Test Models and Coverage Criteria for Automatic Model-Based Test Generation with UML State Machines

DISSERTATION

zur Erlangung des akademischen Grades
doctor rerum naturalium
(Dr. rer. nat.)
im Fach Informatik

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät II
Humboldt-Universität zu Berlin

von
Herr Dipl.-Inf. Stephan Weißleder
geboren am 18.04.1979 in Berlin

Präsident der Humboldt-Universität zu Berlin:
Prof. Dr. Dr. h.c. Christoph Markschies

Dekan der Mathematisch-Naturwissenschaftlichen Fakultät II:
Prof. Dr. Peter Frensch

Gutachter:

1. Prof. Dr. Holger Schlingloff
2. Prof. Dr. Ina Schieferdecker
3. Prof. Dr. Jan Peleska

eingereicht am: 8. Dezember 2009
Tag der mündlichen Prüfung: 26. Oktober 2010

Abstract

Testing is an important means of quality management and is widely used in industrial practice. Model-based functional testing is focussed on comparing the system under test to a test model. This comparison usually consists of automatically generating a test suite from the test model, executing the test suite, and comparing the observable behavior to the expected one. Important advantages of model-based testing are formal test specifications that are close to requirements, traceability of these requirements to test cases, and the automation of test case design, which helps reducing test costs. Testing cannot be complete in many cases: For test models that describe, e.g., non-terminating systems, it is possible to derive a huge and possibly infinite number of different test cases. Coverage criteria are a popular heuristic means to measure the fault detection capability of test suites. They are also used to steer and stop the test generation process.

There are several open questions about test models and coverage criteria. For instance, the UML 2.1 defines 13 different kinds of diagrams, which are often used in isolation although it might be beneficial to combine them. Furthermore, there are several unconnected kinds of coverage criteria. Most of them are very useful and here, too, the question for ways to combine their benefits is very interesting. Moreover, the relation between test models and coverage criteria is not researched thoroughly yet and the question for mutual dependencies remains.

The context of this thesis is automatic model-based test generation with UML state machines. The focus is on test models, coverage criteria, and their relations. We present new approaches to combine coverage criteria, to use model transformations for testing, and to combine state machines with other test models. In detail, we present a test generation algorithm that allows to combine control-flow-, data-flow-, or transition-based coverage criteria with boundary-based coverage criteria. We also show how to transform state machines in order to simulate the satisfaction of coverage criteria, to combine coverage criteria, or to define and implement new coverage criteria. Furthermore, we present ways to combine state machines with class diagrams and with interaction diagrams. We also show how to influence the efficiency of the generated test suite. Finally, we developed the prototype implementation *ParTeG* for the mentioned contributions and applied it to standard examples, academic applications, and industrial case studies.

Zusammenfassung

Testen ist ein wichtiges und weit verbreitetes Mittel des Qualitätsmanagements. Funktionale, modellbasierte Tests vergleichen das zu testende System mit einer Testspezifikation in Form eines Modells: Testsuiten werden auf Basis des Testmodells generiert und gegen das zu testende System ausgeführt – anschließend wird das aktuelle mit dem erwarteten Verhalten verglichen. Wesentliche Vorteile des modellbasierten Testens sind formale und anforderungsnahe Testmodelle, die Rückverfolgbarkeit von Anforderungen, sowie die Automatisierung des Testdesigns und damit auch die Verringerung der Testkosten. Testen kann oft nicht erschöpfend sein: Für viele Testmodelle ist es möglich, eine beliebig hohe Anzahl beliebig langer Testfälle zu generieren. Abdeckungskriterien sind populäre Mittel für das Messen des Fehleraufdeckungspotentials von Testsuiten, sowie für die Steuerung der Testerzeugung und das Stoppen derselben bei Erreichen eines gewissen Abdeckungsgrades.

Zu diesen Themen gibt es etliche offene Punkte. Zum Beispiel definiert die UML 2.1 insgesamt 13 Diagrammtypen, die oftmals in Isolation verwendet werden obwohl eine kombinierte Verwendung vorteilhaft wäre. Weiterhin gibt es verschiedene Arten von Abdeckungskriterien, deren Nutzen in isolierter Verwendung bereits gezeigt wurde. Es stellt sich jedoch die Frage, wie deren Vorteile kombiniert werden können. Darüber hinaus wurden die Beziehungen zwischen Testmodellen und Abdeckungskriterien noch nicht tiefgründig erforscht und die Frage nach gegenseitigen Abhängigkeiten ist offen.

Diese Dissertation befasst sich mit der automatisierten Testerzeugung basierend auf UML Zustandsmaschinen. Der Fokus liegt auf Testmodellen, Abdeckungskriterien und deren Beziehungen. Ich präsentiere verschiedene Ansätze, Abdeckungskriterien zu kombinieren, Modelltransformationen zu nutzen und Testmodelle kombinieren: Ich definiere einen Testgenerierungsalgorithmus, der kontrollfluss-, datenfluss- oder transitionsbasierte Abdeckungskriterien mit grenzwertbasierten Abdeckungskriterien kombiniert. Weiterhin präsentiere ich die Transformation von Zustandsmaschinen, um Abdeckungskriterien austauschbar zu machen, sie zu kombinieren oder die Implementierung neu definierter Kriterien umzusetzen. Ich kombiniere Zustandsmaschinen mit Klassendiagrammen und mit Interaktionsdiagrammen. Die zugehörigen Abdeckungskriterien können ebenfalls teilweise kombiniert werden. Darüber hinaus untersuche ich die Beeinflussung der Testeffizienz durch Eingriffe in die Testerzeugung. Zu den genannten Beiträgen habe ich den prototypischen Testgenerator *ParTeG* entwickelt, der bereits für Standardbeispiele, akademische Anwendungen und industrielle Fallstudien genutzt wurde.

Widmung

Für Tobias, Henni und Barbara.

Contents

Chapter 1

Introduction

1.1 Topic of the Thesis

Testing is one of the most important means to validate the correctness of systems. The costs of testing are put at 50% [Mye79, KFN99, Som01] of the overall project costs. There are many efforts to decrease the costs for testing, e.g. by introducing automation.

There are many different testing techniques, processes, scopes, and targets. This thesis is focused on functional model-based testing. Functional testing consists of comparing the system under test (SUT) to a specification. A functional test detects a failure if the observed and the specified behavior of the SUT differ. Model-based testing is about using models as specifications. Several modeling languages have been applied to create test models, e.g. B [Abr07], Z [Spi92], the Unified Modeling Language (UML) [Obj07], or the Object Constraint Language (OCL) [Obj05a]. Model-based testing allows to derive test suites automatically from formal test models. This thesis is focused on automatic model-based test generation with UML state machines and OCL expressions. Although testing with state charts, state diagrams, or state machines has been investigated for several decades, there are still many unexplored aspects and issues left to be solved.

Testing is often incomplete, i.e. cannot cover all possible system behaviors. There are several heuristic means to measure the quality of test suites, e.g. fault detection, mutation analysis, or coverage criteria. These means of quality measurement can also be used to decide when to stop testing. This thesis is concentrated on coverage criteria. There are many different kinds of coverage criteria, e.g. focused on data flow, control flow, transition sequences, or boundary values. In this thesis, we will present new approaches, e.g., to combine test models or to simulate and combine coverage criteria.

1.2 Contribution of the Thesis

This thesis is focused on automatic model-based test generation with UML state machines as test models and coverage criteria that are applied to them. Figure 1.1 provides an abstract overview of automatic model-based test generation. There are more detailed presentations. For instance, Utting et al. [UPL06] include test execution and requirements. In contrast, this figure just depicts all elements necessary to give an outline of this thesis: The inputs of the model-based test generation process are a test model and coverage criteria to satisfy. The application of a coverage criterion to the test model results in a set of test model-specific test goals. The test goals and the test model are used to automatically generate the test suite.

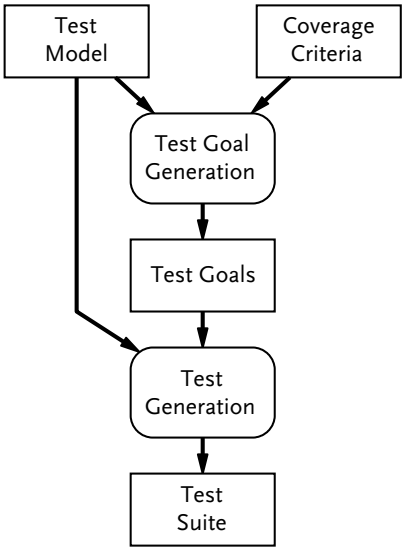


Figure 1.1: Model-based test generation.

The contributions of this thesis are focused on the combination of different test models, the relations of test models and coverage criteria, and the combination of coverage criteria. This thesis contains five contributions.

First, we introduce a novel test generation algorithm based on UML state machines and class diagrams with OCL expressions. The advantage of this algorithm is the combination of abstract test case creation and boundary value analysis. The major contribution is the combination of the corresponding, e.g., transition-based and boundary-based, coverage criteria.

As the second contribution, we investigate the mutual dependency of state machines and coverage criteria in model-based testing. We transform state machines and evaluate the impact of the applied coverage criteria. The

major contribution is that the application of any feasible coverage criterion to the transformed state machine can have the same effect as the application of almost any other feasible coverage criterion to the original state machine. We present an experience report of an industrial cooperation that shows the importance of model transformations for model-based test generation. We define *simulated coverage criteria satisfaction* and present corresponding model transformations. The most important effect is that the satisfaction of a strong coverage criterion on the original state machine can be simulated by satisfying a weak coverage criterion on a transformed state machine. We also show new ways to combine and define coverage criteria. They can also be simulated with existing coverage criteria. This second contribution can be used together with the afore presented test generation approach.

The third contribution is the combination of different test models, which can be used together with the two previously presented contributions. We present the combination of UML state machines with structural models like UML class diagrams and with behavioral models like UML interaction diagrams. Since automatic test generation depends on the provided test models, this combination is advantageous for automatic test generation. Both proposed combinations of test models have advantages that go beyond the separate application of the corresponding single test models. New coverage criteria are presented that are focused on combined test models.

Fourth, we investigate the application of coverage criteria on test models and focus on the resulting set of test-model-specific test goals. The test cases are generated based on the order of the test goals. The contribution is an empirical evaluation of the impact of the test goal order on the efficiency of the generated test suite, e.g. the average number of test cases to execute until detecting a failure. Since this is also a general contribution to automatic model-based test generation, its advantages can be combined with the advantages of the afore three contributions.

These four contributions are interrelated and all support automatic model-based test generation. They are furthermore substantiated by the developed tool support and corresponding case studies. We work on two Eclipse plug-ins based on EMF [Ecl07a] and UML 2.1 [Ecl07b]: The model-based test generation tool *ParTeG* [Weib] implements the novel test generation approach described as the first contribution. It partly supports the transformation and the combination of test models, and performs the ordering of test goals. The tool *Coverage Simulator* [Weia] is currently under development. Its goal is to provide a wide range of test model transformations to support the simulated satisfaction of coverage criteria as presented in the second contribution. ParTeG has been used to generate tests for standard examples as well as academic and industrial test models.

1.3 Structure of the Thesis

The thesis is structured as follows. In Chapter 2, we present all preliminaries of this thesis and formal definitions for coverage criteria in model-based testing. Chapter 3 contains the test generation algorithm that is used to combine the generation of abstract test cases with boundary value analysis. We present several test model transformations that are used to influence the fault detection capability of generated test suites in Chapter 4. In Chapter 5, we investigate the combination of different test models. We consider the test goal order and its influence on test suite efficiency in Chapter 6. Finally, we conclude the thesis in Chapter 7.

Figure 1.2 depicts how the four major contributions besides the case studies fit into the roadmap of model-based test generation in Figure 1.1.

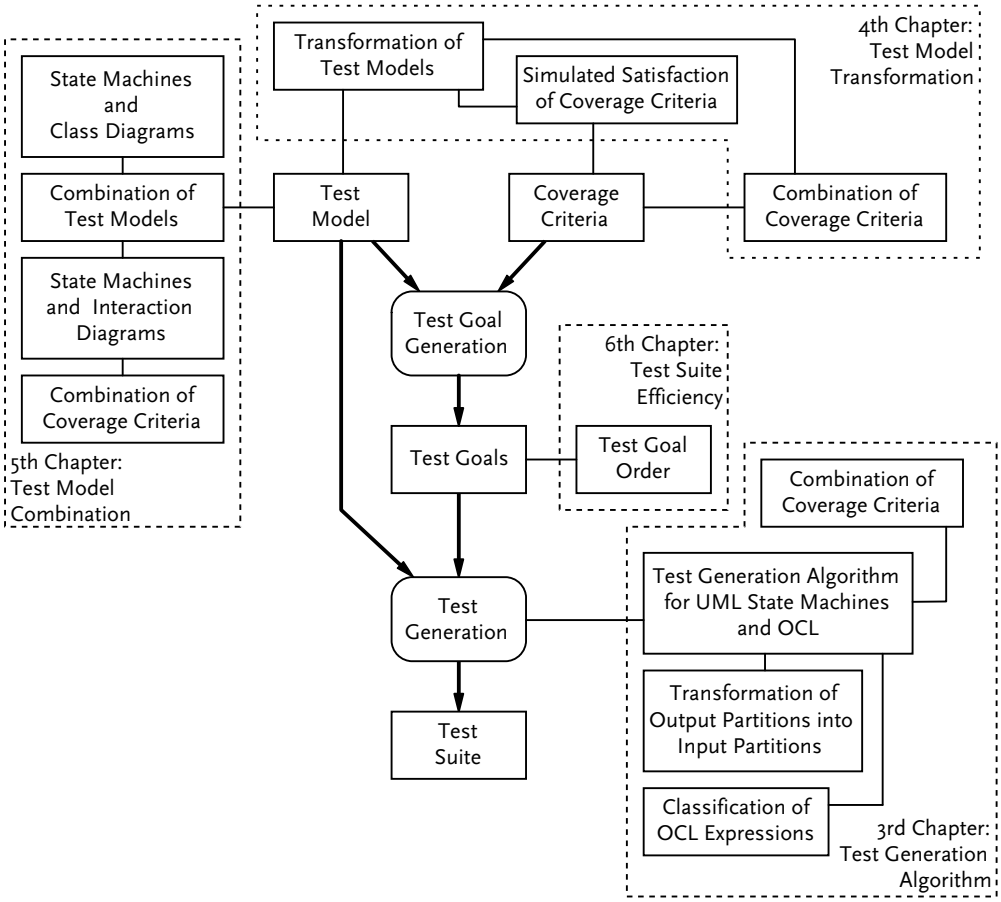


Figure 1.2: Structure of the thesis.

Chapter 2

Preliminaries & Definitions

This chapter contains preliminaries and basic definitions of the thesis. We will give an introduction to existing work about testing in Section 2.1, to modeling languages in Section 2.2, and to model-based testing in Section 2.3. Section 2.4 contains formal definitions of coverage criteria on UML state machines.

2.1 Introduction to Testing

There are many sentiments about testing. For instance, testing is considered an important failure detection technique, a means for system validation, or risk management. There are numerous test purposes, test methods, and test processes. In this section, we provide a survey of testing and position this thesis in the field of testing. First, we define fault, error, and failure in Section 2.1.1. Then, we present definitions of testing in Section 2.1.2 and several test methods in Section 2.1.3. We show test processes in Section 2.1.4 and approaches to measure test quality in Section 2.1.5. Finally, we present further references in Section 2.1.6.

2.1.1 Fault, Error, and Failure

In this section, we clarify the notions of fault, error, and failure. For that, we present the fault/failure model, identify reasons for faults, and a classification of possible consequences of failures.

Fault/Failure Model.

The execution of tests on a system under test (SUT) can result in unexpected behavior. According to Hopper [Hop47], the first such unexpected behavior

was caused by a moth in a relay. That is why it is common to speak of bugs. This term, however, does not describe the different stages of fault, error propagation, and failure detection appropriately. In the following, we introduce the fault/failure model as presented in [Mor83, Off88, Mor90] and [AO08, page 12].

Definition 1 (Fault) *A* fault *is a static defect in a system.*

A static defect is, e.g., a wrong expression in a system's source code. It is often caused by human errors such as misinterpreting a requirement, forgetting a condition, or simply mistyping. As long as the fault is just existing in the system without being executed, it has no effect on the behavior of the system: the fault is said to be *dormant*. If the faulty instruction is executed, then the fault is said to be *activated*. An activated fault can result in an error, but it does not have to.

Definition 2 (Error) *An* error *is a wrong internal state of a running system.*

A wrong internal state of a system can be, e.g., an erroneous program counter or a faulty attribute value. If such wrong values influence the observable behavior of the SUT, the error is said to be *propagated* to the outside. An error that is propagated to the outside can result in a failure.

Definition 3 (Failure) *A* failure *is an observable deviation of the actual from the expected behavior of a system.*

Failures can be detected directly by test cases. Figure 2.1 shows one possible way from a fault to a failure with fault activation and error propagation.

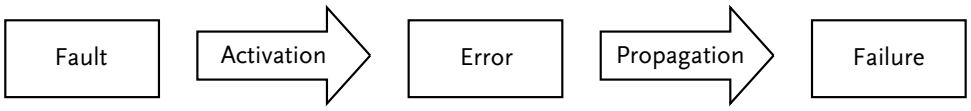


Figure 2.1: Relation of fault, error, and failure.

Since testing can only detect failures, it is a failure detection technique. Nevertheless, it is often called a fault detection technique. We use both terms interchangeably. The fault/failure model [Mor83, Off88, Mor90] defines three conditions that must be fulfilled for that a fault results in a failure: (1) The fault must be reached (*Reachability*). (2) After activating the fault, the system state must be incorrect (*Infection*). (3) The infected system state must be propagated to the outside (*Propagation*).

Causes for Faults.

Here, we present a list of possible causes for faults. The first and quite common cause for faults are missing (or faulty, contradictory, etc.) requirements. In this case, the system engineer missed some important use cases and the corresponding expected behavior of the SUT is undefined. Such faults are most often detected by inspecting the requirements.

Second, there are several kinds of functional faults, i.e., discrepancies between the test specification and the SUT. They are often caused by a defective implementation of the software or the hardware. The effect is that the SUT does not behave corresponding to the test specification. Such faults can be detected by functional testing.

A third kind of faults are non-functional faults. They reference non-functional properties of the SUT, such as performance, security, scalability, or compatibility. The detection of such faults requires theoretical considerations, stress tests, security-relevant specifications, performance tests, and so forth. There is, however, also work about expressing non-functional properties like security in models [BDL05, SEBC09]. Thus, non-functional properties can also be detected by comparing the SUT to the test specification.

Consequences of Failures.

Faults can indirectly result in failures. These failures can be classified corresponding to consequences of their occurrence [Bei90]. This might help to prioritize the abolition of faults. Table 2.1 shows a corresponding failure classification with a short description.

Level	Class	Symptoms
1	Mild	Typing error
2	Moderate	Redundancy, misleading messages
3	Annoying	Erroneous behavior (bills about 0.00$ are sent)
4	Disturbing	Transactions cannot be completed
5	Serious	Transaction and information about it is lost
6	Very serious	Exchange objects inside a transaction (money is transferred to the wrong account)
7	Extreme	Just like 6 but very frequent
8	Intolerable	Unrecoverable errors in a database
9	Catastrophic	System is shutting down on its own
10	Infectious	Consequences for other systems, nuclear power stations, military

Table 2.1: Categorization of failure consequences according to Beizer.

2.1.2 What is Testing and What is not?

This section contains definitions of what is testing and what is not. Furthermore, we present different understandings of testing and sketch a short list of prejudices about testing. Finally, we present definitions of the basic terms used throughout this thesis.

Testing can be Validation or Verification.

Testing can be validation or verification. If it is a part of system engineering that is often part of a programmer's (tester's) daily work, and testers have to derive test cases from requirements specifications manually, testing is a validation technique. In model-based testing, tests are often automatically generated from an abstract description and, in this case, testing is a verification technique. We present a definition of validation and verification according to [AO08, page 11] and compare testing to other techniques:

Definition 4 (Validation) *Validation is the process of evaluating a system to ensure compliance with intended usage or specification.*

Definition 5 (Verification) *Verification is the process of determining if the result of a given phase in system development fulfills the requirements established during the previous phase.*

In general, it would be better to prove a system property instead of creating tests for it. There are, however, many systems that contain components (e.g. battery, display) that cannot be proved or the proofs miss some essential parts of the SUT. A quote of Donald Knuth describes this situation well enough: "Beware of bugs in the above code; I have only proved it correct, not tried it." Testing is for many situations the best technique available.

Definition 6 (Testing) *Testing is the process of systematically evaluating a system by observing its execution.*

Definition 7 (Debugging) *Debugging is the process of finding a fault that causes a given failure.*

Definitions 6 and 7 are adapted from [AO08, page 13]. There are also more detailed definitions [Ber00] that define testing as a dynamic verification with finite set of test cases that are suitably selected to check the system's behavior. Testing can be used to detect failures in the SUT. The detection of the faults that cause these failures is called debugging. Detected failures are the anchor to start the debug process from. Although there are some approaches to automate debugging [SW96, Arc08], it is still a manual task for system engineers.

Correct Understandings of Testing.

There are several understandings of testing. In a sense, many interpretations are true depending on the point of view. Here, we present several correct understandings of testing. Some of these views are also contained in Beizer's testing levels [Bei90].

- *Testing is comparing actual and expected behavior:* Without that comparison it would be impossible to detect functional failures.
- *Testing is detecting failures:* Similar to the previous statement, a failure is the observable deviation of the actual from the expected system behavior.
- *Testing is managing risks:* For many systems, testing cannot be complete and there are only heuristic means of quality measurement. Moreover, Howden [How76] shows that finding all failures of a system is undecidable. Thus, deciding when to stop testing is managing the risk of remaining faults. The test effort depends on the kind of remaining possible faults and the corresponding failures. Thus, the test effort necessary for entertainment systems is probably considerably lower than the test effort for critical systems like airplanes or nuclear power plants.
- *Testing is increasing confidence of testers:* Since testing cannot prove the absence of faults, the goal is to remove at least all detected failures. If the existing test suite does not detect failures, at least the confidence of testers in the correctness of the SUT is increased.
- *Testing is giving continuous feed-back for programmers:* Besides all efforts to measure the quality of programs, testing is also a state of mind. Following this perception, testing is a means to improve the programming skills of the programmer: Test suites are tools to detect errors of programmers just like a spell checker of a text editor detects the typing errors of writers.

Wrong Understandings of Testing.

Corresponding to the various correct interpretations of testing, the following statements describe what testing is not.

- *Testing is not proving the absence of faults:* Each non-trivial system defines an infinite number of possible system execution paths. This can be caused by large value domains of input parameter (long, string) or

a possibly infinite number of loop iterations. Errors can occur at an arbitrary point in a program execution - perhaps in the 5th iteration of a loop or the 1000th. Since it is impossible to execute an infinite number of program instructions in finite time, the absence of faults cannot be proven by testing.

- *Testing is not diagnosing the cause of failures:* Corresponding to Definitions 6 and 7, testing detects the failures but not the causing faults.
- *Testing is not debugging:* Definitions 6 and 7 already describe the difference between testing and debugging. Nevertheless, these two terms are quite often mixed up.

Common Prejudices about Testing.

There are several prejudices about testing and errors. Naming and dispelling them is important to show the limitations of testing. The presented list of prejudices is taken from Beizer [Bei90], and the explanations are adapted to the presented Definitions 1, 2, and 3.

- *Benign Bug Hypothesis:* The belief that failures are friendly, tame, and occur following an easy logical pattern.
- *Bug Locality Hypothesis:* The belief that faults only impact the component in that they exist.
- *Control Bug Dominance:* The belief that failures are easy to detect.
- *Code/Data Separation:* The belief that faults only have an impact on either code or data.
- *Lingua Salvator Est:* The belief that the features of a language prevent faults.
- *Corrections Abide:* The belief that a corrected failure does not appear again.
- *Silver Bullets:* The belief that there is any pattern, tool, or method that prevents the occurrence of faults [Bro87].
- *Sadism Suffices:* The belief that most failures can be detected by intuition or destructive thinking.
- *Angelic Testers:* The belief that testers are better at test design than programmers at code design.

Definitions of Terms.

This section contains basic term definitions that are used throughout the thesis. First of all, there are different notions of test cases. There are the general notions of a test case, of abstract, and concrete test cases.

Definition 8 (Test Case) *A* test case *is a sequence of input stimuli to be fed into a system and expected behavior of the system.*

A test case can exist at many different levels of abstraction. The most important distinction is among abstract and concrete test cases.

Definition 9 (Abstract Test Case) *An* abstract test case *consists of abstract information about the sequence of input and output. The missing information is often concrete parameter values or function names.*

Abstract test cases are often the first step in test case creation. They are used to get an idea of the test case structure or to get information about satisfied coverage criteria. For concrete test cases, the missing information is added.

Definition 10 (Concrete Test Case) *A* concrete test case *is an abstract test case plus all the concrete information that is missing to execute the test case.*

Concrete test cases comprise the complete test information and can be executed on the SUT. A single test case, however, is rarely sufficient for good test execution.

Definition 11 (Test Suite) *A* test suite *is a set of test cases.*

The notions of abstract and concrete test suites can be defined according to the corresponding test case definitions.

Definition 12 (Test Oracle) *An* oracle *is an artifact that comprises the knowledge about the expected behavior of the SUT.*

Each test case must have some oracle information to compare observed and expected SUT behavior. Without it, no test is able to detect a failure. Typical oracles are user expectations, comparable products, past versions of the same program (e.g. regression testing), inferences about intended or expected purpose, given standards, relevant laws, or test specifications.

Definition 13 (Test Specification) *A* test specification *is a description of the system environment or the expected system behavior. It is used to derive test suites and to compare the expected and the observed system behavior.*

Test specifications are used to create test suites. Two specifications can differ in several aspects like abstraction or formalization. Since a test specification is often the result of negotiation between vendor and customer, the important parts for the customer are often accurately specified, whereas unimportant parts are rather sketchy. Consequently, the degree of abstraction and formalization depends on customer wishes and possible consequences of failures (cf. Table 2.1). For executing the test suites, test software and a test framework are needed.

Definition 14 (Test Software) *Test software is any kind of software that can be used in the testing process. Common representatives are test generators, test frameworks, and the (generated) test suite itself.*

Definition 15 (Test Framework) *A* test framework *(or* test harness*) is a framework with the objectives to automate the testing process, execute test suites, and generate the corresponding report.*

There are frameworks that provide automation to a certain extent. For instance, JUnit [EG06] and CppUnit [Sou08] are testing frameworks that allow for the simple definition, integration, and execution of unit test cases. FitNesse [MMWW09] is an example for an acceptance testing framework.

2.1.3 Testing Techniques

Testing can be conducted under several conditions. Two of the most influential aspects are the knowledge and the observability of the SUT's internal matters. In the following, we present black-, white-, and gray-box testing. After that, we sketch further testing techniques.

Black-, White-, and Gray-Box Testing.

In *black-box testing*, the SUT's internal matters are hidden from the tester. The tester only has knowledge about possible input and output values. The SUT appears to be a black-box (see Figure 2.2). Since black-box testing only allows to test input-output functionality, it is often called functional testing. As an advantage, this technique is close to realistic conditions. One important disadvantage is the lack of internal information, which could be useful to generate tests.

Figure 2.2: Black-box testing.

In *white-box testing*, the internals of the SUT are all visible. As a consequence, the knowledge about these internal matters can be used to create tests. Furthermore, white-box testing is not restricted to the detection of failures, but is also able to detect errors. Advantages are tests of higher quality because of the knowledge about system internal matters and tests with stronger oracles because errors are not necessarily propagated to the outside. An important disadvantage is the high effort necessary to look into all aspects of a program. Since white-box testing can access the structure of the SUT, it is often used for structural testing, e.g., focused on covering structural aspects of the SUT.

Black-box testing and white-box testing have both advantages and disadvantages. *Gray-box testing* [LJX+04] is one approach to combine the advantages of both techniques: This testing technique is used to design tests at white-box level and execute them at black-box level. For the tester, this has the advantage of having access to the SUT's internal information while designing tests. The tests are executed, however, under realistic conditions, i.e. only failures are detected. Gray-box testing techniques are used for commercial model-based testing, where, e.g., the test model contains information about the internal structure of the SUT, but the SUT's internal matters themselves are not accessible (e.g. for reasons of non-disclosure).

Further Testing Techniques.

Besides the mentioned testing techniques, there are many useful distinctions of testing approaches. As stated above, functional testing and non-functional testing are distinguished.

Risk-based testing is aimed at detecting serious faults with high follow-up costs (cf. Table 2.1 on page 7). The idea is to define risks for each element of the test specification or the SUT and select test cases with high-ranked elements. The goal is the minimization of the remaining risk. The advantage of this approach is that the risk of faults is taken into account. This can already be considered at the beginning of project planning. The issue of this approach is that the risks are assessed by humans and, thus, that the results can be error-prone. Risks can be also be forgotten or unknown.

One of the major problems of testing is the extent of the test suite and the corresponding test execution time. The test suite should be executed after each change in the SUT. This soon becomes impossible when test suite execution takes several hours or days. In *smoke testing*, only some representatives (e.g. important test cases) of the test suite are executed. This leads to reduced test quality, but the most fundamental aspects are covered, and the test execution time is reduced.

Stress testing is used to evaluate the stability of a system by operating the system beyond the normal level. This testing technique is used to check, e.g., stability or availability of web servers.

There are many more testing techniques. For instance, *alpha testing* and *beta testing* include the users of the SUT inside and outside the company, respectively. These techniques are often used for mass media products. In *acceptance testing*, some representatives of a certain customer decide if the SUT satisfies the requirements. This often includes non-functional requirements. In *regression testing*, the SUT is compared to past versions of the same SUT, e.g. by executing the test suites of past SUT versions on the current SUT.

2.1.4 Test Process

There are several abstraction levels in system development reaching from requirements analysis to the implementation in machine code. Testing can be conducted at all layers of abstraction. We sketch the different test levels of the test process according to the V-model [Rat97]. After that, we present ways to integrate the test process into system engineering.

Testing Levels in V-Model.

There are many different models describing how to manage system development. A prominent representative is the V-Model [Rat97]. It is shown in Figure 2.3. On the left side of the figure, system development is shown as a top-down approach: From requirements, the system is specified, designed, split in units, and implemented. On the right side, testing is shown as a bottom-up process: unit testing for classes, integration testing for components consisting of classes, system testing for integrating all components, and acceptance testing of the customer. It is obvious that the results of late tests have an impact on early development phases. Thus, it is advisable to design and execute tests as early as possible.

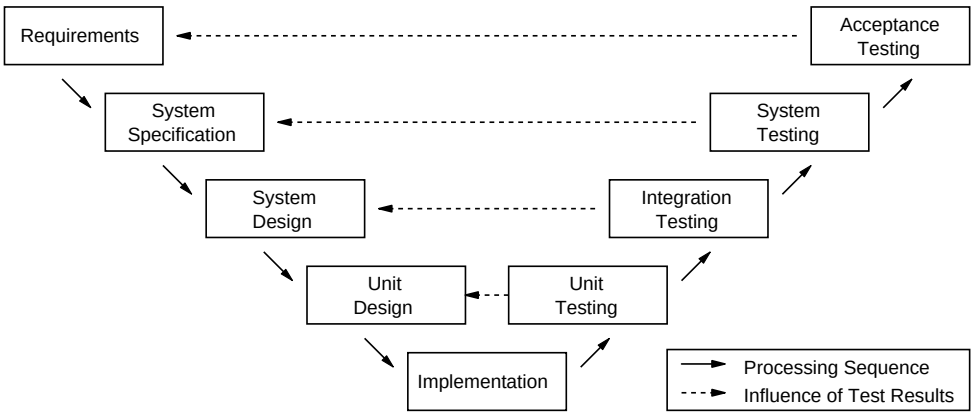
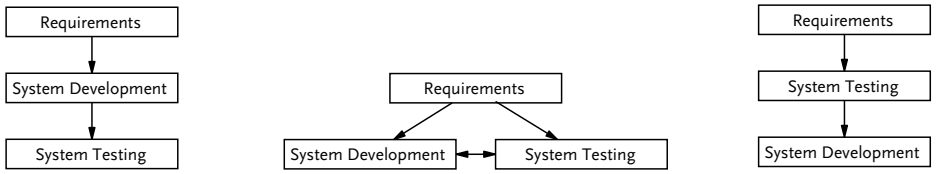


Figure 2.3: The V-Model.

Integrating the Test Process in System Engineering.

In this section, we present three basic approaches of integrating the test process into system engineering: (a) Testing after system development, (b) Running testing and system development concurrently, and (c) Starting with the tests (*test-driven development* [Bec02]). Figure 2.4 depicts these three approaches.



(a) Standard approach. (b) Concurrent approach. (c) Test-first approach.

Figure 2.4: Different approaches to integrate testing into system engineering.

Testing after System Development (a). The standard approach is daily work of software testers: System developers create or maintain components. After that, testers have to validate the SUT. In this approach, testing is conducted after SUT creation. The SUT has to be adapted if the tests detect failures in the SUT. The failures are often caused by incomplete or contradictory requirements. Changing requirements after the implementation phase often results in high costs.

Running Tests and System Development Concurrently (b). There is always limited time for testing. Therefore, it is advisable to start testing

as early as possible. The concurrent development of the test suites and the SUT is a step in this direction: System components are tested as soon as they become available. Faults are detected earlier, and the project management can react faster than in the first alternative. However, the problem of detecting faulty requirements after the implementation phase still remains. So, this approach can also result in high costs.

Test-Driven System Development (c). In test-driven development (see *extreme programming* [Bec00] or *agile system engineering* [BBvB+01]), test cases are created before system implementation starts. Consequently, the test cases fail at first, and the task of system development is to make the test cases pass. When writing down tests before implementation, the number of necessary changes in the requirements at the end of the implementation phase is reduced. A possible disadvantage is that the SUT could just be implemented with the aim to avoid the detection of failures.

2.1.5 Test Quality Measurement

As stressed before, testing cannot be complete in most cases. Common reasons for that are large input domains or infinite loops within the control flow. Even with finite domains and bounded loops, the test effort to cover all domain elements and all repetitions of loops is very high. As a consequence, there is need for other means of quality measurement than completeness. The quality of testing can be measured, e.g., as the probability that there are no faults remaining in the system under test. State of the art is to measure test quality by heuristic means. Beyond pure quality estimation, these means of test quality measurement are often used to steer the test generation and to decide when to stop testing. In the following, we present coverage criteria, mutation analysis, and a short comparison of them.

Coverage Criteria.

Coverage criteria are popular heuristic means to measure the quality of test suites. They can be applied to anything from requirements via models to machine code. There are several kinds of coverage criteria [AOH03, UPL06, UL06, AO08]. We focus on structural coverage criteria: They describe parts of the system behavior that must be covered by tests. They can reference single instructions or value assignments but also long sequences of instructions or data flow paths [FW88, Wey93, Hon01]. Coverage criteria can be compared with the help of *subsumption* relations [CPRZ85, Wei89]: Each test suite that satisfies the *subsuming* coverage criterion also satisfies the *subsumed* coverage

criterion. The subsuming coverage criterion is considered *stronger* than the subsumed one. There is, however, no proof of a relationship between satisfied coverage criteria and the number of detected failures: For instance, the satisfaction of a weaker coverage criterion can accidentally result in a higher number of detected failures than the satisfaction of a stronger coverage criterion. There are studies that show the relation between coverage criteria and the resulting test suite's fault detection capability [HLL94, CM94]. Furthermore, there are also studies that describe situations in which this relation is not substantiated [HT90, WJ91]. The results of these studies are fortified by the success of random testing [WJ91, Nta01, MS06, CLOM07, CPL+08] compared to model-based testing, which brings up the question for cost efficiency of model-based testing [Pre06].

This thesis is focused on model-based test generation from UML models. We will present model-based testing in Section 2.3. There are many coverage criteria for the various aspects of UML [UL06, page 120]. For instance, Nebut et al. [NF06] present a use-case-driven approach of automatic test generation. Andrews et al. present further coverage criteria that are focused on UML diagrams [AFGC03]. Briand et al. [BLL05] present an approach to use data-flow information to improve the cost effectiveness of coverage criteria for state machines. They focus their work on the round-trip-path (transition tree) coverage criterion [Bin99]. The results are that data-flow information can be used to select a better transition tree. We focus on generating tests that satisfy coverage criteria on UML state machines [Obj07, page 519]. Without claiming completeness, we present different kinds of coverage criteria and the subsumption relations between them. For that, we stick to the classification of coverage criteria as presented in [UL06] and [AO08]. We start by providing informal definitions of these coverage criteria. This informality is common for coverage criteria definitions. In Section 2.4, we will present formal definitions for coverage criteria.

Transition-Based Coverage Criteria. The here presented coverage criteria are focused on transition sequences [UL06, page 115]. Note that states are considered as transition sequences of length zero.

- *All-States:* A test suite that satisfies the coverage criterion All-States on a state machine must visit all states of the state machine.
- *All-Configurations:* State machines can contain parallel regions. A configuration is a set of concurrently active states. The satisfaction of All-Configurations requires that all configurations of the state machine's states are visited.

- *All-Transitions:* Satisfying the coverage criterion All-Transitions requires to traverse all transition sequences up to length one. The term "up to length one" includes length one and length zero. This definition of All-Transitions is selected to guarantee that All-Transitions subsumes All-States.

- *All-Transition-Pairs:* Similar to the definition of All-Transitions, All-Transition-Pairs requires to traverse all transition sequences up to length two. For the general case of All-n-Transitions, it is correspondingly necessary to traverse all transition sequences up to length n.

- *All-Paths:* This coverage criterion is satisfied iff all paths of the state machine are traversed. If there are unbounded loops, this criterion is impossible to satisfy or *infeasible*, respectively.

Control-Flow-Based Coverage Criteria. The here presented coverage criteria are focused on control flow, i.e., on value assignments for the state machine's guard conditions.

- *Decision Coverage:* To satisfy Decision Coverage, a test suite must cover the positive and negative evaluation, respectively, of all guard conditions of a state machine. Since it must also be decided whether to traverse transitions without guards, we define that Decision Coverage subsumes All-Transitions. There are other definitions of Decision Coverage corresponding to the focus of coverage criteria definitions in [AO08, page 34]. We come to that later on.

- *Condition Coverage:* Similar to Decision Coverage, Condition Coverage is satisfied iff all atomic boolean conditions of each guard are evaluated to true and false, respectively, at least once.

- *Decision/Condition Coverage:* This criterion is satisfied iff Decision Coverage and Condition Coverage are both satisfied.

- *Modified Condition/Decision Coverage:* Modified Condition/Decision Coverage (MC/DC) [CM94] is focused on the isolated impact of each atomic expression on the whole condition value. For this, the value of the condition must be shown to change if the atomic expression is changed and all other expression values are fixed. MC/DC is proposed in the standard RTCA/DO-178B for airborne systems and equipment certifications [RTC92]. Furthermore, the effort of satisfying MC/DC is linear with the number of atomic expressions of a condition [UL06,

page 114]. Thus, MC/DC is considered a sophisticated control-flow-based coverage criterion.

Due to interdependencies, it might be impossible to change one value and fix all others. For that, there are several kinds of MC/DC [Chi01, Cer01]: Unique-cause MC/DC requires to show the isolated impact of a condition on the guard. Masking MC/DC also allows other values to change as long as they do not influence the guard value. It is also possible to mix both form by applying unique-cause MC/DC as long as possible and applying masking MC/DC only if necessary. There are several applications of MC/DC, e.g. presented in [Pre03].

- *Multiple Condition Coverage:* The satisfaction of Multiple Condition Coverage (MCC) requires to use the values of each row of each guard condition's truth table. In other words, for each guard, all possible value assignments must be included in the test suite.

Data-Flow-Based Coverage Criteria. Data-flow-based coverage criteria are focused on the data flow of variables. Expressions can define and use variables. A variable is said to be *defined* if a new value is assigned to it. If this value is read, the variable is said to be *used*. Two expressions exp_1 and exp_2 along a path form a *def-use-pair* iff exp_1 defines a variable v, exp_2 uses v and there is no other definition of v between exp_1 and exp_2. There are several data-flow-based coverage criteria [UL06, page 114]:

- *All-Defs:* A test suite satisfies All-Defs iff for each definition d_v of a variable v, at least one def-use-pair (d_v, u_v) is tested.
- *All-Uses:* The satisfaction of All-Uses requires to test each existing def-use-pair at least once.
- *All-Def-Use-Paths:* To satisfy All-Def-Use-Paths, a test suite has to execute all the paths between all def-use-pairs.

Boundary-Based Coverage Criteria. Value partitions are constraints that specify sets of objects. Objects that satisfy/violate a value partition are said to be inside/outside the partition. For test data generation, values have to be selected from equivalence classes, i.e. input value partitions of a certain type. For ordered types T, *boundary values* of a partition P are inside of P and have a maximum distance d_{max} to a value outside of P. A *boundary edge* is a constraint of P [KLPU04]. For any partition representatives of type T, there is a distance function $dist : T \times T \rightarrow Integer$. We consider two

partitions $P1$ and $P2$. A boundary value of $P1$ at the edge between $P1$ and $P2$ is defined as a value $x \in P1$ with $\exists_{y \in P2} : dist(x, y) \leq d_{max}$. All boundary values at any edge of a partition P are *boundary values* of P. The following coverage criteria can be found in [KLPU04] and [UL06, page 124]. They are focused on just one value partition:

- *One-Boundary:* The coverage criterion One-Boundary is satisfied iff at least one boundary value of the partition is selected.
- *Multi-Dimensional:* The coverage criterion Multi-Dimensional is satisfied iff each variable is tested with the minimum and the maximum value of the corresponding value partition, respectively.
- *All-Edges:* A test suite that satisfies All-Edges contains at least one boundary value for each boundary edge of the partition.
- *All-Edges Multi-Dimensional:* This criterion is a combination of All-Edges and Multi-Dimensional. It is satisfied iff for each boundary edge each variable takes its minimum and maximum value at least once.
- *All-Boundaries:* This criterion is satisfied iff all boundary values are tested. This criterion is infeasible for anything but tiny domains.

Subsumption Hierarchy. The following figures show the subsumption hierarchies for the presented coverage criteria. Figure 2.5 shows the subsumption hierarchy for transition-based, control-flow-based, and data-flow-based coverage criteria. Figure 2.6 shows the subsumption hierarchy for boundary-based coverage criteria. Subsumption relations are depicted as arrows. Each arrow points from a subsuming coverage criterion to a subsumed one.

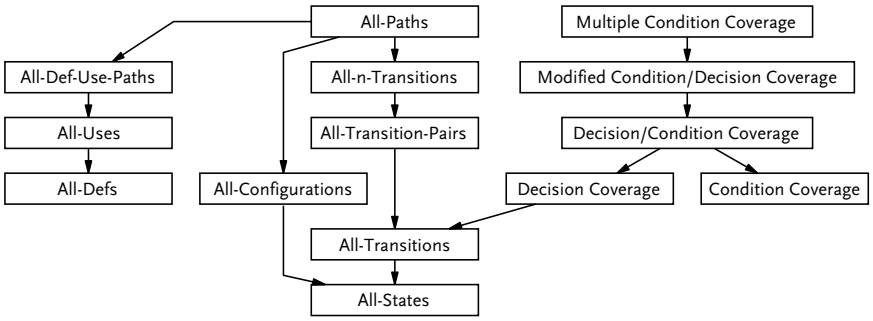


Figure 2.5: Subsumption hierarchy for structural coverage criteria that are focused on transitions, control flow, and data flow.

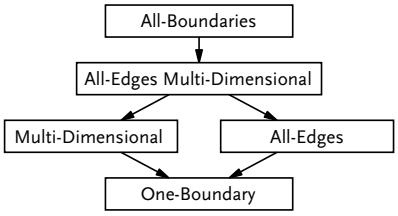


Figure 2.6: Subsumption hierarchy for boundary-based coverage criteria.

Note that there are two fundamental approaches to define coverage criteria. The first approach is focused on subsumption relations and is aimed at defining coverage criteria that subsume others [AO08, page 34]. The second approach is focused on defining each criterion on its own and on combining coverage criteria [UL06, page 133]. We consider subsumption an important means to compare coverage criteria. There are several examples in which assumed subsumption relations do not hold using the second approach. For instance, All-Transitions is assumed to subsume All-States. Using the second approach, All-Transitions just requires to visit all transitions [UL06, page 117]. For a state machine with one state and without transitions, an empty test suite would satisfy All-Transitions but not All-States. Thus, All-Transitions would not subsume All-States, which is a contradiction to the common assumption. We define coverage criteria so that the mentioned subsumption relations are guaranteed. For instance, Ammann and Offutt also define coverage criteria in this way [AO08, page 34]: All-Transitions is satisfied iff all paths up to length one are traversed. Since paths of length zero are interpreted as states, all states are covered if All-Transitions is satisfied. Since All-Transitions is considered the minimum coverage criterion to satisfy [UL06, page 120], it is reasonable to demand that other coverage criteria like Decision Coverage subsume All-Transitions.

As we stated, there are alternative definitions for coverage criteria. For instance, our definition for Decision Coverage represents a combination of Decision Coverage and All-Transitions as defined with the second approach. We formally define the used coverage criteria in Section 2.4.3.

