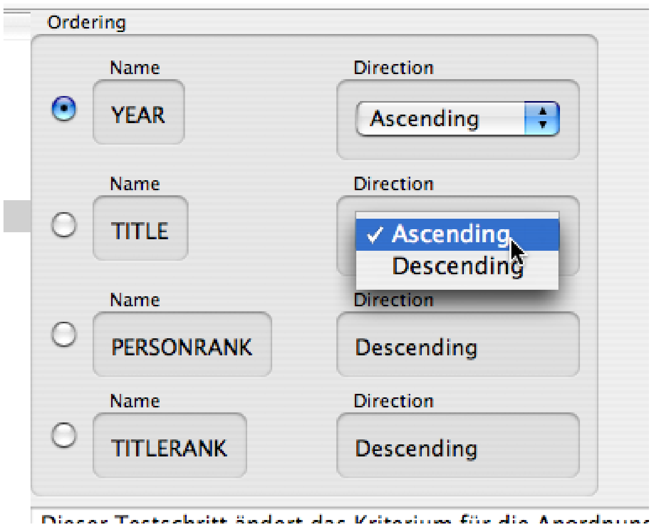
Relevanz (nach Titel)
Titel
Jahr (aufsteigend)
Jahr (absteigend)
Relevanz (nach Schlagwort)
Relevanz (nach Autor)

markieren Markierung entfernen

1 Strategisches Management in Werkstätten für behinderte Menschen management in sheltered workshops in Rhineland-Palatinate)
Leitung: Zink, Klaus J.
Bearbeitung: Leimenstoll, Dieter; Schübel, Ulrich F.

Figure 96: Kopfbereich einer Ergebnisliste in Sowiport

gen in der Ergebnisse angezeigt werden können. Der Testschritt erlaubt sowohl die Auswahl von Sortierungen anhand eines Feldwerts als auch die Anordnung anhand einer von der Anfrage ermittelten Relevanz. Abbildung 97 zeigt die mögliche Parametrisierung des Testschritts für SOWIPORT. Abhängig vom ausgewählten Sortierkriterium ist entweder nur eine absteigende Anordnung möglich oder sowohl ansteigende als auch abfallende Sortierung stehen zur Auswahl.


Ordering

Name	Direction
● YEAR	Ascending
○ TITLE	✓ Ascending / Descending
○ PERSONRANK	Descending
○ TITLERANK	Descending

Figure 97: Optionen zum Beeinflussen der Anzeigereihenfolge in SOWIPORT

Der Testschritt SAVE_FIELD_FROM_DOCUMENT, zu sehen in Abbildung 98 zeigt die mögliche Parametrisierung zur Auswahl eines zu speichernden Feldes aus einem Ergebnisdokument.

Die Parametrisierung des Testschritts wird durch vier Kontrollelemente vorgenommen. Mittels der Auswahlliste Names wird zu speichernde Feld ausgewählt, in diesem Fall im Ergebnisdokument enthaltenen Personen wie etwa Autoren oder Herausgeber. Das Feld DocumentNumber dient dazu festzulegen aus welches Ergebnisdokument von der aktuellen Seite der Ergebnismenge angesprochen wird, bei zehn Dokumenten auf jeder Seite der Ergebnismenge wären mögliche Werte 1 bis 10. Mittels der Kontrolle Field wird festgelegt wie bei mehrfacher Belegung des Felds, etwa bei mehreren Autoren, der in einer Variable zu speichernde Wert ausgewählt wird. Bei Auswahl von AllValues werden alle Belegungen des Feldes gemeinsam in einer Variablen gespeichert, bei Auswahl von Value werden alle Werte gespeichert die identisch mit dem eingegebenen Begriff sind, die Selektion anhand von Teilbegriffen

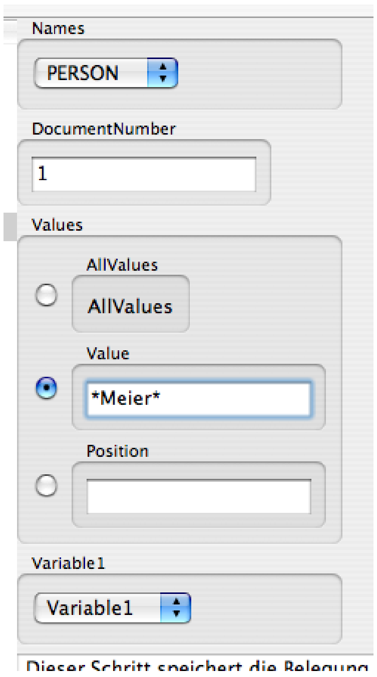

Names
PERSON
DocumentNumber
1
Values
AllValues
AllValues
Value
Meier
Position
Variable1
Variable1
Dieser Schritt speichert die Belegung

Figure 98: Speichern von Feldbelegungen für ein Dokument

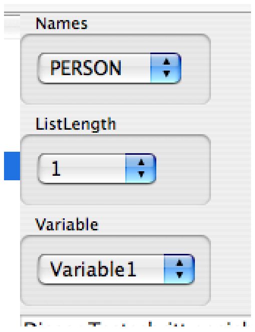

Names
PERSON
ListLength
1
Variable
Variable1

Figure 99: Speichern eines Feldwertes für eine Folge von Dokumenten

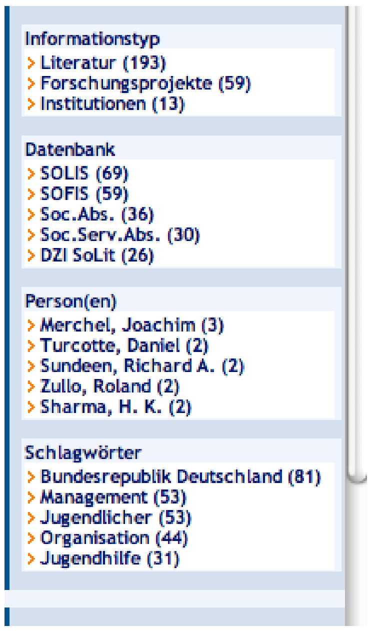
ist mit dem Trunkierungszeichen * möglich. Im in Abbildung 98 gezeigten Beispiel würden alle Autoren gespeichert die den Wert Meier enthalten, also sowohl *August Meier* als auch *Meier, August*. Der letzte Teil der Parametrisierung des Testschritts ist die Angabe zur für die Speicherung zu verwendenden Variablen.

Der Testschritt SAVE_FIELD_FOR_ALL_RESULTS speichert die Belegung eines Feldes für alle Dokumente innerhalb eines Teils der Ergebnisliste, auf diese Weise können etwa alle Titel oder alle Autoren gespeichert werden um zu einem späteren Zeitpunkt zu kontrollieren ob ein bestimmter Autor oder Titel enthalten ist. Abbildung 99 zeigt die mögliche Parametrisierung für diesen Testschritt. Neben dem Feld das gespeichert werden soll und der Variable in der dies geschehen soll muss die Länge der zu speichernden Liste angegeben werden, die Speicherung beginnt mit dem ersten Dokument der Ergebnisliste.

Verwendung von Navigatoren

Navigatoren, auch Facetten genannt, erlauben die Einschränkung einer bestehenden Ergebnismenge. In Sowiport sind Navigatoren über Felder definiert, es gibt beispielsweise Navigatoren für Personen. Abbildung 100 zeigt die verschiedenen Navigatoren für SOWIPORT, durch Betätigen des Links *Literatur* im Navigator *Informationstyp* würde eine bestehende Suche auf Literatur eingeschränkt.

Wie in Abbildung 100 zu sehen ist ist jedem Navigatorwert eine Zählung zugeordnet, im Fall der Datenbank Solis etwa wurden 69 Treffer gefunden. Der Testschritt SAVE_NAVIGATOR_COUNT dient zum Speichern dieser Zählung für einen Navigatorwert in einer Variablen. In Abbildung 101 wird die Schnittstelle zur Parametrisierung des Testschritts gezeigt. Die Auswahl des Navigator erfolgt mittels eines Radiobutton. Der Wert für den die Zählung gespeichert werden soll wird für Felder mit einer kontrollierten und überschaubaren Anzahl von Werten in einer Auswahlliste dargestellt, in der Abbildung ist dies für die Navigatoren DATABASE und INFOTYPE der Fall. Für Navigatoren die sehr viele unterschiedliche Werte annehmen können wie die Felder SUBJECT und PERSON im Beispiel, müssen Werte in ein Textfeld eingetragen werden, auch hier wird die Verwendung des Zeichens * zur Trunkierung unterstützt.

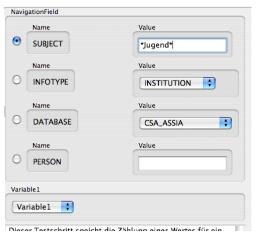

Informationstyp
- Literatur (193)
- Forschungsprojekte (59)
- Institutionen (13)

Datenbank
- SOLIS (69)
- SOFIS (59)
- Soc.Abs. (36)
- Soc.Serv.Abs. (30)
- DZI SoLit (26)

Person(en)
- Merchel, Joachim (3)
- Turcotte, Daniel (2)
- Sundeen, Richard A. (2)
- Zullo, Roland (2)
- Sharma, H. K. (2)

Schlagwörter
- Bundesrepublik Deutschland (81)
- Management (53)
- Jugendlicher (53)
- Organisation (44)
- Jugendhilfe (31)

Figure 100: Navigatoren in Sowiport


NavigationField

	Name	Value
●	SUBJECT	*Jugend*
○	INFOTYPE	INSTITUTION
○	DATABASE	CSA_ASSIA
○	PERSON	

Variable1: Variable1

Dieser Testschritt speicht die Zählung eines Wertes für ein

Figure 101: Speichern eines Feldwertes für eine Folge von Dokumenten

Sollen alle Werte die ein Navigator für ein Feld annimmt ohne die jeweilige Zählung gespeichert werden so wird der Testschritt SAVE_NAVIGATOR_VALUES verwandt. Für diesen Testschritt muss ein zur Verfügung stehenden Navigator ausgewählt werden des Werte in einer Variablen gespeichert werden.

Der Testschritt FOLLOW_NAVIGATOR_LINK dient dazu, einen Navigatorlink auszuwählen und die Ergebnismenge einzuschränken, entspricht also dem Betätigen eines Links. Die in Abbildung 101 dargestellte Schnittstelle für das Speichern von Navigatorwerten wird auch für diesen Testschritt verwandt, da jedoch die Zählung des Navigators nicht gespeichert sondern der Verknüpfung gefolgt wird entfällt die Auswahlmöglichkeit einer Variablen.

Verweise aus Dokumenten

Mit dem Testschritt FOLLOW_LINK_IN_DOCUMENT wird ein Link in einem Ergebnisdokument aufgerufen. In SOWIPORT werden entsprechende Links unteranderem für den Verweis auf die Detailanzeige von Dokumenten, für das Anstossen neuer Suchen nach Personen und Schlagworten und für den Wechsel zu den Collection Level Descriptions verwendet.

Da unterschiedliche Arten von Verweisen innerhalb eines Dokuments vorkommen können bietet der Editor mehrere Instanzen des Testschritt FOLLOW_LINK_IN_DOCUMENT an, die jeweilige Art des Links ist im Dialog zur Auswahl von Testschritten in Klammern hinter dem Names des Testschritts aufgeführt, auf die Bedeutung der einzelnen Links wird im Folgenden eingegangen.

Abbildung 102 zeigt die verschiedenen Verweise in einem Dokument aus der Sowiport Trefferliste farblich hervorgehoben, im Einzelnen existieren die folgenden Verweise.

- Der grün umrandete Titel des Dokuments ist ein Verweis auf die Detailansicht, der entsprechende Testschritt ist durch den Namen DocumentDetails in der Liste der möglichen Testschritte hervorgehoben.

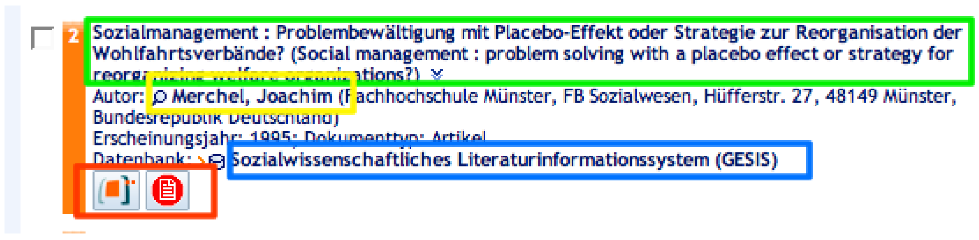

2 Sozialmanagement : Problembewältigung mit Placebo-Effekt oder Strategie zur Reorganisation der Wohlfahrtsverbände? (Social management : problem solving with a placebo effect or strategy for reorganizing welfare organizations?)
Autor: Merchel, Joachim (Fachhochschule Münster, FB Sozialwesen, Hüfferstr. 27, 48149 Münster, Bundesrepublik Deutschland)
Erscheinungsjahr: 1995; Dokumenttyp: Artikel
Datenbank: Sozialwissenschaftliches Literaturinformationssystem (GESIS)

Figure 102: Links in einem SOWIPORT Dokument

- Der gelbe Verweis löst eine neue Suche nach einer Person aus, die Bezeichnung dieses Verweises bei der Auswahl eines Testschritt ist SearchByLink.
- Der blaue Verweise führt zur Beschreibung der Datenbank aus der der entsprechende Treffer stammt, der sogenannten Collection Level Description. Dieser Verweis wird als LinkToCLD bezeichnet.
- Die gelben Verweise führen zu externen Bezugsquellen der durch das Ergebnisdokument repräsentierten Literatur oder Diensten zum Verfügbarkeitsnachweis. In der Liste der Testschritte wird dieser Verweis durch den Titel Availability bezeichnet.

Öffnen der Detailansicht

Aus Sicht des Editors handelt es sich bei der Detailansicht von Dokumenten um eine Liste mit genau einem Dokument. Die Detailansicht unterscheidet sich von der Trefferübersicht darin dass im Fall von Sowiport Aktionen wie das Blättern auf der Ergebnisliste oder die Änderung der Sortierreihenfolge nicht unterstützt werden, um diese Aktionen auszuführen muss die Detailansicht, etwa durch den FOLLOW_LINK, verlassen und die Ergebnisübersicht angesprungen werden.

Als Liste mit nur einem Dokument unterstützt die Detailansicht alle Operationen die auch sonst für Ergebnislisten zur Verfügung stehen, so können Links innerhalb des Dokuments mit dem Testschritt FOLLOW_LINK_IN_DOCUMENT aufgerufen und Feldbelegungen zur späteren Kontrolle gespeichert werden.

Verwendung von Variablen

Variablen werden vom Editor und letztlich in der Testausführung dazu verwendet um Werte für einen späteren Vergleich oder eine Kontrolle zu speichern. Der Editor stellt fünf verschiedene Variablen zur Verfügung die entweder explizit durch die Verwendung eines mit SET_VARIABLE beginnenden Testschritts mit einem von Anwender vergebenen Wert belegt oder durch einen der mit SAVE beginnenden Testschritten zum Speichern von Feldwerten verwandt werden.

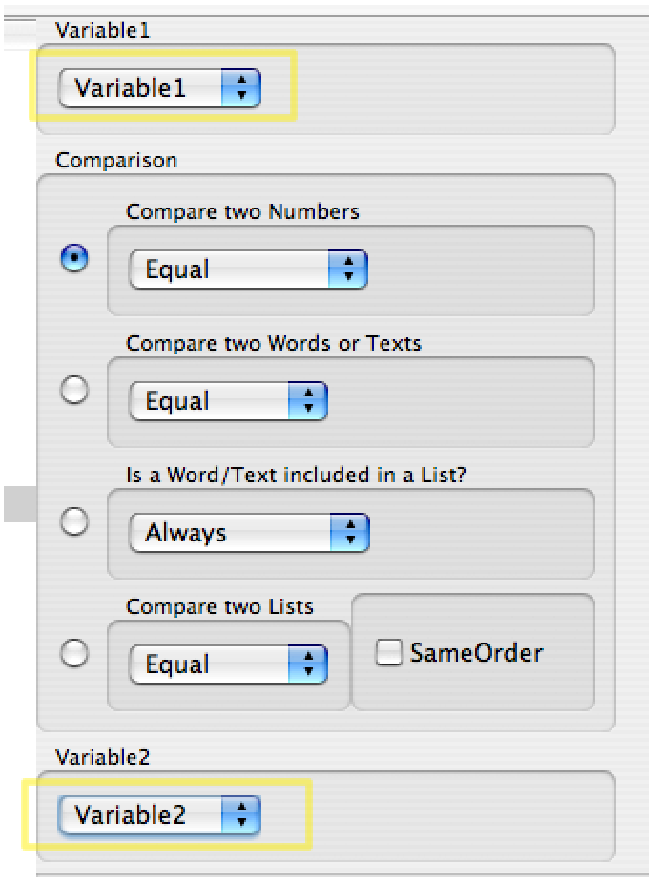
Variablen können Zahlen, Textbestandteile oder Listen von Werten enthalten, abhängig davon von welchem Testschritt eine Variable gefüllt werden. Wird eine bereits gefüllt Variable mit einem Wert belegt wird der enthaltene Wert überschrieben.

Belegen von Variablen

Mittels der drei Testschritte SET_VARIABLE, SET_VARIABLE_LONG_TEXT und SET_VARIABLE_FROM_FILE können Variablen explizit mit Werten belegt werden. Der Testschritt SET_VARIABLE kann eine Variable mit bis zu drei Werten belegen, jeder Wert kann aus mehreren Worten bestehen. Dieser Testschritt wird verwendet wenn etwas das Vorhandensein von mehreren Begriffen in einer Ergebnisliste überprüft werden soll. Der Testschritt SET_VARIABLE_LONG_TEXT erlaubt die Belegung einer Variablen mit einem einzelnen Wert, unterstützt durch einen größeren Textbereich aber auch längere Texte. Der Testschritt SET_VARIABLE_FROM_FILE kann verwandt werden um eine Variable aus mit Werten aus einer Datei zu laden, dies ermöglicht auch die Verwendung von langen Listen von Werten in Variablen.

Vergleich von Variablen

Der Editor stellt eine Reihe von Testschritten zu Verfügung mit denen die Werte in Variablen verglichen werden können. Mit dem Testschritt COMPARE_TWO_VARIABLES werden die Werte von zwei Variablen verglichen. Der Testschritt ist in Abbildung 103 dargestellt. Die Parametrisierung des Testschritts umfasst drei Element: Eine Variable, eine Vergleichsoperation und eine weitere Variable.


Variable1: Variable1

Comparison:
- [✓ Compare two Numbers]: Equal
- Compare two Words or Texts: Equal
- Is a Word/Text included in a List?: Always
- Compare two Lists: Equal; SameOrder ☐

Variable2: Variable2

Figure 103: Testschritt zum Vergleich von zwei Variablen

Insgesamt existieren vier Arten von Vergleichsoperationen, abhängig von dem Inhalt der Variablen.

- Wenn beide Variablen Zahlen enthalten, etwa die Menge von Ergebnissen für eine Anfrage, können diese Zahlen auf Gleichheit und Ungleichheit und auf Größer/Kleiner Beziehungen hin geprüft werden.
- Wenn beide Variablen Worte oder Texte enthalten, etwa Schlagworte, Personennamen oder Titel von Dokumenten, können diese Worte auf Gleichheit und Ungleichheit

verglichen werden. Wenn der Wert einer Variablen mittels einer der SET_VARIBALE Testschritte belegt wurde wird des * Zeichen bei einem Vergleich als Wildcard Symbol interpretiert.