Combination of Coverage Criteria. The presented coverage criteria aim at different elements of system descriptions like state machines. Thus, a combination seems to be reasonable. As presented in [UL06, page 134 ff.], a common approach is to select coverage criteria for requirements, data inputs, transition guards, and transition sequences. Antoniol et al. [ABDPL02] also present a case study that substantiates the advantages of combining coverage criteria. The combination of coverage criteria is also regarded as an alterna-

tive to the absolute comparability of coverage criteria: For instance, Decision Coverage can be satisfied together with All-Transitions and, thus, it does not necessarily has to subsume All-Transitions.

It is intuitive that the satisfaction of more coverage criteria brings a higher degree of test quality. As we presented in [FSW08], coverage criteria can be combined at arbitrary levels of abstraction by, e.g., uniting test suites (with or without traceability), uniting test goals, or uniting coverage criteria (find or define subsuming coverage criteria). However, such combinations are more like a union of coverage criteria than a combination of them. In Chapters 3 and 4, we present new ways of combining coverage criteria.

Mutation Analysis.

Mutation analysis is a technique to measure the fault detection capability of a test suite. It is quite similar to coverage criteria. The most important difference is that activated faults have to be propagated to the outside and also have to be detected for mutation analysis. *Mutation operators* can be applied to different levels of abstraction, e.g. to models [FDMM94, BOY00] or to implementations. We focus our work on applying mutation analysis to implementations. It is said that mutation analysis has first been proposed by Richard Lipton in 1971. The first publications about mutation testing were from DeMillo et al. [DLS78] and Hamlet [Ham77]. The basic idea of mutation analysis (see Figure 2.7) is to inject faults into a correct implementation using mutation operators. The faulty implementations are called *mutants*. The test suite is executed on each mutant with the goal to detect as many mutants as possible. If the test suite detects a failure of a mutant, this mutant is said to be *killed*. The number of all killed mutants divided by the number of all mutants is the *mutation score*. As described in Section 2.1.1, errors must be propagated to the outside before the test suite can detect them. There are two different kinds of mutation analysis that deal differently with this

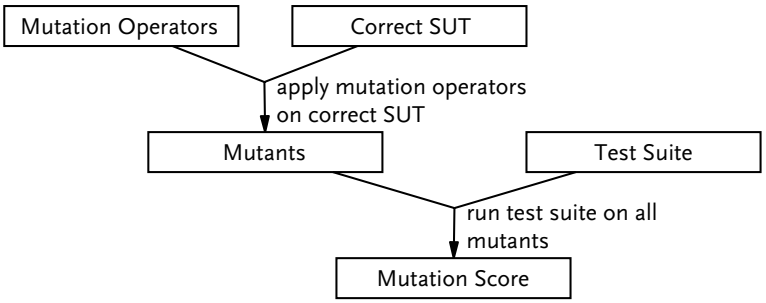


Figure 2.7: The basic process of mutation analysis.

fact: The just presented approach is called *strong mutation analysis* – the propagation is necessary. *Weak mutation analysis* [How82, GW85, WH88, HM90, Mar91, OL91, OL94] is able to detect faults as soon as they result in an error – propagation is not necessary [AO08, page 178].

Mutation Operators. Mutation operators define how to change details of an artifact like an implementation or a model. Theoretically, all mutants can be created manually. However, it is state of the art to create them automatically by using mutation operators. This requires that there is a formal description of mutation operators. Several mutation operators have already been declared for software [OL94] and for specifications [BOY00]. Many languages have been used for mutation analysis. Some examples are [OK87, OLR+96] for Fortran77, [DM96] for C, [Bow88, OVP96] for Ada, and [CTF02, IPT+07, SW07] for Java. In the following, we present the mutation operators that are also used in our case studies.

- *ABS - Absolute Value Insertion:* An absolute value is inserted where a variable was used before.
- *LOR - Logical Operator Replacement:* Logical operators ($\wedge$, $\vee$, *not*) are replaced by other logical operators inside logical expressions.
- *ROR - Relational Operator Replacement:* Relational operators ($<$, $\leq$, $=$, $<>$, $\geq$, $>$) are exchanged inside mathematical expressions.
- *AOR - Arithmetic Operator Replacement:* Arithmetic operators ($+$, $-$, $*$, $/$) are exchanged inside mathematical expressions.
- *UOI - Unary Operator Insertion:* Unary operators are inserted anywhere (for arithmetic expressions: $-$, for boolean expressions: *not*).
- *MCO - Missing Condition Operator:* Subexpressions are removed from the expression. This operator corresponds to "forgetting" parts of expressions.
- *TSO - Target State Operator:* The target state of a transition is changed respectively the corresponding operation call in the SUT.

The first five operators compose the set of *sufficient mutation operators* defined by Offutt et al. [OLR+96]. The sixth operator is presented as a frequently occurring fault in [BOY00]. The last mutation operator is specific for UML state machines.

Comparison of Mutation Operators. The comparison of mutation operators concerning their appropriateness has been the topic of many discussions [OLR+96]. A fundamental question is whether simple mutation operators that change only small details of the system are better than complex mutation operators or vice versa. The difference is obvious: Simple mutation operators produce simple faults, complex mutation operators produce complex faults. The *coupling effect* [DLS78] states that complex and simple faults are coupled in a way that the detection of all simple faults implies the detection of most complex faults. There are also case studies [Off92] and theoretical considerations [Wah03] that support the coupling effect. As a consequence, simple mutants are sufficient for mutation analysis. Thus, we will only apply the presented set of simple mutation operators.

Comparison to Real Faults. The mutants resulting from applying mutation operators have been subject to several case studies [ABL05, Par05, ABLN06, NAM08, SW09]. In [ABLN06], the predictability of a test suite's fault detection capability by detecting mutants derived from mutation operators is investigated. The real faults of the case study were taken from space.c, a program developed by the European Space Agency. Furthermore, Andrews et al. [ABL05] compare the fault detection capability of test suites for real faults of the same case study space.c, mutants derived from mutation operators, and faults that are manually inserted by experienced programmers. The result is that the mutation score of mutation analysis is a good predictor for the test suite's fault detection capability of real faults. Another result is that manually inserted faults are often harder to detect than the average real fault. Thus, predicting the fault detection capability by using manually inserted faults probably underestimates the test suite's fault detection capability.

Coverage Criteria vs. Mutation Analysis.

Coverage criteria and mutation analysis are two means of test quality measurement. Whereas the satisfaction of coverage criteria just requires the test suite to cover certain elements of the system or the test model, mutation analysis (weak or strong) requires the test suite to let the system be in a different state and show a different behavior for mutants, respectively.

There are relations between coverage criteria and mutation analysis. For instance, a test suite that detects a mutant that changes a certain part of the system implies that this test suite also covers this part. For detecting a failure, the propagation of the corresponding error is necessary, which may require further test behavior. Because of the missing need of this propagation

for coverage criteria, weak mutation is more appropriate to compare mutation operators and coverage criteria. If the detection of all mutants for a certain mutation operator results in the satisfaction of a certain coverage criterion, then it is said that the mutation operator *yields* the coverage criterion [AO08, page 186]. Ammann and Offutt present further mutation-based coverage criteria in [AO08].

2.1.6 Further References

In this section, we present a list of testing-related books for further reading and mention standard tools for MBT. Famous books about testing are Beizer's red book "Software Testing Techniques" [Bei90] and "Testing Object-Oriented Systems: Models, Patterns, and Tools" [Bin99] from Binder. Myers published "The Art of Software Testing" [Mye79]. Whittaker provides many practical examples in his book "How to Break Software" [Whi02]. Kaner et al. publish "Testing Computer Software" [KFN99]. Beck puts emphasis on the test-first approach in the book "Test Driven Development: By Example" [Bec02]. In 2007, Ammann and Offutt had a detailed look at many different aspects of software testing in "Introduction to Software Testing" [AO08]. Chapter 5 of SWEBOK [Ber00] also provides a good survey of software testing. Broy et al. published one of the first books about model-based testing: "Model-Based Testing of Reactive Systems: Advanced Lectures (Lecture Notes in Computer Science)" [BJK05]. The book "Practical Model-Based Testing: A Tools Approach" [UL06] from Utting and Legeard was published in 2006 and is also focused on model-based testing. It provides many tool approaches and case studies. There are further introductions to model-based testing, e.g. by Prowell [Pro04], Robinson [Rob06], and Peleska et al. [PML08]. The UML testing profile supports the creation of test models. In the book "Model-Driven Testing – Using the UML Testing Profile", Baker et al. [BDG+07] provide a corresponding survey.

There are many tools for model-based testing. Examples for commercial tools are the Smartesting Test Designer [Sma], Conformiq Qtronic [Con], AETG of Telcordia Technologies [Tel], Microsoft's SpecExplorer [Mic09], PikeTec's TPT [Pik09], and Reactis of Reactive Systems [Rea09]. Several tools for model-based testing are presented and compared in [GNRS09]. Aydal et al. [AUW08] also present a comparison of model-based testing tools. The AGEDIS project with all published documents provide information about the tools necessary for automatic model-based testing [HN04].

2.2 Modeling Languages

Models are purposeful abstractions of a certain artifact. They are instances of meta models or modeling languages, respectively. There are many different modeling languages that have been used to create test models. Some examples are Abstract State Machines [BGN+03], the Unified Modeling Language (UML) [BLC05, SHS03, BBM+01], the Object Constraint Language (OCL) [BBH02, AS05], or Object-Z [MMSC98]. This thesis is focused on model-based test generation from models of the UML and the OCL. We present a short introduction to both languages.

2.2.1 Unified Modeling Language

The UML in version 2.1 is a modeling language with graphical notation that defines 13 diagrams to describe structural and behavioral system properties. It is now under control of the Object Management Group [Obj07]. The initial version of the UML was developed in 1998 by Booch, Rumbaugh, and Jacobson [BRJ98]. It is based on the Meta Object Facility (MOF) [Obj06]. There are many editors, test generators, and further tools that support using the UML notation.

This thesis is mainly focused on UML state machines. In Section 5, we describe new approaches to combine UML state machines with UML class diagrams and UML interaction diagrams. In the following, we briefly sketch the main elements of these three diagrams.

State Machines.

State machines are behavioral diagrams of the UML. They are based on state charts, which were first introduced by Harel [Har87] as finite state machines with hierarchy, concurrency, and communication. State machines are integrated in the UML, i.e., they can reference classes, operations, etc., and they can also be referenced from other diagrams. Each state machine defines states of a system and state changes. UML state machines are specified on more than 50 pages in [Obj07]. This section is restricted to presenting only fundamental aspects: Each state machine contains a set of parallel regions. Each region contains vertices and transitions. Vertices can be pseudostates, states, or connectionpointreferences. If a system object is in a certain state, this state is *active*. Transitions contain events ev, a guard g, and an effect ef. The transition is denoted with $ev[g]/ef$. A transition is *activated* if the transition's source state is active, one of the transition's events is triggered, and the guard condition is true. Activated transitions can be traversed: The

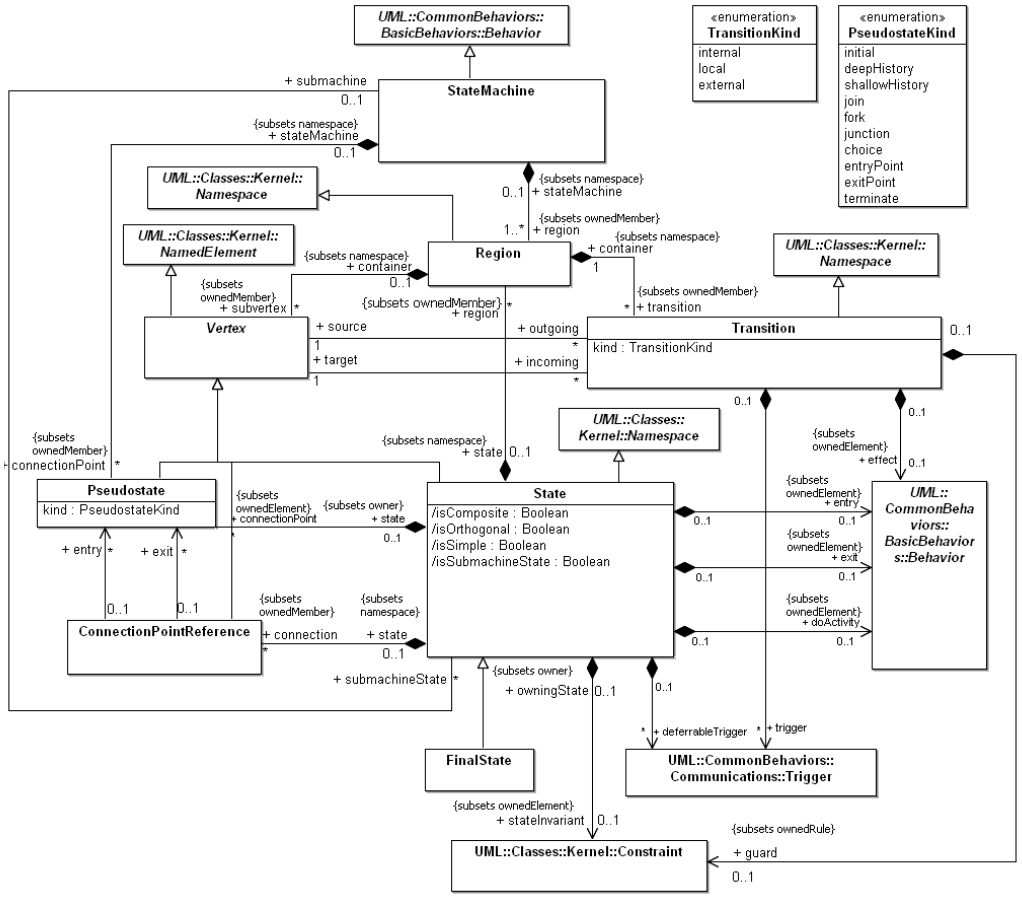


Figure 2.8: Meta model for state machines (UML 2.1 specification).

transition's source state becomes inactive, the transition's effect is executed, and the transition's target state becomes active. Note that concurrently triggering two events can result in non-determinism. Figure 2.8 shows the meta model for UML state machines (corresponding to UML 2.1). Section 3.2 contains several example state machines.

There are many semantics defined for state machines [HG97, LMM99]. The UML is a semi-formal specification, and the semantics of UML state machines are often defined by presenting a transformation to a formal language like Kripke structures [LMM99] or extended finite state machines (EFSMs) [KHBC99]. Harel and Naamad define the STATEMATE semantics for state charts [HN96] – a corresponding tool was developed at I-Logix. Real-time systems can be modeled with the stateflow notation supported by the MATLAB system. In general, there are many different semantics for state machines, and each tool vendor defines a new one.

For these reasons, we use a basic and intuitive definition of state machine semantics: State changes are possible from each state. They are triggered by events like, e.g. operation calls. Transitions are traversed until a state is reached with no activated transition (see *run to completion* [Obj07, page 559] and *compound transitions* [Obj07, page 568]).

State machines can be used for different levels of testing (see Section 2.1.4). For instance, if one class behavior is described, then test generation from state machines can be used for unit testing. If several classes are described that are, e.g., comprised in a subsystem, then this can be used for integration testing. These two options are often applied. Additionally, describing the behavior of all classes would allow to generate tests for system testing. Eshuis and Wieringa [EW00] consider state machine semantics at the requirements level.

Class Diagrams.

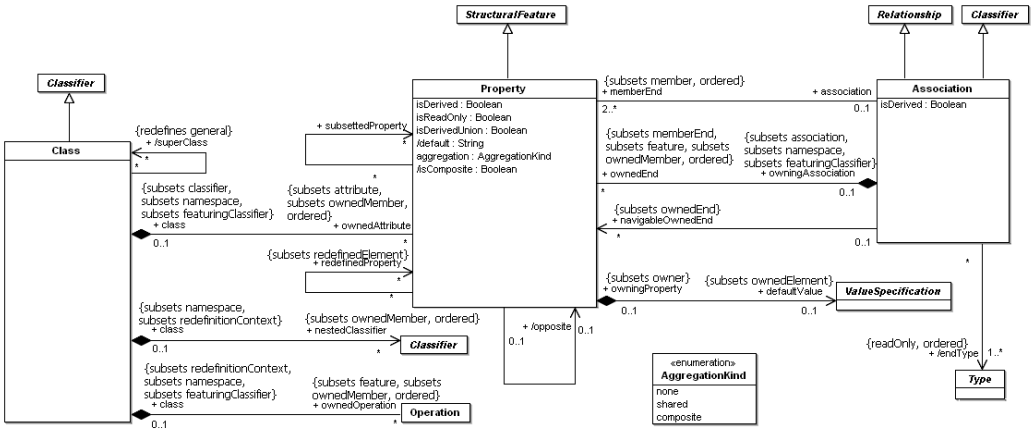


Figure 2.9: Meta model for classes (UML 2.1 specification).

Class diagrams describe the structure of a system. Each class can contain classifiers (e.g. other classes), attributes, and operations. Relations between classes are described using associations, aggregations, and compositions. Classes can be derived from other classes. They can be abstract or concrete. More information is provided in [Obj07]. Figure 2.9 is taken from the UML 2.1 specification and shows the class meta model [Obj07, page 32].

Classes of class diagrams can also contain behavioral system properties like operations with their pre- and postconditions. There are approaches to derive test cases based on the pre-/postconditions of operations [AS05]. State machines are often used to describe the behavior of classes. For that,

the state machine is an element of the class and can reference elements of the class. Operations are behavioral elements and, thus, can be referenced from the effect of a state machine's transition. Our test generation approach in Chapter 3 will make use of that.

Interaction Diagrams.

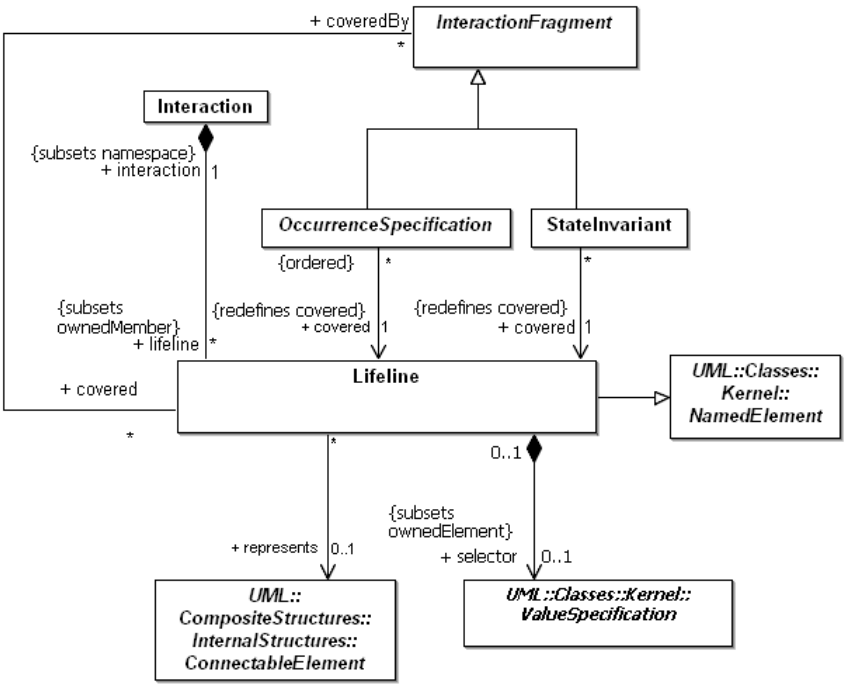


Figure 2.10: Meta model for interactions and life lines (UML 2.1).

Interaction diagrams are behavioral diagrams of the UML. They describe the interaction of several objects. For that, each interaction contains several life lines, which reference the represented objects. Figure 2.10 is taken from the UML 2.1 specification [Obj07, page 459] and shows these relations.

The specification defines that life lines are connected by messages that are exchanged between the corresponding objects. On page 460, the UML specification shows that the class *MessageOccurrenceSpecification* references the class *Event*. The same events (instances of the class *Event*) can also trigger transitions in UML state machines. In Section 5.2, we present the combination of interaction diagrams and state machines based on this usage of the same events.

2.2.2 Object Constraint Language

The Object Constraint Language (OCL) [Obj05a] is a textual language to express constraints. It has been developed by Mark Richters [RG98] and is used to complement the UML. OCL can be used for contract-based design, for which Traon [lT06] also defines vigilance and diagnosibility but does not use it for test case generation. Just like UML, the OCL is now a OMG specification. The current version is 2.0. Several books describe the use of this language in combination with the UML [Jos99, CKM+02, WK03]. It can be used to create queries on the system and to specify invariants, pre-, or postconditions [Obj05a, page 5]. Typical applications of OCL constraints in the afore sketched UML diagrams are state invariants in state machines, class invariants, and pre- and postconditions of operations. There is tool support for OCL like the Dresden OCL toolkit [Fin00], the Object Constraint Language Environment (OCLE) [LCI03], the USE tool [ZG03], and the OCL Eclipse plug-in [Ecl05].

In this section, we present some of the fundamental elements of OCL. Each OCL constraint is described as follows: "context:" <context-def> <kind> <expression>. The context definition (<context-def>) references the namespace of the OCL expression. All elements inside this namespace can be directly used inside the expression. The kind of the expression (<kind>) can, e.g. be "inv:", "pre:", or "post:": State invariants are defined with "inv:", an operation's precondition with "pre:", and an operation's postcondition with "post:". The expression (<expression>) is a boolean or an arithmetical expression. Within postconditions, "@pre" denotes the value of a variable before the execution of the operation.

We present two example statements of OCL expressions:
The following invariant expresses that a company has an employee with the forename Jack:

```
context: Company
  inv: employee->exists(forename = 'Jack')
```

A postcondition can be used to require that the amount of money on a bank account has been raised because of an incoming payment:

```
context: Account::deposit(value : Integer)
  post: money = money@pre + value
```

2.3 Model-Based Testing

Model-based testing usually means functional testing for which the test specification is given as a test model. The test model is derived from the system requirements. There are only a few approaches to use model-based testing for non-functional tests [BDL05, SEBC09]. In model-based testing, test suites are derived (semi-)automatically from the test model. Coverage criteria are often considered at the test model level. The interna of the SUT are not necessarily visible (black-box or gray-box testing). Model-based testing can be applied to all levels from unit tests to system tests. Acceptance tests are usually not covered because user acceptance often also depends on many imprecise expectations. Figure 2.11 shows the kinds of testing, model-based testing can be applied to. Similar graphics are presented by Tretmans [Tre04] and by Utting and Legeard [UL06].

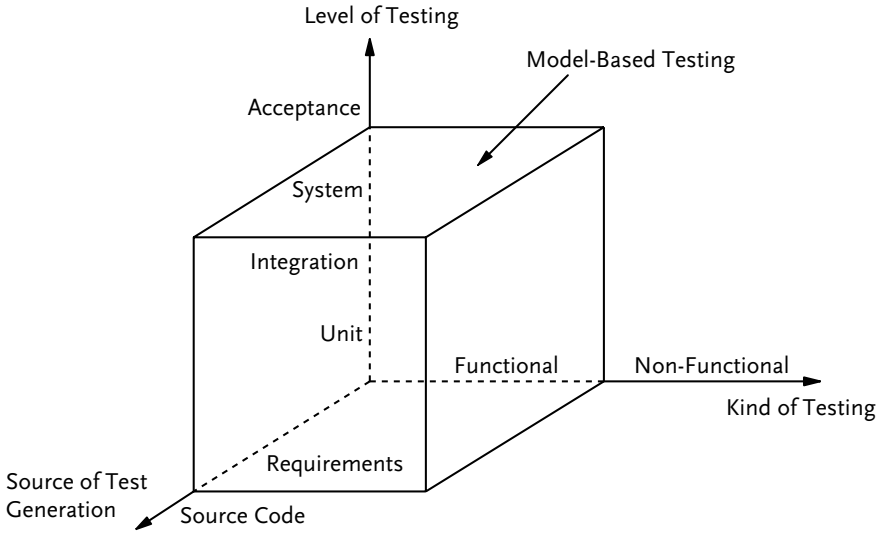


Figure 2.11: Application fields of model-based testing.

Model-based testing plays an important role for model-driven software verification [Utt08]. There are several advantages of model-based testing: First, the test model is usually quite small, easy to understand, and easy to maintain. Second, the use of test models often allows traceability from requirements to test cases. Third, model-based testing can be used for testing after system development as well as for test-first approaches. As discussed in Section 2.1.4, test-first approaches help reducing costs. Furthermore, experience shows that the early creation of formal test models also helps in finding faults and inconsistencies within the requirements [UL06, page 28]. Fourth,

the test model can be used to automatically generate small or huge test suites that satisfy a corresponding coverage criterion. This allows to generate either small and quickly executed test suites or big ones with a high fault detection capability. This thesis is focused on the fourth advantage: automatic test generation. Figure 2.12 depicts the standard procedure for automatic model-based test generation: Automatic test generation creates a test suite that satisfies a certain coverage criterion on the test model [PPW+05]. Afterwards, the test suite is executed. The result is compared using the test oracle.

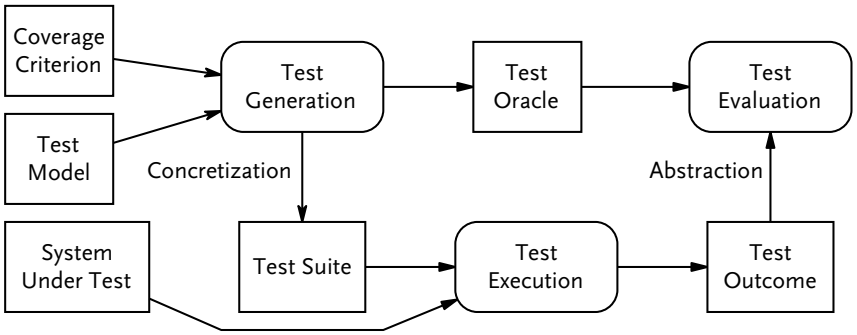


Figure 2.12: Concretization and abstraction in model-based testing.

This section contains a sketch of different test generation approaches, the positioning of this thesis within the field of model-based testing, and a short comparison of model-based testing to conventional testing.

2.3.1 Approaches to Model-Based Testing

In this section, we present a survey of techniques that are used for model-based testing. We shortly sketch graph search algorithms, random testing, evolutionary testing, constraint solving, model checking, static analysis, abstract interpretation, partition testing, and slicing.

Graph Search Algorithms.

UML state machines and many other behavioral models are kinds of graphs. The described behavior is the sum of all possible paths through these graphs. *Graph search algorithms* can be used to find paths with certain properties [Kor90, GMS99, MS04, McM04, LHM08, ATF09]. Chow [Cho95] creates tests from a finite state machine by deriving a testing tree using a graph search algorithm. Offutt and Abdurazik [OA99] identify elements to be covered in a UML state machine and apply a graph search algorithm to cover

them. Other algorithms also include data flow information [BLL05] to search paths. Harman et al. [HHL+07] consider reducing the input space for search-based test generation. Gupta et al. [GMS98] find paths and propose a relaxation method to define suitable input parameters for these paths.

Random Testing.

Many test generation approaches put a lot of effort on generating test cases from test models in a "clever" way. It is to be discussed whether this effort is always justified [Pre06]. Especially in black-box testing, there are many unknown interna, and the derivation of test information is a costly process. *Statistical approaches* to testing like *random testing* are successful in many application areas [BM83, May05, ODC06, CLOM06, UPL06, CLOM07, CPL+08].

In random (fuzz) test approaches, usually a huge number of test cases is created without spending much effort on the quality of the single tests. Statistical considerations assume that faults are randomly spread over the whole program. In such cases, random testing has often advantages over any kind of guided test generation. The assumption that faults are often close to partition boundaries would change this. In [ABLN06], Andrews et al. use a case study to show that random tests can perform considerably worse than coverage-guided test suites in terms of fault detection and cost effectiveness. However, the effort of applying coverage criteria cannot be easily measured, and it is still unclear which approach results in higher costs. Mayer and Schneckenburger [MS06] present a systematic comparison of adaptive random testing techniques. They focus their work also on the comparison of random testing to partition testing. Gutjahr [Gut99], Weyuker, and Jeng [WJ91] also compare random testing to partition testing. Major reasons for the success of random testing techniques are that other techniques are immature to a certain extent or that the used requirements specifications are partly faulty. The main reason for the latter is that humans make errors: developers as well as testers (see Section 2.1.2 for the prejudice *Angelic Testers*). Testers often forget some cases or simply do not know about them.

The question for the applicability of statistical testing is still subject to research. For instance, if the SUT has a complex internal state system, then probably only a few long sequences of input stimuli with a low probability of being generated lead to certain internal states. In such cases, statistical testing will probably return no satisfying results, or it will need a long time until these sequences are generated. An adequate application domain of random testing is, e.g. library testing [CPL+08].

Evolutionary Testing.

There are several approaches to steer test generation or execution with *evolutionary approaches* [MMS97, PHP99, KG04, HM07, WS07]: An initial (e.g. randomly created or arbitrarily defined) set of test input data is refined using mutation and fitness functions to evaluate the quality of the current test suite. For instance, Wegener et al. [JW01] show application fields of evolutionary testing. A major application area are embedded systems [SBW01]. Wappler and Lammermann apply these algorithms for unit testing in object oriented programs [WL05]. Bühler and Wegener present a case study about testing an autonomous parking system with evolutionary methods [OB04].

Baudry et al. [BFJT02] present *bacteriological algorithms* as a variation of mutation testing and as an improvement of genetic algorithms. The variation from the genetic approach consists of the insertion of a new memory function and the suppression of the crossover operator. They use examples in Eiffel and a .NET component to test their approach and show its benefits over the genetic approach for test generation.

Constraint Solving.

The constraint satisfaction problem is defined as a set of objects that must satisfy a set of contraints. The process of finding these object states is known as *constraint solving*. There are several approaches to constraint solving depending on the size of the application domain. We distinguish large but finite and small domains. For domains over many-valued variables, such as scheduling or timetabling, *Constraint Programming* (CP) [RvBW06], *Integer Programming* (IP) [Rav08], or *Satisfiability Modulo Theories* (SMT) [BSST09] with an appropriate theory is used. For extensionally representable domains, using solvers for *Satisfiability* (SAT-Solver) [BHvMW09] and *Answer Set Programming* (ASP) [Bar03, Gel08] are state of the art. SAT is often used for hardware verification [DEFT09].

There are many tools (*solvers*) to support constraint solving techniques. Examples for constraint programming tools are the Choco Solver [The09], MINION [GJK+09], and Emma [Eve09]. Integer programming tools are OpenOpt [Opt07] and CVXOPT [DV09]. An example for SMT solvers is OpenSMT [SBT+09]. There are several competitions for solvers [BDOS08, vMF09, DVB+09]. Constraint solving is also used for testing. Gupta et al. [GMS98] use constraint solver to find input parameter values that enable a generated abstract test case. Aichernig and Salas [AS05] use constraint solvers and mutation of OCL expressions for model-based test generation. Calame et al. [CIvdPS05] use constraint solving for conformance testing.

Model Checking.

Model checking was initially developed by Edmund Clarke and Allen Emerson, and by Jean-Pierre Queille and Joseph Sifakis. As the name of this technique states, properties of a model are checked. The model checking algorithm tries to build a state space from the model to deduce whether the model meets the property for certain (e.g. at least one or all) states. Typical tasks are the detection of deadlocks or livelocks. The representation of all states usually leads to the state explosion problem, which is tried to be avoided by, e.g., partial order reduction, model slicing, or a compact representation of states. If a model checker deduces that a property does not hold, then it returns a path in the model as a corresponding counter-example.

Model checking is often used for automatic test generation [ABM98, GH99, FW08b]. For that, the test model and the coverage criterion to satisfy are used: The coverage criterion is expressed as a set of properties (e.g. a certain state is reached), whose elements are negated and checked by the model checker. If the model checker deduces that the test model does not meet the negated property (respectively meets the original property), it returns a corresponding path. This path meets the original property and is used to create a test case. If there are test cases for all properties of the used coverage criterion, these test cases satisfy the used coverage criterion.

Model checking and test generation have been combined in different settings. Hong et al. [HLSC01] discuss the application of model checking for automatic test generation with control-flow-based and data-flow-based coverage criteria. They define state machines as Kripke structures [CGP00] and translate them to inputs of the model checker SMV [ITC98]. The applied coverage criteria are defined and negated as properties in the temporal logic CTL [CGP00]. Callahan et al. [CSE96] apply user-specified temporal formulas to generate test cases with a model checker. Gargantini and Heitmeyer [GH99] also consider control-flow-based coverage criteria. Abdurazik et al. [AADO00] present an evaluation of specification-based coverage criteria and discuss their strengths and weaknesses when used with a model checker. In contrast, Ammann et al. [ABM98] apply mutation analysis to measure the quality of the generated test suites. Ammann and Black [AB00] present a set of important questions regarding the feasibility of model checking for test generation. Especially, the satisfaction of more complex coverage criteria like MC/DC [CM94, Chi01] is hard because their satisfaction often requires pairs of test cases. Okun and Black [OB03] also present a set of issues about software testing with model checkers. They describe, e.g., the higher abstraction level of formal specifications, the derivation of logic constraints, and the visibility of faults in test cases. Engler and Musuvathi [EM04] compare model

checking to static analysis. They present three case studies that show that model checking often results in much more effort than static analysis although static analysis detects more errors than model checking. In [JM99], a tool is demonstrated that combines model checking and test generation. Further popular model checkers are the SPIN model checker [Bel91], NuSMV [ITC99], and the Java Pathfinder [HVL+99].

Static Analysis.

Static analysis is a technique for collecting information about the system without executing it. For that, a verification tool is executed on integral parts of the system (e.g. source code) to detect faults (e.g. unwanted or forbidden properties of system attributes). There are several approaches and tools to support static analysis that vary in their strength from analyzing only single statements to including the whole source code of a program. Static analysis is known as a formal method. Popular static analysis tools are the PC-Lint tool [Gim85] for C and C++ or the IntelliJ IDEA tool [Jet00] for Java. There are also approaches to apply static analysis on test models for automatic test generation [BFG00, OWB04, CS05, PLK07, PZ07]. Abdurazik and Offutt [AO00] use static analysis on UML collaboration diagrams to generate test cases. In contrast to state-machine-based approaches that are often focused on describing the behavior of one object, this approach is focused on the interaction of several objects. Static and dynamic analysis are compared in [AB05]. Ernst [Ern03] argues for focusing on the similarities of both techniques.

Abstract Interpretation.

Abstract interpretation was initially developed by Patrick Cousot. It is a technique that is focused on approximating the semantics of systems [Cou03, CC04] by deducing information without executing the system and without keeping all information of the system. An abstraction of the real system is created by using an *abstraction function*. Concrete values can be represented as abstract domains that describe the boundaries for the concrete values. Several properties of the SUT can be deduced based on this abstraction. For mapping these properties back to the real system, a *concretization function* is used. The abstractions can be defined, e.g., using Galois connections, i.e., a widening and a narrowing operator [CC92]. Abstract interpretation is often used for static analysis. Commercial tools are, e.g., Polyspace [The94] for Java and C++ or ASTRÈE [CCF+03]. Abstract interpretation is also used for testing [Cou00, PW02].

Partition Testing.

Partition testing consists of creating value partitions of input parameters and selecting representatives from them [HT90, WJ91, Nta01] [BJK05, page 302]. This selection is important to reduce the costs of testing. The category partition method [OB88] is a test generation method that is focused on generating partitions of the test input space. A further prominent approach for category partition is the classification tree method (CTM) [GG93, DDB+05], which enables testers to define arbitrary partitions and to select representatives. The application of CTM to testing embedded systems is demonstrated in [LBE+05]. Alekseev et al. [ATP+07] propose the reuse of classification tree models. Basanieri and Bertolino use the category classification approach to derive integration tests with use case diagrams, class diagrams, and sequence diagrams [BB00]. The Cost-Weighted Test Strategy (CoWTeSt) [BBM, BBM+01] is based on prioritizing classes of test cases in order to restrict the number of necessary test cases. CoWTeSt and the corresponding tool CowSuite have been developed by the PISATEL laboratory [PIS02]. Another means to select test cases by partitioning and prioritization is the risk-driven approach presented by Kolb [Kol03].

Slicing.

Slicing is a technique to remove parts of a program or a model in order to remove unnecessary parts and simplify, e.g., test generation. The idea is that slices are easier to understand and to generate tests from than with the whole program or model [HD95]. Program slicing was introduced in the Ph.D. thesis of Weiser [Wei79]. De Lucia [dL01] discusses several slicing methods (dynamic, static, backward, forward, etc.) that are based on statement deletion for program engineering. Fox et al. [FHH+01] present backward conditioning as an alternative to conditioned slicing that consists of slicing backward instead of forward: Whereas conditioned slicing provides answers to the question for the reaction of a program to a certain initial configuration and inputs, backward slicing finds answers to the question of what program parts can possibly lead to reaching a certain part or state of the program. Jalote et al. [JVSJ06] present a framework for program slicing.

Slicing techniques can be used to support partition testing. For instance, Hierons et al. [HHF+02] use the conditioned slicing [CCL98] tool ConSIT for partition testing and to test given input partitions. Harman et al. [HFH+02] investigate the influence of variable dependence analysis on slicing and present the corresponding prototype VADA. Dai et al. [DDB+05] apply partition testing and rely on the user to provide input partitions. Tip

et al. [TCFR96] present an approach to apply slicing techniques to class hierarchies in C++. In contrast to the previous approaches, this one is focused on slicing structural artifacts instead of behavioral ones.

2.3.2 Positioning of this Thesis

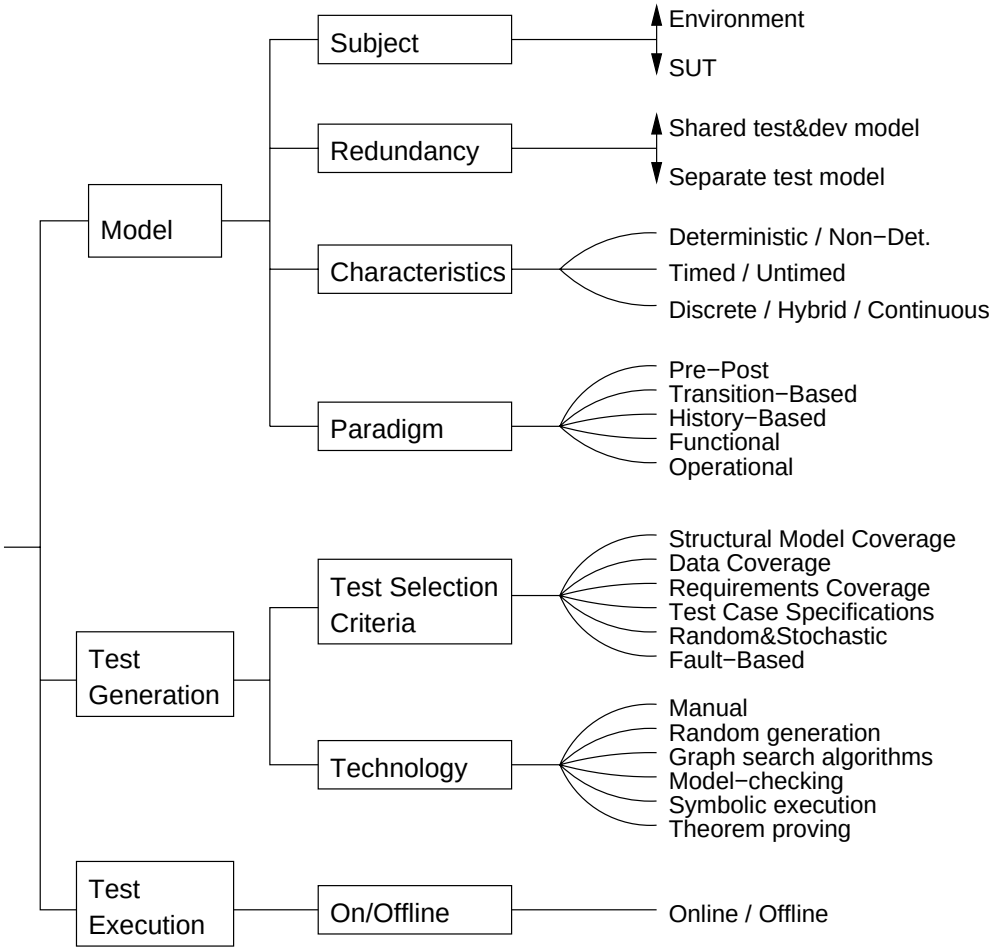


Figure 2.13: Taxonomy according to Utting, Pretschner, Legeard [UPL06] (used with permission).

In this section, we position the used test models and the applied test generation algorithms of this thesis in the context of model-based testing.

Figure 2.13 shows a taxonomy for model-based testing that is taken from [UPL06]. It describes the typical aspects of model-based test generation and test execution. Its focus is on the various kinds of models and test

generation techniques. We use this taxonomy to position our thesis: The subject of our test models is the **SUT**. The test models are **separate** from the development models. Furthermore, the used test models are **deterministic**, **untimed**, and **discrete**. The paradigm of the test model is **pre-post** as well as **transition-based**, i.e. the evaluation of pre-/postconditions is combined with a guided depth-first graph search algorithm. Satisfying **structural model coverage** is used to steer the test generation. The search technology is based on a combination of **graph search algorithm** and **symbolic backward execution**. The test execution is mainly **offline** although online test generation is simulated for the experiments in Chapter 6.

Figure 2.14 is also taken from [UPL06]. It describes the degree to which a test model describes the SUT or its environment. Model *S* describes the SUT but has no information about the environment. This can be an advantage as well as a disadvantage. As an advantage, the SUT is tested independent of the environment and is expected to cope equally well with all environments. As a disadvantage, the tests do not take the conditions of the environment into account, which could help, e.g., to narrow the possible input data space. Model *E* just describes the environment but has no information about the SUT. Model *SE* contains information about the SUT and the environment. For all these models, it is most important to abstract. Otherwise, the test model's complexity would be too high to handle. This is shown with the three models *M1*, *M2*, and *M3*. Our approach is focused on models that describe the SUT. There is, however, no restriction to the extent to which the SUT's environment is included in the test model. Thus, the used models can be described with *M1* or *M2*.

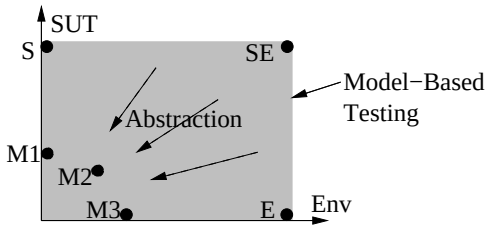


Figure 2.14: Test models describe the SUT or its environment according to Utting, Pretschner, Legeard [UPL06] (used with permission).

2.3.3 Comparison to Conventional Testing

In our context, conventional testing means one of two things: manual test creation or automatic code-based test generation.

In many companies, tests are manually designed by testers or by developers – both approaches have advantages and disadvantages [Bec00]. A disadvantage that they share is the high effort to maintain test suites. There is no traceability from requirements to test cases. As a result, for each change of requirements, someone has to check all existing test suites manually. From our industrial cooperation projects, we know that even for small changes in small projects at least one man-week has to be sacrificed to update the test suite. In model-based testing, the initial test model development is also costly. In contrast to manual testing, however, the test maintenance in model-based testing is cheap: The test model is adapted once and the whole test suite is automatically generated afterwards. Furthermore, the test model is often far more easy to understand than the test suite.

In automatic code-based test generation, the functionality of the system's source code is tested. There are many provers and model checkers that work on source code. They allow to find inconsistencies in the control flow or data flow that may lead to unwanted system behavior, e.g. system crash. Due to the missing redundant test specification, code-based testing is unable to detect functional faults. This also excludes the detection of missing functionality.

As presented in Section 2.1.4, costs for testing increase with the SUT's degree of completion. Manual test creation is costly, and code-based test generation has the drawback that testing can only start after implementation. In contrast, model-based testing allows to find faults before the implementation phase. The creation of formal and coherent test models results in many questions that can often help to detect inconsistencies in the requirements specification. As an example, Holt et al. [HAA+06] present experiences of a case study about the advantages of precise modeling with state machines for safety-critical applications.

2.4 Coverage Criteria Formalization

Coverage criteria are a popular means to measure the fault detection capability of test suites and to steer test suite generation. Their definitions are, however, often vague and informal. It is hard to even find a definition of what coverage criteria actually are. In this section, we name some of the resulting issues and present formal definitions of coverage criteria, test goals, and their satisfaction.

2.4.1 Issues of Current Coverage Criteria Definitions

Although coverage criteria have been used for decades, their descriptions are often vague, vary depending on the artifact they are applied to (e.g. models or source code), and depend on the cited author.

For instance, the coverage criterion All-Transitions requires to traverse all transitions of a state machine. There seems to be no general agreement about the traversal of a composite state's outgoing transitions (cp. [UL06, page 117]): Are they to be traversed for each substate of the composite state or just once? One solution consists of applying the same coverage criterion once to a hierarchical and once to a flattened state machine.

As another example, All-Transition-Pairs requires to cover all adjacent transitions [UL06, page 118]. However, outgoing transitions of a composite state s and incoming transitions of s's final states are consecutively traversed but not adjacent and, thus, not necessarily covered? Flattening the state machine can help to enforce the inclusion of such transition pairs.