- Wenn eine Variable einen Text und die zweite Variable eine Liste von Texten enthält wird mit der dritten Auswahl das Vorhandensein des Textes in der Liste kontrolliert. Ein Anwendungsfall dieses Vergleichs ist die Kontrolle ob ein bestimmtes Wort in einem oder allen Titeln einer Liste von Ergebnisdokumenten erscheint. Im Einzelnen kann verglichen werden ob ein Begriff immer in einer Liste enthalten ist (Always), niemals auftaucht (Never) oder zumindest einmal auftaucht (OnceOrMore).

- Wenn beide Variablen Liste von Werten enthalten können diese Listen verglichen werden. Mittels der Option Equal und NotEqual kann sichergestellt werden das die Listen die gleichen oder unterschiedliche Elemente enthalten. Mittels der Option PartOf wird überprüft ob alle Werte in der ersten Variablen auch Bestandteil der Werte der zweiten Variablen sind. Wenn die Option Disjunct ausgewählt ist dürfen keine Elemente aus der Liste in der ersten Variablen in der Liste in der zweiten Variablen enthalten sein. Mittels der Checkbox SameOrder wird festgelegt das die zwei Listen nicht nur bezüglich der enthaltenen Dokumente verglichen werden sondern auch in Bezug auf deren Reihenfolge.

Ein wichtiger Punkt beim Vergleich von Variablen ist es sicherzustellen dass die korrekten Variablen ausgewählt sind, der Editor selbst kann nicht sicherstellen dass dies der Fall ist.

Der Testschritt CHECK_VARIABLE entspricht dem Testschritt COMPARE_TWO_VARIABLES mit dem Unterschied dass nicht eine Variable mit einer anderen Variablen verglichen wird sondern dass der Vergleichswert direkt im Testschritt angegeben wird. CHECK_VARIABLE ist somit eine Kombination der Testschritte SET_Variable und COMPARE_TWP_VARIABLES.

Der Testschritt CHECK_SORTING_OF_VARIABLE wird verwendet um die Sortierung einer Liste von Werten in einer Variablen zu kontrollieren. Wenn alle Werte in der Liste Zahlen sind wird numerische Sortierung verwandt, ansonsten wird alphabethisch sortiert.

Der Testschritt CHECK_RELEVANCE_AND_PRECISION_RESULTS wird verwendet um eine Liste von Ergebnisdokumenten in absteigender Relevanz mit einer in einer Variablen gespeicherten Liste von relevanten Dokumenten IDs zu vergleichen, Recall oder Precision Werte an einem Punkt der Liste werden wiederum in einer Variablen gespeichert.

Beispiele

In diesem Abschnitt wird die Verwendung des Editors anhand von zwei Beispiele gezeigt, dem Absetzen einer Suche mit folgender Kontrolle der Menge und des Inhalts der Ergebnisse und einer Suche mit anschliessender Verwendung der Navigatoren und schliesslich einer Suche nach einem spezifischen Dokument. Die hier aufgeführten Beispiele können als Beispiel_Suche.xml, Beispiel_Navigatoren.xml und Beispiel_Detailansicht.xml im Editor geladen werden.

Vergleich von Suchergebnissen

Dieser Test ist ein Beispiel für die Verwendung der erweiterten Suche. Nach einer erweiterten Suche nach Geld* im Titel unter Verwendung von Filtern wird kontrolliert ob die Ergebnismenge zumindest 1000 Dokumente umfasst. Eine zweite Kontrolle stellt sicher dass die ersten 1000 Dokumente ein Wort im Titel haben das mit Geld beginnt.

Der Test setzt sich aus den folgenden Testschritten zusammen:

START Dieser Testschritt enthält die Beschreibung des Tests, der einzig mögliche Startzustand für SOWIPORT ist die Einstiegseite.

FOLLOW_LINK Mit diesem Testschritt wird die Einstiegsseite verlassen, wie beschrieben wird beim Aufruf des Testschritts das Ziel des Verweises nicht angegeben, stattdessen findet eine Auswahl der Zielkontexts bei der Auswahl des folgenden Testschritts statt.

FOLLOW_LINK Dieser zweite Linke verlässt die Suche in Sowiport, Intention ist der Wechsel zur erweiterten Suche und zur dortigen Nutzung der Filter.

SET_FILTER In der erweiterten Suche werden diverse Filter gesetzt, das Setzen dieser Filter schickt noch keine Suche ab, dies geschieht im folgenden Testschritt.

START_SEARCH Mit diesem Testschritt wird die Suche im Titel nach Geld abgesetzt.

SAVE_NUMBER_OF_RESULTS Dieser Testschritt speichert die Länge der Trefferliste in der Variablen 1.

CHECK_VARIABLE Hier wird geprüft ob die Zahl 1000 kleiner ist als der Wert in der Variablen 1, ob die Suche nach Geld also zu mehr als 1000 Ergebnissen führte.

SAVE_FIELD_FOR_ALL_RESULTS Mit diesem Testschritt wird der Titel für die ersten tausend Ergebnisdokumente in der Variablen 1 gespeichert, die vorherige Belegung der Variablen wird überschrieben.

CHECK_VARIABLE Mit diesem Testschritt wird kontrolliert ob alle Titel die in Variable 1 gespeichert sind mindestens ein Wort enthalten das mit Geld* beginnt.

Test der Navigatoren

In diesem Test wird eine Suche auf drei Datenbanken ausgeführt, die Treffermenge in einer der Datenbanken wie vom Navigator angezeigt wird gespeichert. Anschliessend wird dem Verweis dieses Navigators gefolgt und die tatsächliche Ergebnismenge wird mit der vom Navigator gemeldeten Menge verglichen

START , FOLLOW_LINK und FOLLOW_LINK diese Testschritte sind identisch mit denen im vorangegangenen Beispiel. Im Startkontext im ersten Testschritt wird die Beschreibung der Suche hinterlegt. In den beiden folgenden Testschritten wird auf die Erweiterte Suche von SOWIPORT gewechselt.

SET_FILTER Hier werden die Filter gesetzt um die folgende Suche auf die Datenbanken Solis, Sofis und Socioguide einzuschränken.

START_SEARCH Hier wird eine Suche nach dem Prefix *soz** gestartet.

SAVE_NAVIGATOR_COUNT In der Ergebnismenge wird die vom Datenbank-Navigator gemeldeten Anzahl der Treffer für Solis gespeichert.

CHECK_VARIABLE In diesem Schritt wird die tatsächliche Zahl der Ergebnisse mit einem Erwartungswert verglichen. Hier wird kein exakter Vergleich ausgeführt, stattdessen wird der Vergleichsoperator Around verwendet der eine Toleranz von 10 Prozent bei vergleichen zulässt.

FOLLOW_NAVIGATOR_LINK In diesem Testschritt wird der Navigator Verweis auf die Solis Daten ausgelöst, die Ergebnismenge wird also auf all jene Treffer eingeschränkt die aus Solis stammen.

SAVE_NUMBER_OF_RESULTS Hier wird die neue, auf Solis eingeschränkte Ergebnismenge in der Variablen 2 gespeichert.

COMPARE_TWO_VARIABLES Hier werden die Variable 1 mit dem Wert der Treffer aus Solis wie sie im Navigator angezeigt wurden und die Variable 2 mit der Trefferanzahl nach der Einschränkung auf Solis miteinander verglichen.

Kontrolle der Detailansicht

In diesem Test wird aus der Schnellsuche auf der Sowiport Eingangsseite eine Suche nach Jugend abgesetzt. Die Detailansicht des ersten Trefferdokuments wird geöffnet und die Quelle des Dokuments wird mit einem Erwartungswert verglichen.

START In diesem Testschritt werden Titel und Beschreibung des Tests gesetzt.

START_SEARCH In diesem Testschritt wird eine Suche nach *Jugend* mit der Suchleiste auf der Sowiport Einstiegsseite abgesetzt.

FOLLOW_LINK_IN_DOCUMENT Dieser Testschritt öffnet durch das Betätigen des Titelverweises im ersten Ergebnisdokument die Detailansicht für dieses Dokument.

SAVE_FIELD_FROM_DOCUMENT Dieser Testschritt speichert den Inhalt des Quellenfelds des Dokuments in der Detailansicht in einer Variablen. Die Dokumentennummer muss trotz der Tatsache angegeben werden dass die Liste der Dokumente nur die Länge 1 hat, mit der Angabe der der Feldposition 1 wird angegeben dass nur der erste Wert gespeichert für das Dokumentenfeld gespeichert wird.

SET_VARIABLE_LONG_TXT In diesem Testschritt wird die Variable 1 mit einem Erwartungswert für den Inhalt des Quellenfelds belegt. Da der Text des Quellenfelds verhältnismässig lang ist wird der Testschritt zur Vergabe langer, mehrzeiliger Variablenwerte verwendet.

COMPARE_TWO_VARIABLES In diesem Testschritt wird die tatsächliche, in einer Variablen gespeicherte Belegung der Variablen mit dem Erwartungswert verglichen.

Glossary

Action	An action is a *Test Step* that changes the state of the testee.
Assertion	An assertion is a Test Step that does not change the state of the testee but either verifies its state or saves it in a variable.
Buildlet	A class that is used by the MTCC editor to instantiate the GUI-representation for a specific feature model.
Composition	The process in which the DomainTestModel, the SystemFeatureModel and the SystemStateModel are combined into the SystemTestModel.
ConfigurationNode	Node in a Feature Model that represents a specialization step applied to a feature, an attribute or a feature group.
Context	In MTCC, a named set of Service instances that can be addressed by Test Steps for a certain state of the testee.
Domain Engineer	Software engineer who is familiar with the implementation of the systems in a system family and with the most relevant concepts of the domain.
Domain Engineering	In the MTCC context, the analysis of the system family for test relevant aspects, the realization of domain-level models and the implementation of testing core assets.
Domain Expert	Stakeholders, experts, or users of a system who have knowledge about the functional requirements for a system and of the subject matter in general, but have no programming or formal modeling skills.
DomainFeatureModel	Collection of feature models for all Services for a system family. Each Service is represented by a feature model that can be specialized to represent every instance of the Service for every system in the system family.
DomainTestModel	Catalog of all Test Steps that are defined for the Services of a system family.
GuiBuilder	Class that coordinates the construction of a GUI representation for the editor from a feature model.
SameStateAction	Test Step that manipulates the state of the testee, but does not change the current context.
Service	An aspect of the functionality of a system that can be tested in separation, represented by a feature model. Defined in the DomainFeatureModel.
Service Instance	Specialized instance of a Service that represents one concrete Service for a system. Defined in the SystemFeatureModel.
Application Engineering	Phase of the MTCC testing process in which one member of the investigated system family is modeled and the Test Adapter for this system is implemented.
SystemFeatureModel	Collection of specialized feature models that represent the Service instances for a specific system.
SystemStateModel	Finite State Machine that represents the possible sequences of Service invocations for a system.

SystemTestModel	Test-level model that describes both the available Test Step instances for a system and the possible sequences to invoke this instances.
Test Engineering	Phase in MTCC testing process in which an editor instances, based on a SystemTestModel is used to model tests for one system.
Test Runner	(COTS) Program used to execute test cases.
Test Script	Program generated by MTCC. Each test case represents one test and is specific to a test runner and a system under test.
Test Step	Single action or assertions that either exercises a Service verifies its state or saves this state.
Test Step Instance	Test Step which was instantiated for one Service instance of unfolding all references to the Service instance.
Test Adapter	Library of routines that serves as a platform for the execution of test cases for one test runner and one system.
Test Configuration	Model that represents a test as a sequence of configured Test Step instances.

Index

Bibliography

Abiteboul, Serge, Buneman, Peter, & Suciu, Dan. 2000. *Data on the Web: From Relations to Semistructured Data and XML.* Morgan Kaufman.

Alanen, Marcus, Lundkvist, Torbjoern, & Porres, Ivan. 2005. Comparison of Modeling Frameworks for Software Engineering. *Nordic Journal of Computing*, **12**(4), 321–342.

Andersson, Johan, & Bache, Geoff. 2004. The Video Store Revisited Yet Again: Adventures in GUI Acceptance Testing. *In:* Eckstein, Jutta, & Baumeister, Hubert (eds), *Extreme Programming and Agile Processes in Software Engineering 5th International Conference XP 2004 Garmisch-Partenkirchen, Germany, June 2004.* Lecture Notes in Computer Science, vol. 3092. Springer.

Andrea, Jennitta. 2004a. Generative Acceptance Testing for Difficult-to-Test Software. *Chap. Generative Acceptance Testing for Difficult-to-Test Software, pages 29–37 of:* Eckstein, Jutta, & Baumeister, Hubert (eds), *Extreme Programming and Agile Processes in Software Engineering.* Lecture Notes in Computer Science, vol. 3092. Springer.

Andrea, Jennitta. 2004b. Putting a Motor on the Canoo WebTest Acceptance Testing Framework. *In:* Eckstein, Jutta, & Baumeister, Hubert (eds), *Extreme Programming and Agile Processes in Software Engineering 5th International Conference XP 2004 Garmisch-Partenkirchen, Germany, June 2004.* Lecture Notes in Computer Science, vol. 3092. Springer.

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

Antkiewicz, Michal, & Czarnecki, Krzysztof. 2004. FeaturePlugin: Feature Modeling Plug-in for Eclipse. *In: OOPSLA'04 Eclipse Technology eXchange (ETX) Workshop,.*

Arango, Guillermo. 1989. Domain Analysis: From Art Form to Engineering Discipline. *Pages 152–159 of: IWSSD '89: Proceedings of the 5th international workshop on Software specification and design.* New York, NY, USA: ACM Press.

Arango, Guillermo. 1994. A Brief Introduction to Domain Analysis. *Pages 42–46 of: SAC '94: Proceedings of the 1994 ACM symposium on Applied computing.* New York, NY, USA: ACM Press.

Armbrust, O., Ochs, M. A., & Snoek, B. 2004. *Stand der Praxis von Software-Tests und deren Automatisierung : Interviews und Online-Umfrage.* Report; Electronic Publication. Fraunhofer Institut Experimentelles Software Engineering.

Bachmann, Felix, & Clements, Paul C. 2005. *Variability in Software Product Lines.* Tech. rept. Carnegie Mellon Software Engineering Institute.

Baerisch, Stefan. 2007. Model-Driven Test-Case Construction. *Pages 597–591 of: The 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering.*

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

Baker, Paul, Dai, Zhen Ru, Grabowski, Jens, Haugen, Øystein, Lucio, Serge, Samuelsson, Eric, & Williams, Ina Schieferdecker Clay E. 2003. *The UML 2.0 Testing Profile.*

Baresi, Luciano, & Young, Michal. 2001. *Test Oracles.* Tech. rept. Department of Computer and Information Science, University of Oregon.

Basili, Victor R. 1996. The role of experimentation in software engineering: past, current, and future. *Pages 442–449 of: ICSE '96: Proceedings of the 18th international conference on Software engineering.* Washington, DC, USA: IEEE Computer Society.

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

Basili, Victor R., Shull, Forrest, & Lanubile, Filippo. 1999. Building Knowledge through Families of Experiments. *IEEE Trans. Softw. Eng.*, **25**(4), 456–473.

Baudry, Benoit, Fleurey, Franck, France, Robert, & Reddy, Raghu. 2006. Exploring the Relationship between Model Composition and Model Transformation. *In: Report of the 7th International Workshop on Aspect-Oriented Modeling.*

Bayer, Joachim, Flege, Oliver, Knauber, Peter, Laqua, Roland, Muthig, Dirk, Schmid, Klaus, Widen, Tanya, & DeBaud, Jean-Marc. 1999. PuLSE: a methodology to develop software product-lines. *Pages 122–131 of: SSR '99: Proceedings of the 1999 symposium on Software reusability.* New York, NY, USA: ACM Press.

Beck, Kent. 2002. *Test Driven Development: By Example.* Addison-Wesley.

Beck, Kent, & Andres, Cynthia. 2004. *Extreme Programming Explained: Embrace Change (2nd Edition).* Addison-Wesley Professional.

Becker, Steffen. 2008. *Coupled Model Transformations for QoS Enabled Component-Based Software Design.* Ph.D. thesis, Carl von Ossietzky Universität Oldenburg, Oldenburg, Germany.

Beizer, Boris. 1990. *Software testing techniques (2nd ed.).* New York, NY, USA: Van Nostrand Reinhold Co.

Beizer, Boris. 1995. *Black-Box Testing: Techniques for Functional Testing of Software and Systems.* New York, NY, USA: John Wiley & Sons, Inc.

Belkin, Nicholas J., & Croft, W. Bruce. 1987. *Retrieval Techniques.* New York, NY, USA: Elsevier Science Inc. Pages 109–145.

Bertolino, A., Marchetti, E., & Muccini, H. 2004. *Introducing a Reasonably Complete and Coherent Approach for Model-based Testing.*

Bertolino, Antonia. 2007. Software Testing Research: Achievements, Challenges, Dreams. *In: 29th Int. Conference on Software Engineering - Future of Software Engineering(FOSE'07).*

Bertolino, Antonia, & Gnesi, Stefania. 2003a. PLUTO: A Test Methodology for Product Families. *In:* van der Linden, Frank (ed), *Proceedings of the Fifth International Workshop on Product Family Engineering (PFE-5).* LNCS 3014. Siena, Italy: Springer Verlag.

Bertolino, Antonia, & Gnesi, Stefania. 2003b. Use case-based testing of product lines. *Pages 355–358 of: ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering.* New York, NY, USA: ACM Press.

Bézivin, Jean. 2005. On the Unification Power of Models. *Software and Systems Modeling*, **4**(2), 171–188.

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

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

Blackburn, Mark, Busser, Robert, & Nauman, Aaron. 2002. *Understanding the Generations of Test Automation.*

Blair, D.C. 1980. Searching biases in large interactive document retrieval systems. *Journal of the American Society for Information Science*, **31**(4), 271–277.

Botaschanjan, Jewgenij, Pister, Markus, & Rumpe, Bernhard. 2003. Testing Agile Requirements Models. *In: In: Proceedings of the First Hangzhou-Lübeck Conference on Software Engineering (HL-SE 03).*

Bracha, Gilad, & Cook, William. 1990. Mixin-based Inheritance. *Pages 303–311 of: OOPSLA/ECOOP '90: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications.* New York, NY, USA: ACM.

Briand, L. C., Labiche, Y., & Cui, J. 2005. Automated support for deriving test requirements from UML statecharts. *Software and Systems Modeling*, **4**, 399–423.

Brooks, Fred P. 1987. No Silver Bullet - essence and accident in software Engineering Computer. *Computer*, **20**(4), 10–19.

Brown, Alan W. 2004. Model driven architecture: Principles and practice. *Software and System Modeling*, **3**(4), 314–327.

Brown, C. Titus, Gheorghiu, Grig, & Huggins, Jason R. 2007. *An Introduction to Testing Web Applications with twill and Selenium.* O'Reilly Media, Inc.

Broy, M., & Rausch, A. 2005. Das neue V-Modell® XT. *Informatik-Spektrum*, **28**(3), 220–229.

Budinsky, Frank, Steinberg, David, Merks, Ed, Ellersick, Raymond, & Grose, Timothy J. 2003. *Eclipse Modeling Framework.* Addison Wesley Professional.

Buschmann, Frank, Meunier, Regine, Rohnert, Hans, Sommerlad, Peter, & Stal, Michael. 1996. *Pattern-Oriented Software Architecture Volume 1: A System of Patterns.* Wiley.

Cabot, Jordi, & Teniente, Ernest. 2006. Constraint Support in MDA tools: a Survey. *In:* Rensink, Arend, & Warmer, Jos (eds), *Model Driven Architecture - Foundations and Applications Second European Conference, ECMDA-FA 2006 Bilbao, Spain, July 10-13, 2006.* Lecture Notes in Computer Science, no. 4066. Springer.