Subsumption is used to compare coverage criteria. It depends on the definition of coverage criteria. For instance, the subsumption relation between All-States and All-Transitions seems to be obvious: If all transitions are traversed, then also all states are visited and, thus, All-Transitions subsumes All-States? As stated above, one problematic (although artificial) scenario is a state machine with just one state and no transitions: An empty test suite would satisfy All-Transitions but not All-States, which contradicts the afore mentioned subsumption relation.

2.4.2 Coverage Criteria and Their Satisfaction

In this section, we formally define coverage criteria, test goals, and how to satisfy them. Figure 2.15 shows all the used symbols. They are explained in the following. $\mathscr{P}(X)$ denotes the power set of a set X.

State Machines:	SM
Step Patterns:	SP
Step Coverage:	$SPCov$
Trace Patterns:	TP
Trace Coverage:	$TPCov$
Atomic Test Goals:	ATG
Complex Test Goals:	CTG
Test Goals:	TG
Coverage Criteria:	CC
Coverage Criteria Satisfaction:	$\models$

Figure 2.15: Names and symbols for formal definitions of coverage criteria.

SM denotes the set of all UML state machines [Obj07, page 519]. Step patterns SP represent the abstract behavior of parts of test cases in a state machine. A step pattern $sp \in SP$ is a 4-tuple (c, E, cva, T) of a state configuration c, a set of events E, a guard condition value assignment cva, and a set of transitions T. It describes a behavior that includes visiting the state configuration c (set of active states) and afterwards triggering one event $e \in E$, satisfying a certain condition value assignment cva for guard conditions, and traversing one of the transitions $t \in T$. Relations of step coverage $SPCov \subseteq SP \times SP$ are used to describe steps that match step patterns. If a step pattern sp_c representing one part of a test case on the level of the state machine meets the description of a step pattern sp, sp_c is said to cover sp: $(sp_c, sp) \in SPCov$. Note that step patterns are used at different abstraction levels to describe the actual behavior as well as the abstract description that is covered by the actual behavior. The wild-card "?" is used if elements of the step pattern do not exist or are unimportant. We clarify these definitions with an example. Figure 2.16 shows a state machine with two states $S1$ and $S2$, one transition t that is triggered by the event e. Consider a test case that visits the state $S1$ and traverses the transition t by calling the event e. A step pattern that describes a this step is $(\{S1\}, \{e\}, ?, \{t\})$. This step pattern covers, e.g., the step patterns $(\{S1\}, ?, ?, ?)$, $(?, \{e\}, ?, ?)$, and $(?, ?, ?, \{t\})$. Attaching "$\neg$" to an element of the tuple describes this element's exclusion from the step pattern. For instance, $(\{S1\}, \{x\}, ?, \{\neg t\})$ describes that the state machine in state $S1$ reacts to the event x for all guard value assignments by doing anything but traversing the transition t. Step patterns do not necessarily describe the traversal of transitions but the general state machine behavior, which also includes remaining in a state.

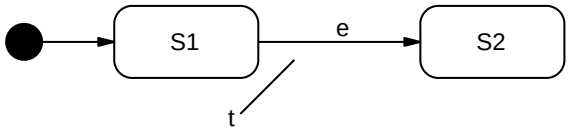


Figure 2.16: State machine example to clarify definitions of coverage.

Trace patterns $tp \in TP$ are sequences (ordered multisets) of step patterns SP. A step pattern sp annotated with a "$*$" describes an arbitrary number of steps that all cover sp. The length of tp is denoted $tp.length$. The n-th step pattern of tp is $tp[n-1]$. Corresponding to step coverage, trace coverage is used to describe traces that match a certain trace pattern: $TPCov \subseteq TP \times TP$. Like for step patterns, trace patterns can describe the actual behavior as well as the described pattern. A trace pattern tp_c describing the abstract behavior of a test case $tc \in TS$ covers a trace pattern tp $((tp_c, tp) \in TPCov)$

iff there is a subsequence tp_{csub} of tp_c so that for each n ($0 \leq n < tp.length$): $(tp_{csub}[n], tp[n]) \in SPCov$. We clarify the notion of trace pattern coverage with the example in Figure 2.16. Imagine a test case that includes a state change from $S1$ to $S2$ using the transition t, which is triggered by the event e. There is no guard condition. A trace pattern that describes this behavior is $((\{S1\}, \{e\}, ?, \{t\}), (\{S2\}, ?, ?, ?))$. In contrast to the previously presented step pattern, the just presented trace pattern also includes the information that state $S2$ is active after traversing t. It covers, e.g., the trace patterns $((\{S1\}, ?, ?, ?))$, $((\{S2\}, ?, ?, ?))$, $((?, \{e\}, ?, ?))$, and $((?, ?, ?, \{t\}))$.

Each atomic test goal $atg \in ATG$ is defined as a set of trace patterns, i.e. $ATG \subseteq \mathscr{P}(TP)$. It is satisfied by a test case iff the test case's trace covers at least one of the test goal's trace patterns. Atomic test goals can be combined by boolean operators to more complex expressions, which we call complex test goals $ctg \in CTG$. The satisfaction of an atomic test goal corresponds to a true value in this boolean expression. A complex test goal ctg is satisfied if the boolean expression of atomic test goals is satisfied, i.e. a sufficient set of included atomic test goals is satisfied. Complex test goals are needed to describe coverage criteria like MC/DC [CM94], for which one test goal can require several atomic test goals to be satisfied, e.g. each by a separate test case. The set of all test goals TG denotes the union of atomic and complex test goals: $TG = ATG \cup CTG$.

Coverage criteria $cc \in CC$ are functions that return test goals for each state machine: $CC \subseteq \{cc \mid cc : sm \rightarrow TGS; sm \in SM; TGS \subseteq TG\}$. Coverage criteria satisfaction $\models$ is a relation between trace patterns describing test cases and test goals: $\models \subseteq \mathscr{P}(TP) \times \mathscr{P}(TG)$. A coverage criterion $cc \in CC$ on a state machine $sm \in SM$ is satisfied by a set of traces $trcs \in \mathscr{P}(TP)$ iff each test goal $tg \in cc(sm)$ is satisfied. This means that at least one trace pattern tp_g of each atomic test goal $tg \in ATG$ is covered by at least one trace of $trcs$: $trcs \models cc(sm)$ iff $\forall_{tg \in cc(sm)} \exists_{tp_t \in trcs, tp_g \in tg} : (tp_t, tp_g) \in TPCov$. For complex coverage criteria like MC/DC, it is adequate to satisfy only a sufficient set of included atomic test goals. There might be infeasible test goals for each coverage criterion. Common reasons are unreachable states or restrictions for guard value assignments. Thus, a coverage criterion is considered satisfied if all feasible test goals are satisfied. In the following, we present formal definitions of coverage criteria.

2.4.3 Formal Definitions of Coverage Criteria

In this section, we apply the afore presented formalization to define the state-machine-related coverage criteria that are presented in Section 2.1.5: We define the transition-based coverage criteria All-States, All-Configurations,

All-Transitions, All-Transition-Pairs, and All-Paths. After that, we define the control-flow-based coverage criteria Decision Coverage, Condition Coverage, Decision/Condition Coverage, MC/DC, and Multiple Condition Coverage. Finally, we present definitions for the data-flow-based coverage criteria All-Defs, All-Uses, and All-Def-Use-Paths. Each coverage criterion is a function that returns a set of test goals for a given state machine. While every concrete coverage criteria definition is not specific to the concrete state machine, the returned test goals are. The test goals can be used to create test cases and to measure the degree of coverage criterion satisfaction. Boundary-based coverage criteria are applied to value partitions. A value partition is defined as a set of constraints that specify a set of objects that are associated with the partition. All already presented definitions are clear and, thus, we desist from presenting formal definitions of boundary-based coverage criteria.

All-States.

The coverage criterion All-States is satisfied on a state machine if each state is visited [UL06, page 117]. Figure 2.17 shows the pseudocode for the definition of All-States. It shows that for each vertex v, an atomic test goal is created that contains a trace that is visiting v – triggered events, value assignments, and traversed transitions are irrelevant.

```
P(TG) All-States(SM sm) {
  testgoals = {};
  for each vertex v in sm {
    testgoals.add(new ATG( (({v}, ?, ?, ?)) ) ); }
  return testgoals; }
```

Figure 2.17: Definition of All-States.

All-Configurations.

All-Configurations is satisfied iff all configurations (sets of concurrently active states) are visited. Figure 2.18 shows the corresponding pseudocode.

```
P(TG) All-Configurations(SM sm) {
  testgoals = {};
  for each configuration c in sm {
    testgoals.add(new ATG( ((c, ?, ?, ?)) ) ); }
  return testgoals; }
```

Figure 2.18: Definition of All-Configurations.

All-Transitions.

The coverage criterion All-Transitions [UL06, page 117] is satisfied on a state machine if all transitions of the state machine are traversed. Figure 2.19 shows the pseudocode for the definition of All-Transitions: For each transition t, an atomic test goal is created with a trace pattern that just traverses t. In order to guarantee the subsumption of All-States, All-Transitions includes all test goals returned by All-States(sm).

```
P(TG) All-Transitions(SM sm) {
  testgoals = All-States(sm);
  for each transition t in sm {
    testgoals.add(new ATG( ((?, ?, ?, {t})) ) ); }
  return testgoals; }
```

Figure 2.19: Definition of All-Transitions.

All-Transition-Pairs.

The coverage criterion All-Transition-Pairs [UL06, page 118] is satisfied on a state machine if all pairs of adjacent transitions are traversed at least once [UL06, page 118]. Figure 2.20 shows the pseudocode for the definition of All-Transition-Pairs: For each pair of adjacent transitions $t1$ and $t2$, an atomic test goal is created that contains a trace pattern with two step patterns that traverse $t1$ and $t2$, respectively. In order to guarantee the subsumption of All-Transitions, All-Transition-Pairs initially calls All-Transitions(sm). For defining All-n-Transitions, the definition of All-Transition-Pairs is extended from pairs of transitions to sequences of length n.

```
P(TG) All-Transition-Pairs(SM sm) {
  testgoals = All-Transitions(sm);
  for each transition t1 in sm {
    for each outgoing transition t2 of the target state of t1 {
      testgoals.add(new ATG( ((?, ?, ?, {t1}), (?, ?, ?, {t2})) )); }}
  return testgoals; }
```

Figure 2.20: Definition of All-Transition-Pairs.

All-Paths.

All-Paths [UL06, page 119] is satisfied if all possible paths of a state machine are traversed at least once. Since any state machine with loops (i.e. transi-

tion sequences with the same start and end configuration) can result in an infinite set of paths, All-Paths is considered an infeasible coverage criterion. Figure 2.21 shows the formal definition of All-Paths. To guarantee the satisfaction of the initial configuration, All-Configurations is called, first. Since All-Paths is infeasible, the pseudocode is not guaranteed to terminate.

```
P(TG) All-Paths(SM sm) {
  testgoals = All-Configurations(sm);
  Configuration c = set of all initial nodes of the regions of sm;
  List transitions = ();
  testgoals.addAll(getPaths(c, transitions));
  return testgoals;
}

P(TG) getPaths(Configuration c, List transitions) {
  testgoals = {};
  TP tracepattern = new TP();
  for i=0 to transitions.length-1 {
    tracepattern.add((?, ?, ?, {transitions.get(i)})); }
  if(transitions.length > 0) {
    testgoals.add(new ATG(tracepattern)); }
  for each vertex v of c {
    for each outgoing transition t of v {
      transitions.addToEnd(t);
      c.remove(v);
      c.add(t.target);
      testgoals.addAll(getPaths(c, transitions));
      transitions.removeFromEnd(t);
      c.add(v);
      c.remove(t.target);
}}}
```

Figure 2.21: Definition of All-Paths.

The pseudocode in Figure 2.21 starts with the initial configuration. The function *getPaths(c,transitions)* uses this configuration and the path that was already traversed to generate atomic test goals. New paths are searched for in a depth-first manner by adding all outgoing transitions of the states that are included in the current configuration. Since this algorithm basically describes all transition execution sequences, it can be arbitrarily complex, e.g., by including outgoing transitions or initial states of composite states.

Decision Coverage.

Decision Coverage [UL06, page 112] is a simple control-flow-based coverage criterion. It is satisfied iff each guard condition is evaluated to true and false,

respectively. Some test goals might be infeasible. For instance, tautologies cannot be negated. We define that default guard conditions of unguarded transitions are satisfied if the transitions are traversed. Thus, Decision Coverage subsumes All-Transitions. Figure 2.22 shows the pseudocode definition of Decision Coverage.

```
P(TG) DecisionCoverage(SM sm) {
  testgoals = All-Transitions(sm);
  for each transition t in sm {
    Expression positive = "false";
    Expression negative = "false";
    for each value assignment va for the guard of t {
      cva = va expressed as a logical formula;
      if the guard of t is satisfied for va {
        positive = positive + " ∨ cva";
      } else {
        negative = negative + " ∨ cva";
      }}
    testgoals.add(new ATG( (({t.source}, t.events, positive, ?)) ));
    testgoals.add(new ATG( (({t.source}, t.events, negative, ?)) ));}
  return testgoals; }
```

Figure 2.22: Definition of Decision Coverage.

The essence of this definition is to evaluate the results of all value assignments and collect them in positive and negative expressions by connecting them by disjunction. As a result, using one of these value assignments at the current guard condition results in satisfying the corresponding positive or negative test goal.

Condition Coverage.

```
P(TG) ConditionCoverage(SM sm) {
  testgoals = {};
  for each transition t in sm {
    for each atomic condition ac of the guard of t {
      testgoals.add(new ATG( (({t.source}, t.events, ac, ?)) ));
      testgoals.add(new ATG( (({t.source}, t.events, ¬ ac, ?)) ));
    }
  }
  return testgoals; }
```

Figure 2.23: Definition of Condition Coverage.

Condition Coverage [UL06, page 112] is satisfied if for each guard condition, all included atomic boolean expressions are evaluated to true and false,

respectively. In contrast to Decision Coverage, this does not imply the satisfaction or violation of the guard. Figure 2.23 shows the pseudocode for the definition of Condition Coverage.

Decision/Condition Coverage.

The coverage criterion Decision/Condition Coverage is the union of Decision Coverage and Condition Coverage. Correspondingly, the algorithm to define this coverage criterion consists of just reusing the definitions of Decision Coverage and Condition Coverage (see Figure 2.24).

```
P(TG) DecisionConditionCoverage(SM sm) {
  testgoals = DecisionCoverage(sm);
  testgoals.addAll(ConditionCoverage(sm));
  return testgoals; }
```

Figure 2.24: Definition of Decision/Condition Coverage.

Modified Condition/Decision Coverage.

The purpose of Modified Condition/Decision Coverage (MC/DC) is to show the isolated impact of each atomic expression on the guard evaluation results [CM94] [UL06, page 114]. The isolated impact is shown by changing only one variable and fixing all others while the evaluation results changes. As stated above, there are different kinds of MC/DC: unique-cause, masking, and the mix of both.

Figure 2.25 shows a possible pseudocode for unique-cause MC/DC. It shows that each atomic condition ac of each transition guard is considered: For each truth table row of the whole guard condition (excluding the currently selected atomic condition ac), the value of ac is set once to true and once to false. If the guard evaluations of the resulting two value assignment conditions differ, ac is said to have an isolated impact on the guard. The atomic test goals $atg1$ and $atg2$ contain the corresponding trace patterns. They are connected by conjunction and add to the complex test goal ctg by disjunction. Such a complex test goal is satisfied if the test suite traces cover the traces patterns of at least one conjuncted pair of atomic test goals. The elements of the pair differ only in the value of ac. There are probably several ways to optimize this algorithm. The intention of displaying the algorithm was to show that the proposed framework can also be used to describe such complex coverage criteria.

```
P(TG) UniqueCauseMCDC(SM sm) {
  testgoals = All-Transitions(sm);
  for each transition t in sm {
    for each atomic condition ac in the guard of t {
      CTG ctg = new CTG();
      testgoals.add(ctg);
      for each value assignment condition cva for the guard of t {
        remove the value asignment for ac from cva;
        value assignment condition cva1 = cva and ac = true;
        value assignment condition cva2 = cva and ac = false;
        if the guard value for cva1 differs from that for cva2 {
          atg1 = new ATG( (({t.source}, t.events, cva1, ?)) );
          atg2 = new ATG( (({t.source}, t.events, cva2, ?)) );
          ctg = ctg ∨ (atg1 ∧ atg2);
  } } } }
  return testgoals;
}
```

Figure 2.25: Definition of unique-cause MC/DC.

Multiple Condition Coverage.

Multiple Condition Coverage (MCC) [UL06, page 114] is satisfied on a state machine if each value assignment of the truth tables of all guards is tested. Figure 2.26 shows the pseudocode for the definition of MCC. It shows that for each guard condition value assignment *cva* representing one row of a guard's truth table, a test goal is created with a trace pattern that references t's source state $t.source$, t's set of events $t.events$, and *cva*. Since the guard of t may not be satisfied by *cva*, t may not be traversed and, thus, no transition is referenced in the test goal.

```
P(TG) MultipleConditionCoverage(SM sm) {
  testgoals = All-Transitions(sm);
  for each transition t in sm {
    for each condition value assignment cva for the guard of t {
      testgoals.add(new ATG( (({t.source}, t.events, cva, ?)) )); }}
  return testgoals; }
```

Figure 2.26: Definition of Multiple Condition Coverage.

All-Defs.

All-Defs [UL06, page 115] is a data-flow-based coverage criterion. It is satisfied iff for each variable *var* and each defining transition t_d of *var*, at least one pair of t_d and any using transition t_u of *var* is tested. Figure 2.27

depicts the pseudocode for All-Defs. For each (t_d, t_u), an atomic test goal is created that demands to traverse the containing transitions t_d and t_u without traversing a transition t_rd in between that redefines var. Since traversing one use for each definition is enough to satisfy All-Defs, we add all atomic test goals of one variable to one complex test goal that combines them by disjunction. Note that a transition that contains a use and a definition of a variable var results in infeasible test goals.

```
P(TG) All-Defs(SM sm) {
  testgoals = {};
  for each variable var in sm {
    for each transition t_d that references a definition of var {
      t_rd = set of all transitions except t_d that define var;
      CTG ctg = new CTG();
      testgoal.add(ctg);
      for each transition t_u that references a use of var {
        ATG atg = new ATG( ((?, ?, ?, {t_d}),
          (?, ?, ?, {¬t_rd})*, (?, ?, ?, {t_u})) ));
        ctg = ctg ∨ atg;
  return testgoals; }
```

Figure 2.27: Definition of All-Defs.

All-Uses.

All-Uses [UL06, page 115] is satisfied on a state machine iff all def-use-pairs (t_d, t_u) of all variables var are tested. Figure 2.28 depicts the pseudocode for All-Uses. For each (t_d, t_u), a test goal is created that demands to traverse the containing transitions t_d and t_u without traversing a transition t_rd in between that redefines var.

```
P(TG) All-Uses(SM sm) {
  testgoals = {};
  for each variable var in sm {
    for each transition t_d that references a definition of var {
      t_rd = set of all transitions except t_d that define var;
      for each transition t_u that references a use of var {
        testgoals.add(new ATG( ((?, ?, ?, {t_d}),
          (?, ?, ?, {¬t_rd})*, (?, ?, ?, {t_u})) )); }}}
  return testgoals; }
```

Figure 2.28: Definition of All-Uses.

All-Def-Use-Paths.

The satisfaction of All-Def-Use-Paths requires to test all paths between all def-use-pairs. Figure 2.29 shows the pseudocode for this coverage criterion. In the main function *All-Def-Use-Paths(sm)*, all def-use-pairs of all variables *var* are identified. The function *getAllTestGoalPathsBetween(var, t_d, t_u)* is called to identify all possible paths between *t_d* and *t_u*. For that, all configurations that may result from traversing *t_d* are used one by one as starting point to identify paths to *t_u*. The function *findAllPaths(var, c, transitions, t_u)* is called to identify these paths. A test goal is created if the generated transition sequence ends with *t_u*. The search is stopped if the current transition redefines the variable *var*. Since All-Def-Use-Paths is considered infeasible, this algorithm is not guaranteed to terminate.

```
P(TG) All-Def-Use-Paths(SM sm) {
  testgoals = {};
  for each variable var in sm {
    for each transition t_d that references a definition of var {
      for each transition t_u that references a use of var {
        testgoals.add(getAllTestGoalPathsBetween(sm, var, t_d, t_u));
  }}}
  return testgoals; }

P(TG) getAllTestGoalPathsBetween(SM sm, Variable var, Transition t_d,
                                 Transition t_u) {
  testgoals = {};
  Vertex v = target state of t_d;
  Set confs = set of all configurations of sm that include v;
  for each configuration c of confs { // use all configurations
    List transitions = (t_d); // the first transition is t_d
    testgoals.addAll(findAllPaths(var, c, transitions, t_u)); }
  return testgoals; }

P(TG) findAllPaths(Variable var, Configuration c,
                   List transitions, Transition t_u) {
  testgoals = {};
  TP tracepattern = new TP();
  if(t_u is equal to transitions.length-1) { // last transition is t_u?
    for i=0 to transitions.length-1 {
      tracepattern.add((?, ?, ?, {transitions.get(i)}));
    }
    testgoals.add(new ATG(tracepattern)); // create test goal
  }
  for each vertex v in c {
    for each outgoing transition t of v {
      if(t does not redefine var) { // redefine var? - stop!
        transitions.addToEnd(t);
        c.remove(v);
        c.add(t.target);
        testgoals.addAll(findAllPaths(c, transitions, t_u));
        transitions.removeFromEnd(t);
        c.add(v);
        c.remove(t.target);
}}}}
```

Figure 2.29: Definition of All-Def-Use-Paths.

Chapter 3

Automatic Model-Based Test Generation

In this chapter, we present a new model-based test generation approach from UML state machines and class diagrams annotated with OCL expressions. Coverage criteria are used to steer the test generation process. The presented test generation approach allows to combine data-flow-based, control-flow-based, or transition-based coverage criteria with boundary-based coverage criteria. The combination of these kinds of coverage criteria is the first contribution of this thesis. We complement it with approaches to combine and transform test models in the following chapters.

This chapter is structured as follows. We present a short motivation for the combination of structural (data-flow-based, control-flow-based, and transition-based) coverage criteria with boundary-based coverage criteria in Section 3.1. This comprises preliminaries like the definition of input partitions, output partitions, and their relations to abstract test cases like, e.g., transition paths in the state machine. After that, we will introduce several example test models in Section 3.2. These examples are used several times as case studies for automatic test generation. In Section 3.3, we present the test goal management, which is focused on managing test goals during automatic test generation. We present the corresponding concrete test case generation approach for single test goals in Section 3.4. In Section 3.5, we present our developed prototype ParTeG (Partition Test Generator) [Weib] and the corresponding results of applying ParTeG to the mentioned test models. Subsequently, we describe the related work in Section 3.6. This chapter ends with conclusion, discussion, and a survey of future work in Section 3.7.

3.1 Motivation

This section contains a motivation for our new test generation approach: We shortly describe the benefits of combining different kinds of coverage criteria.

For the description of reactive systems, graph-based models like UML state machines are often applied. As described above, the quality or the adequacy of test cases is often measured with coverage criteria that are focused on data flow, control flow, or transition sequences [UL06, page 110]. There are many approaches, like model checking, genetic algorithms, or graph search algorithms, that are used to satisfy such coverage criteria for graph-based models (see Sections 2.1.5, 2.3, and 3.6). One important problem of the existing test generation approaches is the selection of concrete input values: The approaches often use random input values and determine whether the specified model elements are covered. There are approaches that allow the user to define input values manually [BSV08]. Such approaches, however, do not comprise information about the quality of the input values. There is need to apply means of representative input value selection.

An alternative approach to user-defined or manually chosen input values is partition testing, which divides the input space into input partitions. Each input partition is a set of constraints for objects that are assumed to trigger the same system behavior. This assumption is called the *uniformity hypothesis* [BGM91]. As a consequence of this assumption, only a few representatives of these equivalence classes are chosen instead of all values. Experience shows that many faults occur near the boundaries of equivalence classes [WC80, CHR82]. There are corresponding boundary-based coverage criteria for the selection of representatives from input partitions [KLPU04]. These coverage criteria can be combined with other approaches from partition value analysis like additionally selecting random elements from the inside of partitions [Bei90]. The issue of input partitions is that they are only applied to the input space of non-reactive systems [BJK05, page 301], i.e. systems that behave like a function: Only the input space is considered and information about state changes or further behavior is neglected.

Both approaches of creating abstract test cases and selecting concrete input values are important aspects of automatic model-based test generation. However, they are often considered in isolation because each of them is only applied to reactive or non-reactive systems, respectively. The core idea of our test generation approach is to combine both approaches and make them applicable to reactive systems. Our approach basically consists of three steps: (1) defining output partitions as visible behavior of a system under test; (2) using abstract behavior information to generate abstract test cases by backward abstract interpretation; and (3) deriving test case-specific input

partitions from output partitions at the same time. The results are abstract test cases that comprise information about input values. This information is used to form input partitions and, subsequently, to select proper input values. The abstract test cases satisfy the chosen structural, e.g., control-flow-based, coverage criterion. For each abstract test case, proper input values can be selected to satisfy a chosen boundary-based coverage criterion. The result of this approach is a test suite that satisfies a combination of both kinds of coverage criteria.

In Section 3.1.1, we introduce input and output partitions as equivalence classes and describe their relations. After these considerations, we show the mutual dependency of input partitions and abstract test cases in Section 3.1.2. Section 3.1.3 contains a description of how to derive input partitions from output partitions. In Section 3.1.4, we describe boundary value analysis as a means to select conrete values from input partitions.

3.1.1 Value Partitions

In this section, we introduce value partitions. Value partitions are constraints that describe a set of objects. We distinguish input and output partitions as well as linear-ordered and unordered partitions.

Input Partitions and Output Partitions.

Input partitions are used to group input parameters of the SUT, e.g. user input data. Corresponding to the uniformity hypothesis [BGM91], all representatives of an input partition are assumed to trigger the same behavior. Thus, only a few representatives are selected for test execution. Otherwise, the definition of the input partitions would be needless. Besides that, selecting all representatives might be impossible. The selection of "good" representatives is an important problem. It depends on two things: the selection of representatives from partitions, and the definition and selection of the partitions themselves. Experience shows that faults often occur near partition boundaries [WC80, CHR82]. In many cases, however, the partitions are defined manually by the user and based on the user's experience and knowledge. A vital threat to this approach is that the input partitions might be invalid. As a consequence, the selected representatives are not boundary values for the real behavior, and the selected elements could only accidentally trigger the expected behavior. In both cases, the selected values are more or less worthless or at least not as meaningful as intended.

Output partitions are used to group observable output data of the system's behavior. Such data can be used to recognize the behavior of the SUT and

to compare it to the expected behavior. Thus, output partitions are used to group equal values for system behavior. They are important for the test oracle, i.e. to decide whether an executed system behavior corresponds to the test specification. They are not used for test input data selection.

In testing, input partitions and output partitions are used for value selection and test oracle, respectively. There is, however, a connection between them as elements of input partitions are fed into the SUT and, afterwards, result in output values of output partitions (see Figure 3.1). If input partitions are defined manually [Con, PE05], this relation is not taken into account. The absolute definition of input partitions assumes that the defined partitions are valid for virtually all possible abstract test cases. As we will show in Section 3.1.2, they are not.

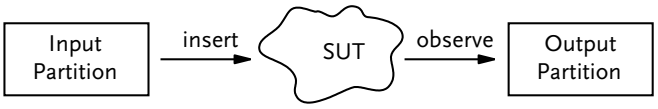


Figure 3.1: Relation between input partitions and output partitions.

Value Type Order.

Each partition is influenced by the type of the contained values. We distinguish linear-ordered types and unordered types (see page 19).

Instances of linear-ordered types are sorted. There is a distance function *dist* for such data types. Boundaries of a partition are defined as all elements of the partition for which an element of another partition is within a certain distance value d_{max} [KLPU04]. Figure 3.2 shows a partition for an

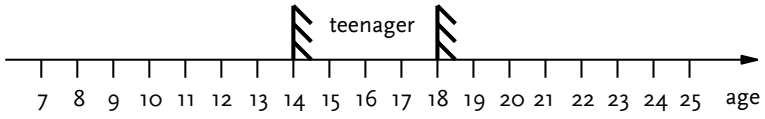


Figure 3.2: Partition for teenager age.

attribute *age* of integer type. Solid lines depict the boundaries of partitions. The hachure shows the partition for which the boundary is inclusive. In this example, the valid age for a teenager is depicted: A person is a teenager if its age is between 14 (inclusive) and 18 (exclusive). The value d_{max} for boundary distance is set to 1, i.e. all elements b are boundaries for which there is another element e from another partition with $dist(b, e) \leq 1$. Correspondingly, the boundary values for the teenager partition are the values 14 and 17.

Instances of unordered types are not ordered. There is no notion of distance between such elements and, thus, there cannot be a definition of partition boundaries based on distances. Each representative of such a partition is statistically as good as any other. A well-known example of unordered types are enumerations. Figure 3.3 exemplary shows two partitions *P1* and *P2* of type enumeration. In this example, the enumeration values are wares for a coffee vending machine. The partitions are used to distinguish hot (*P1*) and cold (*P2*) beverages. There is no obvious way to define that one of the values of one partition is closer to the other partition than another value.

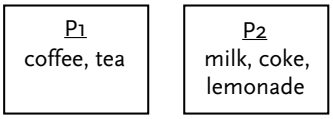


Figure 3.3: Two partitions of an enumeration.

For many applications, there are more than one parameters that influence the result of the output (cf. sorting machine, freight elevator, or triangle classificator in Section 3.2). The linear-ordered partitions of such parameters are combined to create the resulting partitions. They influence each other. Figure 3.4 shows the partitions of the two input parameters of the sorting machine example (Section 3.2.1). The creation of one test case per combination of partitions from each variable would lead to an unnecessarily large number of test cases. The reason is that the partitions for *object.width* are only meaningful for the lower partition of *object.height*. Thus, it is sufficient to consider only the four shown partitions for test generation.

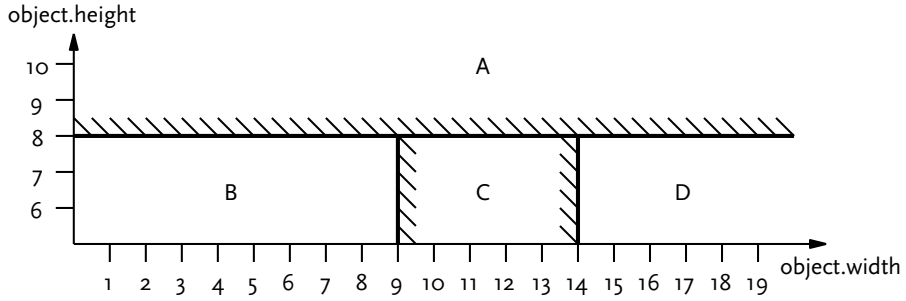


Figure 3.4: Partitions with two dimensions for the sorting machine.

Nevertheless, it is still possible to define value partitions for each of the input parameters without taking the other parameters into account. The only disadvantage is that the test suite contains unnecessary test cases that, e.g., check partition boundaries for *object.width* with values of *object.height* above 8. In such cases, the selection of partitions is influenced by input parameter dependencies. There are more complex dependency relations between input parameters, e.g. when guard conditions depend on several input parameters: Figure 3.5 shows a more complex partition for valid triangles of the triangle classification problem. This partition is influenced by three input parameters. It is impossible to select meaningful boundary values without taking all input parameters into account. All values depend on each other, e.g. in a guard condition $[x < y + z]$. For the sake of simplicity, we set $z = 5$. For the values of x and y, we assume that 10 is the upper boundary for both values. This imposes no general restriction on the size of the triangle. The number 10 is chosen arbitrarily. If we assumed no upper boundary, the partition size would be infinite (which is no problem except we do not like depicting it).

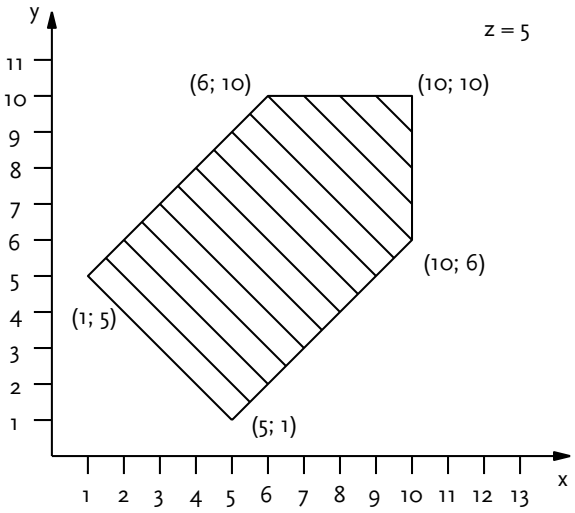


Figure 3.5: Partition for input parameters of valid triangles with $z = 5$.

3.1.2 Value Partitions and Abstract Test Cases

This section describes the mutual dependency of value partitions and abstract test cases. Until now, boundary value analysis has often been considered for non-reactive systems [BJK05, page 301] – input partitions are considered independent of the abstract test cases. In reactive systems, however, input parameters can depend on other input parameters or input events. In test

cases with loops, they may depend on the same input parameter on a transition that is traversed twice. The further application of input parameters can also differ. We present some examples to clarify this point. Figure 3.6 shows a state machine in which the variable x is set to the value of the input parameter a of event *ev1*. Depending on the subsequent event (*ev2* or *ev3*),

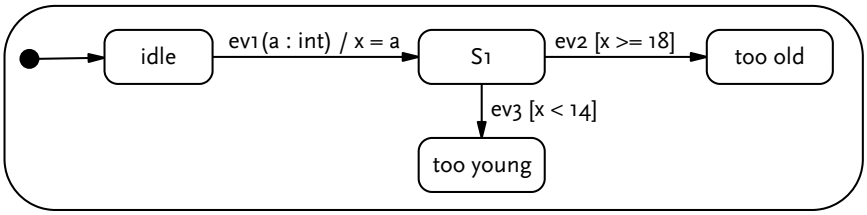


Figure 3.6: State machine to clarify the mutual dependency of input partitions and abstract test cases.

there are different input partitions resulting for a: As shown in Table 3.1, the boundary between the partitions is at 14 or at 18 depending on the received events. The selection of values around 18 is needless if *ev3* is triggered. The same holds for 14 and event *ev2*. In cases where system attributes are incremented depending on the list of triggered events, the partition boundaries may not only be needless but also wrong. Thus, the sole definition of input partitions can be worthless without the knowledge about additional influencing factors such as the sequence of triggered events. To determine input partitions, all information about abstract test cases (the sequence of events, input parameters, other actors' behaviors, etc.) must be included.

Event Sequence	Input Partitions
ev1, *ev2*	$(-\infty; 18)$, $[18; \infty)$
ev1, *ev3*	$(-\infty; 14)$, $[14; \infty)$

Table 3.1: Two partitionings for two event sequences.

3.1.3 Deriving Input Partitions From Output Partitions

As described above, input partitions describe sets of input parameters, and output partitions describe the observable behavior of the SUT. Output partitions are often known before test execution. They can be represented, e.g. as postconditions, system return values, satisfied guard conditions, or state invariants. In contrast, input partitions are often unknown in advance.

However, they are important to derive test input values. In this section, we sketch how to derive input partitions from output partitions: While creating abstract test cases, we transform output partitions into input partitions. The input partitions are used later to derive concrete input values.

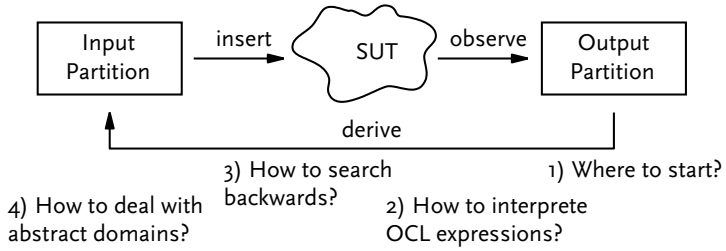


Figure 3.7: Issues of deriving input partitions from output partitions.

We call the used approach *abstract backward interpretation*. It means that we abstractly interpret the test model by traversing transitions backward using the weakest precondition calculus [Dij76]. Figure 3.7 depicts the process of deriving input partitions from output partitions with all described issues. This approach deals with several questions: In forward searching approaches, the state machine transitions are iterated forward starting at the initial node and the question is when to stop. In this approach, we iterate backward until we reach the initial node, and the question is where to start (1). For UML state machines with OCL expressions, the effect of traversing a transition is often described in postconditions, which are logic expressions without the notion of value assignment. The next question is how to interpret postconditions (2). The applied graph search algorithm has to find a transition sequence to the initial node using control flow and data flow information (3). While searching a path to the initial node, the algorithm has to keep track of the valid abstract value partitions. In forward searching approaches, the impact of visited input parameters on system attributes is known. In contrast, there are often attributes in backward searching for which the influence of input parameters is unknown. Furthermore, there may be conditions with several mutually dependent attributes. How are such information about abstract domains handled (4)?

Basically, our approach works as follows. A certain structural, e.g., control-flow-based, coverage criterion is applied to a state machine. The application of this coverage criterion results in a set of test goals (see Section 2.4). Each test goal references, e.g., a certain node, transition sequence, or a node with an event to call and a guard condition to satisfy or violate. From these referenced elements, the search algorithm starts to search a path backward to the initial configuration (1). During the search, all guard

conditions (interpreted as output partitions) on the way are stored and transformed if postconditions are traversed. For that, we present a corresponding interpretation of OCL expressions and corresponding transformation rules that are based on Dijkstra's weakest precondition approach (2). The search algorithm tries to reach transitions that influence the values of these guard conditions. It stops if the initial node is reached or contradictions occur (3). If input parameters are encountered, expressions from the stored conditions are applied to input parameters. Such conditions are interpreted as input partitions whose elements trigger the created test behavior (4).

3.1.4 Boundary Value Analysis

We introduced different kinds of partitions. Boundary value analysis is used to select concrete values from partitions. There are different approaches to deal with partitions and partition boundaries. We present the two main approaches that are focused on partitions or partition boundaries, respectively. We introduce both of them and compare them using the example of partitioning the age of people in Figure 3.2 on page 56. We also motivate why our approach is focused on partitions instead of partition boundaries.

Focus on Partition Boundaries.

The purpose of focusing on partition boundaries is to test the surroundings of them. The motivation for that is that domain faults are detected around domain boundaries [WC80, CHR82]. Beizer [Bei90] describes the concrete value selection for partitions as follows: For each partition boundary of a partition, a value should be selected on the boundary, inside the partition close to the boundary, and outside the partition close to the boundary. Furthermore, an arbitrary number of random elements from inside the partition should be selected. Figure 3.8 shows this exemplary for the partitioning of the age of teenagers: We assume integer value type. The black dots depict the correspondingly selected representatives. As seen in this figure, this approach is often used to distinguish *valid* and *invalid* partitions. All representatives from the valid partitions are expected to trigger the specified

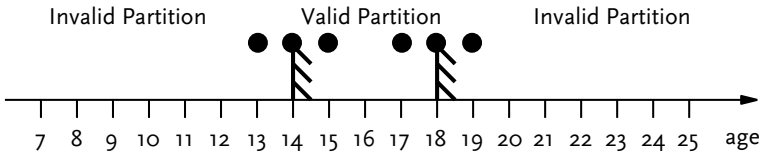


Figure 3.8: Selected values for partition boundaries.

behavior. The representatives of invalid partitions are assumed not to trigger the same behavior – otherwise, they would belong to the same partition.

Focus on Partitions.

If the focus is on partitions, the selected representatives are near the boundaries of the partitions like in the previous approach. For each partition, however, only values inside the partition are selected, and representatives of adjacent partitions are ignored. Thus, users of the second approach do not distinguish valid and invalid partitions. Instead, each partition is an adequate equivalence class on its own. The value on the boundary itself is only selected if the boundary itself is inside the partition. Figure 3.9 shows a possible selection of representatives for the partition B that describes the age of teenagers.

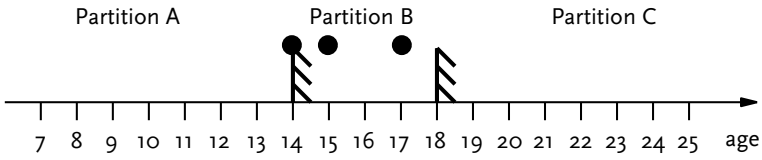


Figure 3.9: Selected values for partition boundaries.

Comparison.

The first approach is focused on boundaries: For each boundary, values on either side are selected. The second approach is similar, but restricted to values inside certain partitions. For the presented example, the following becomes clear: Applying the second approach to all partitions can result in the selection of the same set of representatives as the application of the first approach. Figure 3.10 illustrates this equality. As we see, both approaches are quite similar if we only consider the problem of selecting representatives from input partitions. In our test generation approach, the focus is on creating abstract test cases together with input partitions whose representatives

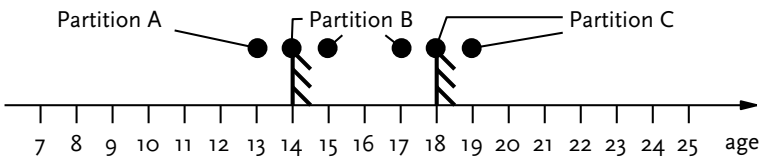


Figure 3.10: Applying the partition-oriented approach to all partitions.

enable these test cases. For the selection of concrete test input values for one test case, only a few value partitions are available for each input parameter. Thus, the selection of a certain, e.g., control-flow-based, coverage criterion determines the available input partitions from which input values are selected.

As Beizer describes, there might be a possibly infinite number of input partitions. Because of this large number of input partitions, the approach that is focused on partition boundaries can be infeasible. The main problem is the proper selection of partitions to test. In contrast, the approach that is focused on partitions selects input values only from some selected partitions. As a consequence, the partition-related approach probably does not all select boundary values and does also not incur the problem of too many input partitions. This selection problem is related to the question of which behavior (or which partitions) of the SUT should be tested at all. Our approach of creating input partitions only for abstract test cases that are generated for structural, e.g., control-flow-based, coverage criteria is a solution for this problem of proper input partition selection: Input partitions selection is determined by a selected structural coverage criterion.

We clarify this point with a small state machine shown in Figure 3.11. In this example, the satisfaction of the coverage criterion Multiple Condition Coverage requires to test the conditions $(x < 14)$, $(x >= 14)$ *and* $(x < 18)$, and $x >= 18$. This results in testing all three input partitions of the teenager partitioning shown in Figure 3.2 on page 56. If only abstract test cases to satisfy All-Transitions or Decision Coverage are created, the conditions $(x >= 14)$ *and* $(x < 18)$ and $(x < 14)$ *or* $(x >= 18)$ are tested, which results in selecting only representatives for two of the three partitions. This substantiates that there is a mutual dependency between input partitions and abstract test cases, i.e. between input partitions and the corresponding structural coverage criteria.

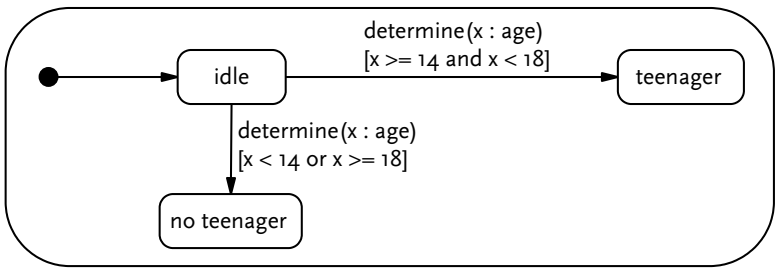


Figure 3.11: State machine describing the reaction to the input values of the teenager partitioning.

3.2 Example Test Models

In this section, we present five test models that are used for automatic test generation: The sorting machine in Section 3.2.1 and the freight elevator in Section 3.2.2 are artificial examples that, descpite their small size, partly describe behavior that is common in practice. The triangle classification example in Section 3.2.3 is a standard example for challenging test input data generation problems [UL06, page 214]. The test model for a track control in Section 3.2.4 is an academic test model that is part of the UnITeD project [PS07] of the University Erlangen. The train control test model presented in Section 3.2.5 is part of our industrial cooperation project with the the German supplier of railway signaling solutions Thales. All state machines describe the behavior of the class that is the context of the state machine and defines all structural elements.

All test models contain attributes of linear-ordered types. Boundary value analysis is an important task for such applications. The corresponding test cases have to contain values that check even small violations of the derived input parameter boundary values. Our prototype implementation ParTeG [Weib] has been used to generate test suites for all these test models. This also includes boundary value analysis. The prototype is used in further industrial applications. However, we consider the number of presented test models sufficient for this thesis. The selection of test models is by no means a restriction of ParTeG's application fields, e.g., to train-related applications. ParTeG is of special value for applications that depend on exact values and boundaries of linear-ordered value types.

3.2.1 Sorting Machine

The sorting machine test model describes the behavior of a machine that wraps up an object and subsequently sorts the resulting package depending on its size. The object is put into a plastic box filled with foam. The package size depends on the size of the object. A possible application field of such machines can be found in post offices.