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

Cechticky, V., Pasetti, A., Rohlik, O., & Schaufelberger, W. 2004. XML-Based Feature Modelling. *Pages 5–9 of: Software Reuse: Methods, Techniques and Tools: 8th International Conference, ICSR*. Springer.

Chakrabarti, S. 2003. *Mining the Web: Discovering Knowledge from Hypertext Data*. Morgan Kaufmann.

Chase, R.B., Aquilano, N.J., & Jacobs, F.R. 2001. *Operations Management for Competitive Advantage*. McGraw-Hill Irwin.

Chen, T. Y. 2005. Are Successful Test Cases Useless or Not? *Pages 2–3 of:* Reussner, Ralf, Mayer, Johannes, Stafford, Judith A., Overhage, Sven, Becker, Steffen, & Schroeder, Patrick J. (eds), *QoSA/SOQUA*. Lecture Notes in Computer Science, vol. 3712. Springer.

Clements, Paul, & Northrop, Linda M. 2001. *Software Product Lines : Practices and Patterns*. 3rd edn. Addison Wesley.

Cleve, H., & Zeller, A. 2000. Finding Failure Causes through Automated Testing. *In:* Ducasse, M. (ed), *Proceedings of the Fourth International Workshop on Automated Debugging*.

Condron, Chris. 2004. Domain Approach to Test Automation of Product Lines. *Pages 27–35 of:* Geppert, Birgit, Krueger, Charles, & Li, Jenny (eds), *Proceedings of the International Workshop on Software Product Line Testing (SPLiT 2004)*.

Cooper, W.S. 1973. On Selecting a Measure of Retrieval Effectiveness. *Journal of the American Society for Information Science*, **24**(6), 413–424.

Coplien, James, Hoffman, Daniel, & Weiss, David. 1998. Commonality and Variability in Software Engineering. *IEEE Software*, **15**(6), 37–45.

Cordy, James R., Lämmel, Ralf, & Winter, Andreas. 2006. Executive Summary – Transformation Techniques in Software Engineering. *In: Transformation Techniques in Software Engineering*. Dagstuhl Seminar Proceedings, vol. 05161. Dagstuhl: Internationales Begegnungs- und Forschungszentrum (IBFI), Schloss Dagstuhl, Germany.

Crosby, P.B. 1979. *Quality is Free: The Art of Making Quality Certain*. McGraw-Hill.

Czarnecki, K., & Kim, C.H.P. 2005. Cardinality-Based Feature Modeling and Constraints: a Progress Report. *In: International Workshop on Software Factories*.

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

Czarnecki, Krzysztof. 2005. Overview of Generative Software Development. *Pages 313–328 of:* et al:, J.-P. Banâtre (ed), *Unconventional Programming Paradigms (UPP) 2004*. Lecture Notes of Computer Science, vol. 3566. Mont Saint-Michel, France: Springer.

Czarnecki, Krzysztof, & Antkiewicz, Michal. 2005. Mapping Features to Models: A Template Approach Based on Superimposed Variants. *In: International Conference on Generative Programming and Component Engineering*.

Czarnecki, Krzysztof, & Helsen, Simon. 2003. Classification of Model Transformation Approaches. *In: OOPSLA'03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*.

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

Czarnecki, Krzysztof, Helsen, Simon, , & Eisenecker, Ulrich. 2005b. Staged Configuration Through Specialization and Multi-Level Configuration of Feature Models. *Software Process Improvement and Practice, special issue on "Software Variability: Process and Management,*, **143 - 169**, 10(2).

Czarnecki, Krzysztof, Kim, Chang Hwan Peter, & Kalleberg, Karl Trygve. 2006a. Feature Models Are Views on Ontologies. *Pages 41–51 of: Proceedings of 10th International Software Product Line Conference (SPLC 2006).*

Czarnecki, Krzysztof, Antkiewicz, Michal, & Kim, Chang Hwan Peter. 2006b. Multi-level customization in application engineering. *Commun. ACM*, **49**(12), 60–65.

Dai, Zhen Ru. 2004. Model-Driven Testing with UML 2.0. *Pages 179–188 of: Second European Workshop on Model Driven Architecture (MDA) EWMDA.*

DeBaud, Jean-Marc, & Schmid, Klaus. 1999. A systematic approach to derive the scope of software product-lines. *Pages 34–43 of: ICSE '99: Proceedings of the 21st international conference on Software engineering.* Los Alamitos, CA, USA: IEEE Computer Society Press.

Deufemia, Vincenzo, Ferrucci, Filomena, & Gravino, Carmine. 2006. Constructing Meta-CASE Workbenches by Exploiting Visual Language Generators. *IEEE Trans. Softw. Eng.*, **32**(3), 156–175. Member-Gennaro Costagliola.

Deursen, Arie, Moonen, Leon, Bergh, Alex, & Kok, Gerard. 2001. *Refactoring Test Code.*

Dijkstra, E.W. 1969. *Notes on structured programming.* Technological University Eindhoven, Netherlands Department of Mathematics, [Eindhoven].

Dmitriev, Sergey. 2004. *Language Oriented Programming: The Next Programming Paradigm.*

Do, Hyunsook, Elbaum, Sebastian, & Rothermel, Gregg. 2005. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empirical Software Engineering*, **10**(4), 405–435.

Duvall, Paul, Matyas, Steve, & Glover, Andrew. 2007. *Continuous Integration: Improving Software Quality and Reducing Risk.* Addison-Wesley Professional.

Edwards, C.D. 1968. The meaning of quality. *Quality Progress*, **1**(10), 36–39.

Eibl, M. 2000. *Visualisierung im Document Retrieval: theoretische und praktische Zusammenführung von Softwareergonomie und Graphik Design.* Ph.D. thesis, GESIS, Informationszentrum Sozialwissenschaften.

Eichmann, D. 1997. Representing Knowledge in Domain Engineering. *In: Proceedings of the Eighth Workshop on Institutionalizing Software Reuse.*

El-Far, Ibrahim K., & Whittaker, James A. 2001. Model-Based Software Testing. *Pages 825–837 of:* Marciniak, J.J. (ed), *Encyclopedia on Software Engineering.* Wiley.

Ellims, Michael, Bridges, James, & Ince, Darrel C. 2006. The Economics of Unit Testing. *Empirical Softw. Engg.*, **11**(1), 5–31.

Emerson, Matthew, & Sztipanovits, Janos. 2006. Techniques for Metamodel Composition. *In: Proceedings of 6th OOPSLA Workshop on Domain-Specific Modeling (DSM'06).* Computer Science and Information System Reports, Technical Reports, no. 37. University of Jyväskylä.

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

Evans, Robert B., & Savoia, Alberto. 2007. Differential testing: a new approach to change detection. *Pages 549–552 of: ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering.* New York, NY, USA: ACM Press.

for Standardization, International Organization. 2001. *ISO 9126 Software engineering – Product quality – Part 1: Quality model.*

Fowler, Martin. 2003. *Patterns of Enterprise Application Architecture.* Addison-Wesley.

Fowler, Martin. 2005a. *A Language Workbench in Action - MPS.*

Fowler, Martin. 2005b. *Language Workbenches and Model Driven Architecture.*

Fowler, Martin. 2005c. *Language Workbenches: The Killer-App for Domain Specific Languages?*

Fowler, Martin. 2006. *Specification By Example.*

France, Robert, & Rumpe, Bernhard. 2007. Model-Driven Development of Complex Software: A Research Roadmap. *In: Future of Software Engineering(FOSE'07).*

Freiling, F., Eusgeld, I., & Reussner, R. (eds). 2008. *Dependability Metrics.* Lecture Notes in Computer Science. Springer-Verlag.

Gälli, Markus. 2006. *Modeling Examples to Test and Understand Software.* Ph.D. thesis, Universität Bern.

Gamma, Erich, Helm, Richard, Johnson, Ralph, & Vlissides, John. 1995. *Design Patterns: Elements of Reusable Object-Oriented Softwar.* Addison-Wesley Professional.

Geppert, Birgit, Krueger, Charles, & Li, J. Jenny (eds). 2004a. *SPLiT 2004 International Workshop on Software Product Line Testing.*

Geppert, Birgit, Li, Jenny, Rößler, Frank, & Weiss, David M. 2004b. Towards Generating Acceptance Test Cases for Product Lines. *In: Proceedings of the Eighth International Conference on Software Reuse.*

Geppert, Birgit, Krueger, Charles, & Trew, Tim (eds). 2005. *SPLiT 2005 2nd International Workshop on Software Product Line Testing.*

Gilb, T., Graham, D., & Finzi, S. 1993. *Software Inspection.* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.

Graham, Dorothy, & Fewster, Mark (eds). 2000. *Software Test Automation: Effective Use of Test Execution Tools.* Addison-Wesley.

Greenfield, Jack, & Short, Keith. 2004. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools.* Wiley.

Grönniger, H., Krahn, H., Rumpe, B., Schindler, M., & Völkel, S. 2006. *MontiCore 1.0 - Ein Framework zur Erstellung und Verarbeitung domänenspezifischer Sprachen.* Tech. rept. Technische Universität Braunschweig, Carl-Friedrich-Gauss-Fakultät für Mathematik und Informatik.

Gurp, Jilles Van, Bosch, Jan, & Svahnberg, Mikael. 2001. On the Notion of Variability in Software Product Lines. *Page 45 of: WICSA '01: Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA '01).* Washington, DC, USA: IEEE Computer Society.

Gutierrez, Javier J., Escalona, Maria J., Mejias, Manuel, & Torres, Jesus. 2006. *Generation of test cases from functional requirements. A survey.* Tech. rept. Department of Computer Languages and Systems, University of Seville.

Hannay, Jo E., Hansen, Ove, Kampenes, Vigdis By, Karahasanovic, Amela, Liborg, Nils-Kristian, & Rekdal, Anette C. 2005. A Survey of Controlled Experiments in Software Engineering. *IEEE Trans. Softw. Eng.*, **31**(9), 733–753. Member-Dag I. K. Sjoberg.

Harel, D., & Rumpe, B. 2004. Meaningful Modeling: What's the Semantics of Semantics? *Computer*, **37**(10), 64–72.

Hartmann, Jean, Vieira, Marlon, & Ruder, Axel. 2004. A UML-based Approach for Validating Product Lines. *Pages 58–65 of:* Geppert, Birgit, Krueger, Charles, & Li, Jenny (eds), *Proceedings of the International Workshop on Software Product Line Testing (SPLiT 2004).*

Hasselbring, W. 1997. Federated integration of replicated information within hospitals. *International Journal on Digital Libraries*, **1**(3), 192–208.

Hasselbring, W. 2002a. Component–Based Software Engineering. *Pages 289–305 of: Handbook of Software Engineering and Knowledge Engineering.* World Scientific Publishing.

Hasselbring, W. 2002b. Web Data Integration for E-Commerce Applications. *IEEE MULTIMEDIA*, 16–25.

Heinold, Ehrhardt F. 2007. *Virtuelle Fachbibliotheken im System der überregionalen Literatur- und Informationsversorgung : Studie zu Angebot und Nutzung der Virtuellen Fachbibliotheken* . Tech. rept. Heinold, Spiller & Partner Unternehmensberatung.

Heinz, Sabine, & Stempfhuber, Maximilian. 2007. Eine Informationsarchitektur für wissenschaftliche Fachportale in vascoda. *Pages 485–507 of:* Oßwald, Achim, Stempfhuber, Maximilian, & Wolff, Christian (eds), *Open Innovation: neue Perspektiven im Kontext von Information und Wissen; Beiträge des 10. internationalen Symposiums für Informationswissenschaft und der 13. Jahrestagung der IuK-Initiative Wissenschaft.* Schriften zur Informationswissenschaft.

Hellweg, H., Krause, J., Mandl, T., Marx, J., Müller, M.N.O., Mutschke, P., & Strötgen, R. 2001. *Treatment of Semantic Heterogeneity in Information Retrieval.* Technical Report 23. GESIS-IZ Sozialwissenschaften.

Herrington, Jack. 2003. *Code Generation in Action.* Manning Publications.

Heymans, Patrick, Schobbens, Pierre-Yves, & Trigau, Jean-Christophe. 2007. Towards the Comparative Evaluation of Feature Diagram Languages. *Pages 1–16 of:* Männistö, Tomi, Niemelä, Eila, & Raatikainen, Mikko (eds), *Software and Service Variability Management Workshop - Concepts, Models, and Tools.* Research Reports, no. 3. Helsinki University of Technology, Software Business and Engineering Institute.

Hicinbothom, J. H., & Zachary, W. W. 1993. A tool for automatically generating transcripts of human-computer interaction. *Page 1042 of: Proceedings of the Human Factors and Ergonomics Society 37th Annual Meeting.*

Hoffman, Daniel, & Strooper, Paul. 2000. Prose + Test Cases = Specifications. *Page 239 of: TOOLS '00: Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 34'00).* Washington, DC, USA: IEEE Computer Society.

Hohpe, G., Woolf, B., & Brown, K. 2004. *Enterprise integration patterns.* Addison-Wesley Boston.

Hudak, Paul. 1996. Building domain-specific embedded languages. *ACM Computing Surverys*, **28**(4es), 196.

Janzen, David, & Saiedian, Hossein. 2005. Test-Driven Development: Concepts, Taxonomy, and Future Direction. *Computer*, **38**(9), 43–50.

Juran, J.M., & Gryna, F.M. (eds). 1988. *Juran's quality control handbook.* McGraw-Hill.

Kalfoglou, Y., & Schorlemmer, M. 2003. Ontology mapping: the state of the art. *The Knowledge Engineering Review*, **18**(01), 1–31.

Kan, Stephen H. 2002. *Metrics and Models in Software Quality Engineering.* 2nd edn. Addison-Wesley Longman.

Kaner, Cem. 2003 (May 12-16). What is a good test case? *In: Software Testing Analysis & Review Conference (STAR) East.*

Kaner, Cem, Bach, James, & Pettichord, Bret. 2001. *Lessons Learned in Software Testing.* Wiley.

Kang, K., Cohen, S., Hess, J., Novak, W., & Peterson, A. 1990 (November). *Feature-oriented domain analysis (FODA) feasibility study.* Tech. rept. CMU/SEI-90-TR-021. Software Engineering Institute, Carnegie-Mellon University,.

Karagiannis, Dimitris, & Kühn, Harald. 2002. Metamodelling Platforms. *In: Proceedings of the Third International Converence EC-Web.*

Kettemann, Stefan, Muthig, Dirk, & Anastasopoulos, Michalis. 2003. *Product Line Implementation Technologies : Component Technology View.* Tech. rept. IESE-Report No. 015.03/E. Fraunhofer Institut Experimentelles Software Engineering.

Kim, Chang Hwan Peter, & Czarnecki, Krzysztof. 2005. Synchronizing Cardinality-Based Feature Models and their Specializations. *Pages 331–348 of:* Hartman, A., & Kreische, D. (eds), *Proceedings of European Conference on Model Driven Architecture – Foundations and Applications (ECMDA-FA'05).* Lecture Notes in Computer Science, no. 3748. Nuremberg, Germany: Springer.

Kim, Soon-Kyeong, Wildman, Luke, & Duke, Roger. 2005. A UML Approach to the Generation of Test Sequences for Java-Based Concurrent Systems. *Pages 100–109 of: Australian Software Engineering Conference.* IEEE Computer Society.

Kitchenham, Barbara A., Dyba, Tore, & Jorgensen, Magne. 2004. Evidence-Based Software Engineering. *Pages 273–281 of: ICSE '04: Proceedings of the 26th International Conference on Software Engineering.* Washington, DC, USA: IEEE Computer Society.

Kolb, Ronny, & Muthig, Dirk. 2006. Making testing product lines more efficient by improving the testability of product line architectures. *Pages 22–27 of: ROSATEA '06: Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis.* New York, NY, USA: ACM Press.

Koll, M. 2000. Information Retrieval. *BULLETIN-AMERICAN SOCIETY FOR INFORMATION SCIENCE*, **26**(2), 16–17.

Koziolek, Heiko. 2008 (January). *Parameter Dependencies for Reusable Performance Specifications of Software Components*. Ph.D. thesis, Carl von Ossietzky Universität Oldenburg.

Krahn, Holger, Rumpe, Bernhard, & Völkel, Steven. 2006. Roles in Software Development using Domain Specific Modelling Languages. *In: Proceedings of 6th OOPSLA Workshop on Domain-Specific Modeling (DSM'06)*. Computer Science and Information System Reports, Technical Reports, no. 37. University of Jyväskylä.

Krause, J. 1996 (September). *Informationserschliessung und -bereitstellung zwischen Deregulation, Kommerzialisierung und weltweiter Vernetzung - Schalenmodell*. Tech. rept. 6. GESIS-IZ Sozialwissenschaften.

Krause, Jürgen. 2006a. Shell Model, Semantic Web and Web Information Retrieval. *Pages 95 – 106 of:* Harms, Ilse, Luckhardt, Heinz-Dirk, & Giessen, Hans W. (eds), *Information und Sprache: Beiträge zu Informationswissenschaft, Computerlinguistik, Bibliothekswesen und verwandten Fächern; Festschrift für Harald H. Zimmermann*. München: Saur.

Krause, Jürgen. 2006b. Visual Interaction on the Basis of the WOB-Model. *In:* Rapp, Reinhad, Seldmeier, Peter, & Zunker-Rapp, Gisela (eds), *Perspectives on Cognition - A Festschrift for Manfred Wettler*. Pabst Science Publishers.

Kruchten, P. 2003. *The Rational Unified Process: An Introduction*. Addison-Wesley Professional.

Kulak, Daryl, & Guiney, Eamonn. 2003. *Use Cases: Requirements in Context*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Larman, Craig. 2004. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. 3rd edn. Pretentice Hall PTR.

Larman, Craig, & Basili, Victor R. 2003. Iterative and Incremental Development: A Brief History. *Computer*, **36**(6), 47–56.

Lewandowski, Dirk. 2007. Mit welchen Kennzahlen lässt sich die Qualität von Suchmaschinen messen? *In:* Machill, Marcel, & Beiler, Markus (eds), *Die Macht der Suchmaschinen / The Power of Search Engines*. Köln: Herbert von Halem Verlag.

Lewandowski, Dirk, & Hoechstoetter, Nadine. 2007. Qualitätsmessung bei Suchmaschinen: System- und nutzerbezogene Evaluationsmaße. *Informatik Spektrum*, **30**(3).

Li, Ping, Huynh, Toan, Reformat, Marek, & Miller, James. 2007. A practical approach to testing GUI systems. *Empirical Software Engineering*, **12**(4), 331 – 357.

Linzhang, Wang, Jiesong, Yuan, Xiaofeng, Yu, Jun, Hu, Xuandong, Li, & Guoliang, Zheng. 2004. Generating Test Cases from UML Activity Diagram based on Gray-Box Method. *apsec*, **00**, 284–291.

Loughran, Neil, Groher, Iris, & Rashid, Awais. 2006. Good Practice Guidelines for Code Generation in Software Product Line Engineering. *In:* Clements, Paul, & Muthig, Dirk (eds), *Variability Management – Working with Variability Mechanisms:Proceedings of the Workshop held in conjunction with the 10th Software Product Line Conference (SPLC-2006)*.

Louridas, Panagiotis. 2006. Static Code Analysis. *Software, IEEE*, **23**(4), 58–61.

Ludewig, Jochen. 2003. Models in software engineering – an introduction. *Software and Systems Modeling*, **2**, 5–14.

Maletic, Jonathan I., Soliman, Khalid S., Moreno, Manuel A., & Mercer, William M. 2000. *Identification of Test Cases from Business Requirements of Software Systems.*

Mandl, Thomas. 2008. *Automatische Bewertung der Qualität von Web-Seiten.* Habilitationsschrift.