Figure 3.12 shows a state machine and Figure 3.13 shows the corresponding class diagram of such a sorting machine. The diagrams define the rules for the packaging as follows: Due to wrapping up the objects, the original width of the object should be doubled by foam plus two extra size units for each side of a plastic box: $(width = (object.width + 2) * 2)$. The height is handled a bit different, and the corresponding equation is: $(height = (object.height * 2) + 2)$. Our sorting machine's task is to sort incoming items depending on the size after their wrapping so that they fit

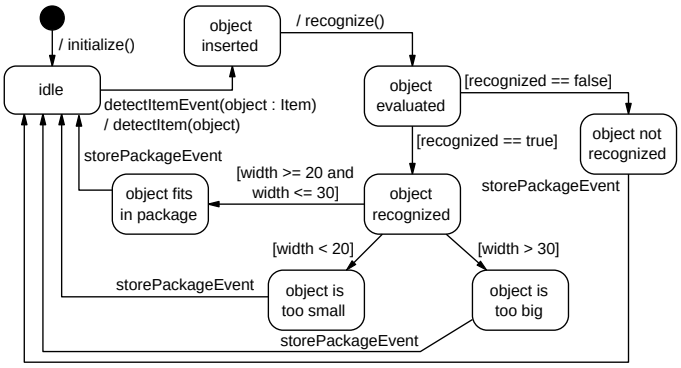


Figure 3.12: State machine of a sorting machine.

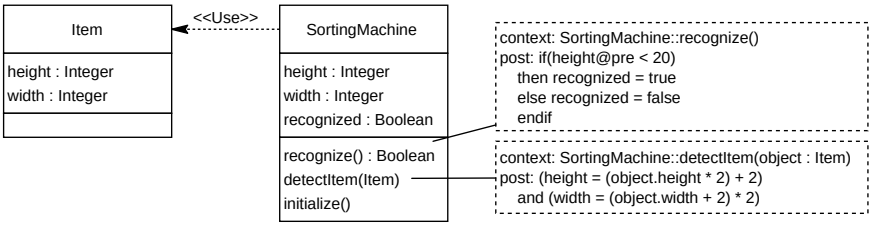


Figure 3.13: Class diagram of a sorting machine.

into given transport containers. The operations of the class *SortingMachine* contain postconditions and are referenced from the state machine. The postconditions of these operations influence the behavior of the state machine: The size of the package is described in the postcondition of the operation *detectItem(Item)*. The sorting is determined by the postcondition of *recognize()* and by the guard conditions of the outgoing transitions of the states *object recognized* and *object evaluated*: Objects that are too tall are sorted out using the guards of the outgoing transitions of state *object evaluated*. These objects are considered as *not recognized*. The remaining objects are sorted depending on their width. The values of *height* and *width* of the parameter of *detectItem(Item)* both influence the packaging. The only exceptions are tall objects, for which the width is unimportant.

3.2.2 Freight Elevator

The freight elevator behaves similar to a normal elevator. It can carry all weights up to a maximum weight, and the user can press a button to select the target floor of the elevator. Additional features are that the elevator can move faster if it is empty and that certain floors require a certain minimum user rank or authority.

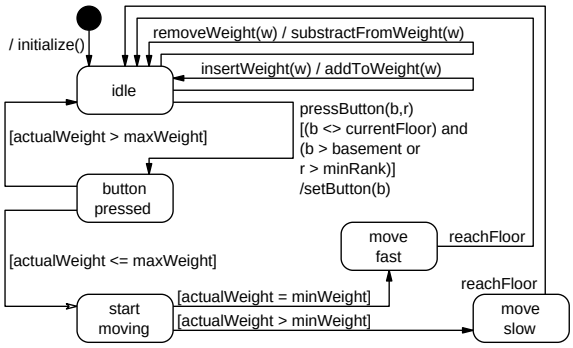


Figure 3.14: State machine of the freight elevator control.

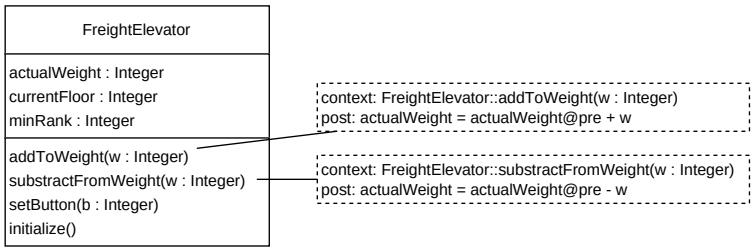
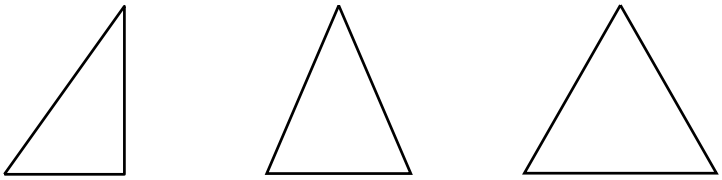


Figure 3.15: Class diagram of the freight elevator control.

Figure 3.14 shows the state machine that describes the behavior of a freight elevator. If the elevator is in state *idle*, the user of the elevator can insert and remove weight. The user can also press a button for the elevator's next target floor. As a reaction, the elevator checks the user's rank and whether the current weight exceeds the maximum weight. Subsequently, the elevator begins to move at an appropriate velocity until the target floor is reached. Like in the previous example, the postconditions of operations in the class diagram (see Figure 3.15) influence the behavior the of the state machine.

3.2.3 Triangle Classification

The triangle classification example is a standard example in testing literature [Mye79, UL06]. The task is to classify triangles whose three side lengths are described by three integer values. The triangle is *invalid* if one side length is less or equal to zero or if the sum of two side lengths is less or equal to the third side length. Valid triangles can be *scalene*, *isosceles*, or *equilateral* (see Figure 3.16). The issue of this example is the huge number of possible parameter value combinations.



(a) Scalene triangle. (b) Isosceles triangle. (c) Equilateral triangle.

Figure 3.16: Valid triangle classification results.

For this example, boundary value analysis is an good means to select proper input values. Since the conditions for the triangle classification include many mutual dependencies of the input parameters, there are many possible boundary values. This results in many correspondingly necessary test cases.

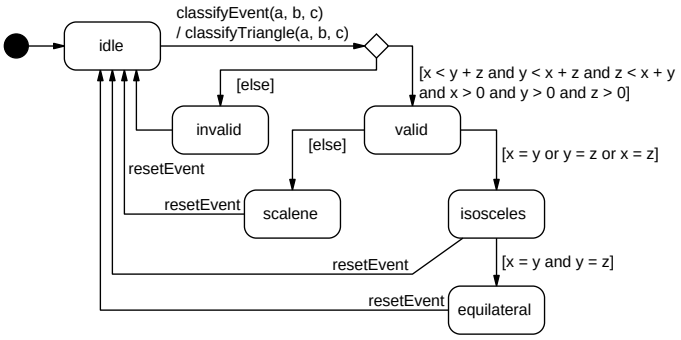


Figure 3.17: State machine for the triangle classification.

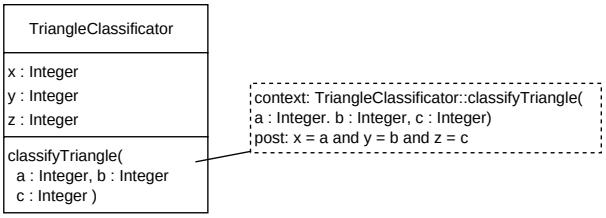


Figure 3.18: Class diagram for the triangle classification.

There are many ways to describe the triangle classification problem. For instance, Utting and Legeard [UL06, page 214] define OCL expressions to classify triangles. We use UML state machines. Figure 3.17 shows such a state machine. The corresponding class is shown in Figure 3.18. The postcondition of the operation *classifyTriangle(a, b, c)* represents a simple mapping from input parameters a, b, and c to system attributes x, y, and z.

3.2.4 Track Control

The track control example is part of the UnITeD project [PS07] of the University Erlangen. This example describes the control of a railway track and includes the steering of trains depending on their priority, their train number, and the track status. The original test model contains 15 states and 27 transitions. We present only a part of the test model to show the frequent use of linear-ordered types. Nevertheless, we use the complete model for test generation. Further information is available at the UnITeD homepage [PS07].

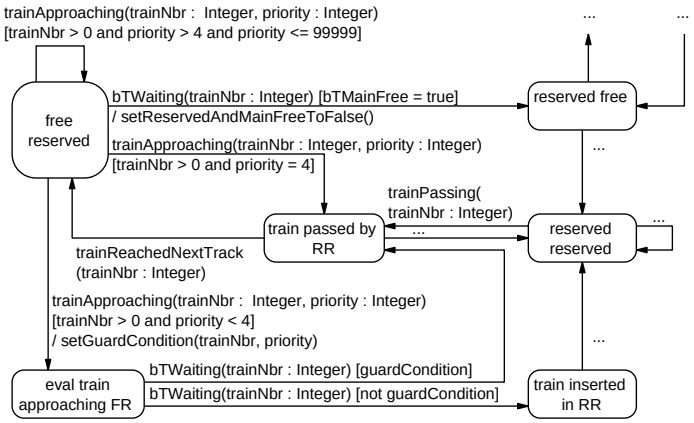


Figure 3.19: State machine of the track control.

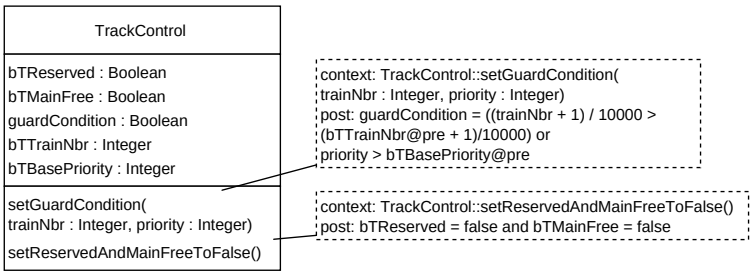


Figure 3.20: Class diagram of the track control.

Figure 3.19 depicts a part of the state machine and Figure 3.20 the corresponding class diagram. Note that there are several integer attributes for which boundary value analysis is a proper means to improve the test results.

3.2.5 Train Control

The test model for a train control is part of our industrial cooperation with the German supplier of railway signaling solutions Thales. The test model

describes the communication behavior between modules of a train and the railroad line. The task of these modules is to determine the train's position. Since safety is an important aspect for this company, the model describes several cases for emergency situations. It comprises about 35 states, 70 transitions, and composite states with a hierarchy of depth 4. Transitions are all triggered by call events. All generated tests are functional tests without time information. In order to protect the intellectual property of the company, we only provide an anonymised version of the test model. Due to the test model size and complexity, Figure 3.21 depicts only a part of the state machine. Since the model is anonymised, there is no sense in presenting the class diagram. Again, we use the complete model for test generation.

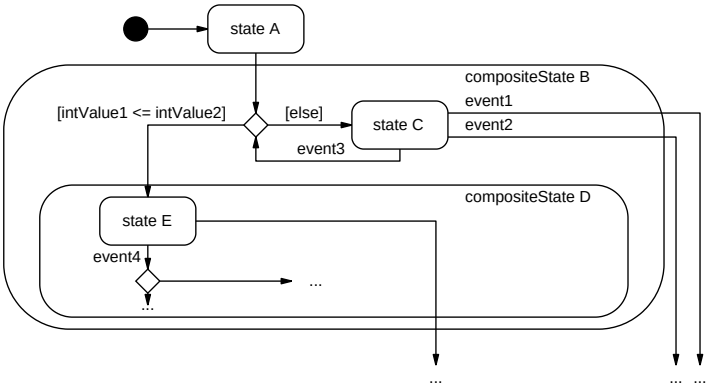


Figure 3.21: Anonymised part of the train control state machine.

3.3 Test Goal Management

Since our approach is focused on the satisfaction of coverage criteria, coverage criteria play an important role for the management of test generation. Here, we present the management of coverage criteria for automatic test generation. We generate test goals according to the approach presented in Section 2.4.3 using the multi-purpose language Java [Sun95].

First, we describe the general test goal management process during the automatic test generation in Section 3.3.1. Then, we present refinements that consist of transforming all expressions referenced by the test goals to disjunctive normal form in Section 3.3.2 and by extending or restricting test goals in Section 3.3.3. After showing the standard test suite generation management, we present some limitations to the presented approach in Section 3.3.4. The test case generation for the single test goals is presented in Section 3.4.

3.3.1 General Test Goal Management

The proposed test generation approach is focused on the satisfaction of coverage criteria on the test model level. The model-specific representation of a coverage criterion is a set of test goals. Thus, the management of test suite generation is based on test goals. Figure 3.22 depicts all tasks that are comprised in test goal management. The single steps are described in the following.

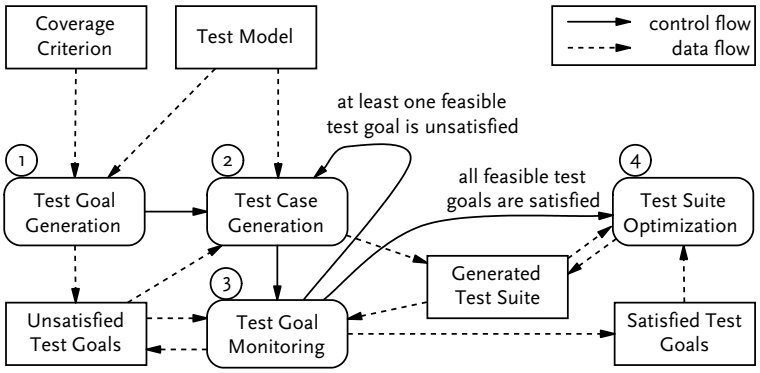


Figure 3.22: Test goal management for test suite generation.

First, the initial set of unsatisfied test goals has to be generated. This is done by applying the selected coverage criterion to the used test model. Corresponding algorithms for coverage criteria definitions that return a set of test goals are described in Section 2.4.3.

Second, the test goals of the resulting set of unsatisfied test goals are used one by one for test case generation. The process of using one test goal to generate a test case is described in Section 3.4.2.

Third, after each test case generation, the set of satisfied test goals is determined: Each generated test case was intended to satisfy a certain test goal. We say that this test goal is satisfied *intentionally* by the test case. There are typically more test goals that are *accidentally* satisfied by the test case, e.g., by visiting certain elements on the path from the initial configuration to the elements referenced by the intentionally satisfied test goal. For these accidentally satisfied test goals, no further test case has to be created. This process of constant test goal satisfaction checking is also known as *monitoring* [FW08a].

Finally, if there are no unsatisfied feasible test goals left, the created test suite is optimized: A test case is *redundant* if it only satisfies test goals that are also satisfied by other test cases. All test cases are checked one by one – if they are redundant, then they are removed. Afterwards, each remaining test

case satisfies at least one test goal that is not satisfied by another test case, and all test goals are still satisfied. Thus, the resulting test suite contains no redundant test cases, anymore.

Depending on the sequence of test case removals, there are several optimal test suites possible. Furthermore, there are other possible optimization steps, e.g., that are focused on the number of test cases and their respective length [FS07]. In most cases, however, these aspects are only useful for certain application areas and are not further considered here.

3.3.2 Expressions in Disjunctive Normal Form

Test goals, e.g., for control-flow-based coverage criteria, contain logical expressions. All these expressions are transformed into disjunctive normal form (DNF). Using DNF instead of the original form has some advantages. For instance, the evaluation of complex conditions, e.g., consisting of several *if-then-else* constructs, might be cumbersome, whereas the satisfaction of a guard condition in DNF is evaluated by checking whether one of the DNF's conjunctions holds. As a consequence, our search algorithm, which has to evaluate such conditions, is less complex because a) the conjunction to evaluate is usually shorter than the whole expression, and b) all interdependencies between the expression subparts are obvious. This approach has been used already several times [DF93] [AO08, page 138].

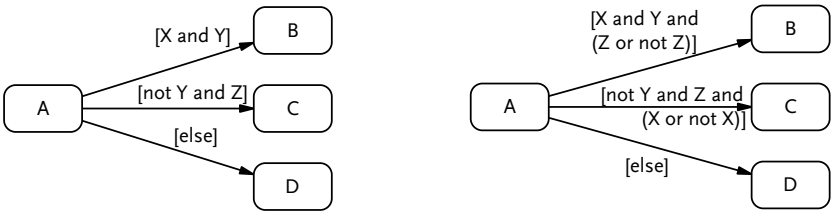
The only restriction to using DNF is that test goals must be created before expressions are transformed into DNF. The reason for that is that the structure of expressions is shown to have an impact on the number of test goals, i.e., the effect of the applied coverage criterion (cp. [HHH+04, FS07, RWH08, Wei09b]). This impact can have unexpected and possibly disadvantageous effects.

3.3.3 Test Goal Extension and Restriction

There is sometimes the need to adapt test goals, i.e., to extend or restrict them. This section describes problems resulting from incomplete guard conditions and how to solve them by adapting the test goals returned by the coverage criterion for a specific test model.

Definition 16 ***(Influencing Expression Set).*** *Each guard condition of a transition is composed of a set of atomic boolean expressions. For each state s of a state machine, we call the union of all atomic boolean expressions of s's outgoing transitions' guards s's* influencing expression set.

Definition 17 ***(Incomplete Guard Condition).*** *A guard condition that does not reference all elements of its transition's source state's influencing expression set is* incomplete.



(a) Incomplete guard conditions.

(b) Complete guard conditions.

Figure 3.23: Add missing elements of A's influencing expression set.

The state A of the state machine in Figure 3.23(a) has three outgoing transitions. Its influencing expression set is $\{X, Y, Z\}$. The guards are mutually exclusive. Nevertheless, all guards are incomplete because no guard references all elements of A's influencing expression set. The following issue arises: One test goal for the satisfaction of Multiple Condition Coverage for *[X and Y]* consists of satisfying the condition *[X and not Y]* in state A. For this condition, there are several possible resulting target states (C and D) – depending on the value of Z. As a consequence, only from the satisfaction of *[X and not Y]*, the test oracle cannot deduce the traversed transition. This restricts the means to check the correct behavior of the SUT for such scenarios, e.g., by evaluating the target state's invariant.

How does a solution look like? Transition-based coverage criteria [UL06, page 115] are focused on sequences of transitions but not focused on value assignments for guard conditions and, thus, their satisfaction is no solution. A possible solution seems to be the extension of existing guard conditions with the missing elements of A's influencing expression set. Figure 3.23(b) shows the corresponding state machine. This test model transformation inserts the missing elements of A's influencing expression set and seems to solve the described problem. As described above, however, the effect of control-flow-based coverage criteria depends on the structure of conditions. Since test model transformations may possibly change the structure of expressions in a disadvantageous manner, we have to avoid this approach.

Instead, we propose the dynamic adaptation of test goals. This means that each test goal is checked for whether there is more than one resulting target state. In such a case, the conditions to satisfy the test goal are extended so that they satisfy the original test goal and missing elements of the influencing expression set are added. In our example, one test goal requires the

satisfaction of *[X and not Y]*. The test goal adaptation results in splitting the original trace pattern (see Section 2.4.2) into two alternative trace patterns that require to test *[X and (not Y) and Z]* and *[X and (not Y) and (not Z)]*, respectively. The test goal is satisfied if at least one of the trace patterns is covered. For each of the new conditions, there is only one resulting target state. As a consequence, the oracle of each test case can predict the traversed transition and the target state for each event trigger. Thus, the test cases have higher fault detection capabilities.

3.3.4 Limitations to Test Goal Management

In this section, we discuss limitations to the sketched test goal management. One can assume that the test goal management process creates exactly one test case for each test goal. There are a few exceptions to that assumption: First, test goals cannot always be covered because, e.g., the corresponding transitions cannot be traversed. The corresponding problem of finding feasible test goals respectively paths that satisfy them has been shown to be undecidable [GWZ94]. Second, the coverage criterion unique-cause MC/DC [CM94, Chi01] is only satisfied iff the isolated impact of each atomic expression on the whole guard evaluation is shown. Thus, the described test goal actually consists of pairs of atomic test goals, each of which is referencing atomic expressions that satisfy or violate the guard, respectively. There are test models in which the satisfaction of such test goal pairs does not necessarily lead to more than one test case (e.g. by traversing loops). In most cases, however, it does. If one of the corresponding atomic test goals is infeasible, this may result in test cases that have to be removed because no matching partner test case could be created.

3.4 Test Case Generation Algorithm

In this section, we present the algorithm for our test case generation approach. The test case generation is subdivided into the generation of abstract test cases and the subsequent generation of input parameter values for each abstract test case. Possible inputs of our test generation algorithm are UML state machines and class diagrams annotated with OCL expressions. The OCL expressions are used to describe, e.g., guards or state invariants of the state machine and pre-/postconditions of the class diagram's operations. Class operations are referenced from the state machine and, thus, their pre-/postconditions can be evaluated together with the conditions of the state machine. Before explaining the generation of abstract test cases, we show

how to interpret OCL expressions in Section 3.4.1. This is necessary for our abstract backward interpretation. The generation of abstract test cases is presented in Section 3.4.2. The corresponding input value selection for each abstract test case is presented in Section 3.4.3. A short example to support the explanation of the algorithm is given in Section 3.4.4. Section 3.4.5 is concerned with the complexity of the algorithm. In Section 3.4.6, the limitations of the test generation approach are listed.

3.4.1 Interpreting OCL Expressions

Postconditions are used to express the effects of operations. However, OCL does not include a notion of assignment, which would be necessary to identify the variables of a postcondition that are actually changed. In this section, we present our approach to interpret OCL expressions: Depending on this interpretation, equations in postconditions can be interpreted as assignments. Furthermore, this also allows to derive useful information from inequations of postconditions.

Classification of OCL Expressions.

We present a classification of all atomic expressions to identify the changeable and the fixed elements in OCL expressions. We motivate this characterization with a short example of the postcondition $X = Y$. This postcondition allows several interpretations: **a)** the old value of Y is assigned to X, **b)** the old value of X is assigned to Y, or **c)** a new value is assigned to both variables. Without further investigation, it would be hard to derive any information beyond the equality of X and Y. There are, however, some influencing factors that allow further results. For instance, the value type of Y can be constant, Y can be a fixed input parameter, or an "@pre" can be attached to Y. In these cases, Y is fixed for the postcondition, X can be changed, and consequently the postcondition is satisfied by assigning the value of Y to X (interpretation **a**). In the following, we present the classification of OCL expressions:

An OCL expression is composed of several atomic predicates that consist of variables *var*, relations between them, and operations on them. A predicate that contains *var* is a *context predicate* of *var*. We classify the test model's variables (attributes, input parameters, or constants) and define *dependent* and *independent* variables. We distinguish two characteristics of OCL variables *var*: their dependency and their context.

Definition 18 (Variables) *The elements in OCL expressions are constants (e.g. numbers or strings), class attributes, or input parameters. We call all elements of these types* variables.

Definition 19 (Dependency of Variables) *Constant class attributes, input parameters, and constants are* independent variables. *Their values are fixed. Non-constant class attributes are* dependent variables. *They depend on previous behavior and value assignments.*

Definition 20 (Context of Variables) *The context of a variable is the kind of OCL expression the variable is referenced from.*

Possible contexts of a variable are, e.g., transition guards, state invariants, preconditions, and postconditions. In the following, we apply the dependency and the context of variables to classify variables. Each variable can have one of two possible classifications: The variable is *changeable* or *fixed*, which basically means that its value can be changed or not.

Only postconditions describe the situation after an action and, thus, express changes. All dependent variables *var* that are contained in postconditions and have no "@pre"-attachment can be changed – *var* is changeable. In all other cases, the value of *var* cannot be changed (because of type, context, or an attached "@pre"), and *var* is fixed. Table 3.2 shows the classification.

Expression Context	Dependent Variable	Independent Variable
postcondition (no "@pre" attached)	changeable	fixed
postcondition ("@pre" attached)	fixed	fixed
all other contexts	fixed	fixed

Table 3.2: Classification of variables in OCL expressions.

Definition and Use of Variables in OCL Expressions.

UML state machines typically reference several OCL expressions. Consequently, paths of a state machine also often reference more than one OCL expression. In order to execute the test case corresponding to a path in a state machine, each referenced expression has to be satisfied or violated, i.e. must have a determined result.

Several expressions that reference the same variable can be connected by this variable. Depending on the variable's classification in both expressions

and the relations between them, one expression can influence the other. We use the terms from data flow analysis to describe these relations:

Definition 21 (Definition of a variable) *A variable* var *is* defined *in an expression* exp *iff* var *is classified as changeable in* exp.

Note that constants and input parameters cannot be defined at a transition. Their values are always fixed.

Definition 22 (Use of a variable) *A variable* var *is* used *in an expression* exp *iff* var *is classified as fixed in* exp.

Definition 23 (Def-use-path of a variable) *A* def-use-path *of a variable* var *is a path from a definition* def(var) *of* var *to a use* use(var) *of* var. *The part of the path between definition and use of* var *must be* definition-free, *which means that there is no other definition of* var *in-between.*

Definition 24 (Def-use-pair) *Expressions* def(var) *and* use(var) *that are connected by a def-use-path of any variable* var, *are called the* def-use-pair (def(var),use(var)).

Definition 25 (Connecting variable) *The variable* var *of a def-use-pair* (def(var),use(var)) *is called a* connecting variable *of* def(var) *and* use(var).

All definitions of defining and using expressions are also used for the corresponding containing transitions. Note that there can several connecting variables of two expressions.

Transforming OCL Expressions.

With the help of the presented classification of OCL variables, expressions can be interpreted as definitions and uses of variables. In this section, we describe how expressions influence each other and how we process them while traversing transitions backward. The approach of transforming OCL expressions corresponds to finding the weakest precondition [Dij76, Whi91, CN00] where *use(var)* denotes the postcondition and *def(var)* denotes the command for computing the weakest precondition. The task is to transform a given expression *use(var)* using the expression *def(var)* so that the satisfaction of the transformed expression and *def(var)* imply the satisfaction of *use(var)*. The final goal of computing the weakest precondition is to replace all dependent system attributes with input parameters so that the corresponding expressions can be used as conditions for selecting input parameter values.

The test algorithm iterates all transitions backward. Thus, for any def-use-pair *(def(var),use(var))* on the path, *use(var)* is visited before *def(var)*. The expression *def(var)* can be used to transform *use(var)* by isolating the connecting variable *var* in both expressions on one side and replacing *var* in *def(var)* with the remaining part of *use(var)*. Any connecting variable must be dependent. Otherwise, it cannot be defined in *def(var)*. The connecting variable must also be fixed in *use(var)*. Otherwise, it would not be used but defined in *use(var)*. In the following, we describe how to deal with *use(var)* depending on the number of included dependent fixed variables.

One Dependent Fixed Variable. The expression *use(var)* contains exactly one dependent fixed variable *var*. Since *var* is the only dependent fixed variable in *use(var)*, it is also the connecting variable of any *def(var)* and this *use(var)*. Thus, *def(var)* influences *use(var)* only via *var* – all other variables in *def(var)* do not influence the satisfaction of *use(var)*.

Our basic approach to deal with this situation is to isolate the connecting variable *var* in *def(var)* and *use(var)* on one side and exchange *var* in *use(var)* with the expression on the other side in *def(var)*. As a result, the satisfaction of *use(var)* is now expressed with all influencing variables (e.g. input parameters or constants) used in *def(var)*, and the postcondition *def(var)* is described with elements that are valid before. For instance, if *def(var)* is $var = p$ with p as an input parameter and *use(var)* is $var > 0$, then the transformed expression is $p > 0$. This transformation is based on the fact that $p > 0$ and $var = p$ imply $var > 0$. As stated above, we have to find a new expression for *def(var)* and *use(var)* so that *use(var)* will be satisfied if the new expression and *def(var)* are satisfied. The occurring relation symbols are important for this replacement: In Table 3.3, we present a list of relation symbols for the resulting expression depending on the expressions

def(var) *var ? expr1*	*use(var)* *var ? expr2*	transformed expression *expr1 ? expr2*
$<$	$<$	$\leq$
$<$	$\leq$	$\leq$
$\leq$	$<$	$<$
$\leq$	$\leq$	$\leq$
$=$	$<$	$<$
$=$	$\leq$	$\leq$
$=$	$=$	$=$
$=$	$<>$	$<>$
$<>$	$<>$	$=$

Table 3.3: Relation symbols for expression transformation.

def(var) and *use(var)*. The term *var* ? *expr* means that *var* is isolated on the left-hand side, *expr* is isolated on the right-hand side, and the relation symbol is shown in the corresponding column.

The same rules are valid if we exchange all $>$ for $<$ and $\geq$ for $\leq$. All remaining combinations of relation symbols cannot be evaluated. For instance, there is no condition about $expr1$ and $expr2$ that could be satisfied before traversing a transition to enforce that the satisfaction of the transition's postcondition $var < expr1$ results in the satisfaction of $var > expr2$.

We present a part of the sorting machine defined on page 64 to clarify this transformation: The postcondition of the operation *detectItem(Item)* is a conjunction that contains the expression $width = (object.width + 2) * 2$. There are several transitions that reference the attribute *width* in a guard condition. One such guard condition is $width < 20$. In *use(var)*, *width* is the only dependent fixed variable – in *def(var)*, *object.width* is an input parameter and 2 is a constant value. The variable *width* is isolated on one side of each expression. According to Table 3.3, these two expressions can be transformed to the new precondition $(object.width + 2) * 2 < 20$. Isolating *object.width* on the left side results in $object.width < 8$, which has to be satisfied in order to satisfy the guard $width < 20$.

Note that this approach also allows to interpret inequations in postconditions. For instance, a postcondition $x < y@pre$ can be used to satisfy a subsequent guard $x < 10$ if a preceding condition requires that $y \leq 10$. According to our experiences in case studies, one dependent fixed variable in a use expression is the standard case.

Several Dependent Fixed Variables. If there are several dependent fixed variables in *use(var)*, the satisfaction of *use(var)* can depend on more than one defining expression. As a consequence, the relationship between these variables is often not easily determined and a simple replacement of variables as soon as there are possible matches is no solution: One reason for this is that this replacement would mask dependencies between the two connecting variables. Since these dependencies can influence the evaluation of other expressions, this would probably change the meaning of the resulting expression. Instead, we do not process expressions with more than one connecting variable but collect all such expressions in groups and evaluate these groups after a candidate test case is found. This current evaluation approach for groups consists of creating transformed expressions for all possible combinations of expressions. Based on this whole set of expressions, possible partitions for each input parameter are derived. Validity checks are used to determine whether intermediate expression groups are contradictory.

As a special case, there may be several expressions in *use(var)* that all reference the same variables. The triangle classification example in Section 3.2.3 contains several such guard conditions. For such examples, the costs of evaluating the collected expression groups is exponential.

3.4.2 Generating Abstract Test Cases

In this section, we describe the abstract test case generation algorithm, whose purpose is the creation of an abstract test case with abstract information about input parameters.

The algorithm starts at a certain point in the test model described by a given test goal. From that point, the algorithm iterates backward in the state machine to the initial configuration with a guided depth-first graph search approach and creates a corresponding trace. When iterating backward, the algorithm collects all conditions and keeps them in a consistent set of data-flow information. According to the satisfiability of this data-flow information, we call the resulting traces *valid*, *contradictory*, or *indetermined*:

Definition 26 (Valid trace) *In a* valid trace*, all guard conditions of the traversed transitions are satisfied. The trace contains one def-use-path for each used variable. This means that all variables necessary to determine the evaluation of the conditions along the trace are determined by initial values, constants, and input parameters.*

Definition 27 (Contradictory trace) *A* contradictory trace *contains contradictions for at least one condition of the path. Thus, it is impossible to find input data that enables the path.*

Definition 28 (Indetermined trace) *An* indetermined trace *contains variable uses without preceding definitions to determine their values. Thus, the values for the respective variables are undefined. Correspondingly, there may be settings in which the path is valid or contradictory, respectively.*

The presented search algorithm only returns abstract test cases from valid traces. The following subsections contain the description of the basic graph search algorithm and possible extensions to steer the algorithm.

The Search Algorithm.

This section contains the description of our test generation algorithm that returns for each test goal an abstract test case together with input parameter constraints. The idea is to find a path by traversing state machine transitions

backward and keeping track of all the guard conditions to satisfy. For the latter, we define two kinds of expressions:

Definition 29 (One-dimensional expression) *All expressions that contain only one dependent variable or input parameter are called* one-dimensional expressions.

Definition 30 (Multidimensional expression) *All expressions that contain more than one dependent variable or input parameter are called* multidimensional expressions.

The *one-dimensional expressions* contain exactly one dependent variable or input parameter. The expression satisfaction depends only on the value of this variable. As soon as the corresponding definition of this variable is identified, the expression can be evaluated. The expressions contained in *multidimensional expressions* contain several dependent variables. As a consequence, the validity of these expressions can only be evaluated after detecting all corresponding input parameters.

The algorithm consists of the following steps:
1) Extending the trace pattern defined by the test goal to possible end parts of the created path (see Section 3.3.3): The result is a sequence of visited states, traversed transitions, and event calls that match the trace pattern defined by the current test goal. The end point of each such extended path is a possible start point for searching backward. Furthermore, outgoing completion transitions (that may contain guards) can also be included in the extended path. This last step is optional. Experience show, however, it is advisable to include it (see Section 4.1.2).
2) Searching a trace backward to the initial configuration: For creating a transition sequence, the same basic algorithm is applied for each encountered state configuration: All incoming transitions are evaluated according to the following aspects: First, the contained postconditions are applied to transform expressions as described in Section 3.4.1 about OCL expression transformations. Second, the transition's guard condition is added to the one-dimensional and multidimensional expressions, respectively. If the evaluation of one-dimensional or multidimensional expressions only depends on input parameters, the expressions are used to determine the concrete input parameter values. The algorithm stops if the initial state is reached and the satisfaction of all encountered one-dimensional and multidimensional expressions can be determined only with input parameters.

The pseudocode in Figure 3.24 shows how trace extensions are created for each trace pattern of the given test goal (line 03). The extension consists of

```
01 TraceExtensions extendPathForTestGoal(tg : TestGoal) {
02   set traceExtensions = {} // keeps track of the current extensions
03   for each trace pattern tp of the test goal tg {
04     traceExtension = ();
05     traceExtension = buildTraceExtension(tp, 0, traceExtension);
06   }
07   return traceExtensions;
08 }
09
10 TraceExtensions buildTraceExtension(
11   tp : TracePattern, i : index, tE : traceExtension) {
12   set traceExtensions = {};
13   if(i < tp.steppatterns.size()) { // add further information
14     sp = tp.steppatterns.get(i); // get current step pattern
15     trans = all transitions that match tE; // candidate extensions
16     for each transition t in trans {
17       add t to tE;
18       traceExtensions.addAll(
19         buildTraceExtension(tp, i+1, tE)); // recursion with i+1
20       remove t from tE;
21     }
22   } else { // create additional extension from tE
23     s = target state of last transition in tE;
23     for all outgoing completion transitions t of s {
24       add t to the end of tE;
25       traceExtensions.addAll(
26         buildTraceExtension(tp, i+1, tE)); // recursion with i+1
27       remove t from tE;
28     }
29     if there are no outgoing completion transitions of s {
30       add copy of tE to traceExtensions;
31     }
32   }
33   return traceExtensions;
34 }
```

Figure 3.24: Test goal extension for all all trace patterns of a test goal.

concrete trace information that match the trace pattern. At the beginning, it's an empty sequence (line 04). In line 05, *buildTraceExtension()* is called for each trace pattern: All step patterns that are contained in the current trace pattern are used for the extension. The index i describes the current step pattern (line 11). If there are still unused step patterns in the trace pattern (line 13), then all possibly matching transitions are identified and buildTraceExtension() is called for the next step pattern (i+1 in line 19). If all step patterns of the trace pattern have been used (line 22), then completion transitions [Obj07, page 568] are added to the extension (lines 23-28). If there are also no more completion transitions, the current extension is saved and returned (lines 18, 25, 30, and 33).

```
36 TestCase createTestCase(te : TraceExtension) {
37   n = target node of the last transition of te;
38   TestCase tc = searchBackwardsFromNode(n, te);
39   if(tc is a valid test case) { return tc;}
40   else { return null;}
41 } }
42
43 TestCase searchBackwardsFromNode(n : Node, te : TraceExtension) {
44   if(n is initial node and all expressions are satisfied) {  // valid
45     return test case that contains the current path information;
46   }
47   TestCase tc = null;
48   if(n has a transition t that is part of te) {
49     tc = traverseTransition(t, te);
50     if(tc != null) return tc;
51   } else {
52     for each incoming transition t of n {
53       tc = traverseTransition(t, te);
54       if(tc != null) return tc;
55   } }
56   return null; }
57
58 TestCase traverseTransition(t : Transition, te : TraceExtension) {
59   transform all one-dimensional expressions with t's postcondition;
60   classify precondition of t and add it to the one-dimensional or
61     the multidimensional expressions;
62   tc = searchBackwardsFromNode(t.sourceNode, te);
63   if (tc is valid test case) { return tc;}
64   else { return null;}
65 }
```

Figure 3.25: Pseudocode for generating test cases by searching backward.

In Figure 3.25, the pseudocode for the creation of test cases is shown. The just generated trace extensions are used as starting points for searching

backward. They are used one by one for the operation *createTestCase()*. As soon as a valid test case is returned, all remaining trace extensions of the test goal are discarded. If the use of no trace extension leads to the creation of a test case, the test generation for that test goal fails.

The target node of the trace extension's last transition is the start node (line 37) for the search algorithm implementation in the function *searchBackwardsFromNode()* (line 38). Line 44 shows that the algorithm stops as soon as an initial configuration is found and all one-dimensional and multidimensional expressions are satisfied. If the initial configuration is not reached yet, then it is checked if one incoming transition of the current node n is part of the trace extension (line 48). The described trace pattern should be included once at the beginning of the path search – afterwards, the trace extension plays no role for selecting transitions. If yes, then this transition is traversed (line 49). Otherwise, all incoming transitions of the current node are investigated (line 52). The function *traverseTransition()* describes how to deal with the expressions attached to transitions (lines 58 – 65). For each such transition t, *t's* postcondition is used to transform the current one-dimensional expressions (line 59) as described in Section 3.4.1. Furthermore, the guard of t is added to the one-dimensional or multidimensional expressions and used for further computations (lines 60, 61). Afterwards, the search is continued for the source node of t (line 62). If this search was successful, then the current test case is returned (line 63), and all remaining trace extensions are discarded. If the search is not successful, no test case is returned (lines 40, 56, and 64).

Applying Search Techniques.

The algorithm is presented on an abstract level. There are many details that can influence the search algorithm. Some of them are used in the implementation of the corresponding prototype ParTeG. First of all, it is shown in [GWZ94, JBW+94] that the problem of detecting infeasible test paths is undecidable. As a consequence, there may be many improvements, but there will never be a guarantee that all feasible test cases will be detected. Since a perfect, deterministic search algorithm is impossible, we only sketch some improvements.

For instance, the application of standard search algorithms like the presented depth-first results in high costs. For small standard examples like the freight elevator or the triangle classificator in Section 3.2.3, the costs of test generation are not important. When we applied ParTeG to the academical and industrial test model in Sections 3.2.4 and 3.2.5, we ran into exponential runtime and stopped test generation after several minutes. Brute force

depth-first search was infeasible and, thus, we implemented data-flow-driven approaches to steer the search backward to the closest definition of a variable. One important problem of our test generation approach to solve is the inclusion of expressions in the path that define the variables that are used subsequently in the path. If these define-expressions are not included, the resulting test case is considered indetermined. As a solution, the algorithm steered the selection of the current nodes and the next transition to traverse backward in order to find postconditions that set the values of variables that are used but not defined, yet. The results of these approaches was that satisfying test suites were generated for both mentioned academical and industrial test models within a few seconds.

There are further approaches to utilize data-flow information for test generation. For instance, Briand et al. [BLL05] and Weyuker [Wey93] consider data-flow information to improve test case generation. Korel [Kor90] pursues a forward searching approach but also applies data analysis in order to determine the input parameters that influence the evaluation of a guard value.

3.4.3 Selecting Input Values

In this section, we describe how to select concrete input values for abstract test cases generated in the previous section. The basis for input data selection are abstract value domains of input parameters. As described in Section 3.1.4, it is advisable to select only a few representatives at proper positions inside of each input partition. In Section 2.1.5, we described that there are corresponding coverage criteria for value selection from partitions [KLPU04]. In the following, we show how to utilize the abstract information about input partitions to create representatives that satisfy boundary-based coverage criteria like Multi-Dimensional.

Multi-Dimensional.

The criterion Multi-Dimensional [KLPU04] is satisfied if each variable that is contained in describing an input partition takes its minimum and maximum value at least once. The task is to select these values for each referenced input variable instance in each abstract test case. As defined on page 80, we distinguish one-dimensional expressions and multidimensional expressions.

One-dimensional expressions only contain information about one parameter. The coverage criterion Multi-Dimensional is satisfied for one-dimensional expressions by initially creating an unbounded abstract value partition for the input parameter and restricting it step-wise for each one-dimensional ex-

pression. The result is a value partition whose representatives satisfy all used expressions. The boundary values of this value partition are the minimum and maximum values, respectively, for the corresponding input parameter. Figure 3.26 depicts this approach: The three expressions about x ($x > 5$, $x > 7$, $x < 11$) are combined to a value space from 7 to 11 (both exclusive as the hachures show).

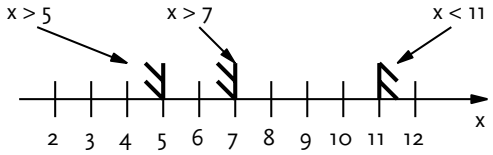


Figure 3.26: Value partition for one-dimensional expressions.

For multi-dimensional expressions, the satisfaction of Multi-Dimensional is not that easy. The obvious reason is that there are more than one input parameters. The triangle classification is a good example to clarify this. There are several multidimensional expressions that are all focused on three parameters x, y, and z that correspond to the three sides of a triangle. There are several possible value combinations. Our approach is a step-wise refinement of the expressions. For that, we focus on only one parameter. We evaluate all expressions like for one-dimensional expressions and determine temporary value partitions. After that, we use all expressions and the already existing temporary value partitions to determine the value partition for another variable and so forth. This is necessary because some of the multidimensional expressions can only be evaluated if there is partly knowledge about other parameters. For instance, $x < y + z$ can only be evaluated for x if y and z are known in advance. Note that this is an initial solution, and the use of constraint solvers may lead to a more efficient computation of more complex expressions.

3.4.4 Example

In this section, we explain the presented test goal management approach with the example of the sorting machine as presented in Section 3.2.1. In this example, the goal is to satisfy All-States on the state machine and Multi-Dimensional for each resulting abstract test case. Figures 3.27 and 3.28 show the state machine and class diagram of the test model.

As presented in Section 3.3.1, the first step is to create test goals by applying the coverage criterion All-States to the state machine. We select an

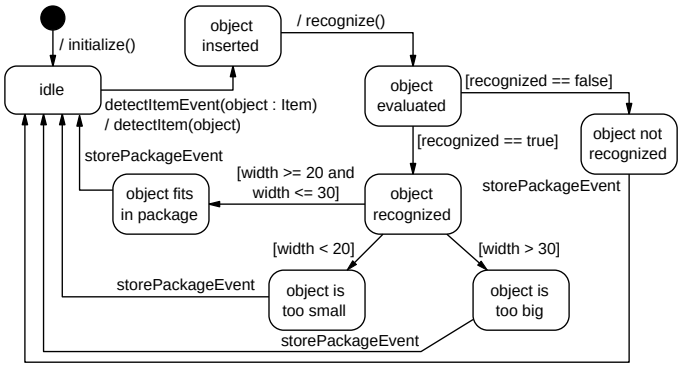


Figure 3.27: State machine of a sorting machine.

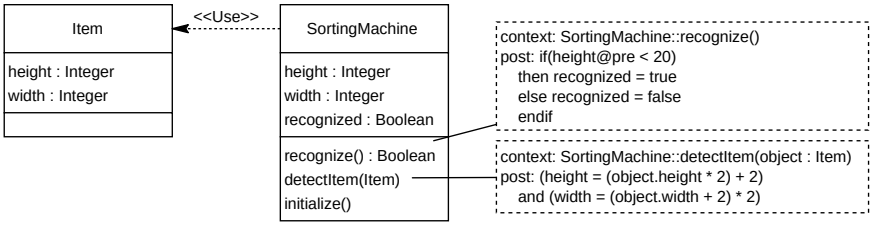


Figure 3.28: Class diagram of a sorting machine.

arbitrary test goal with the trace pattern *((object recognized, ?, ?, ?))*, which is the test goal that is covered by visiting the state "object recognized".

The second step is the test case generation for this test goal. First, the given trace pattern is extended: The referenced state has several outgoing completion transitions. Corresponding target states without completion transitions are "object is too big", "object is too small", and "object fits in package". The algorithm creates corresponding trace extensions. Afterwards, these extensions are used for test generation. We start with the extension that ends in the state "object is too big". There is only one shortest path back to the initial node. Traversing two transitions backward, the algorithm collects two one-dimensional guard expressions: $width > 30$ and $recognized = true$. The next transition references the operation *recognize()*, whose postcondition is used to transform $recognized = true$ into $height < 20$. Here, the transformation of the expressions into DNF is used: The postcondition of *recognize()* is transformed so that there is a conjunction ($height@pre < 20$ *and* $recognized = true$), which can be used to derive the new condition $height < 20$. The next transition references a postcondition that is used to transform both one-dimensional expressions into $(object.width+2)*2 > 30$ and $(object.height*2)+2 < 20$, respectively. These

expressions can be transformed to $object.width > 13$ and $object.height < 9$, respectively. After that, the initial state can be reached with one transition traversal. All one-dimensional expressions are mapped to input parameters, i.e. a corresponding definition is found. The expressions are not contradicting. Thus, the generated abstract test case is valid. The remaining task is to identify the concrete input values. As stated in Section 3.4.3, this is quite easy for one-dimensional expressions: *object.height* is set to 8 and *object.width* is set to 14. The result is a concrete test case. All remaining trace extensions are discarded.

In the third step, we monitor the test goal satisfaction. All generated test goals contain trace patterns that reference single states. The created path visits the states "object is too big", "object recognized", "object evaluated", "object inserted", and "idle". Thus, it also satisfies the corresponding test goals. As a result, the test case generation only has to be executed for the remaining three states.

Step four is about optimizing the resulting test suite. Since we present the generation of only one test case, we just sketch this step. The generated test case visits the state "object is too big" and satisfies the corresponding test goal. The test goal monitoring guarantees that there will be no further test case generation for this test goal. Since "object recognized" has to be visited to visit "object is too big" and the monitoring guarantees that "object recognized" has not been visited before, there has been no test case to satisfy "object is too big" before generating the test case for "object recognized". Thus, there is no further test case that visits "object is too big". The just created test case is not redundant and will not be removed.

3.4.5 Complexity

This section contains a description of the presented approach's complexity. We consider all phases in the test goal management as presented in Section 3.3.1 and further problems that depend on the selected target language.

Test Goal Management.

The first phase is the test goal generation. It depends on the selected coverage criterion. Simple coverage criteria like All-States or All-Transitions create one test goal per state or transition, respectively. For such coverage criteria, the effort is linear with the expression size, e.g. measured as the number of variables in a guard. Most control-flow-based coverage criteria subsume All-Transitions. Thus, their effort is at least linear with the test model size. They depend, however, also on the complexity of the considered conditions:

Decision Coverage requires two condition evaluations and its effort is constant with the condition size for each condition. MC/DC is linear with the model size [CM94]. Multiple Condition Coverage requires that each possible value assignment of a condition's truth table is used and its effort is, therefore, exponential ($0(c^n)$). There are, however, even worse coverage criteria: All-Paths is satisfied iff all paths of the state machine are traversed. Since loops can cause an infinite number of paths, the test goal generation may never stop. A stop timer like it is used in ATG [IBM] might be a solution to prevent infinite test generation. For that, we propose to satisfy all subsumed feasible coverage criteria in a step-wise manner.

The second phase is the test case generation. It depends on many factors and is basically undecidable [GWZ94]. It's complexity depends on the number of attributes that influence the path search, the number of transitions and how many loops they form, the number of parallel regions in the state machine, and the overall size and complexity of all data-flow information. We described in Section 3.4.2 some techniques to improve the test generation.

Test goal monitoring is the third phase. The brute force approach is to check for each existing test goal whether the generated test case covers it. Test goals do not match the whole test case but only steps of it. Thus, each single step of each test case must be compared to each step pattern of each test goal. This results in n comparisons, where n is the number of steps per test case times the number of test cases times the number of test goals: $n = \#steps \times \#testcase \times \#goals$. As a rule of thumb, the number of test goals, the number of test cases, as well as their length (number of steps) depend on the size of the test model. As a consequence, the effort of test goal monitoring is about the size of the test model to the power of three ($0(n^3)$). However, this is only an upper boundary: First, if the test cases are longer, then there is a high probability that many test goals are accidentally satisfied and fewer test cases have to be created and checked for test goal satisfaction. Furthermore, the implementation contains a map from model elements to test goals, which requires to compare each step of a test case only to a small subset of all test goals.

In the fourth and last phase, the test suite is optimized. This means that all satisfied test goals of a test case are checked for whether they are also satisfied by other test cases. If each test goal is also satisfied by another test case, then the current test case is redundant and can be removed. The basic effort for these operations is high: For each test case and all test goals that are satisfied by it, all other test cases have to be analyzed whether they satisfy the same test goals. In the implementation, this issue is solved by creating maps from test cases to the test goals they satisfy and from test goals to the number of satisfying test cases. Both maps are created and filled during

the monitoring phase. No further comparisons are necessary. The overall task is only to check for each test case whether all mapped test goals are in turn mapped to more than one test case. In this case, all test goals of the test case are also satisfied by other test cases, the test case is removed, and the number for each referenced test goal is decreased by one. The remaining effort corresponds to the product of test cases and their satisfied test goals.

As we have seen, many aspects of the presented algorithm's complexity analysis depend on the details of the used state machine. That is why the main statements are rather sketchy and the whole question whether the algorithm terminates at all (if not stopped by a time limit or buffer overflow) is undecidable [GWZ94]. Nevertheless, the current implementation performed good in comparison to some commercial tools. For instance, for the state machine of the industrial train control test model of Section 3.2.5, ParTeG needed only 10 seconds to create a test suite that satisfies a combination of Multiple Condition Coverage and Multi-Dimensional whereas the evaluation version of the commercial tool Smartesting Test Designer [Sma] in version 3.2.1 (Leirios) needed about 25 minutes for creating a test suite that satisfies only All-Transitions.

Expression Solving.

The OCL expressions that are currently used can contain equations, inequations, and logical operators. After transforming all expressions into expressions that only contain constant values and input parameters, these new expressions are used to derive concrete input parameter. There might be complex expressions like polynomial, exponential, or further mathematical functions for which it is not easy to derive concrete representatives. In all used examples, the academic, and the industrial cooperation, there were only linear expressions. Using the agile approach [BBvB+01], there was no need to implement further support. We are aware, however, that the problem of finding a solution for a set of expressions is non-trivial. There are several kinds of SAT solving [GMS98, AS05, HGK06] that are focused on the problem of detecting representatives that satisfy a set of constraint expressions.

Target Language Selection.

This section contains several target-language-dependent problems that we faced for the created test suites. The first approach of printing test files for mutation analysis in Java was to print all code into one static function. This works well for all coverage criteria applied to the sorting machine example and the freight elevator. Using the train control state machine already resulted

in a Java error because the size of the function exceeded the 65536 characters limit. To solve this issue, we split up the generated code into one function per test case. Another problem was printing all data into one stream. This resulted in stack overflow of ParTeG. As a consequence, we flushed the used streams at constant intervals into the file to keep memory usage low. A C++-specific problem occurred when compiling the CppUnit [Sou08] files with MS Visual Studio. This resulted in an internal C++ compiler error because there were too many object instances in one file. The corresponding solution consisted in splitting the test suite over several output files. We added a corresponding switch to ParTeG to define the number of output files. All test cases are equally distributed to the generated output files.

3.4.6 Restrictions

In this section, we describe the restrictions of the presented test generation approach. We include the supported coverage criteria, state machines, and OCL expressions.

In Section 2.4.3, we presented the formal definitions of coverage criteria. We listed many important coverage criteria without claiming completeness. Transition-based and control-flow-based coverage criteria are considerably easy to use for test generation, because they describe connected sets of model elements to start the the search for the initial configuration. For data-flow-based coverage criteria, the algorithm has to start at the use of a variable, has to include a definition of this variable, and afterwards has to find a path to the initial configuration. This corresponds to the concatenation of two searches and, thus, also does not pose a problem. Infeasible coverage criteria like All-Paths can result in a non-terminating test goal generation.

Basically, there is no restriction to the used state machines other than undecidability. The proposed search algorithm is based on a graph search algorithm. Since state machines are graphs, there is no general restriction to graph constructs that cannot be evaluated or traversed, respectively. Of course, the test generation algorithm may be impossible, e.g., because of an infeasible coverage criterion or an unbounded loop in the state machine. Such restrictions, however, are common to all test generators.

Our test generation approach is based on state machines and class diagrams with OCL expressions. Section 3.4.1 includes descriptions of how to interpret OCL expressions. The main point is transforming OCL expressions using the weakest precondition approach. As a consequence, our test generation is restricted to OCL expressions that can be used to deduce preconditions. For all others, the test generation algorithm just fails to deduce concrete input values or abstract test cases at all. Again, this restriction is

common to all test generators because it depends on the test model's degree of completeness and abstraction.

3.5 Case Studies

We implemented the presented test generation approach in the tool ParTeG (Partition Test Generator) [Weib]. Here, we shortly describe our prototype and the experiments for combining coverage criteria: We describe the implementation of ParTeG in Section 3.5.1. In Section 3.5.2, we present the mutation analysis on artifical implementations. We summarize the results of the experiments in Section 3.5.3.

3.5.1 Prototype Implementation

ParTeG is an Eclipse plug-in that is based on the Eclipse Modeling Framework (EMF) [Ecl07a] and is able to create test suites for UML state machines and class diagrams that are described with the Eclipse UML 2.1 plug-in [Ecl07b]. EMF is a modeling framework comparable to EMOF (Essential Meta Object Facility). EMOF is the OMG standard to describe metamodels like the UML. The used UML 2.1 plug-in is an implementation of the UML description based on EMF. Besides the used modeling frameworks, ParTeG is implemented in the multi-purpose language Java. Possible editors for creating the test models are the editors of the UML 2.1 plug-in or the open source editor TOPCASED [Ope09]. The current version of ParTeG is 1.3.2.

The tool ParTeG is a prototype implementation to evaluate the approach of combining test path generation with boundary value analysis. It supports the satisfaction of the transition-based coverage criteria All-States and All-Transitions, as well as the control-flow-based coverage criteria Decision Coverage, masking MC/DC, and Multiple Condition Coverage (see Section 2.1.5). ParTeG also prints a log containing all considered test goals. Input parameter values for the generated abstract test cases are selected randomly or by applying the boundary-based coverage criterion Multi-Dimensional with or without additional random values and values near absolute type minima and maxima. The evaluated OCL expressions are restricted to boolean and linear arithmethic expressions. Constraint solvers can be used to increase the set of supported OCL expressions. ParTeG implements the whole test goal management as described in Section 3.3. It uses an independent internal test case graph metamodel, on which the test generation algorithms run. All used state machines and class diagrams are transformed into that model. The purpose of the internal metamodel is the reusability of

all implemented algorithms for other modeling languages like UML activity diagrams or UML protocol state machines: The remaining work consists of transforming these models into test case graph models. In the current version, ParTeG supports the code generation for JUnit 3.8, JUnit 4.3 [EG06], CppUnit 1.12 [Sou08], and Java Mutation Analysis – a proprietary code format for automated mutation analysis.

There are several possible extensions to ParTeG concerning the supported elements of UML state machines and OCL expressions. For instance, history states are not supported yet, and also OCL collection expressions are not fully supported. The current version supports parallel regions only at the uppermost state machine level, all kinds of boolean and linear arithmetical OCL expressions, and all vertices except history states. Further extensions comprise a plug-in for a user-defined output format, e.g. by providing corresponding pattern files. The current implementation is sufficient to generate test cases for all presented example models.

3.5.2 Mutation Analysis

In our case studies, we use ParTeG to automatically generate test suites from a UML state machine, the corresponding context class, and OCL expressions. To evaluate the generated test suites, we use mutation analysis (cf. Section 2.1.5). The mutation analysis is executed on manually created implementations of the test models. For running mutation analysis, we use the existing mutation analysis tool Jumble [UTC+07] and the Java Mutation Analysis output format of ParTeG. Jumble is an open source mutation analysis tool that works at byte code level to evaluate the fault detection capability of JUnit 3.8 test suites. Jumble in version 1.1 returns only percental integer values of the mutation score but no absolute numbers. The percental values are imprecise if there are more than 100 mutants. To get the absolute numbers, we adapted the provided source code so that absolute numbers of detected mutants are provided. Although we configured Jumble so that the maximum number of available mutants is used, we have only restricted control over the applied mutation operators or the place where they are applied [UTC+07]. Therefore, we created the mutants that are used for the ParTeG Java Mutation Analysis manually. For mutation analysis, they are all provided by an external mutant factory, which returns all mutants one by one for evaluation. This gives us improved control over the application of mutation operators. As a result, Jumble and Java Mutation Analysis used different sets of mutants. Thus, the maximum number of killed and detectable mutants differ for both approaches. For both approaches, we identified the detectable mutants from the set of all mutants – we measured

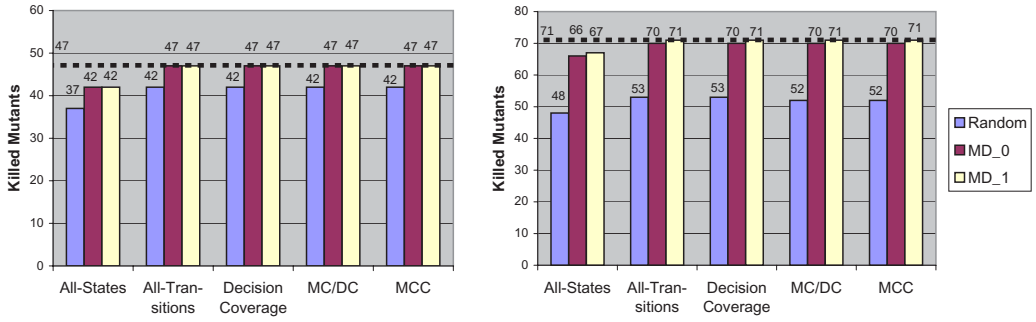
the fault detection capability only with them. In all figures, we used a black dashed line to depict the number of detectable mutants. Note that all presented results are no comparison of Jumble and the Java Mutation Analysis, but studies about the impact of combining coverage criteria as supported by ParTeG. Furthermore, for evaluating the impact of a coverage criterion, the numbers of detected and undetected mutants are important.

We use both mutation analysis approaches to compare the fault detection capabilities of test suites that satisfy single transition-based or control-flow-based coverage criteria to that of test suites that satisfy the combined coverage criteria supported by ParTeG. For that, we measured the effect of the coverage criteria in terms of detected mutants and test suite size of the generated test suites for all the test models that are presented in Section 3.2. For creating abstract test cases, we use the coverage criteria All-States, All-Transitions, Decision Coverage, masking MC/DC, and Multiple Condition Coverage. For selecting input parameter values, we applied random selection (*Random)*, Multi-Dimensional that only creates values near partition boundaries between two partitions (*MD_0*) and Multi-Dimensional in combination with one randomly selected value of the partition and values at absolute type boundaries (*MD_1*). The terms in parentheses denote the terms used in the following figures.

We know of no tool that is able to include boundary value analysis for the automatically generated input partitions of all generated abstract test cases. For this reason, the results of the random value selection approach corresponds to the state of the art of commercial tools. The two other criteria for concrete input parameter selection MD_0 and MD_1 are only supported by ParTeG. In the beginning of 2008, we used a temporary, scientific version of the Smartesting Test Designer [Sma] version 3.2.1 and Rhapsody ATG 6.2 [IBM] for a comparison with ParTeG. Mutation analysis (see Section 2.1.5) showed that both commercial tools detected only 50% of the mutants detected by ParTeG. As mentioned above, the Test Designer also needed considerably more time than ParTeG. For each mutation analysis, we show three bars for each used structural coverage criterion. Each bar presents the detected mutants for the test suites that are generated by combining the structural coverage criterion with *Random*, *MD_0*, and *MD_1*, respectively.

Sorting Machine.

Figure 3.29 shows the results for the mutation analysis with Jumble and with Java Mutation Analysis for the sorting machine example. Jumble used 47 semantically different mutants – Java Mutation Analysis used 71. The black dashed lines show these values.



(a) Jumble. (b) Java Mutation Analysis.

Figure 3.29: Mutation analysis for the sorting machine example.

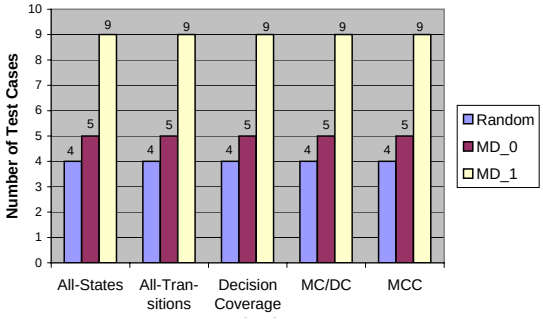


Figure 3.30: Test suite sizes for the sorting machine.

For Jumble, the application of MD_0 or MD_1 results in an increase of 5 killed mutants compared to the random value selection, which corresponds to an average increase of 13%. MD_1 has no additional advantage over MD_0.

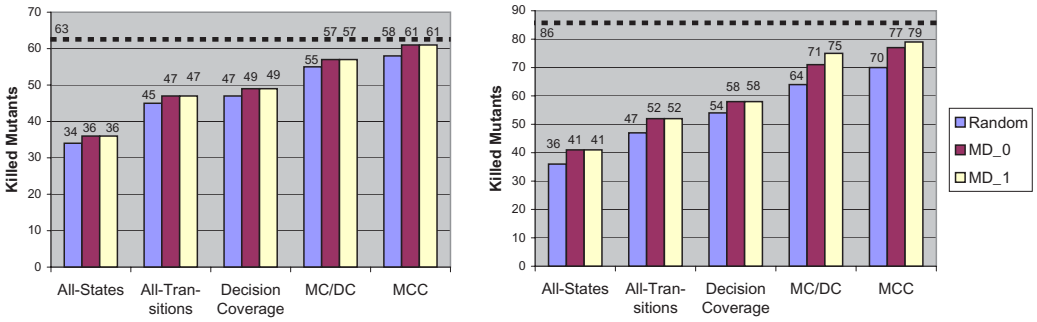
The mutation analysis with the Java Mutation Analysis output of ParTeG shows a similar picture. Only the extent of the improvement is larger. On average, the number of killed mutants is increased by 18, which corresponds to an increase of more than 34%. The application of MD_1 instead of MD_0 results in an additional advantage of 1 killed mutant.

Note that the numbers of undetected mutants are even more important to measure the impact of coverage criteria combination: For both measurement approaches, the numbers of undetected mutants are reduced to 0.

Figure 3.30 shows the test suite sizes for all coverage criteria combinations. The test suite sizes increase for all coverage criteria from 4 (Random) to 5 (MD_0) and 9 (MD_1), which corresponds to an increase of 25% and 125%, respectively. Note that the test suite size is equal for both kinds of mutation analysis: The test suites are almost equal – their main difference is only the output format (Java Mutation Analysis or JUnit 3.8, respectively).

Freight Elevator.

Here, we evaluate the fault detection capabilities of test suites generated for the freight elevator example. Figure 3.31 shows the corresponding numbers. Jumble used 63 detectable mutants. Java Mutation Analysis executed the test suite for 86 mutants. Again, the dashed lines depict the total numbers of detectable mutants.



(a) Jumble. (b) Java Mutation Analysis.

Figure 3.31: Mutation analysis for the elevator control example.

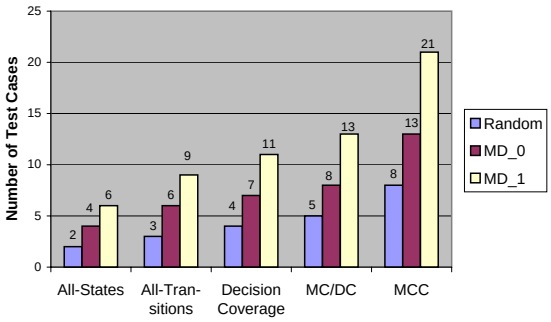


Figure 3.32: Test suite sizes for the elevator control example.

The application of Jumble brought the following results: For all coverage criteria, the number of killed mutants increases by 2 to 3 mutants, which corresponds to an increase of 5.1% to 5.8%. Again, selecting MD_1 instead of MD_0 has no impact.

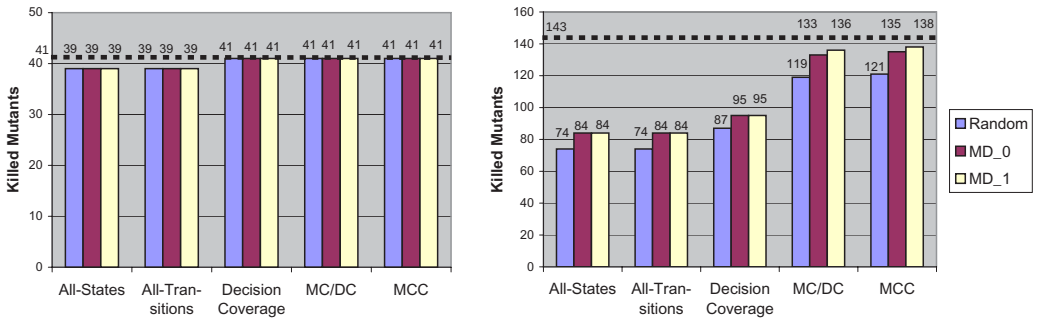
The evaluation with Java Mutation Analysis shows a stronger increase: For MD_0, the number of killed mutants increases for all applied coverage criteria by 4 to 7 mutants, which corresponds to an increase of 6.2% and 14.3%, respectively. The selection of MD_1 instead of MD_0 brings only advantages for MC/DC and MCC: The number of killed mutants is increased

by another 6 and 2 mutants, respectively. Note again, that the number of undetected mutants is considerably decreased – for MCC, from 16 to 7.

In Figure 3.32, the test suite sizes for all coverage criteria combinations are shown. Compared to Random, the test suite size increases for MD_0 between 62.5% and 100% and for MD_1 between 162.5% and 200%. Again, the size of the test suites are equal for both Jumble and Java Mutation Analysis. This also holds for all following evaluations.

Triangle Classification.

The triangle classification example is a standard example for automatic test generation (cp. [UL06, page 214]). The challenge of this example is the large number of interdependencies of the three input parameters. This makes this example especially hard for boundary value analysis. Figure 3.33 shows the results for the corresponding mutation analysis. The number of detectable mutants for Jumble is 41 – for Java Mutation Analysis, it is 143.



(a) Jumble. (b) Java Mutation Analysis.

Figure 3.33: Mutation analysis for the triangle classification.

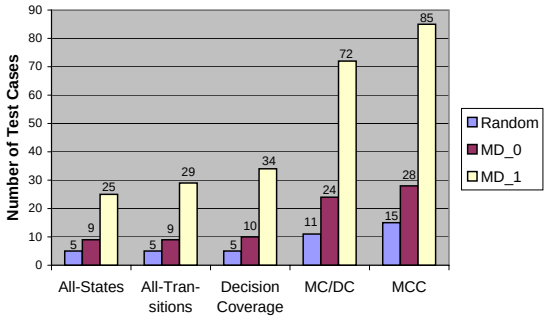


Figure 3.34: Test suite sizes for the triangle classification.

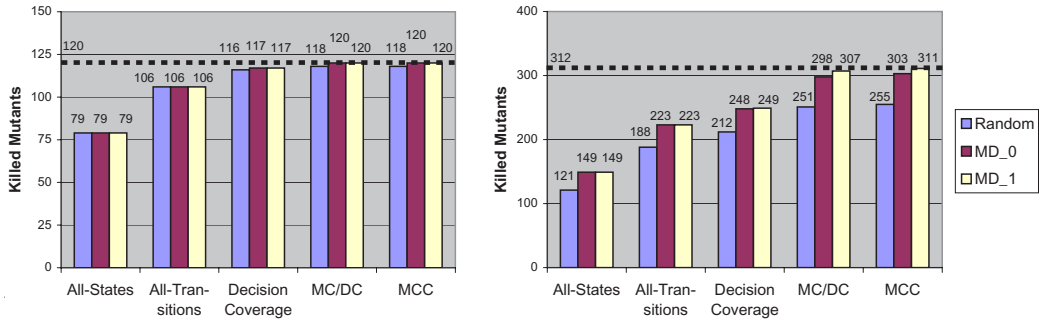
Jumble does not yield an improvement of the mutation score. This is caused by the used mutation operators, on which we have only restricted influence: The application of Decision Coverage, MC/DC, or MCC already results in killing all detectable faults. This means that there is no room for improvement for a combination of coverage criteria.

The mutation analysis using Java Mutation Analysis shows an increase for MD_0 and MD_1. The application of MD_0 instead of Random results in an increase of the killed mutants between 8 and 14. This corresponds to an increase of 9.2% to 13.5%. For MC/DC and MCC, the application of MD_1 instead of MD_0 brought an additional effect: For both, the number of detected mutants was additionally increased by 3. Furthermore, the numbers of undetected mutants were decreased considerably: For MCC, the number is decreased from 22 (Random) to 5 (MD_1).

Figure 3.34 depicts the corresponding test suite sizes. The test suite sizes of this example increase considerably: For MD_0, the test suite size is doubled compared to Random. For MD_1, it is about sixtupled.

Track Control.

The track control example contains a test model that is taken from the UnITeD project [PS07] of the University Erlangen. Figure 3.35 shows the number of killed mutants. Jumble used 120 detectable mutants. For the Java Mutation Analysis, 312 mutants were tested.



(a) Jumble. (b) Java Mutation Analysis.

Figure 3.35: Mutation analysis for the track control example.

Jumble detected no increase of the fault detection capability for All-States and All-Transitions. For the stronger structural coverage criteria, the application of MD_0 and MD_1 results in an increase of the killed mutants between 1 and 2, which correspond to 0.8% and 1.7%, respectively. Since

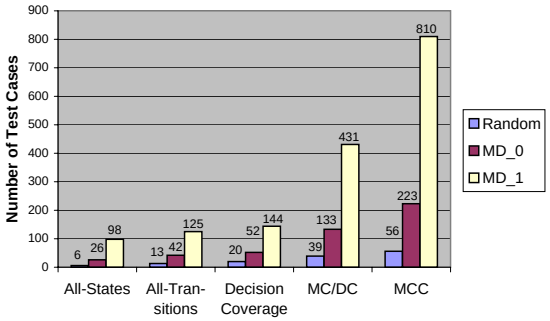


Figure 3.36: Test suite sizes for the track control example.

the combination with MD_0 results in killing all detectable mutants, the application of MD_1 instead of MD_0 brings no additional gains.

The mutation analysis with Java Mutation Analysis also shows a strong impact of the coverage criteria combination on the number of killed mutants: For MD_0, it increases between 28 and 48. Applying MD_1 instead of MD_0 brings for MC/DC and MCC another advantage of 8 or 9 killed mutants. Combining MCC with MD_1 instead of Random results in a decrease of the undetected mutants from 57 to 1.

Figure 3.36 shows the test suite sizes for this example. The test suite sizes increase considerably for MD_0 and MD_1: The test suite size is multiplied with 4 for MD_0 and with 15 for MD_1.

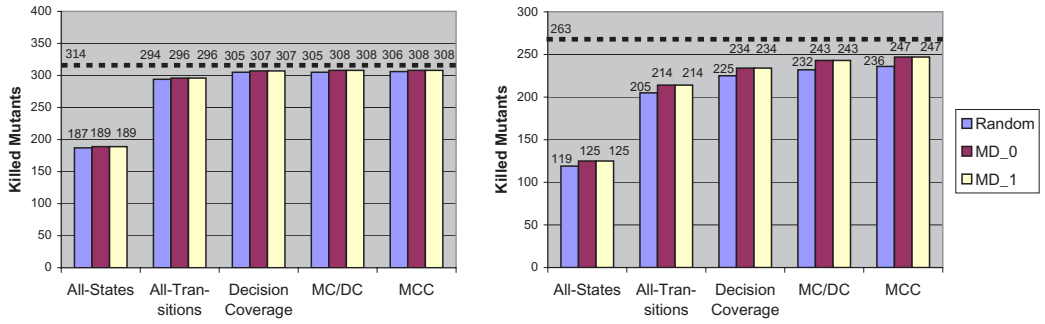
Train Control.

The test model of the train control example was used in an industrial case study. Figure 3.37 shows the results of the mutation analysis with Jumble and the Java Mutation Analysis output of ParTeG. Jumble applied 314 mutants, and Java Mutation Analysis applied 263 mutants.

The number of killed mutants for Jumble is only increased by 2 to 3. For MCC and MD_1, only 6 of 314 mutants remain unkilled.

The mutation analysis with Java Mutation Analysis shows a different result: For all coverage criteria, there is an increase of killed mutants between 6 and 13, which corresponds to an increase of about 5%. The selection of MD_1 instead of MD_0 brings no additional advantage.

Figure 3.38 shows the test suite sizes for the case study. The test suite sizes increase almost linearly: For all coverage criteria, the application of MD_0 doubles the test suite size and the application of MD_1 triples the test suite size compared to random value selection.



(a) Jumble. (b) Java Mutation Analysis.

Figure 3.37: Mutation analysis for the train control example.

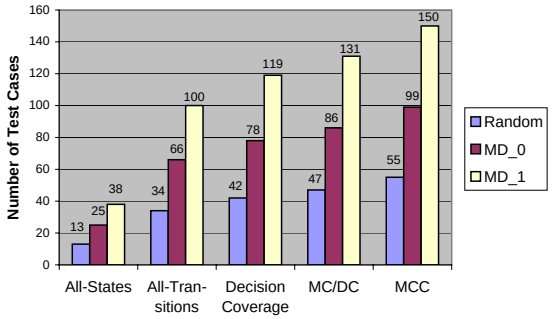


Figure 3.38: Test suite sizes for the train control example.

3.5.3 Results of Mutation Analyis

The fundamental result of the presented mutation analysis is that the combination of coverage criteria as presented in this chapter usually cause an increase of the mutation score. Given the importance of detecting the last undetected mutants [ABLN06], this increase is an important advantage of automatically generated model-based test suites – in some cases, the number of detectable but unkilled mutants is reduced to almost zero. As a second result, the test suite sizes are increased considerably when applying MD_0 or MD_1. Since there are case studies [ABLN06] that estimate an exponential growth of the test suite size for the detection of the last undetected mutants, this increase is not surprising. After all, applying the presented coverage criteria combinations is still a decision of the overall test risk management. The combinations of coverage criteria are a means to increase the fault detection capability beyond the ones for single coverage criteria. Given enough time and a strong interest in test quality, the application of these combinations is a valuable instrument.

The impacts measured with Jumble are always lower than the ones measured with Java Mutation Analysis. Furthermore, applying MD_1 instead of MD_0 never brings an additional impact for Jumble. One reason for this is that Jumble only supports the change of linear-ordered type instances if they are defined as literal constants [UTC+07], which is often not the case in the used implementations. However, Jumble also shows a positive impact of coverage criteria combination on the test suite's fault detection capability in most cases. The Java Mutation Analysis of ParTeG is based on manually created mutants according to the mutation operators presented in Section 2.1.5. The measured impact of applying the boundary-based coverage criteria for the Java Mutation Analysis was always stronger than the impact measured with Jumble. Furthermore, applying MD_1 instead of MD_0 often has an additional impact for Java Mutation Analysis.

Combining coverage criteria has an impact on all considered coverage criteria. In most cases, the absolute impact is similar for all coverage criteria. Since there are less undetected mutants for stronger coverage criteria like MCC than for weaker coverage criteria like All-States, we consider the undetected mutants for stronger coverage criteria harder to detect and the relative impact of combining coverage criteria bigger for stronger coverage criteria. A similar effect can be observed for the application of MD_1: For the elevator control, the triangle classification, and the track control, applying MD_1 instead of MD_0 only has an effect for MC/DC and MCC.

The identification of undetectable mutants requires a considerable amount of manual work: We manually checked all unkilled mutants to identify the undetectable mutants. Since running the test suites on the correct SUT does not result in the detection of any failures, the executed test suites only kill detectable mutants. In the following, we summarize the results of the mutation analysis for the generated test suites to satisfy MCC. Table 3.4 shows the Jumble mutation scores for all combinations of MCC with Random, MD_0, and MD_1, respectively. Table 3.5 shows the corresponding results for applying the Java Mutation Analysis output.

Boundary-Based Coverage Criterion	Sorting Machine	Freight Elevator	Triangle Classification	Track Control	Train Control
Random	42/47	58/63	41/41	118/120	306/314
MD_0	47/47	61/63	41/41	120/120	308/314
MD_1	47/47	61/63	41/41	120/120	308/314

Table 3.4: Comparison of Jumble mutation analysis for MCC with all boundary-based coverage criteria.

Boundary-Based Coverage Criterion	Sorting Machine	Freight Elevator	Triangle Classification	Track Control	Train Control
Random	52/71	70/86	121/143	255/312	236/263
MD_0	70/71	77/86	135/143	303/312	247/263
MD_1	71/71	79/86	138/143	311/312	247/263

Table 3.5: Comparison of Java Mutation Analysis for MCC with all boundary-based coverage criteria.

3.6 Related Work

All the general work about model-based testing has been introduced in Section 2.3. In this section, we focus on work related to the presented test generation approach. For instance, partition testing and boundary testing are testing techniques that are focused on the selection of test input values. Several approaches validate predefined boundaries [Kor90, HHF+02] but do not provide means to derive these boundaries from test models. As an exception, Legeard, Peureux, and Utting claim that they developed a method for automated boundary testing from the textual languages Z and B [LPU02]. In contrast to this approach, however, we additionally consider the combination of control-flow-based, data-flow-based, or transition-based coverage criteria with boundary-based coverage criteria. Unfortunately, the corresponding prototype BZ-TT were not available for evaluation and comparison due to operation system incompatibilities of the development machines. This shows the general advantage of implementing ParTeG in Java. There are coverage criteria for boundary values of finite partitions [KLPU04]. In contrast, our approach is not restricted to finite input partitions. A prominent approach for manual partition definition is the classification tree method (CTM) [GG93, DDB+05, LBE+05, ATP+07]. For instance, Hierons et al. [HHS03] specify an automatic test generation approach focused on boundary values. Their approach, however, is also based on CTM, which requires the tester to manually define the equivalence partitions. In contrast, our approach is able to derive the input partitions based on control-flow-based, transition-based, or data-flow-based coverage criteria. This gives our approach the additional advantage of restricting the number of considered partitions according to a selected, e.g., control-flow-based, coverage criterion. Gupta et al. [GMS98] propose a relaxation method to create input parameters for automatically detected feasible paths. However, they start with arbitrary input parameters and adapt them using constraint solvers to find any solution for input parameters. Their work is not focused on finding boundary values. A similar approach has been proposed in [GMS99].

Simon Burton describes in his PhD thesis [Bur01] a different way of including boundary value analysis into automatic test generation from test specifications in Z. For that, he defines heuristics that are used to transform the test model, e.g. by splitting one inequation of the test model into several ones. For instance, he transformed an expression that corresponds to a guard $[x >= y]$ into three guards $[x = y]$, $[x = y + 1]$, and $[x > y + 1]$. The idea is that the satisfaction of each separate guard forces the test generator to select boundary values for the guard. There are similar approaches for model checkers that include the transformation or mutation of the test model. Our approach for integrating boundary value analysis has some advantages over model transformations for boundary value analysis. We briefly discuss two of the most important advantages. Figure 3.39 presents such a tranformation adapted to UML state machines.

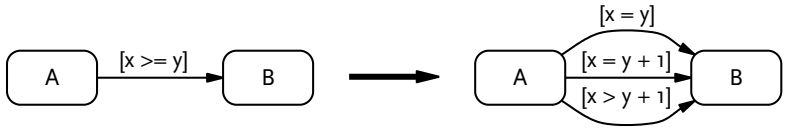


Figure 3.39: Semantic-preserving test model transformation by Burton.

First, we consider boundary value analysis important for the place where input parameters are specified and not where they or derived attributes are used (at the guard). The reason is that the input parameters are used to steer the SUT and there are many situations in which the attributes of the transition guard cannot be directly accessed. It is not guaranteed that the concrete attribute values of the guards can be equal to the constants they are compared to. For the example of the sorting machine in Section 3.2.1, it would be necessary to create boundary values for $[x = y + 2]$ instead of $[x = y + 1]$ because x is twice the value of an integer input parameter. In the ParTeG approach, boundary values for input parameters are derived from the guard conditions to satisfy and all value changes on the path to reach that guard condition. Figure 3.40 shows a similar extension of the test model

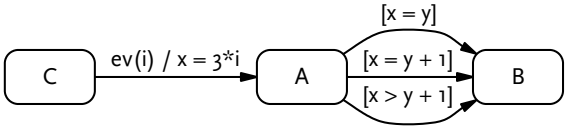


Figure 3.40: Problematic scenario for Burton's approach.

example for Burton of Figure 3.39: The value of x is set to three times the value of the input parameter i. If we assume that x, y, and i are integers

and y is fix, it is easy to see that at most two of the three guards can be satisfied and that $x = y$ and $x = y + 1$ can never be jointly satisfied by any set of test cases for one test model. As a consequence, at most one of the corresponding boundary values will be selected for test generation - the remaining one is a random value for $x > y + 1$. In contrast, our approach creates partitions on the input values and selects values that are closest to the boundary, e.g. if y is divisible by 3, then $i = y/3$ and $i = y/3+1$ are selected, which results in $x = y$ and $x = y + 3$, respectively. The second boundary value can be created by ParTeG but not by the approach of Burton. The approach of Burton can be extended by including static analysis and creating the transition guard $[x = y + 3]$ instead of $[x = y + 1]$. This would solve the presented issue. There may be, however, several paths leading to that guard, e.g., one that sets x three times the value of an input parameter and one that sets it to the value times seven. Figure 3.41 shows a corresponding state machine. This is an example for the violation of the Markov property

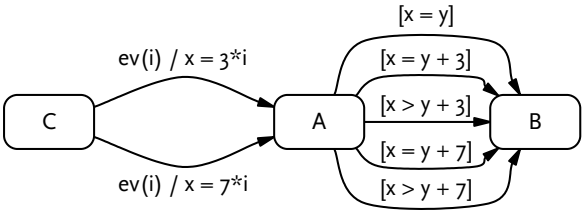


Figure 3.41: Different computations for the same input parameters.

that states that only the current state machine state and the future input are important for future behavior. As a consequence, the traversed transition sequences are important and, thus, the transformation of guard conditions into trace-specific conditions of input parameters is important. In this case, several guards would have to be created. In a scenario where x is increased depending on the number of transition loops, this would result in a potentially

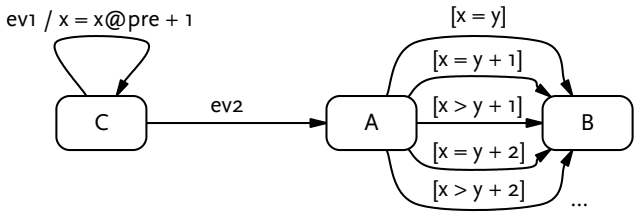


Figure 3.42: Boundary values depending on the number of loop iterations.

infinite number of additional guards to cover all possible paths, which makes this approach infeasible for such scenarios (see Figure 3.42).

Second, we are interested in executing boundary value analysis for each abstract test case. There might be several transition sequences leading to the transformed guard condition. Splitting the guard condition as presented has the effect that only one of these paths has to satisfy a certain boundary value: the satisfaction of any control-flow-based coverage criterion only demands to satisfy a guard without considering the transition sequence that leads to it. So the number of transition sequences per visited guard condition for which boundary value analysis must be executed is limited to 1. Figure 3.43 depicts such a scenario: Satisfying $[x = y]$ for the event *ev1* implies that it is not necessary to satisfy this guard for the event *ev2*. In contrast, the ParTeG approach includes boundary value analysis for each created abstract test case. Transforming the test model graph into a tree could help here. This approach, however, is infeasible for loops. The tool Qtronic [Con] implements Burton's approach to boundary value analysis.

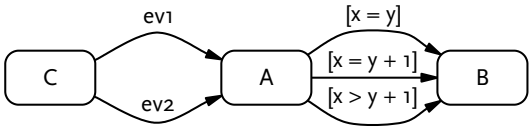


Figure 3.43: Another problematic scenario for the approach of generating boundary values with model transformations.

Finally, we also created two test suites that were generated for the combination of Multiple Condition Coverage and Multi-Dimensional as provided by ParTeG and for Multiple Condition Coverage on a transformed model as proposed by Burton, respectively. We applied mutation analysis to compare the fault detection capabilities of both test suites for the redundant test model (see Section 4.1.1) of the industrial train control with the Java Mutation Analysis output: The test suite for the Burton approach detected 834 mutants. The test suite for the combined coverage criteria as implemented in ParTeG detected 839 mutants. Thus, all theoretic advantages were fortified by a short experiment including an industrial test model.

Cause-effect graphs [Elm73, Mye79] are "a graphical representation of inputs or stimuli (causes) with their associated outputs (effects), which can be used to design test cases" (BS 7925-1. British Computer Society Specialist Interest Group in Software Testing (BCS SIGIST)). They are an alternative way to describe dependencies between input partitions and output partitions. Basically defined for any kind of expressions, they can also be applied for partition descriptions. Cause-effect graphs have been used to derive test cases [Mye79, PTV97]. Schroeder and Korel [SK00] try to reduce the number of black-box tests using input-output analysis. They identify relationships

between inputs and outputs. We can also describe these relations between inputs and outputs with cause-effect graphs [WS08b]. In contrast, however, we also consider the transition paths that connect input and output partitions.

Fox et al. [FHH+01] present backward conditioning as an alternative to conditioned slicing. Backward slicing provides answers to the question of what program parts can possibly lead to reaching a certain part of the program. The same intention applies to the backward search approach of ParTeG: In order to satisfy each test goal of a coverage criterion, we are only interested in finding a path that can lead to covering this test goal.

OCL is a language to express constraints on models [RG98, ZG03, lT06]. Sokenou [Sok06a] translates OCL expressions of the model into Java code to use them as a test oracle. Hamie et al. consider OCL in combination with state machines and classes [HCH+99]. Our algorithm deviates in that we also evaluate OCL constraints and use them to derive test input value partitions. Smartesting claims that their Test Designer for UML [UL06] can evaluate OCL expressions like pre-/postconditions or transition guards for boundary value analysis. They use an operational interpretation of equations in OCL postconditions. As a consequence, their approach is not able to deal with inequations and logical expressions beyond conjunction (e.g. "or", "not") in postconditions. Furthermore, the Smartesting approach is only able to create min-max values, which may be insufficient to satisfy the boundary-based coverage criterion All-Edges [KLPU04] for edges that do not contain minimal or maximal partition values of a certain parameter (see definition in Section 2.1.5). In 2008, we received and evaluated a scientific version of Smartesting (Leirios) Test Designer 3.2.1, which was not able to create boundary values at all. In [BLC05], Briand et al. present an approach to automate the evaluation of OCL expressions. They assume that the transition sequences are already generated and focus on the remaining task of solving the constraints given as OCL expressions. In contrast, our work includes the generation of transition sequences. Since transition sequences can be infeasible due to the attached OCL constraints, the inclusion of OCL expression evaluation in the test generation process is important.

We transform and evaluate OCL expressions to deduce input parameter values. There are many constraint solving techniques that are applied to finding object states that satisfy a set of constraints [RvBW06, Rav08, BSST09, BHvMW09, DEFT09]. Our approach, however, is not focused on finding only any possible solution as generated by most constraint solvers. Instead, we aim at generating boundary values. There are also constraint solvers that include linear optimization [BEN04]. Such solvers can probably be used to support the evaluation of further OCL expressions or non-linear mathematical functions that are out of the scope of OCL (e.g., *sin*, *cos*).

Dijkstra's weakest precondition calculus [Dij76, Gri81, Whi91, Win93, CN00] is often used for reasoning and proofs. We know of no backward interpretation approach that uses Dijkstra's weakest precondition for generating test data in model-based testing.

As presented in Section 2.3.1, there are several approaches to model-based testing with model checking [CSE96, ABM98, GH99, HLSC01]. A major draw-back of model checking is the approach to build up a complete state space. For input values of ordered type (e.g. integer), this often results in a restriction of the value space, e.g., from 0 to 100. In contrast, our approach is based on abstraction of concrete values and, thus, there is no need to restrict data types. We used ParTeG to create boundary values for arbitrary integer values ranging from the minimum possible Java integer or double values to the corresponding maximum.

Abstract interpretation [Cou03, CC04] is a technique to interpret programs at a higher abstraction level than the program itself. The representation at the higher level is formal. The major advantage is to derive information without executing the program. We also apply abstract interpretation in our approach: While searching a path backward to the initial configuration, the test generator keeps track of abstract information that restrict the values of the input parameters. After creating a feasible abstract test case, this abstract information is used to derive concrete input parameter values.

Other models or formalisms than the UML (e.g. extended finite state machines [CK93, BDAR97]) are also used for test generation. However, they do not adequately support object-oriented systems.

Besides the already mentioned Test Designer of Smartesting [Sma], many commercial tools support model-based test generation based on UML state machines. The tool Rhapsody ATG [IBM] is based on UML state machines. It generates and executes test cases with respect to coverage criteria like MC/DC. Boundary value selection is not included. The Conformiq tool Qtronic [Con] supports parallelism and concurrency in UML state machines and supports the boundary value analysis approach of Simon Burton, but also faces the mentioned disadvantages. The algorithm of the tool AETG [Tel] depends on user-defined values and boundaries and is not able to derive them automatically from the test model. In contrast, we derive input partitions automatically. To our knowledge, no commercial tool creates test cases by explicitly deriving input partitions from conditions.

3.7 Conclusion, Discussion, and Future Work

In this section, we present conclusion, discussion, and possible future work for the presented test generation approach.

3.7.1 Conclusion

In this chapter, we presented a new approach to test suite generation, which is focused on combining data-flow-based, control-flow-based, or transition-based coverage criteria with boundary-based coverage criteria. The approach is based on a guided depth-first search algorithm. After providing motivations, we described the test goal management as well as the corresponding test case generation approach for single test goals. We also introduced example test models from literature, academia, and industry, and we used them to evaluate the impact of combined coverage criteria on the test suites generated by our prototype implementation ParTeG. As a major result of the corresponding mutation analysis, the satisfaction of combined coverage criteria results in a higher fault detection capability of the generated test suite than the satisfaction of single coverage criteria. Since the latter is the state of the art, we also pointed out our contribution and showed the advantages of combining coverage criteria. A small comparison with test suites generated by commercial tools further substantiates the advantages of ParTeG over the state of the art in automatic model-based test generation.