Manning, Christopher D., Raghavan, Prabhakar, & Schütze, Hinrich. 2008. *Introduction to Information Retrieval.* Cambridge University Press.

Martens, Anne. 2007 (November). *Empirical Validation of the Model-Driven Performance Prediction Approach Palladio.* M.Phil. thesis, Carl von Ossietzky Unversität Oldenburg.

Matinlassi, Mari. 2004. Comparison of Software Product Line Architecture Design Methods: COPA, FAST, FORM, KobrA and QADA. *Pages 127–136 of: ICSE '04: Proceedings of the 26th International Conference on Software Engineering.* Washington, DC, USA: IEEE Computer Society.

Mayr, Philipp. 2006. *IZ-Arbeitsbericht Informationsangebote für das Wissenschaftsportal vascoda - eine Bestandsaufnahme.* Tech. rept. 37. In formationszentrum Sozialwissenschaften der Arbeitsgemeinschaft Sozialwissenschaftlicher Institute e.V. (ASI).

Mayr, Philipp, & Petras, Vivien. 2008. Cross-concordances: terminology mapping and its effectiveness for Information Retrieval. *In: IFLA World Library and Information Congress.*

Mayr, Philipp, Stempfhuber, Maximilian, & Walter, Anne-Kathrin. 2005. Auf dem Weg zum wissenschaftlichen Fachportal – Modellbildung und Integration heterogener Informationssammlungen. *In: 27. DGI-Online-Tagung in Frankfurt am Main.*

McConnell, Steve. 2004. *Code Complete, Second Edition.* Microsoft Press.

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

McKeeman, William M. 1998. Differential Testing for Software. *Digital Technical Journal,* **10**(1), 100–107.

McNeile, Ashley. 2003. *MDA: The Vision with the Hole?*

Meister, Juergen. 2006a. *Produktgetriebene Entwicklung von Software-Produktlinien am Beispiel analytischer Anwendungssoftware.* Ph.D. thesis, Carl von Ossietzky Universitat Oldenburg.

Meister, Jürgen. 2006b. Software-Produktlinien. *Pages 370–393 of:* Reussner, Ralf, & Hasselbring, Wilhelm (eds), *Handbuch der Software-Architektur.* dpunkt.verlag.

Mellor, Stephen J., Scott, Kendall, & Uhl, Axel. 2004. *MDA Distilled: Principles of Model-Driven Architecture.* Addison-Wesley.

Memon, A.M. 2001. *A COMPREHENSIVE FRAMEWORK FOR TESTING GRAPHICAL USER INTERFACES.* Ph.D. thesis, University of Pittsburgh.

Memon, Atif M., Pollack, Martha E., & Soffa, Mary Lou. 2001. Hierarchical GUI Test Case Generation Using Automated Planning. *IEEE Trans. Softw. Eng.,* **27**(2), 144–155.

Memon, Atif M., Banerjee, Ishan, & Nagarajan, Adithya. 2003. What Test Oracle Should I Use for Effective GUI Testing? *Pages 164–173 of: 18th IEEE International Conference on Automated Software Engineering (ASE 2003), 6-10 October 2003, Montreal, Canada.* IEEE Computer Society.

Mernik, Marjan, Heering, Jan, & Sloane, Anthony M. 2005. When and how to develop domain-specific languages. *ACM Comput. Surv.*, **37**(4), 316–344.

Meszaros, Gerard. 2007. *xUnit Test Patterns*. Addison Wesley.

Morisio, M., & Torchiano, M. 2002. Definition and Classification of COTS: A Proposal. *LECTURE NOTES IN COMPUTER SCIENCE*, 165–175.

Mugridge, R., & Cunningham, W. 2005a. Agile Test Composition. *Pages 137–144 of: Extreme Programming And Agile Processes in Software Engineering: 6th International Conference, XP 2005, Sheffield, UK*. Lecture Notes in Computer Science, vol. 3556. Springer.

Mugridge, Rick. 2004. Test Driving Custom Fit Fixtures. *In:* Eckstein, Jutta, & Baumeister, Hubert (eds), *Extreme Programming and Agile Processes in Software Engineering 5th International Conference XP 2004 Garmisch-Partenkirchen, Germany, June 2004*. Lecture Notes in Computer Science, vol. 3092. Springer.

Mugridge, Rick, & Cunningham, Ward. 2005b. *FIT for Developing Software. Framework for Integrated Tests*. Prentice Hall PTR.

Mutschke, P. 2003. Mining Networks and Central Entities in Digital Libraries. A Graph Theoretic Approach Applied to Co-author Networks. *LECTURE NOTES IN COMPUTER SCIENCE*, 155–166.

Myers, Glenford J., Sandler, Corey, Badgett, Tom, & Thomas, Todd M. 2004. *The Art of Software Testing, Second Edition*. 2nd edn. John Wiley & Sons.

Myllännistö, Varvana, & Soininen, Timo. 2005. Kumbang Configurator - A Configuration Tool for Software Product Families. Presented at the . *In: International Joint Conference on Artificial Intelligence 2005 Configuration workshop,*.

Mylopoulos, John. 2005. Goal-Oriented Requirements Engineering. *apsec*, **0**, 3.

Nebut, Clementine, & Fleurey, Franck. 2006. Automatic Test Generation: A Use Case Driven Approach. *IEEE Trans. Softw. Eng.*, **32**(3), 140–155. Member-Yves Le Traon and Member-Jean-Marc Jezequel.

Neighbors, James M. 1989. Draco: A Method for Engineering Reusable Software Systems. *Pages 295–319 of:* Biggerstaff, Ted J., & Perlis, Alan J. (eds), *Software Reusability – Concepts and Models*, vol. I. ACM Press.

Nielsen, J. 1994. Guerrilla HCI: Using Discount Usability Engineering to Penetrate the Intimidation Barrier. *Pages 245–272 of:* Bias, Randolph G., & Mayhew, Deborah J. (eds), *Cost-Justifying Usability*. Morgan Kaufmann.

Nielsen, J. 2001. *How to Conduct a Heuristic Evaluation*. http://www.useit.com/papers/heuristic/heuristic_evaluation.html.

Norman, Donald. 2006. Why doing user observations first is wrong. *interactions*, **13**(4), 50–ff.

of Electrical, Institute, & Staff, Electronics Engineers (eds). 1991. *IEEE Computer Dictionary - Compilation of IEEE Standard Computer Glossaries, 610-1990*. Institute of Electrical and Electronics Engineers.

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

Patzke, Thomas, & Muthig, Dirk. 2002. *Product Line Implementation Technologies : Programming Language View.* Tech. rept. IESE-Report No. 057.02/E. Fraunhofer Institut Experimentelles Software Engineering.

Patzke, Thomas, & Muthig, Dirk. 2003. *Product Line Implementation with Frame Technology: A Case Study.* Tech. rept. IESE-Report No. 018.03/E. Fraunhofer Institut Experimentelles Software Engineering.

Pedersen, S., & Hasselbring, W. 2004. Interoperabilität für Informationssysteme im Gesundheitswesen auf Basis medizinischer Standards. *Informatik-Forschung und Entwicklung*, **18**(3), 174–188.

Perry, D.E., Porter, A.A., & Votta, L.G. 2000. Empirical studies of software engineering: a roadmap. *Proceedings of the conference on The future of Software engineering*, 345–355.

Petras, Vivien. 2006. *Translating Dialects in Search: Mapping between Specialized Languages of Discourse and Documentary Languages.* Ph.D. thesis, University of California, Berkeley.

Petras, Vivien, Baerisch, Stefan, & Stempfhuber, Maximillian. 2007. The Domain-Specific Track at CLEF 2007. *In:* Nardi, Alessandro, & Peters, Carol (eds), *Working Notes for the CLEF 2007 Workshop, 19-21 September, Budapest, Hungary.*

Pilato, C. Michael, Collins-Sussman, Ben, & Fitzpatrick, Brian W. 2004. *Version Control with Subversion.* O'Reilly Media.

Pohl, Klaus, & Metzger, Andreas. 2006. Software product-line testing. *Communication of the ACM*, **49**(12), 78–81.

Poston, Robert. 1998. Making Test Cases from Use Cases Automatically. *In: Proceedings of Quality Week Europe 1998.*

Poston, Robert M. 1996. *Automating Specification-Based Software Testing.* Institute of Electrical & Electronics Engineering.

Prähofer, Herbert, Hurnaus, Dominik, & Mössenböck, Hanspeter. 2006. Building End-User Programming Systems Based on a Domain-Specific Language. *In: Proceedings of 6th OOPSLA Workshop on Domain-Specific Modeling (DSM'06).* Computer Science and Information System Reports, Technical Reports, no. 37. University of Jyväskylä.

Prechelt, L. 2001. *Kontrollierte Experimente in der Softwaretechnik: Potenzial und Methodik.* Springer.

Prinz, Andreas, Nytun, Jan P., & Tveit, Merete S. 2006. Automatic Generation of Modelling Tools. *In:* Rensink, Arend, & Warmer, Jos (eds), *Model Driven Architecture - Foundations and Applications Second European Conference, ECMDA-FA 2006 Bilbao, Spain, July 10-13, 2006.* Lecture Notes in Computer Science, no. 4066. Springer.

Punter, Teade, Ciolkowski, Marcus, Freimut, Bernd, & John, Isabel. 2003. Conducting Online Surveys in Software Engineering. *Page 80 of: ISESE '03: Proceedings of the 2003 International Symposium on Empirical Software Engineering.* Washington, DC, USA: IEEE Computer Society.

Quix, Christoph Josef. 2003. *Metadatenverwaltung zur qualitätsorientierten Informationslogistik in Data-Warehouse-Systemen.* Ph.D. thesis, Rheinisch-Westfälischen Technischen Hochschule Aachen.

Rahm, E., & Bernstein, P.A. 2001. A survey of approaches to automatic schema matching. *The VLDB Journal The International Journal on Very Large Data Bases*, **10**(4), 334–350.