3.7.2 Discussion

The above results show the prospective strengths of our approach. We also discussed the related work in the previous section. There are, however, further interesting points to discuss. First of all, the presented algorithm and the corresponding tool are not as mature as other (commercial) products. Some constructs for UML state machines are not supported. We already named them in Section 3.5.1 and accounted for that with the agile development process. The results of applying ParTeG successfully in industrial applications, however, shows the relative strength of this academic tool.

Furthermore, the completeness of the test model and its pre-/postconditions strongly influences the quality of the generated test suite. As any other model-based testing approach, our approach can only transform expressions into input value partitions if the model comprises the corresponding dependencies between definition and use of variables. In general, the detection of feasible paths is undecidable. Correspondingly, our approach cannot guarantee the satisfaction of a selected coverage criterion. However, ParTeG returns

a detailed list containing the unsatisfied test goals, which gives the tester the opportunity to check the test model or manually create missing test cases.

There are several reasonable adaptations and extensions for ParTeG. For instance, initial configurations can be expressed in object diagrams. Since names of test model elements may differ from the used names of SUT elements, it may be necessary to adapt the resulting test suite in a subsequent step or to support the creation of a test adapter. The use of only one state machine can also be discussed – especially for distributed systems or distributed development processes: It would be possible to define several active classes of the SUT and define one state machine for each active class.

3.7.3 Future Work

ParTeG has already been applied in two industrial cooperations. Measured in terms of the number of killed mutants, the results for ParTeG were always better then the results of the selected commercial product.

However, there are some things that could be improved for this tool. We plan to support a broader range of constraints in OCL postconditions including expressions about collections. The selection of concrete input parameters from a set of restricting constraints is not a trivial problem. We think about including an existing constraint solver into ParTeG to be able to derive concrete test input values. Possible solvers might be lp_solve [BEN04], which also includes linear optimization, or SAT4J [Con09], which is already shipped with the ParTeG development platform Eclipse 3.5 (Galileo). The transition sequence search using data-flow information (e.g. for loops) could be improved as well as the support of the missing UML state machine elements like parallel composite states or history states. The set of uncovered test goals could be returned, e.g., in an XML document, to enable other test generators to search for the remaining test cases, automatically. Furthermore, the current approach satisfies boundary-based coverage criteria for each created abstract test case, which results in a sharp increase of the test suite size compared to selecting only one representative input value for each partition. In one industrial project, we realized that some of these values do not need to be selected twice if the corresponding transition paths in the state machine overlap to a certain degree. We think that this can be solved by determining the overlapping def-use-paths. The criteria for identifying these overlapping paths, however, still need to be elaborated. As another possible improvement, model slicing could be used to reduce the effort of test generation. Before slicing away parts of the model, however, the data-flow has to be carefully analysed in order to prevent the removal of essential aspects. Moreover, the presented test generation algorithm combines pre-post and transition-based

algorithms (cf. classification in Section 2.3.2). An open question is whether further combinations with other test generation techniques may bear similar advantages like the presented combination of coverage criteria. Finally, the user interaction can be improved, and other input languages than state machines can be supported.

Chapter 4

Test Model Transformation

In this chapter, we present test model transformations as a means to influence the automatic model-based test generation based on UML state machines. This is the second contribution of this thesis. The transformations are used to change and improve the effect of the applied coverage criteria. Thus, this contribution can be used in conjunction with the first contribution of this thesis, which is the automatic generation of test cases.

This chapter is structured as follows. We present experiences from an industrial cooperation that show the potential impact of test model transformations in Section 4.1. We provide the preliminaries for all subsequent considerations in Section 4.2. In Section 4.3, we present the simulated satisfaction of coverage criteria as the satisfaction of coverage criteria on the original test model by satisfying other, possibly weaker coverage criteria on a transformed test model. We sketch further effects of model transformations in Section 4.4. After that, we present related work in Section 4.5 and conclusion, discussion, and future work in Section 4.6.

4.1 Industrial Cooperation

In this section, we report on a cooperation [Wei09a] with the German supplier of railway signaling solutions Thales. The results of this report are our initial motivation for considering model transformations as a means to improve the quality of automatically generated test suites. The test suites in the company are usually created manually. For our industrial partner, the objective of this cooperation was to investigate the use of model-based testing before adopting it as an additional testing technique. Our task was to automatically generate unit tests based on a given UML state machine. For reasons of nondisclosure, the SUT was not provided. Instead, we manually

created artificial implementations of the test model and conducted mutation analysis on them to measure the fault detection capability of the generated test suite. This mutation analysis showed that the application of existing coverage criteria on the given state machine often does not result in a satisfying fault detection capability of the test suite. As a consequence, we tuned several influencing factors of the model-based test generation process to improve the results of mutation analysis: We transformed the test model, adapted the test goals of the applied coverage criteria, and combined coverage criteria. We also used the cooperation as a test for the presented prototype implementation ParTeG [Weib] under realistic conditions.

Section 4.1.1 contains preliminaries. In Section 4.1.2, we report on the cooperation and all the steps to improve the fault detection capability of the generated test suites. Section 4.1.3 contains conclusion and discussion.

4.1.1 Preliminaries

In this section, we present the preliminaries of the cooperation report: We introduce the notions of artificial SUTs as well as efficiency and redundancy for SUT and test models.

Our aim was to convince our client of the advantages of model-based testing. Thus, we wanted to maximize the generated test suite's fault detection capability for the company's SUT. Since this SUT was not provided, we created and used artificial SUTs, instead: (1) a small SUT with almost no redundancy - we call it the *efficient* SUT - and (2) a cumbersome SUT with a lot of copied source code and redundant function definitions - we call it the *redundant* SUT. The two SUTs are two extreme implementations regarding source code efficiency.

```
if(eventIs('ev1')) {
  if(inState('B') ||
    inState('C') ||
    inState('D')) {
    if(a < b) {
      setState('E');
}}}
```

(a) Efficient condition definition.

```
if(eventIs('ev1')) {
  if(inState('B') && a < b) {
    setState('E'); }
  if(inState('C') && a < b) {
    setState('E'); }
  if(inState('D') && a < b) {
    setState('E'); }}
```

(b) Redundant condition definition.

Figure 4.1: Examples for efficient and redundant SUT source code.

Figure 4.1 shows two small examples. Both SUTs are manually implemented in Java and show the same behavior as the test model. Since the redundant SUT contains more similar code snippets than the efficient SUT

and each snippet can contain a fault, there are more possible places for faults in the redundant SUT and, thus, these faults are assumed to be harder to detect than faults in the efficient SUT. We were aware that the company's SUT can be totally different to our artificial SUTs, and we do not claim that our approach is the best one. However, in the described situation we had no access to the company's SUT and considered the use of artificial SUTs a good solution for finding state-related failures. These artificial SUTs gave us at least an indication for the possible performances of the generated test suites. Since the focus of mutation operators is on syntactic changes, the real SUT is important. Different SUTs can have the same behavior but different structures. Thus, in general, good mutation scores on artificial SUTs are no guarantee for good mutation scores on the company's real SUT.

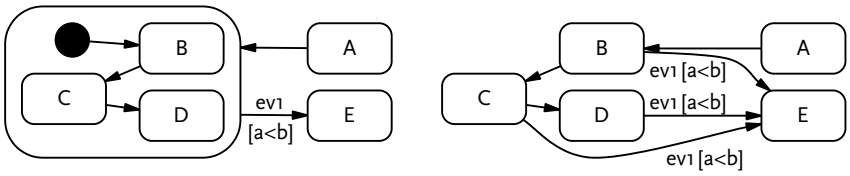


Figure 4.2: Hierarchical and flat state machine.

Likewise, we also call test models *efficient* or *redundant*: For instance, a hierarchical state machine is often more efficient than a flattened one because it needs less model elements to describe the same behavior (see Figure 4.2). The provided test model of the cooperation contains almost no redundancy. Our prototype ParTeG partly supports the insertion of redundancy such as flattening state machines, which allows us to automatically generate redundant test models from efficient ones. During the cooperation, we considered two scenarios most interesting. (1) The test suite is generated from the efficient test model and executed on the efficient SUT. (2) The test suite is generated from the redundant test model and executed on the redundant SUT. Additionally, we also applied the test suite derived from the efficient test model on the redundant SUT, and we applied the test suite derived from the redundant test model on the efficient SUT in Section 4.1.2. This gives us further information about the impact of the used SUT's structure.

4.1.2 Report on the Industrial Cooperation

This section contains our report on the industrial cooperation with the German supplier of railway signaling solutions Thales. A UML state machine was provided to automatically generate unit tests from it. The test model is described in Section 3.2.5.

For reasons of nondisclosure, the company's SUT was not provided. Instead, we had access only to the UML state machine to generate test suites from. We used ParTeG for automatic test generation. The test oracle was contained in the state machine (e.g. as state invariants), and the corresponding oracle code was also generated automatically. After generating a test suite, we measured its fault detection capability with mutation analysis and manually identified all undetectable mutants. Furthermore, we investigated the reasons for detectable but undetected mutants, came up with solutions to detect them, and repeated the test suite generation. We applied the coverage criteria All-Transitions, masking MC/DC, and Multiple Condition Coverage (MCC) to the test model (see Section 2.1.5). In the following, we describe all adaptations of the test generation process and present their impacts on the generated test suites' fault detection capabilities. In Section 3.2.5, we introduced the test model of the industrial cooperation already. The presented figure, however, was focused on showing the use of linear ordered types. Figure 4.3 shows another anonymised part of the provided state machine that contains only model elements for the aspects that were adapted during the cooperation. All following figures depict parts of Figure 4.3.

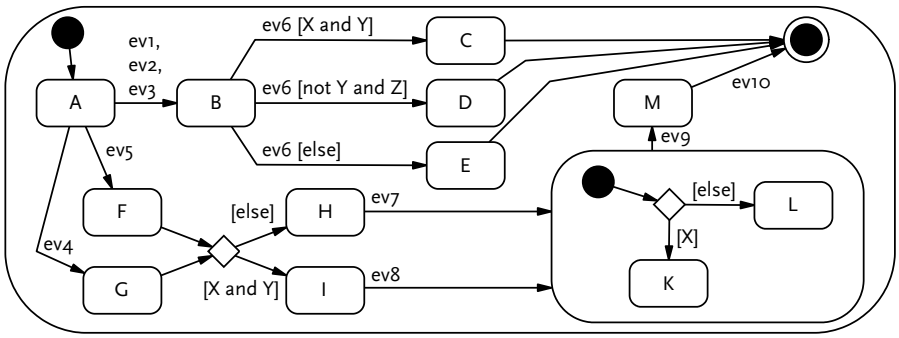


Figure 4.3: Anonymised part of the provided state machine.

Initial Results.

This section contains a description of the cooperation's initial results. Table 4.1 shows the results of mutation analysis for the efficient and the redundant SUT with test suites generated from the efficient and the redundant test model (TM), respectively, without test goal adaptations (cf. Section 3.3.3). The need for test goal adaptation was identified during this cooperation. All following tables contain numbers in parentheses that describe the absolute impact of the described adaptation on the *test suite size* as the number of test cases and on the *mutation score* as the percentage of killed mutants.

Coverage Criterion	Efficient TM/SUT		Redundant TM/SUT	
	Test Suite Size	Mutation Score	Test Suite Size	Mutation Score
All-Transitions	33	185/255	117	610/872
masking MC/DC	46	212/255	197	790/872
MCC	54	217/255	257	810/872

Table 4.1: Results of initial mutation analysis.

Transition Trigger Distribution.

Some transitions of the provided UML state machine are triggered by multiple events (e.g. from state A to state B). None of the applied coverage criteria is focused on events, but the SUT can contain separate source code snippets for each transition trigger. Thus, the satisfaction of any of the used coverage criteria does not necessarily result in the detection of a fault in each corresponding implementation branch. In theory, testing all (even the non-triggering) events for all transitions can be covered with sneak path analysis [REG76, PS96]. Sneak path analysis is focused on finding state transitions that are unintended and, thus, unmodeled. This analysis, however, is costly and we know of no supporting test tool [BSV08].

We considered two solutions: the implementation of a better test generator and the transformation of the test model. For users of a (commercial) model-based testing tool, the improvement of the test generator is almost impossible. Even if the tool vendor is willing to implement the necessary aspects, this change would probably be costly. Transforming the test model, however, seems to be easy: The transformation consists of creating several copies of the corresponding transitions, each of which is triggered by exactly one of the original transition's events. Figure 4.4 shows the original and the transformed test model.

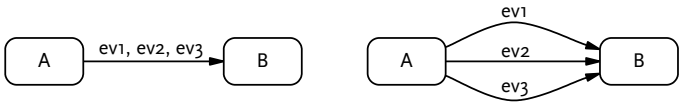


Figure 4.4: Splitting transitions according to their triggering events.

We implemented this solution in ParTeG and repeated the test suite generation. The results of the subsequent mutation analysis are presented in Table 4.2. The numbers in parentheses describe the change caused by this test model transformation. The presented adaptation had a positive impact for the pair of redundant test model and SUT (TM/SUT) and almost no impact on the efficient TM/SUT. Since the redundancy of the company's SUT is unknown, however, we consider this transformation valuable.

Coverage Criterion	Efficient TM/SUT		Redundant TM/SUT	
	Test Suite Size	Mutation Score	Test Suite Size	Mutation Score
All-Transitions	36 (+3)	185/255 (+0)	134 (+17)	627/872 (+17)
masking MC/DC	49 (+3)	212/255 (+0)	214 (+17)	807/872 (+17)
MCC	57 (+3)	217/255 (+0)	274 (+17)	827/872 (+17)

Table 4.2: Mutation analysis after limiting triggers per transition to 1.

Dynamic Test Goal Adaptation.

Dynamic test goal adaptation as a means to deal with incomplete guard descriptions has already been described in Section 3.3.3. Including this step in automatic test generation was motivated during this cooperation. The idea is to adapt generated test goals (see Section 2.4.2) so that there is only one possible target state for each test goal. As a consequence, the oracle of each test case can predict the expected target state for each event trigger, and the corresponding test case is able to detect more mutants. We implemented this dynamic test goal adaptation in ParTeG and generated the test suites again. Table 4.3 shows the results of the subsequent mutation analysis.

Coverage Criterion	Efficient TM/SUT		Redundant TM/SUT	
	Test Suite Size	Mutation Score	Test Suite Size	Mutation Score
All-Transitions	34 (-2)	189/255 (+4)	131 (-3)	634/872 (+7)
masking MC/DC	50 (+1)	226/255 (+14)	215 (+1)	820/872 (+13)
MCC	57 (+0)	229/255 (+12)	274 (+0)	842/872 (+15)

Table 4.3: Mutation analysis with additional dynamic test goal adaptation.

Choice Pseudostate Splitting.

The state machine in Figure 4.3 contains completion transitions, which are not triggered explicitly. A problem occurs if a vertex has several incoming transitions and several outgoing completion transitions with guard conditions. The satisfaction of a control-flow-based coverage criterion such as MC/DC or MCC is focused on value assignments for guard conditions. It is not influenced by traversed transition paths, and there may be different paths for each necessary guard value assignment. Figure 4.5(a) shows a corresponding part of the state machine. The outgoing transitions of the choice pseudostate are not directly triggered by events. Each control-flow-based coverage criterion is already satisfied, e.g. if the guard *[X and Y]* is satisfied on a path including the state *F* and if *[else]* is satisfied on a path including the state *G* (see Figure 4.5(a)). Consequently, *[X and Y]* may not be satisfied for paths including state *G* and *[else]* may not be satisfied for paths including state *F*. All corresponding mutants will remain unkilled.

(a) Original choice pseudostate. (b) Splitted choice pseudostate.

Figure 4.5: Split the choice pseudostate.

The application of transition-based coverage criteria [UL06, page 118] like All-Transition-Pairs instead of MCC is no satisfying solution because All-Transition-Pairs only requires to test one of the three value assignments to satisfy $[else]$. We know of no coverage criterion that is focused on transition sequences and on the value assignment of guards' atomic conditions at the same time. Our solution comprises a test model transformation – each choice pseudostate is split up according to its incoming transitions: Each new choice pseudostate is attached to only one incoming transition but to all outgoing transitions of the original choice pseudostate (see Figure 4.5(b)). As a consequence of this transformation, the satisfaction of control-flow-based coverage criteria implies that each guard condition on outgoing transitions of choice pseudostates has to be covered for each source state of the choice pseudostate (F and G in the example). We implemented this test model transformation and rerun the test generation. Table 4.4 shows the results of the subsequent mutation analysis.

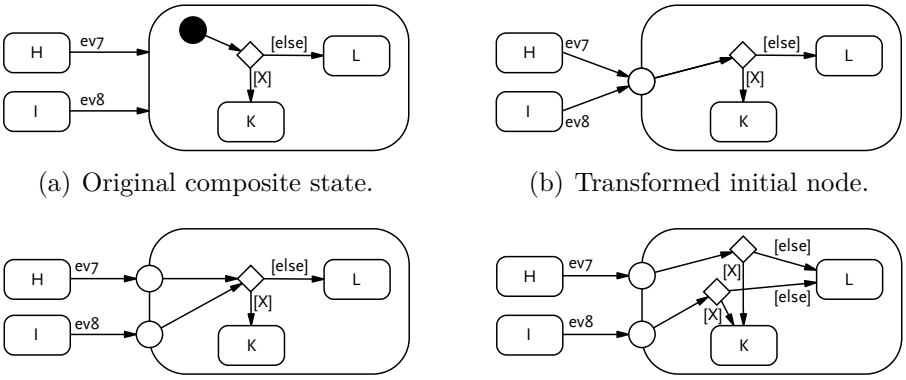
Coverage Criterion	Efficient TM/SUT		Redundant TM/SUT	
	Test Suite Size	Mutation Score	Test Suite Size	Mutation Score
All-Transitions	47 (+13)	209/255 (+20)	144 (+13)	660/872 (+26)
masking MC/DC	61 (+11)	239/255 (+13)	226 (+11)	840/872 (+20)
MCC	68 (+11)	241/255 (+12)	285 (+11)	860/872 (+18)

Table 4.4: Results of mutation analysis with splitted choice pseudostates.

Composite States Transformation.

Several choice pseudostates of the test model are contained in composite states and are directly connected to the composite state's initial state (see Figure 4.6(a)). There is only one incoming transition for such choice pseudostates and all compound transitions [Obj07, page 568] from outside the composite state are united in the initial state. Thus, the previous test model transformation has no effect. For this case, we have to split incoming compound transitions instead of splitting incoming transitions: We transform

the initial state into an entry point and connect it to all incoming transitions of the composite state (see Figure 4.6(b)). After that, this entry point is duplicated so that there is only one incoming transition for each entry point (see Figure 4.6(c)). As a consequence, the choice pseudostate has now several incoming transitions and the previously presented transformation about choice pseudostate splitting produces several choice pseudostates (see Figure 4.6(d)). As a result of this transformation, the guard conditions of choice pseudostates are also tested across boundaries of composite states for each start state of compound transitions. We implemented this model transformation in ParTeG and regenerated the test suite. Table 4.5 shows the results of the subsequent mutation analysis.



(a) Original composite state. (b) Transformed initial node.

(c) Duplicated entry point. (d) Splitted choice pseudostate.

Figure 4.6: Transform composite states.

Coverage Criterion	Efficient TM/SUT		Redundant TM/SUT	
	Test Suite Size	Mutation Score	Test Suite Size	Mutation Score
All-Transitions	51 (+4)	213/255 (+4)	148 (+4)	661/872 (+1)
masking MC/DC	65 (+4)	242/255 (+3)	229 (+3)	843/872 (+3)
MCC	72 (+4)	244/255 (+3)	288 (+3)	863/872 (+3)

Table 4.5: Mutation analysis with additionally transformed composite states.

Coverage Criteria Combination.

Mutation analysis shows that the generated test suites do not kill all detectable mutants. The remaining unkilled mutants are caused by small changes of boundary values in conditions with (in-)equations and variables of ordered types. For instance, if the correct SUT contained a condition $[x > 5]$, then the unkilled mutants could contain $[x > 4]$ or $[x \geq 5]$, instead.

To detect such mutants, boundary value analysis has to be included in the test suite generation. As described in Chapter 3, ParTeG includes boundary value analysis in automatic test generation. Boundary coverage criteria like Multi-Dimensional (MD) [KLPU04] are combined with structural, e.g., control-flow-based, coverage criteria: For each abstract test case generated to satisfy one of the three investigated coverage criteria, the concrete input values are selected according to the boundary coverage criterion MD. These combined coverage criteria are denoted with a preceding MD. We regenerated the test suites for all three considered coverage criteria combinations. The subsequent mutation analysis showed that the test suites that satisfy MDMCC (combination of MD and MCC) on the efficient and redundant test model killed all mutants of the corresponding SUT (see Table 4.6).

Combined	Efficient TM/SUT		Redundant TM/SUT	
Coverage Criterion	Test Suite Size	Mutation Score	Test Suite Size	Mutation Score
MDAll-Transitions	102 (+51)	222/255 (+9)	296 (+148)	672/872 (+11)
masking MDMC/DC	128 (+63)	251/255 (+9)	456 (+227)	852/872 (+9)
MDMCC	140 (+68)	255/255 (+11)	572 (+284)	872/872 (+9)

Table 4.6: Mutation analysis with additionally combined coverage criteria.

Impact of the SUT.

In the previous sections, we presented the results of running test suites derived from the efficient state machine on the efficient SUT and of running test suites derived from the redundant state machine on the redundant SUT. The results for both scenarios are comparable. In both cases, we assumed that the SUT and the test model have a similar degree of redundancy. However, the implementation details of the company's SUT are unknown. Thus, we also investigated the impact of the SUT redundancy on the fault detection capability of the test suite. For reasons of conciseness, we present the mutation analysis results just once for all presented adaptations.

Combined	Efficient TM/Redundant SUT		Redundant TM/Efficient SUT	
Coverage Criterion	Test Suite Size	Mutation Score	Test Suite Size	Mutation Score
MDAll-Transitions	102	240/872	296	226/255
masking MDMC/DC	128	285/872	456	251/255
MDMCC	140	289/872	572	255/255

Table 4.7: Combinations of efficient and redundant test models and SUT.

Table 4.7 shows the results of the mutation analysis for the combination of efficient test model and redundant SUT as well as the results for the combination of redundant test model and efficient SUT: If the test model

is efficient but the SUT is redundant, then the fault detection capability of the generated test suite is low. None of the used coverage criteria was able to kill one third of all detectable mutants! The test suites derived from the redundant state machine that were applied to the efficient SUT are also unsatisfactory: The test suite generated from the efficient state machine using MDMCC already killed all detectable mutants and no improvement of the fault detection capability is possible. Instead, the test suite size increased: the number of test cases is more than four times higher than for the satisfaction of MDMCC on the efficient test model.

For the time of the industrial cooperation and also during presented test generation process, the company's SUT was unknown to us. After delivering the test suites, however, we got feed-back about the number of failed test cases (see Table 4.8) and a short analysis of the reasons. Following that analysis, the test suite generated from the efficient test model detected one fault that caused four test cases to fail and the test suite generated from the redundant test model detected two faults that caused 32 test cases to fail. In random testing, such results are expected – big test suites are likely to detect more faults than small test suites. Here, however, the test suites are generated from the same behavioral information using the same coverage criteria. Thus, the test model redundancy is important: The test suite for the redundant test model contains more test cases that detect faults, but it also detects more faults. This substantiates the importance of the described adaptations and transformations in realistic scenarios as well as the importance of considering different levels of test model redundancy.

Combined	Efficient Test Model		Redundant Test Model	
Coverage Criterion	Failed Tests	Detected Faults	Failed Tests	Detected Faults
MDMCC	4/140	1	32/572	2

Table 4.8: Failed tests and detected faults on the company's SUT.

4.1.3 Conclusion and Discussion

In this section, we conclude and discuss the presented report on the industrial cooperation. Related work is presented in Section 4.5.

Conclusion.

We reported on an industrial cooperation with the German supplier of railway signaling solutions Thales. We described the initial situation and the occurred challenges of model-based black-box testing. We provided solutions

for all of these challenges. The application of the solutions resulted in the detection of all detectable mutants. We measured the fault detection capabilities of the generated test suites with mutation analysis on artificial SUTs and also got feed-back about the execution of the final test suites on the company's SUT. The goal of the presented cooperation was to investigate model-based testing before adopting it as a new testing technique. We were able to convince our client of the benefits of model-based testing: The quality of the generated test suites was comparable to manually created test suites in terms of detected faults and covered source code elements. However, model-based testing requires considerably lower maintenance effort than manual test creation. Moreover, the test execution effort can be considerably influenced by the selection of a weaker or stronger coverage criterion.

Furthermore, we got interesting feed-back about the importance of integrating boundary value analysis in test generation. We were told that once a whole train communication was shut down because one boundary value was not implemented correctly: The distance to a certain point was exactly zero, a highly unlikely event. Boundary value analysis probably would have revealed this fault.

The contribution of this section is the presented procedure for model-based testing in an industrial scenario with a hidden SUT. The main benefit of this procedure is the increased fault detection capability for automatically generated test suites. Novel elements of this report are the application of artificial SUTs, the purposeful transformation of test models, the adaptation of test goals, and the combination of coverage criteria in an industrial application.

Discussion.

Most of the presented results can also be reached by improving the used test generator. Usually, however, the tester is just a user and has no influence on the used test generator. The presented test model transformations are the only way to increase the generated test suite's fault detection capability without changing the test generator.

One result is that test suites generated from redundant test models have a higher fault detection capability than test suites from efficient test models. The result of this report is not a recommendation to create redundant test models as a new kind of "modeling paradigm". Such models would be hard to maintain. Instead, we recommend to create and maintain efficient test models, to transform copies of them automatically, and to use these transformed and redundant copies for automatic model-based test generation. Test models have to be adapted if the SUT is changed. Since test models are in general

easier to understand than source code and test suites are generated automatically, we consider the corresponding effort lower than the effort for adapting manually created test suites.

We were skeptical about using artificial SUTs. There is no guarantee that a good mutation score for artificial SUTs implies a good mutation score for the company's SUT. Reasons can be that some information is missing in the test model and that the artificial implementations only contain state-based information that is also contained in the test model. Furthermore, the presented influencing factors were only investigated for artificial SUTs, and it is an open issue whether their application also leads to the detection of all mutants in the company's SUT. In our case, however, this technique was quite successful. As shown in the section about the impact of the SUT, the artificial SUTs were helpful to improve the fault detection capability of test suites generated from redundant test models. Furthermore, the presented report shows that the adaptations also have a positive impact on the fault detection capability of the generated test suites for the company's SUT.

We presented the incremental effect of the adaptations, i.e., the results of each adaptation already included the results of all previous adaptations. This is especially obvious for the choice pseudostate splitting and the composite state transformation, for which the second one is only intended to improve the effect of the first one. An isolated investigation of all adaptations would be interesting. This report just presents our experiences during the cooperation.

Furthermore, the combination of coverage criteria resulted in doubling the test suite size. Case studies [ABLN06] estimate an exponential growth of the test suite size with respect to the number of already killed mutants. These case studies also point out the importance of killing the last 10%-20% of all detectable mutants.

Moreover, we used a technique to dynamically adapt test goals in Section 4.1.2. As presented in [FW08a], the adaptation of test goals is an important research topic. It would be interesting to also categorize possible test goal adaptations and identify their impact on the test generation process. Chapter 6 is focused on test goal prioritizations.

Finally, the presented adaptations resulted from undetected mutants of one concrete test model, and it is interesting whether they also have an impact on other scenarios. We generated test suites from all the test models presented in Section 3.2 and measured the impact of the adaptations on the corresponding mutation scores presented in Section 3.5.2. Additionally, we applied the adaptation of splitting inequations in guards as presented in [Wei08]. The adaptations can only have an additional impact to the test suites that do not already kill all detectable mutants. We sketch the results for the satisfaction of the strongest control-flow-based coverage criterion Mul-

tiple Condition Coverage: For the track control, the last remaining mutant is killed for Java Mutation Analysis. Applying the adaptations also results in killing five of the seven remaining mutants for Java Mutation Analysis and both missing mutants for Jumble. For the triangle classification, all five remaining mutants are killed for Java Mutation Analysis. Table 4.9 shows the corresponding mutation scores before and after applying the mentioned adaptations. For all examples except the freight elevator for Java Mutation Analysis, the application of all mentioned adaptations results in killing all detectable mutants.

Mutation Analysis	Sorting Machine	Track Control	Freight Elevator	Triangle Classification
Java Mutation Analysis	before: 71/71 after: 71/71	before: 311/312 after: 312/312	before: 79/86 after: 84/86	before: 138/143 after: 143/143
Jumble	before: 47/47 after: 47/47	before: 120/120 after: 120/120	before: 61/63 after: 63/63	before: 41/41 after: 41/41

Table 4.9: Impact of test model adaptation on mutation analysis for the four remaining example models.

4.2 Preliminaries

In this section, we present several test model transformations to support automatic model-based test generation. Motivation is the improved fault detection capability of test suites generated from transformed test models as shown in the previous section. As we have seen, model transformations can be used to increase even the effect of the strongest control-flow-based coverage criterion. This leads us to the question whether model transformations can also be used to improve the effect of weaker coverage criteria so that they correspond to stronger ones.

Model transformations [Küs06] are used to change models. They can be used to transform almost every model into every other model. The two models can be instances of different meta models [Fav] or the same one.

A state machine is a behavioral abstraction of the SUT. Two structurally different state machines can describe the same behavior. A state machine can be transformed and used afterwards for test generation if the test model transformation preserves the behavior described by the model. The test goals generated from the test model and the coverage criterion depend on the structure of the test model (see Section 2.4.2). Test model transformations for coverage criteria can be used to accentuate certain parts of the test model: The idea is that coverage criteria on transformed test models produce

more test goals that result in more test cases with a higher fault detection capability.

Coverage criteria are focused on covering model elements and not on detecting failures – the aspect of propagating the error to the outside is left out (see Section 2.1.1). For the sole satisfaction of such coverage criteria, it is irrelevant whether the corresponding test cases detect a failure, i.e., whether the used model transformation preserves the semantics of the state machine. In our context, however, the state machines should be used to generate test suites and, thus, failure detection and semantics preservation are important. Before thinking about how to preserve the semantics, we have to define the semantics. This is an issue: First, the UML specification contains many semantic variation points and, thus, the semantics of state machines is not defined by the OMG specification. Second, there are many different definitions of state machine semantics [Har87, HN96, EW00]. Finally, the proposed test model transformations [Wei09b] are intended to support test generation tools with a limited ability to satisfy coverage criteria. Developers of such tools often have their own understandings of state machine semantics, and the tools often do not support the full state machine specification. For these reasons, it is impossible to define just one kind of state machine semantics preservation and claim the general applicability of the corresponding model transformations to all existing tools. Instead, we restrict ourselves to an "intuitive semantics" of state machines and claim that the presented transformations can be adapted to existing tool semantics. We use Smartesting LTD [Sma] as an example: This tool supports only test generation from deterministic, untimed, and flat state machines and is restricted to the satisfaction of All-Transitions on the used test model. According to that, we will restrict the presented model transformations to such UML state machines and, amongst others, present model transformations to enhance the effect of All-Transitions.

In the following, we present definitions of model transformations and abstract representations of test suites on state machines to compare coverage criteria applied to different test models. Furthermore, we present semantic-preserving test model transformation patterns that are used to compose the transformations in the further sections of this chapter.

4.2.1 Definitions

Here, we present several definitions that are used to clarify the concepts of coverage criteria, test goals, and satisfying test cases. For that, we extend the definitions presented in Section 2.4.2. For reasons of clarification, we show all the used symbols again. All the definitions from step patterns to

Test Suites:	TS
State Machines:	SM
State Machine / Model Transformations:	MT
Step Patterns:	SP
Step Coverage:	$SPCov$
Trace Patterns:	TP
Trace Coverage:	$TPCov$
Atomic Test Goals:	ATG
Complex Test Goals:	CTG
Test Goals:	TG
Coverage Criteria:	CC
Coverage Criteria Satisfaction:	$\models$
Abstract State Machine Representation:	$asmr$

Figure 4.7: Names and symbols for test model transformations.

coverage criteria satisfaction can be looked up in Section 2.4.2 on page 41 ff. Here, we only define the additionally presented names.

Test suites TS are sequences of parameterized events that trigger a certain behavior of the test model and (oracle) information corresponding to the expected behavior. The corresponding behavior on a state machine is expressed using the function *asmr*. Since the considered test models are untimed, the model instantly reacts to the events and there are no event conflicts. SM denotes the set of all UML state machines [Obj07, page 519]. All model transformations of state machines are described with MT, which is a set of functions $\{mt \mid mt : sm1 \rightarrow sm2; sm1 \in SM; sm2 \in SM\}$.

The abstract state machine representation $asmr : ts \times sm \rightarrow TPS$ (with $ts \in TS$, $sm \in SM$, and $TPS \subseteq TP$) is a function that represents the behavior of test suites ts on the level of state machines sm as a set of trace patterns TPS.

We already defined coverage criteria satisfaction in terms of covered trace patterns in Section 2.4.2. Here, we are interested in test suites whose corresponding trace patterns at state machine level should be covered. Like for the previous definition of coverage criteria satisfaction in Section 2.4.2, for complex coverage criteria like MC/DC, it is adequate to satisfy only a sufficient set of included atomic test goals. For that, we present an extended definition of coverage criteria satisfaction: A coverage criterion $cc \in CC$ on a state machine $sm \in SM$ is satisfied by a test suite $ts \in TS$ iff all test goals $tg \in cc(sm)$ are covered by the traces of $asmr(ts, sm)$, i.e., at least one trace pattern tp_g of each atomic test goal tg is covered by the traces of $asmr(ts, sm)$:
$asmr(ts, sm) \models cc(sm)$ iff $\forall_{tg \in cc(sm)} \exists_{tp_t \in asmr(ts,sm), tp_g \in tg} : (tp_t, tp_g) \in TPCov$.

4.2.2 Basic Transformation Patterns

In this section, we present basic transformation patterns that are used to assemble test model transformations of UML state machines. Due to the afore mentioned restrictions of semantics specification, we restrict ourselves to show that the provided transformation patterns preserve the intuitive semantics. As a consequence, all transformations that are composed of these transformation patterns are also semantic-preserving. All subsequently shown pseudocodes use the UML notation.

Add Variables.

The transformation *Add Variables* consists of adding the definition of a new variable to a transition's effect. The variable does not influence the control flow of the state machine. Thus, this transformation preserves the state machine's semantics. A test generation postprocessor should be used that removes these new and artificial variables so that they do not occur in the generated test suite. Figure 4.8 shows the pseudocode for the corresponding transformation.

```
addVariable(Effect ef, Variable var, String value) {
  add "var = value" to ef;
}
```

Figure 4.8: Transformation that inserts a new variable into a transition effect.

Insert Node in Transition.

```
insertNodeInTransition(Transition t) {
  Vertex v = new ChoidePseudoState();
  Transition t_new = new Transition();
  Vertex target = t.target;
  t.target = v; v.incoming = {t}; target.incoming.remove(t);
  t_new.source = v; v.outgoing = {t_new};
  t_new.target = target; target.incoming.add(t_new);
}
```

Figure 4.9: Transformation that inserts a choice pseudostate into a transition.

The transformation *Insert Node in Transition* consists of inserting a choice pseudostate v into a transition t. Figure 4.9 shows the corresponding pseudocode. The new node v has only one outgoing transition t_new, which has no guard, no effect, and leads to the former target state of t. Since t_new

is connected to t via a pseudostate, both are part of the same compound transition. Consequently, they are executed in one step, and this transformation preserves the semantics of the state machine.

Move Effect.

Each transition t of a state machine can contain an effect. The transformation *Move Effect* consists of inserting a choice pseudostate v into t (*Insert Node in Transition*) and moving the effect *ef* of t to the new outgoing transition *t_new* of v (see Figure 4.9). Both transitions are part of the same compound transition, *t_new* has no guard, and after executing this compound transition, *ef* will be executed. Thus, this transformation preserves the semantics of the state machine. The pseudocode that describes this transformation is shown in Figure 4.10.

```
moveEffect(Transition t) {
  insertNodeInTransition(t);
  Transition t_new = t.target.outgoing->get(0);
  t_new.effect = t.effect;
  t.effect = null;
}
```

Figure 4.10: Transformation that moves the effect of a transition.

Copy Vertices.

States of a state machine do not necessarily correspond bijectively to variable value assignments of the SUT. Thus, there are often states that describe a set of value assignments of the SUT – this is part of the abstraction. But there may also be several states of the state machine that describe the same set of value assignments of the SUT. The aim of the atomic transformation *Copy Vertices* is to create copies of a vertex v in the state machine that reference the same set of SUT value assignments as v.

This transformation creates copies of a vertex depending on the number of its incoming transitions. For each copy c of the original vertex v holds: All properties (e.g. internal transitions, state invariants, etc.) and all outgoing transitions are copied. The source vertices of the copied transitions are set to c – the target states are equal to the target state of the original transition. Thus, each input event causes the state machine to reach the same target vertex. Consequently, the behavior is unchanged and this transformation preserves the semantics. Figure 4.11 shows an algorithm for copying vertices.

```
copyVertex(SM sm, Vertex v) {
  Vertex c = new Vertex();
  set all properties of c to the ones of v;
  for all outgoing transitions t of v {
    create a copy of t and set its source vertex to c;
} }
```

Figure 4.11: Transformation that creates copies of state machine vertices.

Exchange Transition Targets for Vertex Copies.

In addition to copying vertices as presented above, the transitions that point to the original vertex v can also point to any copy c of v or vice versa. Since v and c imply the same behavior, the effect of traversing any transition is the same no matter whether the transition's target vertex is v or c. Thus, exchanging a transition's target vertex with a copied vertex preserves the semantics of the state machine. Figure 4.12 shows the transformation.

```
exchangeTransitionTarget(Transition t, Vertex new_Target) {
  Vertex old_Target = t.target;
  if (new_Target is copy of old_Target or vice versa) {
    t.target = new_Target;
} }
```

Figure 4.12: Transformation to exchange a transition's target vertex.

Create Self-Transitions for Unmodeled Behavior.

Events in UML state machines can trigger transitions. This depends on the current configuration, i.e., the set of concurrently active states. There may be states for which no outgoing transition is triggered by a certain event, e.g., because the corresponding guard is not satisfied or there is simply no corresponding transition. The proposed model transformation pattern *Create Self-Transitions for Unmodeled Behavior* consists of creating self-transitions of states for combinations of events and guard value assignments that activate no existing transition. The effect of the transformation is that all combinations of event and guard value assignment that initially do not result in traversing transitions now do at least once. To prevent that the new transitions trigger entry or exit actions of states, their type is set to *internal* [Obj07, page 574]. The semantics of the state machine is unchanged and, thus, the presented model transformation is semantic-preserving. Figure 4.13 shows the pseudocode for the corresponding algorithm.

```
addSelfTransitions(SM sm, Vertex v) {
  for all pairs of event e and guard conditions g that do not result
    in transition traversal from v {
    Transition t = new Transition();
    t.guard = g; t.event = e; t.source = v; t.target = v;
    if (e = null) { // completion transition
      create new variable var;
      add "var = true" to t.guard;
      add "var = false" to t.effect;
      add "var = true" to the effect of all incoming transitions of v
        that are not self-transitions;
} } }
```

Figure 4.13: Create self-transitions for unmodeled behavior.

There may be an issue: If such self-transitions are completion transitions [Obj07, page 568], this transformation event has a higher priority than all other events [Obj07, 569] and can lead to an unlimited loop execution. To prevent this, a new variable *var* is created for each newly inserted completion transition. The value of *var* is set to false in the effect of the new self-transitions and to true in the effect of all incoming transitions of the self-transition's source state that are not self-transitions. Additionally, the guards of all newly defined completion self-transitions include *var* so that the self-transitions can be traversed just once.

Split Transition.

Each transition of a state machine can contain a guard condition. The model transformation *Split Transition* consists of splitting a transition t according to its guard g: First, t is removed and two copies $t1, t2$ of t are created that have the same source vertex, target vertex, event, and effect like t. Then, the transition guard g is split up into two new complete and mutually exclusive expressions $g1$ and $g2$ (with $g1 \vee g2 = g$ and $g1 \wedge g2 = false$). The new expressions are assigned as guards to $t1$ and $t2$, respectively. Since for each value assignment of t's guard, exactly one of $t1$ and $t2$ has the same effect as t, this transformation preserves the state machine semantics.

Transitions can also be split into more than two transitions. To create new guards from the guard of a transition to split, we use the conjunctions that correspond to the rows in the truth value table of the guard to set the new guards g_1 and g_2. For instance, the guard $[a \vee b]$ can be used to create the guards $[a \wedge b]$, $[a \wedge (\neg b)]$, $[(\neg a) \wedge b]$, and $[(\neg a) \wedge (\neg b)]$. This set also contains guards that are satisfied if the original guard is violated. Thus, the

```
splitTransition(SM sm, Transition t) {
  for all conjunctions c of the truth value table of t.guard {
    Transition t_new = new Transition();
    t_new.source = t.source;
    t_new.event = t.event;
    t_new.guard = c;
    if(c implies t.guard) {
      t_new.target = t.target;
      t_new.effect = t.effect;
    } else {
      t_new.target = t.source;
} } }
```

Figure 4.14: Split transitions according to their guards.

effect and the target of any newly created transition t_1 are only set to the values of the original t iff the satisfaction of $t_1.guard$ implies the satisfaction of $t.guard$ – otherwise, they are created as self-transitions without effect (see pattern *Create Self-Transtions for Unmodeled Behavior*). Figure 4.14 shows the pseudocode for a model transformation that creates a new transition for each row of the original guard's truth value table.

Note that it is necessary to consider also the guards of other outgoing transitions. Basically, new transition guards must not imply the satisfaction of an already existing transition guard, if both transitions are triggered by the same event. Since these problems can be easily solved by just excluding the mentioned event-guard-pairs, we abstract from such problematic scenarios.

4.3 Simulated Coverage Criteria Satisfaction

The industrial report of Section 4.1 already revealed that model transformations are a means to increase the fault detection capability of a test suite that is generated based on a test model. This section contains the definition of simulated coverage criteria satisfaction [Wei10]. The simulated satisfaction is an important means to satisfy coverage criteria whose satisfaction is not directly supported by the used test generator. The basic idea is to transform the test model in such a way that any test suite that satisfies a supported coverage criterion on the transformed test model also satisfies the desired but unsupported coverage criterion on the original test model. Simulated satisfaction is used to exchange coverage criteria in a way that the satisfaction of weak coverage criteria can have the same effect as the satisfaction of strong ones. Furthermore, the simulated satisfaction relations can be used to relate coverage criteria that are not connected by subsumption.

4.3.1 Introduction

Throughout the thesis, we considered the whole expressiveness of UML state machines. In this section, we concentrate on model-based testing with deterministic untimed flat UML state machines. All presented transformations are focused on them. The transformations necessary to deal with timed or hierarchical state machines are similar but contain many special cases that we do not explain in detail here. It is impossible to guarantee coverage for non-deterministic models in general. Such restrictions of our presentation do not imply restrictions of our general approach. We will also sketch some of the mentioned special cases and the corresponding solutions.

Model-based coverage criteria can be compared by subsumption if they are applied to the same model. The subsuming coverage criterion is considered stronger than the correspondingly subsumed coverage criterion. For instance, All-Transitions [UL06, page 117] is considered the minimum coverage criterion to satisfy. There are many commercial test generators that are only able to satisfy rather weak coverage criteria [BSV08]. As an example, Smartesting LTD [Sma] is only able to satisfy All-Transitions. If users of such tools want stronger coverage criteria to be satisfied, their only choice is often to buy a new test generator. Here, we present model transformations such that the satisfaction of a weak coverage criterion on the transformed test model implies the satisfaction of a stronger one on the original test model [Wei09b]. This is an important support for (commercial) model-based test generators that are only able to satisfy a limited set of coverage criteria.

This section is structured as follows. Section 4.3.2 contains the preliminaries. In Section 4.3.3, we present test model transformations that are used to simulate the satisfaction of coverage criteria. The transformations are assembled from the transformation patterns presented in Section 4.2.2. We present a resulting simulated satisfaction graph in Section 4.3.4.

4.3.2 Preliminaries

Here, we present a small example of a parallel coffee dispenser. We use it to illustrate the results of the presented model transformations. Furthermore, we formally define the notion of simulated coverage criteria satisfaction.

Example.

Figure 4.15 depicts a UML state machine showing the behavior of a coffee dispenser. The details are explained in the following. The left region of the state machine allows a user of the coffee dispenser to select a beverage (*sel_bev*).

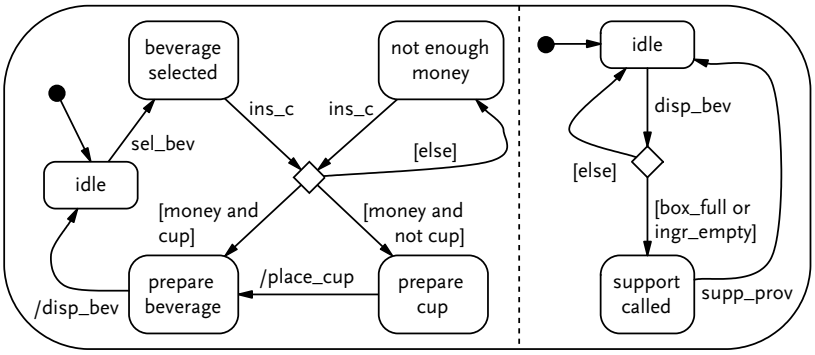


Figure 4.15: UML state machine describing the behavior of a coffee dispenser.

Afterwards, he has to insert a certain amount of coins (*ins_c*) until the beverage is paid (*money = true*). Furthermore, the user can decide to place his own cup in the dispenser (*cup = true*) or to use the standard plastic cup (*cup = false*). If the user does not use his own cup, then the dispenser places one on the collection tray (*place_cup*). Afterwards, the beverage is dispensed into the cup. The right region of the coffee dispenser describes the support of the coffee dispenser: Each time a beverage is dispensed (*disp_bev*), the coffee dispenser checks whether the cash box is close to being full (*box_full = true*) or the boxes with the ingredients for the beverages are almost empty (*ingr_empty = true*). In this case, the support is called. The coffee dispenser remains in this state until the support is provided (*supp_prov*).

Simulated Satisfaction of Coverage Criteria.

In this section, we propose to compare the satisfaction of coverage criteria on source and target model of model transformations. All test model transformations are composed of atomic, semantic-preserving transformations that were presented in Section 4.2.2. We consider only semantic-preserving model transformations. As presented in Section 4.2, the semantics of the concrete state machine depends on the tool to support. Thus, we restrict ourselves to an *intuitive semantics*. It can be described as follows. A certain active state configuration describes the current object state. As the reaction to a trigger, a transition can be traversed if its guard condition is satisfied. The effect of a transition can, e.g. change value assignments or trigger further transitions. Besides this basic description, there are several undefined details concerning simultaneous events or composite states. As mentioned above, the UML specification [Obj07] refers to such details as semantic variation points. Consequently, vendors of existing test generation tools are free in defining details of state machine semantics. Since our approach is generally applicable to all

test generation tools, we do not restrict it to a certain vendor-dependent interpretation but focus our approach on the described intuitive semantics. We claim, however, that there are only a few modifications necessary to adapt the model transformations to the semantics of a certain model-based test generation tool. For this reason and for reasons of simplicity, all subsequent explanations are just focused on deterministic flat untimed UML state machines. We will discuss the corresponding restrictions and how to resolve them in Section 4.6. Here, we present some basic definitions. Many language semantics are defined by the set of the accepted words. UML state machines, however, are input enabled, and accept all possible input sequences. Thus, the semantics is defined as the observed behavior or the assigned attribute values, respectively.

Definition 31 (Semantics of UML State Machines) *The* semantics *of a UML state machine is its reaction to all sequences of input stimuli, i.e., the observed behavior or the attribute values and state invariants of all reached states.*

Definition 32 (Semantic-Preserving Model Transformation) *A model transformation of a UML state machine is* semantic-preserving *iff the semantics of the source model is equal to the semantics of the corresponding target model.*

Definition 33 (Simulated Satisfaction) *We consider semantic-preserving model transformations* $mt \in MT$ *of state machines* $sm \in SM$*. The coverage criterion* cc_1 *is applied to* mt(sm) *and the coverage criterion* cc_2 *is applied to* sm*. If the satisfaction of* cc_1 *on* mt(sm) *implies the satisfaction of* cc_2 *on* sm *for all test suites, then the satisfaction of* cc_1 *is said to* simulate the satisfaction *of* cc_2 *with the model transformation* mt*. If there exists such a model transformation, we say that* cc_1 simulates cc_2*:* $cc_1 \Rrightarrow cc_2$*.*

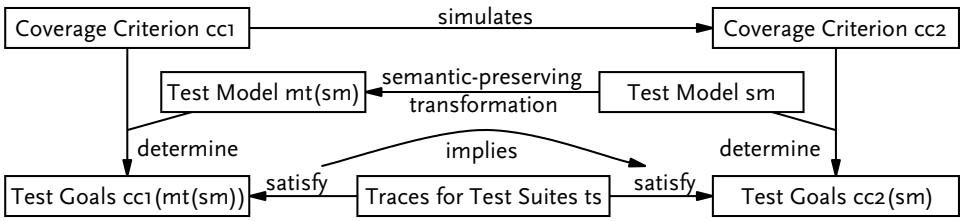


Figure 4.16: Simulated satisfaction of coverage criteria.

Figure 4.16 depicts the notion of simulated satisfaction. The coverage criterion $cc1$ simulates the coverage criterion $cc2$ iff for each semantic-preserving

model transformation mt, the satisfaction of $cc1$ on the transformed state machine implies the satisfaction of $cc2$ on the original state machine for all test suites $ts \in TS$:

$$(cc1 \Rrightarrow cc2) \text{ iff } (\exists_{mt \in MT} \forall_{sm \in SM, ts \in TS} : \\ (asmr(ts, mt(sm)) \models cc1(mt(sm))) \rightarrow (asmr(ts, sm) \models cc2(sm))).$$

In the following, we present model transformations that are used to show the simulation of many coverage criteria from the subsumption hierarchy presented in Figure 2.5 on page 20.

To show that the transformations have the desired effect, we have to show (1) that the model transformations preserve the semantics of the state machine and (2) that the simulated satisfaction relations between the coverage criteria are valid. For (1), the model transformations we consider are composed of basic transformation patterns that are presented in Section 4.2.2. The transformation patterns are shown to preserve the state machine semantics. Thus, the composed transformations also do. For (2), we apply the formal definitions of this section for coverage criteria *cc1* and *cc2* to show that the satisfaction of a coverage criterion *cc1(mt(sm))* implies the satisfaction of *cc2(sm)*. We prove this by showing the contraposition, i.e., if *cc2(sm)* is not satisfied then *cc1(mt(sm))* is also not. Thus, we show that one unsatisfied test goal (see Section 2.4.2) of *cc2(sm)* implies that a test goal of *cc1(mt(sm))* is also unsatisfied. This approach has the advantage that we can use the model transformations for the proof without requiring that the transformations are bidirectional. We desist from defining a formal semantics of state machines for proving the correctness of the presented transformations. One reason is that all transformations are simple and intuitive and, thus, such a formal framework would be overkill. Another one is that, due to the mentioned issues with state machine semantics definition, all proofs would have to be adapted for each tool vendor, anyway.

4.3.3 Simulated Satisfaction Relations

In this section, we apply the previously presented state machine transformation patterns to compose test model transformations that are used to establish simulated satisfaction relations. All analyzed coverage criteria are presented in [UL06] and formally defined in Section 2.4.3.

All-Transitions Simulates Multiple Condition Coverage.

A test model transformation to simulate Multiple Condition Coverage (MCC) by satisfying All-Transitions consists of the following: For each vertex v, we

split all outgoing transitions t of v (see pattern *Split Transition*) once for each value assignment cva of t's guard. The result of the transformation is a set of transitions t_new for each t and cva so that only t_new is activated if t was activated for cva in the original test model.

The corresponding pseudocode is shown in Figure 4.17. Figure 4.18 shows the transformed test model of the coffee dispenser example.

```
simulateMCCWithAllTransitions(SM sm) {
  for each vertex v in sm {
    for each outgoing transition t of v {
      splitTransition(sm, t);
} } }
```

Figure 4.17: Transformation that splits transitions according to guards.

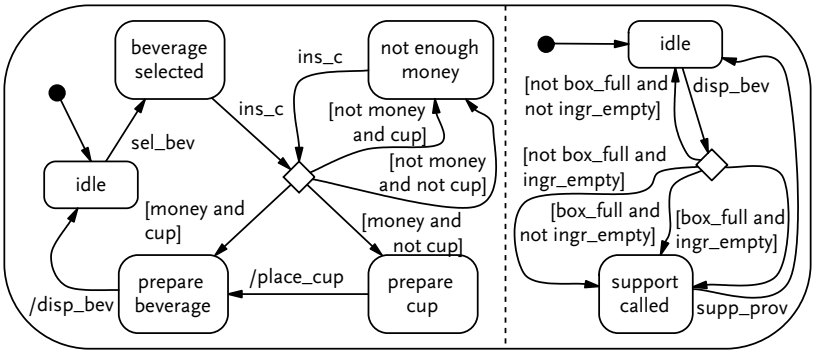


Figure 4.18: Transformed test model to satisfy MCC by satisfying All-Transitions.

The presented model transformation justifies the simulation of MCC with All-Transitions. We prove this by showing the contraposition.

Proof. If MCC is unsatisfied, then there is a transition t and a guard value assignment cva so that there is an unsatisfied test goal for MCC that contains the trace pattern $((\{t.source\}, t.events, cva, ?))$. The transformation creates a separate transition ct that is only activated if one event of $t.events$ is triggered from the state $t.source$ with the condition cva. Thus, the transformation of $((\{t.source\}, t.events, cva, ?))$ results in the trace pattern $((\{t.source\}, t.events, cva, \{ct\}))$. Since ct cannot be activated for any other state, event, or value assignment, the trace pattern $((?, ?, ?, \{ct\}))$ is also not covered by any trace of the test suite. Thus, the test goal of All-Transitions on $mt(sm)$ that contains this trace pattern is not satisfied, and also All-Transitions is unsatisfied on the transformed model. ∎

All-Transitions Simulates All-Transition-Pairs.

To simulate the satisfaction of All-Transition-Pairs with All-Transitions, we present a model transformation that consists of two phases: In the first phase, we consider all transitions but self-transitions. For each vertex v with $n > 1$ incoming transitions, we create $n - 1$ copies $c1$ of v and spread the incoming transitions of v over all $c1$ and v so that each of these vertices has exactly one incoming transition (see pattern *Copy Vertices*). In the second phase, we create another copy $c2$ of v for each self-transition st of v and set the target state $st.target = c2$ (see pattern *Exchange Transition Targets for Vertex Copies*). We also set the target vertex of each copy of st to $c2$. As a result, for each pair of adjacent transitions $(t1, t2)$ of the original test model, the transformed test model contains a copy of $t2$ that can only be traversed if a copy of $t1$ has been traversed before. Figure 4.19 depicts the pseudocode for this transformation. Figure 4.20 shows one possible transformed model of the coffee dispenser.

```
simulateAllTransitionPairsWithAllTransitions(SM sm) {
  for each vertex v in sm {
    for all but 1 incoming transitions t of v {
      copyVertex(sm, v);
      c1 = the copy of v;
      t.target = c1;
      addVerticesForSelfTransitions(c1); }
    addVerticesForSelfTransitions(v); }
}

addVerticesForSelfTransitions(Vertex v) {
  m = empty map from transition to vertex;
  for each self-transition st of v {
    take existing copy c2 of v or create a one with copyVertex(sm, v);
    set st.target = c2;
    m.put(st, c2); } // map for target states of transitions
  // set target states for copies of self-transitions
  for each new copy c2 of v {
    for all copies ct of st that are self-transition of cv {
      set ct.target = m.get(st);
} } }
```

Figure 4.19: Transformation for the simulated satisfaction of All-Transition-Pairs with All-Transitions.

The satisfaction of All-Transitions on this transformation's target model implies the satisfaction of All-Transition-Pairs on the source model. Again, we prove this by showing the contraposition.

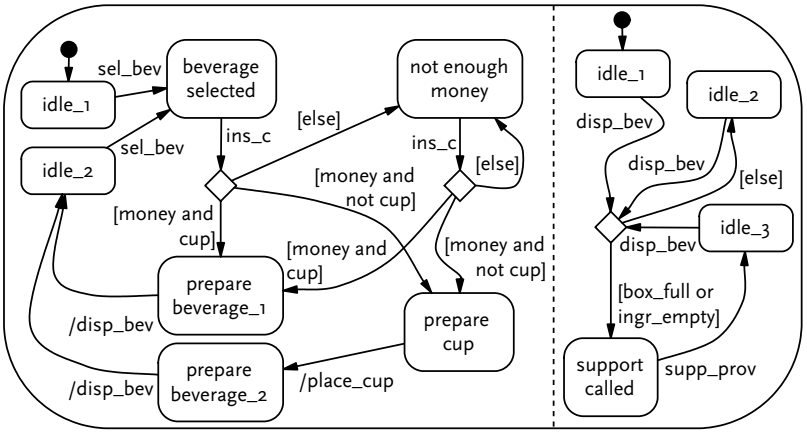


Figure 4.20: Transformed test model to satisfy All-Transition-Pairs by satisfying All-Transitions.

Proof. If All-Transition-Pairs on the original state machine is unsatisfied, then a test goal with the trace pattern $((?,?,?,\{t1\}),(?,?,?,\{t2\}))$ for the transitions $t1$ and $t2$ is unsatisfied, too. For all adjacent $t1$ and $t2$, the model transformation creates copies of $t1$ and $t2$ so that a copy $ct2$ of $t2$ can only be traversed after a copy $ct1$ of $t1$. The transformation creates copies of transitions. Traversing a transition copy on the transformed state machine corresponds to traversing the corresponding original in the original state machine. Thus, if any $((?,?,?,\{t1\}),(?,?,?,\{t2\}))$ is not covered, then all corresponding trace patterns $((?,?,?,\{ct1\}),(?,?,?,\{ct2\}))$ are also not covered. The transition $ct2$ can only be traversed after any copy of $t1$. Thus, if all $((?,?,?,\{ct1\}),(?,?,?,\{ct2\}))$ are not covered, then $((?,?,?,\{ct2\}))$ is also not. Consequently, the corresponding test goal is unsatisfied on the target model and All-Transitions is unsatisfied on the target model, too. ∎

All-Transitions Simulates All-Uses.

In this section, we present a model transformation to simulate All-Uses with All-Transitions: For each existing transition *t_d* that defines a variable *var*, a new variable *var_new* is set to true at *t_d* and to false at all other definitions *t_rd* of *var* (see pattern *Add Variables*). For each transition *t_u* that includes a use of *var*, a use of *var_new* is added: If *t_u* has an effect *ef*, *ef* is moved (see pattern *Move Effect*) to the newly created transition *t_u_new*. Afterwards, a choice pseudostate *c* is inserted in *t_u* (see pattern *Insert Node in Transition*) and the new outgoing transition of *c* is split according to the value of *var_new* (see pattern *Split Transition*): One transition has the guard [*var_new*] and the other transition has the guard [*else*].

The effect is moved to prevent that it has any influence on the evaluation of $[var\_new]$. The result of this transformation is at least one new transition t for each def-use-pair (t_d, t_u) for which t can only be traversed if (t_d, t_u) is tested. Figure 4.21 shows the pseudocode for this model transformation. Figure 4.22 shows the transformation result exemplary for the coffee dispenser with two new attributes $a1$ and $a2$ for the value of $money$, which is defined at transitions triggered by ins_c and used in the outgoing transitions of the adjacent decision state.

```
simulateAllUsesWithAllTransitions(SM sm) {
  for each variable var in sm {
    for each transition effect ef that includes a definition of var {
      create a new variable var_new;
      addVariable(ef, var_new, "true");
      for all other defining transition effects ef2 of var {
        addVariable(ef2, var_new, "false"); }
    }
    for each transition t that includes a use of var (guard or effect) {
      if t has an effect { moveEffect(t); }
      insertNodeInTransition(t);
      split the outgoing transitions of t.target according to var;
} } }
```

Figure 4.21: Transformation for simulating All-Uses with All-Transitions.

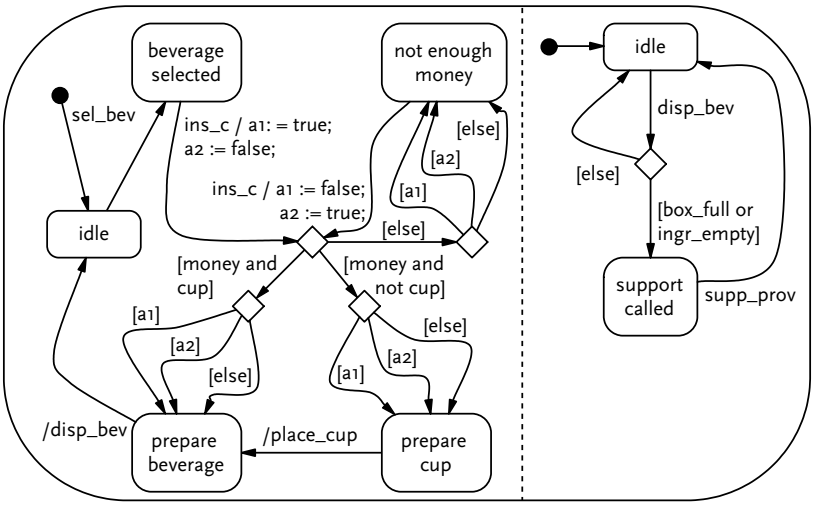


Figure 4.22: Transformed model to simulate All-Uses with All-Transitions.

We prove that this model transformation witnesses the simulated satisfaction of All-Uses by All-Transitions by showing the contraposition.

Proof. If All-Uses is unsatisfied on the original test model, then a test goal with the trace pattern $((?,?,?,\{t\_d\}),(?,?,?,\{\neg t\_rd\})*,(?,?,?,\{t\_u\}))$ is also unsatisfied. The used transformation inserts a new variable v_new and sets it to true at t_d and to false at each t_rd. Thus, the presented trace pattern expresses for the transformed model that v_new has to be set to true and never set to false before reaching t_u. Furthermore, a transition t is added to the same compound transition as t_u. The guard of t is $[v\_new]$. As a result of this transformation, the value of v_new is true iff the presented trace pattern is covered. Thus, if $((?,?,?,\{t\_d\}),(?,?,?,\{\neg t\_rd\})*,(?,?,?,\{t\_u\}))$ is not covered by a test suite trace, then $((?,?,?,\{t\}))$ also is not. Consequently, the corresponding test goal and All-Transitions are unsatisfied on $mt(sm)$, too. ∎

All-Defs Simulates All-Uses.

All-Uses subsumes All-Defs. Since there can be several uses for the definition of a variable *var*, All-Defs does not necessarily subsume All-Uses. Here, we present a model transformation $mt \in MT$ to show that All-Defs simulates All-Uses: For each use t_u of a def-use-pair (t_d, t_u) in sm, a newly inserted attribute a (see pattern *Add Variables*) is defined at all definitions t_d (e.g. set to true) and used at t_u (e.g., by adding the tautology *[a or (not a)]* to the corresponding guard). The intention of this transformation is to create variables for which there is exactly one use. This is done for all existing def-use-pairs. Consequently, the satisfaction of All-Defs implies the satisfaction of All-Uses. Since the new guard imposes no restrictions, the newly inserted def-use-pairs do not influence the behavior of the test model. Thus, the semantics of $mt(sm)$ is the same as the semantics of sm.

```
simulateAllUsesWithAllDefs(SM sm) {
  for each t_u of def-use-pairs (t_d, t_u) {
    create a new attribute a;
    for each t_d that defines the variable that is used in t_u {
      addVariable(t_d.effect, a, "true"); }
    add the usage of a to t_u (by adding a tautology to the
      corresponding guard, e.g., [a or not a]);
} }
```

Figure 4.23: Transformation for the simulation of All-Uses with All-Defs.

Figure 4.23 depicts the pseudocode for this model transformation. Figure 4.24 shows a partly transformed test model of the coffee dispenser. Like for the previous transformation, the focus is on the variable *money*, which is

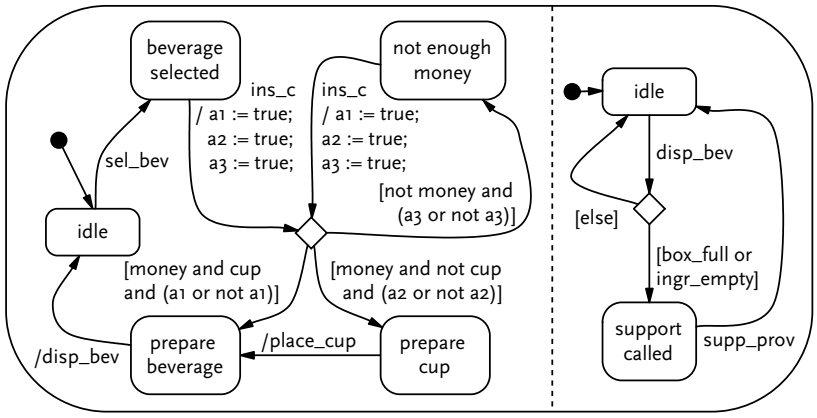


Figure 4.24: Transformed test model to simulate All-Uses with All-Defs.

defined at the transitions triggered by ins_c and used in the three outgoing transitions of the choice pseudostate.

We prove that using this model transformation leads to the simulated satisfaction of All-Uses with All-Defs by showing the contraposition.

Proof. If All-Uses is unsatisfied on the original state machine, then a test goal with a trace pattern $((?,?,?,\{t\_d\}),(?,?,?,\{\neg t\_rd\})*,(?,?,?,\{t\_u\}))$ is also unsatisfied. The presented model transformation adds a new variable that is used only at t_u and defined at t_d and all t_rd. Transitions are unchanged. Only their guards and effects are changed. Applying All-Defs on the transformed model results in a set of test goals, one of which is equal to the presented test goal of All-Uses and, thus, is unsatisfied. Consequently, All-Defs are unsatisfied on $mt(sm)$, too. ∎

All-States Simulates All-Configurations.

All-Configurations subsumes All-States. A model transformation $mt \in MT$ to witness that All-States simulates All-Configurations consists of transforming each configuration c of parallel states into one state s. All outgoing transitions of c are copied to the set of s's outgoing transitions in the target model. The traversal of one of s's outgoing transitions t in $mt(sm)$ leads to a state representing the target state configuration that would have been reached in the original test model sm. The presented model transformation does not add or remove any details. Instead, only the representation is not parallel anymore. The semantics of the state machine is unchanged.

The pseudocode for this transformation is shown in Figure 4.25. Figure 4.26 shows the transformed test model of the coffee dispenser example.

```
simulateAllConfigurationsWithAllStates(SM sm) {
  create a new region r in sm;
  for each configuration c in sm {
    create a new state s in r; }
  for each configuration c in sm {
    s = the state that was created for c in r;
    for all outgoing compound transitions t of c {
      copy t (including pseudostates) to s.outgoing;
      set the t.target to the state in r that describes the corresponding
        target state configuration of traversing t from c;
    }
    create an initial node in r with an outgoing transition
      to the state that corresponds to the initial configuration of sm;
  }
  remove all regions from sm except for r;
}
```

Figure 4.25: Transformation for the simulation of All-Configurations with All-States.

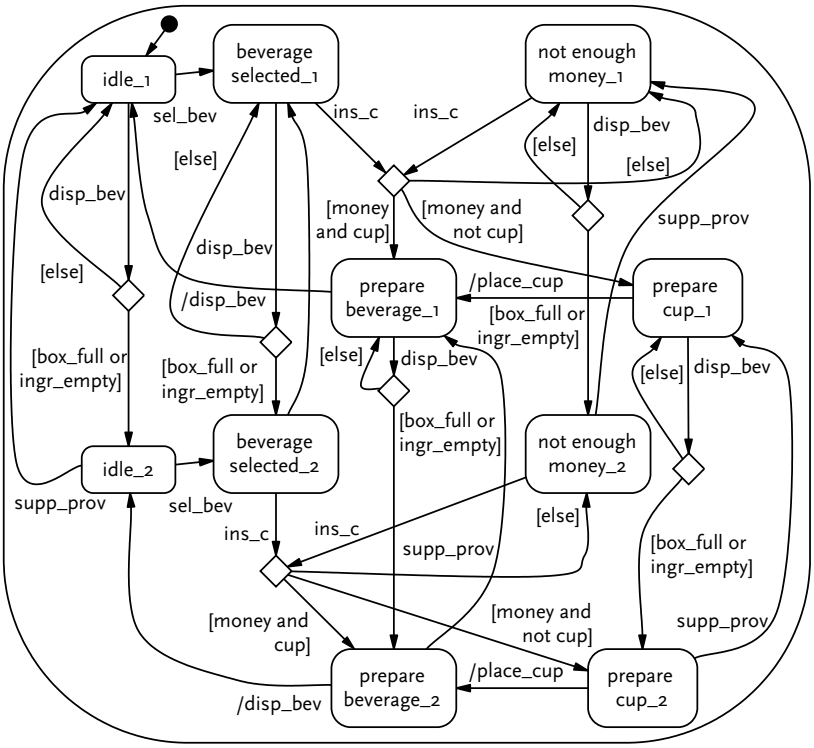


Figure 4.26: Transformed test model to satisfy All-Configurations by satisfying All-States.

All states whose names end with "1" correspond to configurations that include the state *idle* of the right region. The "2" stands for the inclusion of the state *support called* of the right region.

We will prove that the presented transformation can be used to simulate All-Configurations with All-States by showing the contraposition.

Proof. If All-Configurations is unsatisfied on the original state machine, then a test goal with a trace pattern $((c, ?, ?, ?))$ for configuration c is also unsatisfied. The target model of the presented model transformation describes all configurations of the original model as single states. This means that all configurations are represented by states and the relations between the states are the same as the ones between configurations. Thus, there is a state s in $mt(sm)$ that corresponds to the configuration c in sm so that the test goal $((\{s\}, ?, ?, ?))$ is also unsatisfied for $mt(sm)$. Consequently, All-States is unsatisfied on $mt(sm)$, too. ∎

All-States Simulates All-Transitions.

All-Transitions subsumes All-States. The model transformation pattern *Insert Node in Transition* is enough to show that All-States simulates All-Transitions: For each transition $t1$ of the test model sm, a new choice pseudostate c and a new transition $t2$ are created with: $t2.target := t1.target$, $t1.target := c$, and $t2.source := c$. After the transformation, c is the new target state of $t1$, $t2$ points from c to the former target state of $t1$, $t1$ is the only incoming transition of c, and $t2$ is c's only outgoing transition. Since c is a pseudostate with one incoming and one outgoing transition, $t2$ is part of the same compound transition as $t1$ (see "compound transition" [Obj07, page 568] and "run-to-completion" [Obj07, page 559]). Since $t2$ has no additional trigger, guard, nor effect, the model transformation does not impact the original state machine semantics.

```
simulateAllTransitionsWithAllStates(SM sm) {
  for all transitions t in sm {
    insertNodeInTransition(t);
} }
```

Figure 4.27: Transformation for the simulated satisfaction of All-Transitions with All-States.

Figure 4.27 depicts the pseudocode that describes the model transformation. Figure 4.28 shows the transformed test model of the coffee dispenser example.

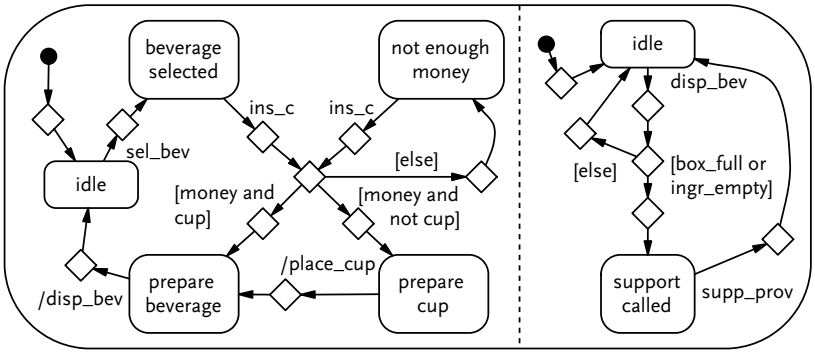


Figure 4.28: Transformed test model to satisfy All-Transitions by satisfying All-States.

We prove that the presented transformation can be used to simulate All-Transitions with All-States by showing the contraposition.

Proof. If All-Transitions is unsatisfied on the original state machine, then a test goal with a trace pattern $((?, ?, ?, \{t1\}))$ for the transition $t1$ is unsatisfied in sm. The model transformation adds new choice pseudostates c into the transition of $t1$. Thus, the trace pattern $((?, ?, ?, \{t1\}), (\{c\}, ?, ?, \{t2\}))$ is also unsatisfied in $mt(sm)$. The vertex c can only be reached by traversing its incoming transition $t1$. Thus, c is not reached iff $t1$ is not traversed. Correspondingly, the test goal with the trace pattern $((\{c\}, ?, ?, ?))$ is not satisfied and All-States is unsatisfied on $mt(sm)$, too. ∎

Condition Coverage Simulates All-Transitions.

Condition Coverage is satisfied if all values for each atomic condition are tested. Since this does not always imply the satisfaction of the whole guard condition, not all transitions are necessarily traversed. Thus, Condition Coverage does not subsume All-Transitions. A model transformation $mt \in MT$ that shows that Condition Coverage simulates All-Transitions may consist of extending each transition $t1$ by inserting a choice pseudostate and a transition $t2$ as described in the pattern *Insert Node in Transition*. Additionally, a new attribute a is created and $t2$'s guard condition is set to a tautology that includes a, like *[a or (not a)]*. This guard is always satisfied. Thus, the transition $t2$ can always be executed from its source state and does not impact the state machine semantics.

Figure 4.29 shows the pseudocode for the presented model transformation. In Figure 4.30, the transformed coffee dispenser test model is shown.

```
simulateConditionCoverageWithAllTransitions(SM sm) {
  create the new variable a;
  for all outgoing transitions t of all initial nodes {
    addVariable(t.effect, a, "true"); }
  for all transitions t1 in sm {
    insertNodeInTransition(t1);
    Transition t2 = t1.target.outgoing->get(0);
    t2.guard = "[a or (not a)]";
} }
```

Figure 4.29: Transformation for the simulation of Condition Coverage with All-Transitions.

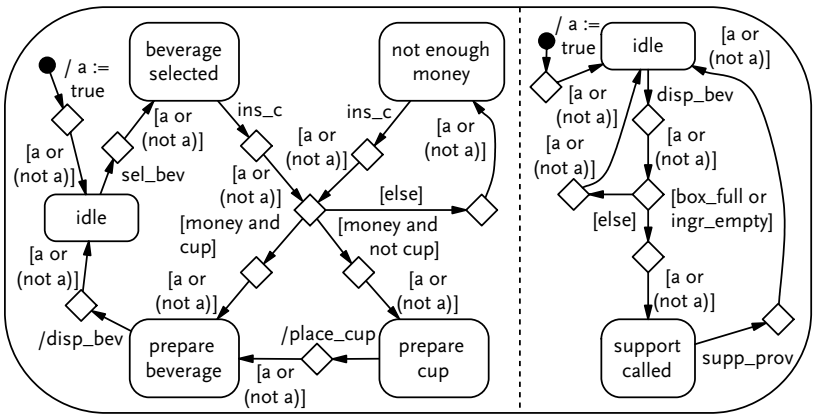


Figure 4.30: Transformed test model to simulate All-Transitions with Condition Coverage.

The presented model transformation justifies that Condition Coverage simulates All-Transitions. We prove this via contraposition.

Proof. If All-Transitions is unsatisfied on the original state machine, then there is a transition $t1$ and an unsatisfied test goal with the trace pattern $((?, ?, ?, \{t1\}))$. The model transformation adds a transition $t2$ to each transition $t1$. They are connected by a pseudostate and are, therefore, part of the same compound transition. Each added $t2$ contains a guard condition that is always satisfied, i.e., the use of any condition value assignment cva results in traversing $t2$. Since $t1$ is the only incoming transition of $t2$'s source state, $t1$ must be traversed before $t2$. Since $t1$ is never traversed and $t1$ is the only incoming transition of $t1.target$, there will be also an unsatisfied test goal with the trace pattern $((\{t1.target\}, ?, cva, \{t2\}))$. As a consequence, Condition Coverage is unsatisfied on $mt(sm)$. ∎

All-Def-Use-Paths Simulates All-Paths.

The satisfaction of All-Paths requires to traverse all paths. If the state machine contains loops, this may result in an infinite number of possible paths. The same applies to All-Def-Use-Paths. Thus, both coverage criteria are known to be infeasible. All-Paths subsumes All-Def-Use-Paths. A model transformation $mt \in MT$ that shows that All-Def-Use-Paths simulates All-Paths is equal to the transformation for the simulated satisfaction of All-Transitions with Condition Coverage. Thus, it also preserves the intuitive state machine semantics.

Since the transformation is the same as the previous one, we reference Figure 4.29 for the pseudocode of this transformation and Figure 4.30 for the transformed coffee dispenser example. The model transformation also justifies that All-Def-Use-Paths simulates All-Paths. The proof is given by showing the contraposition.

Proof. Since each compound transition contains a use of the variable a and there is only one definition of a in the very first transition, each path is also a def-use-path. So, if there is a not covered path in the original test model, there will be a corresponding not covered def-use-path in the transformed test model. Hence, not satisfying All-Paths on sm implies not satisfying All-Def-Use-Paths on $mt(sm)$. ∎

All-Defs Simulates All-Transitions.

All-Defs is satisfied iff at least one def-use-pair (t_d, t_u) is tested for each defining transition t_d of each variable var [UL06, page 115]. Traversing transitions and testing def-use-pairs are not related. Consequently, neither All-Defs subsumes All-Transitions nor vice versa. A possible model transformation $mt \in MT$ that witnesses that All-Defs simulates All-Transitions consists of adding a new transition $t2$ for each transition $t1$ (see pattern *Insert Node in Transition*) and an attribute a to the test model (see pattern *Add Variables*). The new attribute a is defined in each $t1$ and used in each $t2$. As a consequence, traversing any original transition corresponds to defining and using the new attribute, which has to be done to satisfy All-Defs. This model transformation adds definitions and uses of a newly defined attribute that has no impact on the rest of the test model. Consequently, mt does not change the semantics of the test model sm.

Figure 4.31 shows the corresponding pseudocode for this model transformation. The transformed coffee dispenser test model is depicted in Figure 4.32.

```
simulateAllTransitionsWithAllDefs(SM sm) {
  create a new attribute a;
  for all transitions t1 in sm {
    insertNodeInTransition(t1);
    addVariable(t1.effect, a, "true");
    Transition t2 = t1.target.outgoing->get(0);
    add the use "[a or (not a)]" of a to the guard of t2;
  }
}
```

Figure 4.31: Transformation for the simulated satisfaction of All-Transitions with All-Defs.

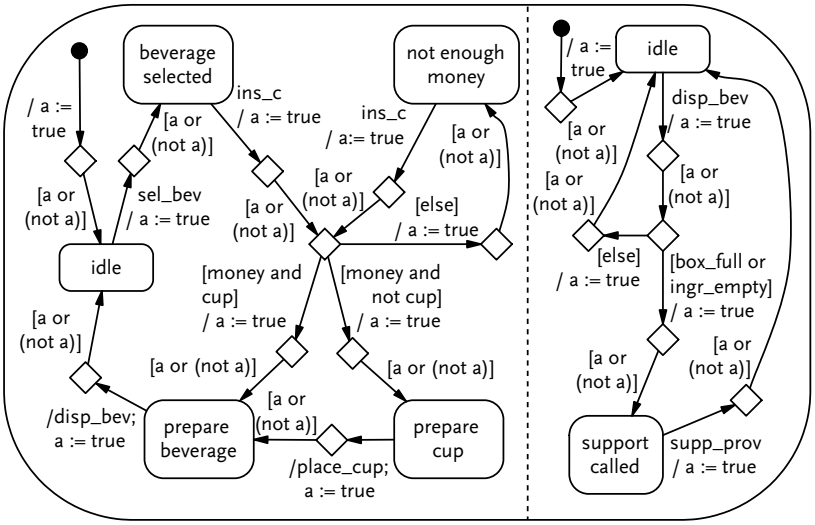


Figure 4.32: Transformed test model for the simulated satisfaction of All-Transitions with All-Defs.

The presented model transformation justifies that All-Defs simulates All-Transitions. Again, we prove this by showing the contraposition.

Proof. If All-Transitions is unsatisfied, then there is a transition $t1$ so that a corresponding test goal with the trace pattern $((?, ?, ?, \{t1\}))$ is uncovered. The presented model transformation adds new transitions $t2$ to the compound transition of $t1$ and adds a variable a that is defined at each $t1$ and used at each $t2$. The transition $t2$ can only be traversed if $t1$ has been traversed before. If $t1$ is not traversed, then the test goal with the trace pattern $((?, ?, ?, \{t1\}), (?, ?, ?, \{t2\}))$ is also unsatisfied. Consequently, All-Defs is not satisfied on the transformed test model $mt(sm)$. ∎

Decision Coverage Simulates All-Transitions.

As presented in Section 2.1.5, there are several interpretations of subsumption relations between certain coverage criteria. For instance, we state that Decision Coverage subsumes All-Transitions, but there are also other opinions. Here, we assume that there is no such subsumption relation and briefly sketch the corresponding simulation relation: The used model transformation is exactly the one presented for the simulation of All-Transitions with Condition Coverage. As a result of the transformation, each compound transition contains at least one transition with a guard. Consequently, satisfying all guards results in traversing all transitions. Thus, Decision Coverage simulates All-Transitions.

4.3.4 Simulated Satisfaction Graph

In this section, we combine the presented relations of simulated coverage criteria satisfaction. We present them as an extension of the corresponding coverage criteria's subsumption graph. Figure 4.33 shows the simulated satisfaction graph.

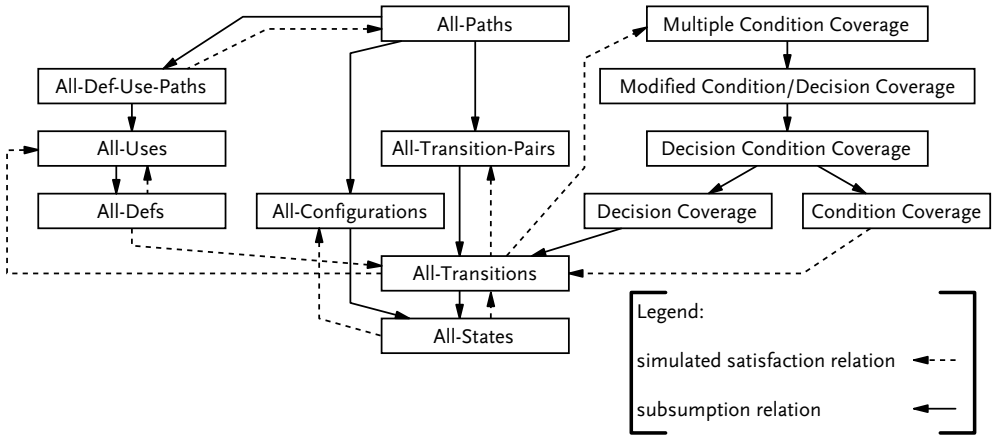


Figure 4.33: Simulated satisfaction graph.

Interesting results can be derived from this graph. First, simulated satisfaction relations between coverage criteria partly invert subsumption relations. For instance, All-Transitions simulates Multiple Condition Coverage, which subsumes All-Transitions in turn. All-Transitions simulates All-Transition-Pairs, which transitively subsumes All-Transitions. There are more examples. Second, many coverage criteria can be simulated with coverage criteria that are not directly related to them. By combining the corresponding model transformations, even more pairs of coverage criteria can

be used for simulation. For instance, All-States simulates All-Transitions, and All-Transitions simulates All-Uses. Consequently, All-States simulates All-Uses with the corresponding concatenated model transformations. Likewise, Condition Coverage simulates All-Uses, All-Defs simulates All-Configurations, and Condition Coverage simulates All-Transition-Pairs. That is, any feasible coverage criterion can be chosen to simulate almost any other feasible coverage criterion.

There are a few exceptions. For instance, All-Paths and All-Def-Use-Paths are considered infeasible, because they often return an infinite set of test goals and also require an infinite test suite to be satisfied. It is impossible to transform a finite set of test goals into an infinite one with a finite transformation. Thus, there is no generally applicable transformation that admits the simulation of any infeasible coverage criterion with a feasible one. As an exception to that, it might be possible to use a step-wise transformation, e.g., for online testing: For instance, it is possible to repeatedly transform the model so that visiting a newly generated state corresponds to traversing a path. This model transformation can be done for an arbitrary number of paths. Complete execution would take infinite time and, thus, has to be stopped at some point.

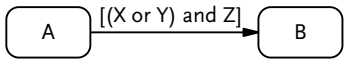


Figure 4.34: Simple guarded transition.

Another example is the simulation of "complex" coverage criteria with "simple" ones. For instance, All-Transitions simulates MC/DC because All-Transitions simulates Multiple Condition Coverage, which subsumes MC/DC. The problem, however, is that Multiple Condition Coverage requires exponential effort whereas MC/DC requires only linear effort depending on the guard condition size. The task is to keep the additionally introduced test effort of simulated satisfaction as low as possible. So the question is whether there are more efficient model transformations. Each test goal for MC/DC requires the satisfaction of several atomic test goals. This often results in pairs of test cases that are necessary to satisfy MC/DC. The satisfaction of All-Transitions, All-Defs, All-Transition-Pairs and alike only require a set of single test cases to be generated. Depending on the used test model, it might be hard to find single test cases that have the same effect as pairs of test cases. For instance, Figure 4.34 shows a simple transition with a guard for which we want to satisfy MC/DC. Satisfying MC/DC means generating pairs of test cases that show the isolated impact of all variables. For each pair, there have to be two distinct transitions, each satisfying one element of the pair.

Figure 4.10 shows the truth table for this example. There are several minimal sets of value assignments that can be used to satisfy MC/DC. For the presented example, it might be sufficient to test the value assignments for the rows 3, 4, 5, and 7: Rows 3 and 7 show the impact of X, rows 5 and 7 for Y, and rows 3 and 4 for Z. Due to other model elements, however, testing row 4 might be infeasible. To deal with this problem, we could also test row 6, which could be used together with row 5 to show the impact of Z. If, however, rows 4 and 6 are infeasible, then we have to test rows 1 and 2 to show the impact of Z. As we see, a fundamental problem of satisfying MC/DC is the existence of infeasible value assignments. Due to this problem, we can only exclude tests for rows that cannot be used in combination with any other row to show the impact of any variable – only row 8 for this example. We have to include all other rows. If all the corresponding value assignments are feasible, however, this results in unnecessary test effort.

Row	X	Y	Z	(X or Y) and Z
1	1	1	1	1
2	1	1	0	0
3	1	0	1	1
4	1	0	0	0
5	0	1	1	1
6	0	1	0	0
7	0	0	1	0
8	0	0	0	0

Table 4.10: Truth table for (X *or* Y) *and* Z.

Finally, it is easy to see that all subsumption relations are also relations of simulated satisfaction by using identy as model transformation: We claim that simulated satisfaction is a more general relation than subsumption, i.e., all pairs of coverage criteria that are related by subsumption are also related by simulated satisfaction.

Theorem 4.3.1 *Simulated satisfaction includes subsumption.*

We prove this theorem by presenting one corresponding model transformation:

Proof. The identical transformation results in $sm = mt(sm)$. If both considered coverage criteria are applied to the same model, then the simulated satisfaction implies subsumption. ∎

This does not mean that simulated satisfaction is somehow superior to subsumption. As we see in Figure 4.33, simulated satisfaction is too broad for comparing coverage criteria. Instead, it can be used to exchange coverage criteria. It is also a powerful means to support existing model-based test generators.

4.4 Further Effects of Model Transformations

Beyond simulated satisfaction of coverage criteria, there can be several further effects of test model transformations for coverage criteria that are applied to UML state machines. In the following, we sketch the combination of different coverage criteria in Section 4.4.1 and the definition of new coverage criteria in Section 4.4.2, both of which can also be implemented with model transformations and existing coverage criteria. We present general considerations about the impact of test model transformations on coverage criteria satisfaction in Section 4.4.3.

4.4.1 Coverage Criteria Combinations

In this section, we present the use of model transformations for the combination of coverage criteria. For that, we define different kinds of coverage criteria combinations. After that, we present concrete test model transformations to simulate combined coverage criteria with single coverage criteria.

Kinds of Coverage Criteria Combinations.

In this section, we present several interpretations of coverage criteria combination. Some of them have also been presented in [FSW08]. We define three kinds of coverage criteria combinations based on their complexity.

Definition 34 (Level-1-Combination) *The combination of two or more coverage criteria by satisfying each of them with a separate test suite and combining these test suites, afterwards, is a* level-1-combination.

Level-1-combinations can be achieved by creating and uniting two or more test suites that satisfy one the coverage criteria, each. Both may be supported by the corresponding test generator. Furthermore, the two coverage criteria can also be simulated, e.g., by satisfying All-Transitions two times on the correspondingly transformed test models. The advantage of level-1-combinations is their easy creation. The disadvantage is the overhead of unnecessarily generated test cases caused by overlapping coverage criteria.

Definition 35 (Level-2-Combination) *A* level-2-combination *of two coverage criteria consists of creating test cases to satisfy the first coverage criterion, monitoring (measuring the satisfaction of) the second coverage criterion, and creating test cases to satisfy only the still unsatisfied test goals of the second coverage criterion.*

Just like level-1-combinations, level-2-combinations are focused on combining the isolated satisfaction of two coverage criteria. Since level-2-combinations are monitored, they do not result in the test case overhead of level-1-combinations. Additionally, all test cases of the first coverage criterion that are unnecessary because of test cases for the second coverage criterion can be removed. The monitoring makes the creation of level-2-combinations harder. Level-2-combinations can also be achieved by satisfying only one coverage criterion on several transformed test models.