Reeves, Carol A., & Bednar, David A. 1994. Defining Quality: Alternatives and Implications. *The Academy of Management Review, Special Issue: Total Quality*, **19**(3), 419–445.

Ricca, Filippo. 2003. *Analysis, Testing and Re-structuring of Web Applications.* Ph.D. thesis, Universität Genua.

Ricca, Filippo, & Tonella, Paolo. 2001. Analysis and Testing of Web Applications. *In: ICSE.*

Robertson, SE, van Rijsbergen, CJ, & Porter, MF. 1980. Probabilistic models of indexing and searching. *Proceedings of the 3rd annual ACM conference on Research and development in Information Retrieval*, 35–56.

Royce, W. W. 1987. Managing the development of large software systems: concepts and techniques. *Pages 328–338 of: ICSE '87: Proceedings of the 9th international conference on Software Engineering.* Los Alamitos, CA, USA: IEEE Computer Society Press.

Rumbaugh, James, Jacobson, Ivar, & Booch, Grady. 2004. *The Unified Modeling Language Reference Manual.* 2nd edn. Addison Wesley Professional.

Ryser, Johannes. 2003. *Szenariobasiertes Validieren und Testen von Softwaresystemen.* Ph.D. thesis, Wirtschaftswissenschaftliche Fakultät der Universität Zürich.

Saeki, Motoshi, & Kaiya, Haruhiko. 2006. On Relationships among Models, Meta Models and Ontologies. *In: Proceedings of 6th OOPSLA Workshop on Domain-Specific Modeling (DSM'06).* Computer Science and Information System Reports, Technical Reports, no. 37. University of Jyväskylä.

Salton, G., & Buckley, C. 1988. Term-weighting approaches in automatic text retrieval. *Information Processing and Management: an International Journal*, **24**(5), 513–523.

Salton, Gerard, Fox, Edward A., & Wu, Harry. 1982. *Extended Boolean Information Retrieval.* Tech. rept. Cornell University, Ithaca, NY, USA.

Schamber, L., Eisenberg, M.B., & Nilan, M.S. 1990. A re-examination of relevance: Towards a dynamic, situational definition. *Information Processing and Management*, **26(6)**, 755–776.

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

Scott, Michael L. 2005. *Programming Language Pragmatics.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Seifert, D., & Souquieres, J. 2008. *Using UML Protocol State Machines in Conformance Testing of Components.* Tech. rept. 00274383. Institut National Polytechnique de Lorraine Institut National Polytechnique de Lorraine.

Sensler, Carsten, Kunz, Michael, & Schnell, Peter. 2006. Testautomatisierung mit modellgetriebener Testskript-Entwicklung. *Objektspektrum*, **3**, 74–84.

Shvaiko, P., & Euzenat, J. 2005. A Survey of Schema-Based Matching Approaches. *LECTURE NOTES IN COMPUTER SCIENCE*, **3730**, 146.

Simonyi, Charles. 1995. *The Death Of Computer Languages, The Birth of Intentional Programming.* Tech. rept. Microsoft Research.

Simonyi, Charles, Christerson, Magnus, & Clifford, Shane. 2006. Intentional software. *Pages 451–464 of: OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming languages, systems, and applications.* New York, NY, USA: ACM Press.

Spinellis, Diomidis. 2001. Notable design patterns for domain-specific languages. *The Journal of Szstems and Software*, **56**, 91–99.

Spinellis, Diomidis. 2003. On the Declarative Specification of Models. *IEEE Computer*, **20(2)**, 93–95.

Stachowiak, Herbert. 1973. *Allgemeine Modelltheorie.* Springer, Wien.

Stahl, Thomas, & Völter, Markus. 2006. *Model-Driven Software Development.* Wiley & Sons.

Stempfhuber, Maximilian. 2003. *Objectorientierte Dynamische Benutzungsoberflächen - Odin:Behandlung semantischer und struktureller Heterogenität in Informationssystemen mit den Mitteln der Softwareergonomie.* Ph.D. thesis, Universität Koblenz-Landau.

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

Thomas, Dave. 2004. MDA: Revenge of the Modelers or UML Utopia? *IEEE Softwware*, **21**(3), 15–17.

Thurmeier, M. 2007. *Erwartungen an wissenschaftliche Fachportale : Ergebnisse einer qualitativen Befragung von Wissenschaftlern.* Tech. rept. in to mind: Institut für Marketingforschung.

Tichy, Walter F. 2000. Hints for Reviewing Empirical Work in Software Engineering. *Empirical Softw. Engg.*, **5**(4), 309–312.

Tichy, W.F., Lukowicz, P., Prechelt, L., & Heinz, E.A. 1995. Experimental evaluation in computer science: A quantitative study. *The Journal of Systems & Software*, **28**(1), 9–18.

Tolvanen, Juha-Pekka, & Kelly, Steven. 2004. Domänenspezifische Modellierung. *OBJEKTspektrum*, **04**, 30–35.

Tolvanen, Juha-Pekka, & Rossi, Matti. 2003. MetaEdit+: defining and using domain-specific modeling languages and code generators. *Pages 92–93 of: OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications.* New York, NY, USA: ACM.

Tsai, Wei-Tek, Bai, Xiaoying, Paul, Raymond A., & Yu, Lian. 2001. Scenario-Based Functional Regression Testing. *Pages 496–501 of: Computer Software and Applications Conference (COMPSAC).*

Uflacker, Matthias. 2005 (Januar). *Entwicklung eines Editors für die modellgetriebene Konstruktion komponentenbasierter Software-.* M.Phil. thesis, Universität Oldenburg, Department für Informatik.

Uhl, Axel. 2006. Model-Driven Architecture. *Pages 103–128 of:* Reussner, Ralf, & Hasselbring, Wilhelm (eds), *Handbuch der Software-Architektur.* dpunkt.

Utting, M., Pretschner, A., & Legeard, B. 2005. *A Taxonomy of Model-Based Testing.* Tech. rept. Department of Computer Science The University of Waikato.

van Deursen, A., & Klint, P. 2002. Domain-Specific Language Design Requires Feature Descriptions. *Journal of Computing and Information Technology*, **10**(1), 1–17.

van Deursen, Arie, Klint, Paul, & Visser, Joost. 2000. Domain-Specific Languages: An Annotated Bibliography. *SIGPLAN Notices*, **35**(6), 26–36.

van Lamsweerde, Axel. 2001. Goal-Oriented Requirements Engineering: A Guided Tour. *Page 249 of: RE '01: Proceedings of the 5th IEEE International Symposium on Requirements Engineering*. Washington, DC, USA: IEEE Computer Society.

Voelter, Markus. 2003. *A collection of Patterns for Program Generation*. EuroPLoP.

Voelter, Markus, & Groher, Iris. 2007. Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. *In: 11th international Software Product Line Conference 2007*.

Völter, Marcus. 2006a. Im Fokus: Sprachen, Modelle und Fabriken in der Softwareentwicklung. *IX Magazin für Professionelle Informationstechnik*, **10**, 123–127.

Völter, Markus. 2005. Kaskadierung von MDSD und Modelltransformationen. *JavaSpektrum, 05/2005*, **05/2005**, ?

Völter, Markus. 2006b. Best Practices for Model-to-Text Transformations. *In: Eclipse Summit 2006 Workshop: Modeling Symposium*.

Völter, Markus. 2006c. oAW xText - A framework for textual DSLs. *In: Eclipse Summit 2006 Workshop: Modeling Symposium*.

Völter, Markus. 2007. *Domain Specific Languages : Implementation Technologies*.

Voorhees, Ellen M. 2002. The Philosophy of Information Retrieval Evaluation. *Pages 355–370 of: CLEF '01: Revised Papers from the Second Workshop of the Cross-Language Evaluation Forum on Evaluation of Cross-Language Information Retrieval Systems*. London, UK: Springer-Verlag.

Wagelaar, Dennis, & Straeten, Ragnhild Van Der. 2006. A Comparison of Configuration Techniques for Refactoring and Refinement Transformations. *In:* Rensink, Arend, & Warmer, Jos (eds), *Model Driven Architecture - Foundations and Applications Second European Conference, ECMDA-FA 2006 Bilbao, Spain, July 10-13, 2006*. Lecture Notes in Computer Science, no. 4066. Springer.

Weiss, David M., & Lai, Chi Tau Robert. 1999. *Software product-line engineering: a family-based software development process*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Weiss, S.M. 2005. *Text Mining: Predictive Methods for Analyzing Unstructured Information*. Springer.

Winter, Mario. 1999. *Qualitätssicherung für objektorientierte Software: Anforderungsermittlung und Test gegen Anforderungsspezifikationen*. Ph.D. thesis, Fernuniversität Hagen.

Wohlin, Claes, Runeson, Per, Hoest, Martin, Ohlsson, Magnus, Regnell, Björn, & Wesslen, Anders. 2000. *Experimentation in Software Engineering : An Introduction*. Kluwer Academic Publishers.

Womser-Hacker, Christa. 2006. An Information Retrieval Prototype for Research and Teaching. *Pages 85–101 of:* Eibl, Maximilian, Wolff, Christian, & Womser-Hacker, Christa (eds), *Designing Information Systems: Festschrift für Jürgen Krause*. Schriften zur Informationswissenschaft, vol. 43. Hochschulverband für Informationswissenschaft.

Zelkowitz, MV, & Wallace, DR. 1998. Experimental models for validating technology. *Computer*, **31**(5), 23–31.

Zeller, A. 2005. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA.

Zeng, M.L., & Chan, L.M. 2004. Trends and issues in establishing interoperability among knowledge organization systems. *Journal of the American Society for Information Science and Technology*, **55**(5), 377–395.

Zhu, Hong, Hall, Patrick A. V., & May, John H. R. 1997. Software Unit Test Coverage and Adequacy. *ACM Comput. Surv.*, **29**(4), 366–427.

Zubrow, D. 1998. Measurement with a Focus: Goal-Driven Software Measurement. *CROSSTALK: The journal of Defense Software Engineering.*