Definition 36 (Level-3-Combination) *A* level-3-combination *consists of satisfying test goals from one coverage criterion for the test goals of another coverage criterion.*

Level-3-combinations are not focused on the isolated satisfaction of coverage criteria. Instead, the test goals of one coverage criterion are satisfied for each test goal of another coverage criterion. This can be done, e.g., by applying the second coverage criterion only on the model elements that are referenced in the test goals of the first coverage criterion. We presented one example in Section 4.1.2 that contains the description of choice pseudostate splitting. This splitting results in satisfying a control-flow-based coverage criterion, like Decision Coverage for All-Transition-Pairs, by using all guard value assignments necessary to satisfy Decision Coverage for the second transition of every pair of adjacent transitions. Another example is the combination of coverage criteria as described in Chapter 3: A certain boundary-based coverage criterion is satisfied for the test cases of all test goals of any structural, e.g., control-flow-based, data-flow-based, or transition-based, coverage criterion. These two examples show two different approaches. In the first one, all test goals for the combined coverage criterion can be identified statically for the state machine. We call this a *static combination*. For the second example, the test goals of the boundary-based coverage criterion can be defined for the input partitions of the generated abstract test cases for the, e.g., control-flow-based, coverage criterion. Since these paths are not statically predefined, we call this *dynamic combination*.

All presented combinations can be executed for more than just two coverage criteria. We assume that level-1-combinations and (to a certain extent) level-2-combinations are common knowledge and also easy to construct. In the following, we concentrate on level-3-combinations and sketch how to achieve them by satisfying only one coverage criterion on a transformed test model.

Level-3-Combinations of Coverage Criteria.

In this section, we present several level-3-combinations of transition-based, control-flow-based, data-flow-based, and boundary-based coverage criteria.

First, we present level-3-combinations of transition-based and control-flow-based coverage criteria. The purpose of such combinations is to test all relevant guard value assignments for all considered transition sequences. A corresponding combination can be achieved by splitting the intermediate states of the transition sequence as presented in the industrial report in Section 4.1.2 (Choice Pseudostate Splitting). For all pairs of adjacent transitions of the original state machine, the transformed test model contains a single transition that is only traversed by a test case iff the test case traverses also the transition pair of the original state machine. The satisfaction of any control-flow-based coverage criterion on the transformed test model corresponds to the satisfaction of the level-3-combination. We present the level-3-combination of All-Transition-Pairs and Decision Coverage as one example: A test suite that satisfies this combined coverage criterion satisfies and violates each guard condition of the second transition of each pair of adjacent transitions. We call the level-3-combination of All-Transition-Pairs and Decision Coverage *All-Transition-Pairs-Decisions*. To clarify the meaning of All-Transition-Pairs-Decisions, we present a formal definition of this coverage criterion in Figure 4.35.

```
P(TG) All-Transition-Pairs-Decisions(SM sm) {
  testgoals = All-Transition-Pairs(sm);
  for each transition t1 in sm {
    for each outgoing transition t2 of the target state of t1 {
      Expression positive = "false";
      Expression negative = "false";
      for each value assignment va for the guard of t2 {
        cva = va expressed as a logical formula;
        if the guard of t2 is true for va {
          positive = positive + "or cva";
        } else {
          negative = negative + "or cva";
      } }
      testgoals.add(new ATG( ((?, ?, ?, {t1}),
                          ({t2.source}, t2.events, positive, {t2})) ));
      testgoals.add(new ATG( ((?, ?, ?, {t1}),
                             ({t2.source}, t2.events, negative, ?)) ));
  } }
  return testgoals; }
```

Figure 4.35: Definition of All-Transition-Pairs-Decisions.

The level-3-combination of transition-based and control-flow-based coverage criteria is a static combination. Figure 4.36 shows one part of a such transformed model: A copy of the state $S3$ is created. As a result, the guard conditions on $S3$'s outgoing transitions are also copied. Satisfying and violating the guards on the target model corresponds to satisfying and violating them for each incoming transition of $S3$ in the original model.

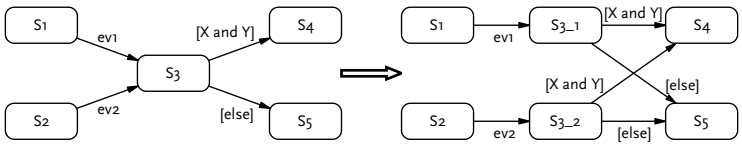


Figure 4.36: Example for a model transformation to support the level-3-combination of control-flow-based and transition-based coverage criteria.

Second, we sketch the level-3-combination of transition-based and data-flow-based coverage criteria. Data-flow-based coverage criteria are focused on testing def-use-pairs of variables. The corresponding test goals refer to single transitions. Transition-based coverage criteria are focused on traversing adjacent transition sequences. Combinations of such coverage criteria may have several results: For instance, for each def-use-pair, all sequences of adjacent transitions could have to be tested, for which one of them is a defining or using transition of a def-use-pair. This is a static combination.

Third, control-flow-based coverage criteria can be combined with data-flow-based ones. As one example, all values of guard conditions that are referenced by a defining or a using transition have to be tested for each def-use-pair of the data-flow-based coverage criterion. One result of this combination would be that all guard value assignments are tested for paths that lead to the guard-referencing transition. For instance, the level-3-combination of All-Uses and MC/DC could require that MC/DC is satisfied for each using transition of each def-use-pair. The effect would be that all possible effects of variable definitions on the guard evaluation are included, which could be beneficial. This combination is also static.

We proposed two level-3-combinations with data-flow-based coverage criteria. We note that these level-3-combination have not yet been shown to be beneficial. For this reason, we desist from considering them in further detail.

A level-3-combination of control-flow-based, data-flow-based, and transition-based coverage criteria can consist of the following: All guards for all transition sequences that contain defining and using transitions have to be tested for each def-use-pair. This combination appears artificial, and the combined test goals are complex. We just want to point out that more complex combinations than pair-wise are possible.

The presented combinations are all static. Furthermore, there are also ways to combine the mentioned coverage criteria dynamically. For instance, the test goals of one coverage criterion would have to be satisfied for each test case that was generated to satisfy a test goal of another coverage criterion. The combination of boundary value analysis and abstract test case generation in Chapter 3 is one example. Since satisfying level-3-combinations of control-flow-based, data-flow-based, or transition-based coverage criteria also results in abstract test cases, boundary-based coverage criteria can also be combined with level-3-combinations of the mentioned three kinds of coverage criteria.

4.4.2 Coverage Criteria Definitions

In this section, we describe how to use model transformations to support the definition and implementation of new coverage criteria. We also present a few examples for such definitions.

First, we define the new coverage criterion *All-Subsequent-Transition-Pairs*. The purpose of this coverage criterion is to cover all transition pairs that can be executed subsequently. This includes all adjacent transitions but also transitions from parallel regions or transitions on different hierarchy levels. Figure 4.37 shows the formal definition of All-Subsequent-Transition-Pairs using the definitions of Section 2.4.2. The definition is similar to the one of All-Transition-Pairs on page 45. The first difference is the consideration of all states of one configuration to include parallelism. The second difference is the consideration of the outgoing transitions of superstates and the outgoing transition of a composite state's initial and history states to include hierarchy.

```
P(TG) All-Subsequent-Transition-Pairs(SM sm) {
  testgoals = All-Transition-Pairs(sm);
  for all state configurations c of sm {
    S = set of all states that are included in c;
    S = S and all initial and history states of the states in S;
    S = S and all superstates of the states in S;
    for all incoming transitions t1 of states in c {
      for all outgoing transitions t2 of states in S {
        testgoals.add(new ATG( ((?, ?, ?, {t1}), (?, ?, ?, {t2})) )); }}}
  return testgoals; }
```

Figure 4.37: Definition of All-Subsequent-Transition-Pairs.

Combining all regions into one and flattening the state machine would be a alternative model transformation. On the transformed test model, all

subsequently executable transitions are also adjacent. Consequently, the satisfaction of All-Transition-Pairs on the transformed test model corresponds to the satisfaction of All-Subsequent-Transition-Pairs on the original test model – All-Transition-Pairs simulates All-Subsequent-Transition-Pairs. Likewise, such criteria can be defined for any length of transition sequences. We call the corresponding coverage criterion *All-n-Subsequent-Transitions*. By flattening the state machine, All-n-Transitions simulates this new coverage criterion.

Furthermore, it might be beneficial to combine guard value assignment considerations of several transition guards. For instance, the guards of pairs of adjacent transitions could be considered – all feasible combinations of their guard value assignments could be tested. We define a corresponding coverage criterion *Multiple Condition Coverage Pairs* in Figure 4.38. Multiple Condition Coverage Pairs subsumes Multiple Condition Coverage.

```
P(TG) MultipleConditionCoveragePairs(SM sm) {
  testgoals = MultipleConditionCoverage(sm);
  for each transition t1 in sm {
    for each value assignment condition cva1 for the guard of t1 {
      tExec = one transition that is active for cva1 and one event of t1;
      for each outgoing transition t2 of the target state of tExec {
        for each value assignment condition cva2 for the guard of t2 {
          testgoals.add(new ATG( (({t1.source}, t1.events, cva1, tExec),
                                  ({t2.source}, t2.events, cva2, ?)) ));
  } } } }
  return testgoals; }
```

Figure 4.38: Definition of Multiple Condition Coverage Pairs.

A corresponding model transformation corresponds to the one of simulating Multiple Condition Coverage wih All-Transitions (see Section 4.3.3 on page 134). The transformed test model would contain transitions that correspond in sum to all guard value assignment of the original test model – one transition for each assignment. The satisfaction of All-Transition-Pairs on the transformed test model would result in testing all pairs of guard value assignments for guards on adjacent transitions.

There are many more examples for new and feasible coverage criteria definitions. Figure 4.39 shows a subsumption hierarchy that includes the three newly defined coverage criteria into the existing subsumption hierarchy. Note that all new coverage criteria can be simulated with existing ones.

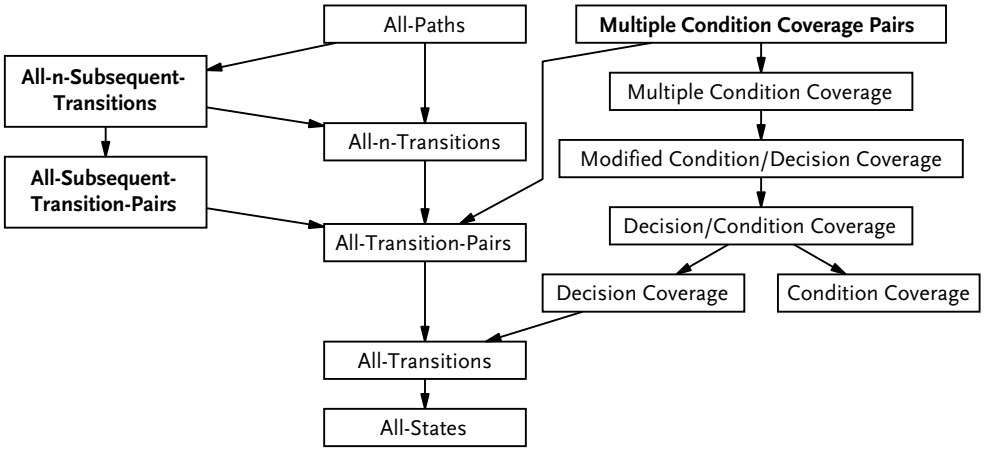


Figure 4.39: Subsumption hierarchy with new coverage criteria.

4.4.3 General Considerations

In this section, we present general considerations about the impact of test model transformations on coverage criteria satisfaction. We presented several transformations that yield coverage criteria simulation relations. All presented transformations had a positive impact, i.e. the satisfaction of a certain coverage criterion on the transformed test model always implied the satisfaction of a stronger or a different kind of coverage criterion. General examples for test model transformations with a positive impact are the flattening of hierarchical state machines, the removal of parallel regions, or the creation of unmodeled self-transitions to support sneak path analysis. Such transformations can be used to increase the effect of several coverage criteria. However, there is no recommendation to apply certain model transformations. Testing is still risk management. This also includes the use of test model transformations with the corresponding gains and costs.

On the other hand, there are also coverage criteria for which the application of certain model transformations results in disadvantages. For instance, inserting nodes into transitions as presented in Section 4.2.2 can be disadvantageous for the coverage criterion All-3-Transitions. Whereas its application to the original test model results in testing all triples of adjacent transitions, its application to the target model corresponds only to the satisfaction of All-Transition-Pairs on the original model. This holds for further transition-based coverage criteria. As another example, Rajan et al. [RWH08] show that splitting guards into several ones can result in disadvantages for the satisfaction of control-flow-based coverage criteria. As a result, transformations should be selected carefully for each individual pair of coverage criteria.

Finally, there are test model transformations that have a positive impact on some coverage criteria and test model transformations that have a negative impact. Creating a complete classification of test model transformations or the corresponding test models is left to future work.

4.5 Related Work

Model transformations have been used for testing before. For instance, Friske and Schlingloff [FS07] instrument conditions of the state machine's transitions to simulate (although they used different terms) All-Transition-Pairs with MC/DC: The authors add variables to the test model so that the satisfaction of MC/DC on the adapted test model has the same effect as the satisfaction of All-Transition-Pairs on the original test model. Badban et al. [BFPT06] use semantic-preserving model transformations to simulate MC/DC with Decision Coverage. Ranville [Ran03] adapts the test model to simulate MC/DC with All-Transitions: He splits transitions like presented in Section 4.2.2. As discussed previously, however, MC/DC is a complex coverage criterion that cannot be simulated with All-Transitions or Decision Coverage without unnecessary test effort. Thus, the existing approaches to simulate MC/DC are either too optimistic to be generally applicable or result in more effort than necessary to satisfy MC/DC. Rajan et al. [RWH08] examine the impact of the model's and the program's structure on the satisfaction of MC/DC. Santiago et al. [SVG+08] describe the flattening of state machines or state charts with a reachability tree generator. The satisfaction of All-Transitions on this reachability tree implies the satisfaction of All-Configurations on the original state machine. In contrast, this chapter is not focused on defining single relations between coverage criteria, but on showing the general relations between model transformations and coverage criteria: Coverage criteria satisfaction depends on the structure of the test model. This can be used to simulate coverage criteria, combine them, or define and implement new ones. Furthermore, Harman et al. [HHH+04] consider source code transformations (*testability transformations*) as a means to support evolutionary test generators. The presented transformations are focused on flag removal, which consists of converting two-valued variables into many-valued variables, whose values can slowly approximate a certain optimum. In contrast to our work, however, they do not use transformations as a means to influence the effect of coverage criteria. This chapter is furthermore not focused on test generation algorithms. Instead, the existence of such algorithms is assumed. Since most commercial test generators support the satisfaction of a limited set of coverage criteria [BSV08], the test model

transformations that yield simulated satisfaction relations are of high value. In [Wei09b], we present that model transformations can be used to make coverage criteria interchangeable. The report of the industrial cooperation has been published in [Wei09a].

Model transformations are used to convert one model into another model. In order to apply model transformations to all models defined by a meta model [Fav], the transformations are defined at the meta model level. There are several connections between testing, model transformations, and meta models. For instance, we propose a technique to test meta models based on mutation [SW08]. Küster [Küs06] presents an approach for the systematic validation of model transformations by translating models to text and applying rule-based coverage criteria to them. In [CH03], Czarneki and Helsen compare different approaches to model transformations. Wang et al. [WKC06] validate model transformations and apply coverage criteria to the used source and target meta model of the model transformation. Brottier et al. [BFS+06] propose to apply coverage criteria to meta models in order to generate tests for model transformations. There are many transformation languages and tools that support model transformations, e.g. ATL [Ecl09] or QVT [IKV]. These languages should be considered for the implementation of the proposed test model transformations. In contrast to the cited work, this chapter is focused on the use of model transformations to support testing instead of testing model transformations.

Several commercial model-based test generators are based on UML state machines and follow the approach of applying a coverage criterion to the test model to create a set of test-model-specific test goals. For instance, the Smartesting Test Designer LTD [Sma] supports All-Transitions and handles each transition as a target. IBM Rhapsody ATG [IBM] creates test suites and measures coverage criteria satisfaction with calculated test goals [IBM04].

4.6 Conclusion, Discussion, and Future Work

Here, we conclude this chaper, discuss important aspects, and present future work.

4.6.1 Conclusion

This chapter is focused on the importance of test model transformations for coverage criteria in model-based testing with deterministic flat untimed UML state machines. Our main contribution is the definition of simulated coverage criteria satisfaction. We presented several pairs of coverage criteria $(cc1, cc2)$

together with model transformations mt for which $cc1$ applied to each target model of mt simulates $cc2$ applied to a corresponding source model of mt. The presented simulation relations are joint into a corresponding simulated satisfaction graph that contains the known subsumption hierarchy. This graph contains cycles and shows that each feasible coverage criterion can be used to simulate any other feasible coverage criterion. We used the formal definitions of coverage criteria in Section 2.4.3 to prove the correctness of the presented model transformations. The results of an industrial cooperation show the importance of test model transformations for realistic scenarios.

One interesting aspect of simulated satisfaction of coverage criteria is that stronger coverage criteria can be simulated with weaker ones. This is especially useful for model-based test generators that satisfy only a restricted set of rather weak coverage criteria. For instance, Smartesting Test Designer [Sma] only supports the satisfaction of All-Transitions. The presented model transformations enable users of this tool to satisfy, e.g., Multiple Condition Coverage, All-Configurations, or All-Uses. We consider this an important contribution for existing model-based test generation tools. Furthermore, the impact of coverage-criteria-based comparisons of model-based test generation tools like presented in [BSV08] should be reconsidered. We defined several kinds of coverage criteria combinations. The satisfaction of All-Transitions is sufficient to satisfy a level-3-combination of, e.g., control-flow-based and data-flow-based, coverage criteria. With the approach presented in Chapter 3, it is furthermore possible to combine these criteria with boundary-based coverage criteria like Multi-Dimensional.

As we also presented in Section 4.4.3, test model transformations do not necessarily influence model-based test generation in a positive manner, but can also result in disadvantages. So, they have to be chosen carefully.

4.6.2 Discussion

The presented model transformations and the obtained results leave room for discussion. For instance, coverage criteria have properties aside from the fault detection capability, e.g., the necessary test effort to satisfy them. The presented test model transformations, however, are just focused on efficient results for certain pairs of coverage criteria. As an example, we presented a model transformation to show that All-Transitions simulates Multiple Condition Coverage. Multiple Condition Coverage subsumes MC/DC and, thus, All-Transitions also simulates MC/DC. However, Multiple Condition Coverage requires exponential test effort whereas MC/DC requires only linear test effort relative to the condition size. Correspondingly, the transformation for simulating Multiple Condition Coverage with All-Transitions leads to expo-

nential space complexity depending on the transition guards. All transformations are based on adding elements to the test model. Thus, test model transformations should be selected carefully to prevent them from adding unnecessary complexity to the test model. The best solution is to define a separate model transformation for each pair of coverage criteria. There may be exceptions like MC/DC, for which we showed that its simulation with All-Transitions results in unnecessary complexity.

Our aim was not to create efficient model transformations, but to show their impacts at all. Beyond that, each model transformation is aimed at transforming the structure of the state machine so that the test goals of a complicated coverage criterion are presented in a way that also simple coverage criteria include them. Thus, we think that the number of test goals is almost unchanged by the model transformation. At least for the presented coverage criteria simulation relations, we think the corresponding model transformations do not result in unnecessary test effort.

Another interesting question is whether there are further relations between the original and the transformed state machine. For instance, strong or weak (bi-)simulations [Par81] are means to compare state-based behavioral descriptions. Since there are several transformations that insert states in the state machine, strong (bi-)simulation does not have to be satisfied. Furthermore, there are basic transformation patterns like *Move Effect* that spread the behavior of one transition to two transitions. Thus, the transitions really differ and, thus, also weak (bi-)simulation does not necessarily have to be satisfied. The fundamental reason is that the transformations are focused on preserving the semantics of compound transitions and not of single ones. If we interpreted bisimulation correspondingly, there could be corresponding relations between the original and the transformed test model.

We presented All-Transitions as the minimum coverage criterion to satisfy and showed how to use it to simulate other coverage criteria. It was also very interesting to simulate All-Transitions with even weaker coverage criteria like All-States [UL06, page 117], e.g., by applying the transformation pattern *Insert Node in Transition*: A new choice pseudostate is inserted in a transition so that it is visited if and only if the transition is traversed. Since each feasible coverage criterion can be used to simulate any other feasible one, another result is that there is no need to define a minimal coverage criterion to satisfy for an appropriately transformed test model.

Furthermore, there may be interpretations of All-States that require to visit all states but no pseudostates. In this case, the presented model transformation to establish the simulated satisfaction of All-Transitions with All-States is not sufficient. The main reason for this are compound transitions [Obj07, page 568] that represent semantically complete paths from one

state to another and that may contain pseudostates. They can only be extended by inserting pseudostates, but not by inserting states. The insertion of a real state may also be sufficient because completion transitions have a higher priority than explicitly triggered transitions [Obj07, 569]. Furthermore, most commercial model-based tools are able to satisfy at least All-Transitions and, thus, this issue is no threat to this chapter's contribution. Moreover, we can think of a model transformation that creates a tree-like state machine with at most one incoming transition ($|v.incoming| \leq 1$) for each vertex v: for each test goal tg of any feasible coverage criterion (with a finite test goal set) there could be a path with a separate target state to achieve tg. Although this might result in large test models with no room for efficient test suite generation, the satisfaction of All-States on the transformed test model would be sufficient to satisfy any other feasible coverage criterion – or even an infeasible one in a step-wise online test generation process.

Moreover, we consider model transformations as an answer to many open discussions about coverage criteria. For instance, the existence of composite states leads to discussions about the interpretation of All-Transitions: Are outgoing (high-level) transitions [Obj07, page 568] to be traversed for each included substate or just once? By applying a model transformation that flattens the state machine, this question can be settled. Furthermore, All-Transition-Pairs is defined for adjacent transitions. As we explained for the case study, there are pairs of subsequently traversed transitions that are not adjacent. We also sketched a corresponding model transformation that transforms initial states into entry points. Many alike discussions can easily be abolished by transforming the used test model and applying a formally defined coverage criterion to the resulting target test model.

We restricted ourselves to flat UML state machines to show the feasibility of our approach without making the presented transformations too complex. Since state machines can be flattened, however, this means no general restriction to our approach. Hierarchical state machines contain some elements that result in a higher effort. For instance, transitions inside a composite state have a higher priority than transitions on the outside of that state [Obj07, page 561]. The transformation shown in Section 4.2.2 can insert new transitions for missing guard conditions of internal transitions. These new transitions, however, would override outer transitions. A more complicated transformation would be necessary to show that these new internal transitions have the same effect as the existing outer transitions. The same holds for tricky elements of composite states like history states.

The application of constraints to the test model may result in problems. For instance, the OCL expression *oclIsInState* [Obj05a, page 139] may refer

to states for which copies are created (see pattern *Copy Vertices*). Although these copies show the same behavior as the original state, they are not the original state and, thus, the OCL expression that returns *true* for the original state returns *false* for a copy. A solution may consist in moving the original state and all its copies into a new composite state, which is named like (and identified with) the original state. Since all the moved states are substates of this composite state, the *self* object is the composite state if one of the moved state is active. Thus, the *oclIsInState* will return *true* for all of them. There may be further issues like time events, for which we are also confident that solutions based on model transformations can be found.

Model transformations can be used to increase the generated test suite's fault detection capability and size. As stated in Section 4.4.3, testing is a risk management. Thus, the additional gain has to be traded against the additional cost, and there can be no general recommendation for applying certain model transformations.

4.6.3 Future Work

We presented theoretic results about the effects of test model transformations on coverage criteria satisfaction and coverage criteria combination. Some of the presented test model transformations are already implemented in the prototype test generation tool ParTeG. A corresponding industrial case study showed the advantages of these transformations. In order to make these advantages available to users of commercial test generators, we plan to create an independent test model transformator that can be applied in a test generation preprocessing step. For that, we already created the sourceforge project *Coverage Simulator* [Weia] and started the work. Our goal is to implement all proposed model transformations and to make them available to users of model-based testing tools.

This thesis is focused on model-based test generation and, in particular, on test models and coverage criteria that are applied to them. The presented contributions can also be transferred to other testing fields like source code-based testing. An interesting work would be an automatic source code transformator that transforms the source code of a SUT in order to support the application of coverage criteria to source code.

Coverage criteria can be compared by subsumption relations. There are several subsumption hierarchies. We presented several model transformations and showed some of their positive and negative impacts. We think that a classification of test models according to their structures and the transformations that are advantageous when satisfying a certain coverage criterion may also be beneficial. Accordingly, we plan to compare test model struc-

tures with respect to certain coverage criteria to satisfy. One result would be a complete coverage-criterion-oriented survey of advantageous and disadvantageous model transformations.

Chapter 5

Test Model Combination

In this chapter, we propose the combination of several UML models for automatic test generation. Such combinations have the advantage of providing more information to the test generation process and, thus, increasing the generated test suite's fault detection capability. We consider this chapter the third contribution of the thesis. It enhances all already presented contributions. The presented approaches are partly implemented in the test generator ParTeG and were already used for automatic test generation.

We focus on two scenarios. In both scenarios, UML state machines play an important role. First, we describe the combination of state machines and class diagrams. The approach of Section 3 already included the automatic test generation from a state machine that describes the behavior of one class. Here, we extend this approach to a class hierarchy represented in a class diagram. Second, we describe the combination of state machines with interaction diagrams. Interaction diagrams are often used in combination with use cases and are, therefore, closer to requirements than state machines. We show how to use state machines to combine interaction sequences automatically. The two approaches are quite different according to the used techniques, the intentions, and the achieved goals. For that reason, we present related work and discussion separately.

5.1 State Machines and Class Diagrams

In this section, we describe the combination of UML state machines with UML class diagrams. The idea of combining state machines and class diagrams is to reuse state machines with context classes that are derived from the original class. This approach can be especially useful for test generation in the context of product lines.

This section is structured as follows. In Section 5.1.1, we provide an introduction to the combination of state machines and class diagrams. We introduce state machine inheritance as a way of using state machines and class diagrams together in Section 5.1.2. We present related work in Section 5.1.3 and conclusion, discussion, and future work in Section 5.1.4.

5.1.1 Introduction

In this section, we provide an introduction to the combination of state machines and class diagrams. First, we introduce feature models as a means to describe product lines. Then, we present an example of a product line for a car audio system. This example is not listed in the previous chapter because it is only used to clarify the approach, but no test suites have been created from it. Finally, we present "150% models" as a means to present more information in one model than could ever be used in one execution.

Feature Models.

A product line consists of products, e.g., software or hardware with essential similar features. The dissimilar features are known as product variation points. Activating or deactivating different variation points results in different product variants. Product lines are common in mass production. German car industry provides a good example for the possible complexity of such product lines: Given the number of possible car product variants and the number of sold cars, statistically, for each product variant there is only one car. If a feature should be used in a certain product variant, the feature is said to be *active*. Features can be activated in all products or just some of them. The activation of one feature can prohibit the activation of another one. Correspondingly, the possible features are classified as either common, optional, or alternative. The application of product lines to software products results in software product lines (SPLs), which are sets of similar software products. Feature models are a popular means to describe features of a product line [LKL02]. They are used to relate the basic product and the features. For developing and testing product lines, it is beneficial to deal with their similar features prior to their dissimilar features. SPLs are subject to model-based testing [McG01, McG05, OG05].

Figure 5.1 depicts an example feature model. The uppermost rectangle with the name "Product" describes the basic product feature. All other rectangles represent features or subfeatures. The arcs between the rectangles depict relations between features: A filled black circle depicts a common feature, i.e. the feature at the end of the black circle is always active if the

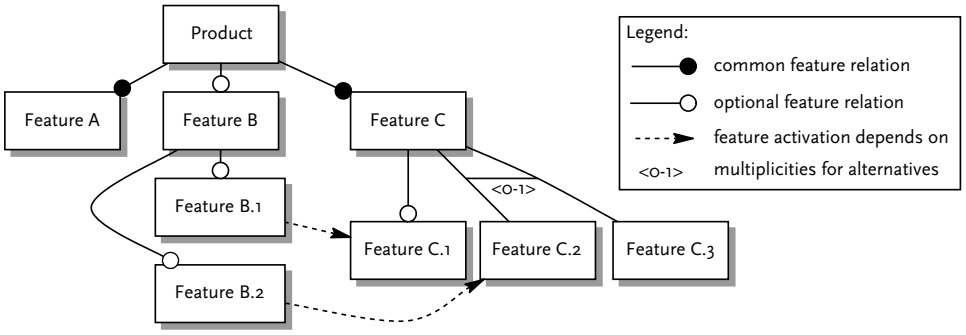


Figure 5.1: Example of a feature model.

other feature is active, too. For instance, "Feature A" is active in every product variant. An empty circle represents an optional feature, i.e. the feature can be active but does not have to. In Figure 5.1, "Feature B" is optional. Dotted arcs with an arrow depict dependencies between feature activations. For instance, "Feature B.2" can only be activated if "Feature C.2" is activated. Arcs without circles and arrows represent alternatives. The attached numbers show possible multiplicities. In the example, zero or one feature among "Feature C.2" and "Feature C.3" can be activated.

Example: Product Lines for Car Audio Systems.

Here, we present an example for the combination of state machines and class diagrams: a car audio system. Such a system has several possible features, many of which are common to all product variants. Therefore, the car audio system can be appropriately described with a feature model. Figure 5.2 shows a feature model with some reasonable features of car audio systems.

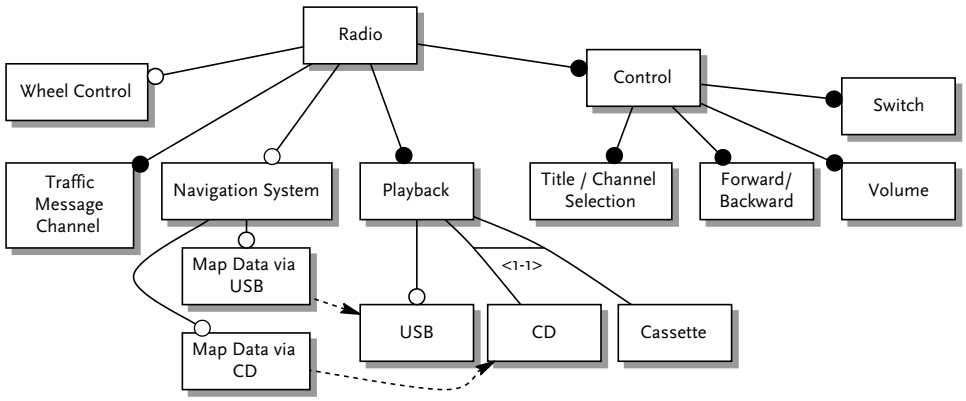


Figure 5.2: Feature model of car audio systems.

For instance, the basic controls are common to all radios: A radio must provide an option to switch its mode, e.g., to toggle radio and playback media. It must allow the selection of channels or tracks, the change of the volume, and the search for new channels or new tracks. Depending on the current mode and received events (e.g. emergency messages via the traffic message channel), the basic controls like forward and backward search can have different meanings. Furthermore, the choice of playback media (CD or cassette player) also influences the behavior. Whereas a forward event for a cassette player results in winding the tape, a forward event for a CD player results in the immediate selection of the next track. The traffic message channel is common to all car audio systems. The availability of a USB port, the wheel control, and the navigation system are optional features. Depending on the playback media, the update of map data for the navigation system is an optional feature.

150% Models.

Testing consumes a large amount of the overall development costs. High effort is put into keeping costs for testing as low as possible. In model-based testing, test maintenance costs are comparably low. The test model creation effort is high. The reuse of certain artifacts, e.g., like test suites in regression testing, helps reducing development and maintenance costs. In the field of model-based testing, the reuse of test models also significantly contributes to reducing such costs.

As stated above, industrial product lines from the automotive domain can contain thousands of product variants. Creating and maintaining all corresponding behavioral and structural models is infeasible. Instead, 150% models are proposed as a solution for reusing models [GKPR08, DW09]: A 150% model contains more information than necessary for one instance. It integrates all the information of all product variants. Separate configuration files supplement these models and determine the actually used model elements. In [DW09], the use of 150% models is illustrated with real applications.

A state machine describes the behavior of a class. The goal of combining state machines and class diagrams is to create 150% state machines and use classes to configure them. Classes can influence the behavior of state machines with their attributes (e.g. for guard conditions), their operations (e.g. for the effect of state machine transitions), or their relations to other concrete classes and their properties.

5.1.2 State Machine Inheritance

UML state machines [Obj07] are often used to model the behavior of product variants. In our context, they are used as test models to automatically generate test suites. There are many ways to reuse these state machines as test models in order to reduce the costs for test development and test maintenance. In this section, we present two ways to reuse UML state machines for test case generation: the one proposed by the Object Management Group (OMG) and our own approach. We sketch advantages and disadvantages of both approaches in the context of product line testing.

The OMG proposes to generalize and specialize state machines for UML. Rationale of this is to enable the redefinition of a general classifier's behavior [Obj07, page 561]: *"A specialized state machine is an extension of the general state machine, in that regions, vertices, and transitions may be added; regions and states may be redefined (extended: simple states to composite states and composite states by adding states and transitions); and transitions can be redefined."* Furthermore, the specification [Obj07, page 562] states: *"A submachine state may be redefined. The submachine state machine may be replaced by another submachine state machine, provided that it has the same entry/exit points as the redefined submachine state machine, but it may add entry/exit points. Transitions can have their content and target state replaced, while the source state and trigger are preserved."*

In principle, the specialized state machine is read similarly to "non-specialized" state machines. Interesting features can be added when it is necessary and, consequently, agile development is supported. Nonetheless, some important problems are unsolved. For instance, transitions can be redefined but there is no way to mark them as reused. Consequently, all transitions have to be re-drawn in the specialized state machine. Changes can have significant effects on the whole state machine. The relations between the general and the specialized state machine are not defined. Consequently, the effects of changes in the general state machine on the specialized state machines are also undefined: Does the change of a state in the general state machine also change the corresponding inherited state in the specialized state machine? Do reused elements, in turn, reference their general original? Submachine state machines, transitions, and regions can be replaced, added, or redefined. Summing up, the proposed way to specialize state machines is focused on structural aspects and behavioral aspects are not considered. Even for small examples, it is obvious that the development as well as the maintenance of such descriptions are costly because the basic behavior has to be adapted and re-drawn for each product variant. To our knowledge, there is no tool support for this approach.

Our approach to reuse state machines leaves the state machine unchanged but changes its context class. As specified in [Obj07], the context of a state machine is a class; transitions of the state machine can refer to properties and operations of this class via events, guards, and effects. Instead of using an inheritance relationship between state machines, we propose to use the inheritance relationship between classes and reuse a state machine as a behavioral description of these classes. For that, we define one state machine as a 150% model for a general class. Since all specialized classes contain the same operations, attributes, and associations, this state machine can also describe the behavior each specialized class. A motivation for this approach is Liskov's substitution principle [Lis88]. This principle states that all properties of a class also have to hold for its subclasses. Obviously, the behavior of a class is such a property. Since a state machine is the model of a class' behavior, it is also a model of a class' properties and can, thus, be a model for the behavior of each subclass. Consequently, each subclass of the state machine's originally defined context class is a possible context class (see Figure 5.3).

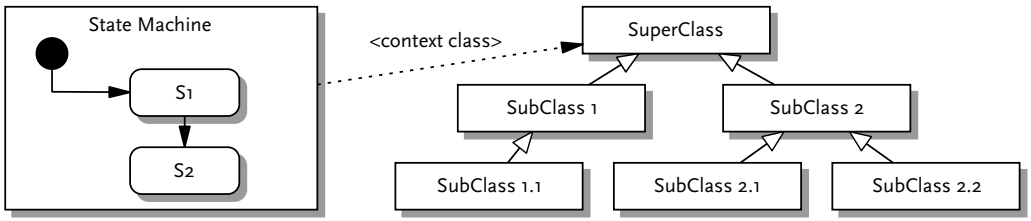


Figure 5.3: Each class can be the context of the state machine.

In the test generation approach presented in Chapter 3, the referred properties and the postconditions of the operations in the selected context class influence the behavior of the state machine via guard, transition effects, and state invariants. Consequently, the behavior described by the state machine depends on the selected context class and the modeled behavior changes with the selection of a new context class. By using submachine states, the clarity of the test model is kept even for large systems. Figure 5.4 shows a reusable state machine for our example, and Figure 5.5 shows corresponding context classes. Note that our approach is not only feasible for testing one class, but also for systems with many classes. For each product variant, each active class is then described by a state machine.

This state machine describes a part of the car audio system's complete behavior. Each car audio system is in one of two states: *On* or *Off*. The state *On* is a composite state and contains two regions. The upper region deals with the common feature *Switch* between sources of the current playback: Users can *switch* between several sources. A *TMCEvent* is triggered if a

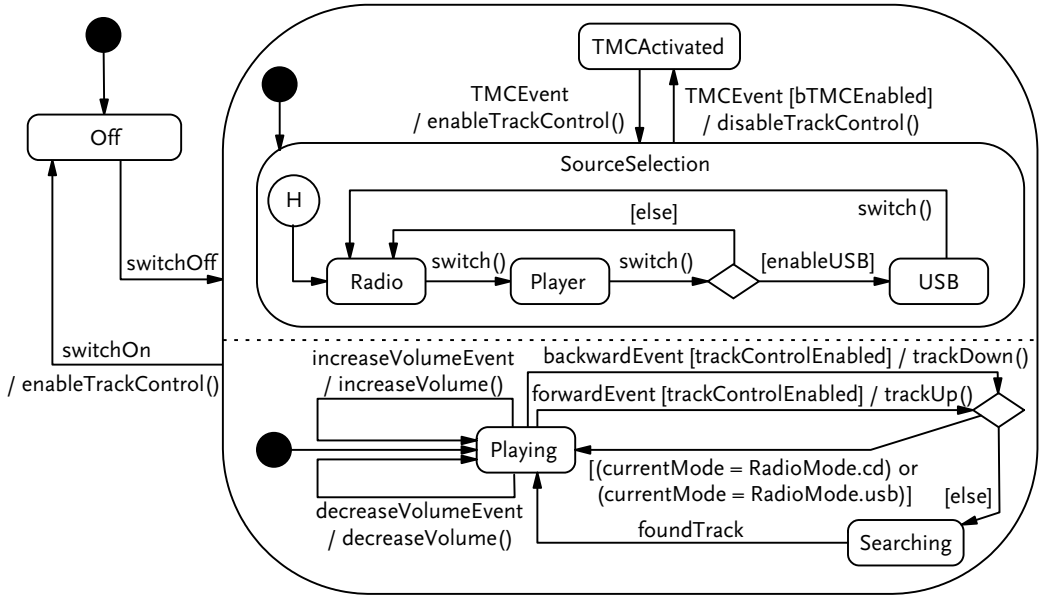


Figure 5.4: A state machine describing an extract of the general car audio system behavior.

traffic-relevant message is received. If the traffic message control is activated ($bTMCEnabled = true$), then the input media will change to the traffic message channel. If the message broadcast is finished or the radio is turned off and on again, then the radio is reset to the state *SourceSelection*. The lower region describes some of the remaining common control features, like volume control or track control. Both regions are handled in parallel. Depending on the active states of the car audio system, the effects of the events differ. For instance, if the radio receives a traffic message, the volume is set to a higher value and the track control is disabled. Receiving the *switch* event in the state *Player* results in entering the state *USB* iff the value of *enabledUSB* is true, which depends on the selected context class. Furthermore, the handling of the features *Forward* and *Backward* depends on the current product variant: For USB and CD, the corresponding events result in the immediate selection of a new track; all other modes include *searching* (scanning for radio, winding for cassette player).

As described above, product variants are described with single classes. The class diagram in Figure 5.5 shows several context classes, two of which describe one product variant, each: *DefaultRadioConfiguration* and *ComfortRadioConfiguration*. The class *AbstractRadioConfiguration* contains all attributes and operations that are referenced by the state machine. Each specialized class describes one product variant and can redefine the attributes

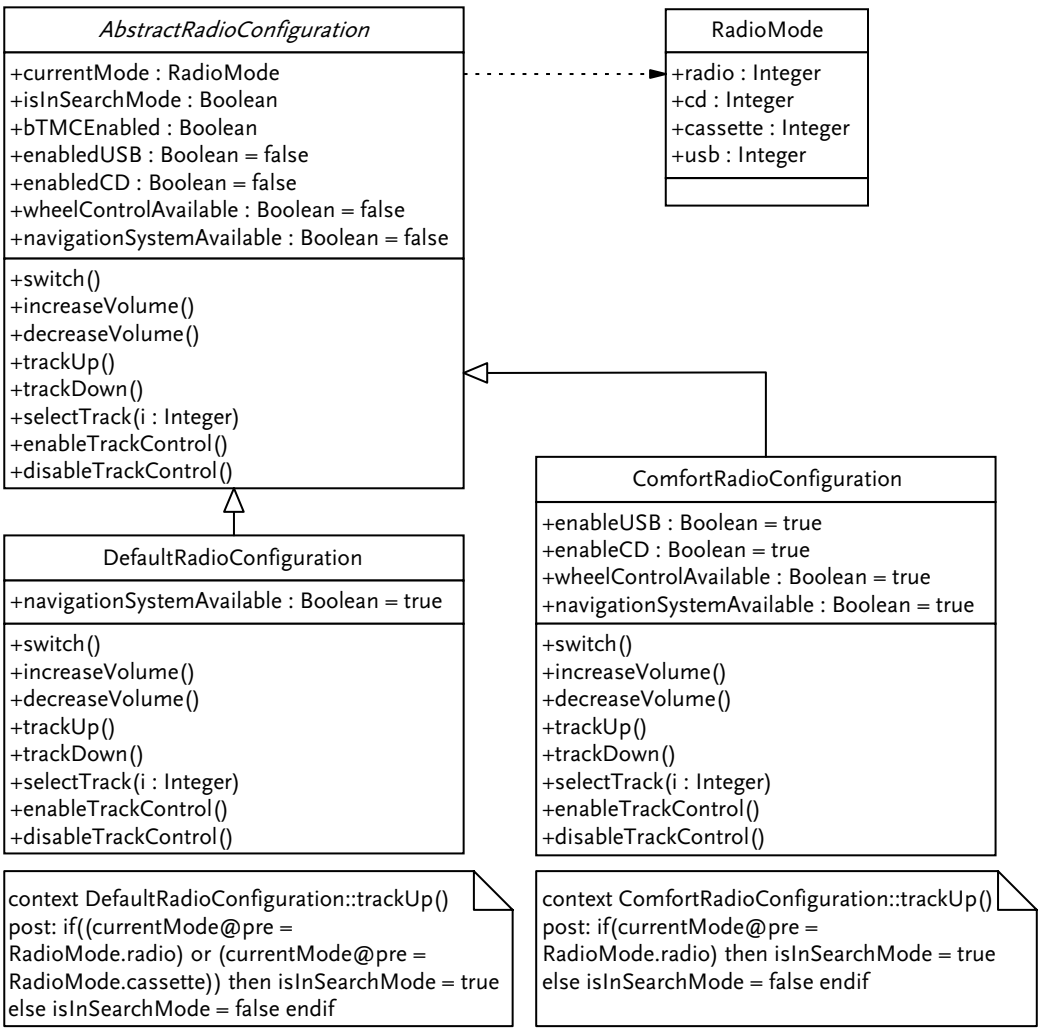


Figure 5.5: Extract of classes describing two product configurations for a car audio system.

and operations of the base class necessary to adapt the behavior. For instance, the default values of *enabledUSB* show that USB support is activated for the configuration *ComfortRadioConfiguration*, whereas *DefaultRadioConfiguration* offers no support for USB. Changed operation effects are expressed by different postconditions.

5.1.3 Related Work

In this section, we present related work on the combination of state machines and class diagrams. State machines have often been used for automatic test generation [CK93, BDAR97, OA99, SHS03, BLC05]. In this section, however, the focus is on the combination with class diagrams for product lines. There are many model-based approaches that aim at reducing test execution costs. We think that it is possible to combine such techniques with our technique to further reduce test costs.

We introduced a possibility to reuse test models for product lines. In order to derive tests for a product variant, a behavioral description of this variant is required. For instance, Kishi and Noda [KN04] model one state machine for each feature-supporting component. Kahsai et al. [KRS08] concentrate on adding features to an existing software product line. Geppert et al. [GLRW04] introduce a decision model for feature selection and use it for test selection. In our approach, we describe the behavior of all product variants with one state machine. There are several other approaches focused on product lines. For instance, El-Fakih et al. [EFYvB02] consider testing during product life cycle. They reduce the test effort by testing only the modified elements of a new product version. Since different product variants of a product line may be developed simultaneously, this approach is not able to reduce test costs for product variants of product lines. In contrast, our approach allows to test all products of a product line based on one behavioral 150% test model. However, a combination of our approach with the approach of El-Fakih et al. seems to be reasonable. In [McG01], McGregor points out the importance of a well-defined process for testing software product lines. Kolb [Kol03] discusses the problem of selecting a suitable software product line testing strategy that takes the reuse of variable elements into account. As an extension to that, we focus on the reuse of models for automatic test generation to reduce test development costs. Furthermore, Gomaa [Gom04] introduces the PLUS models and method. He uses Boolean feature conditions to activate product line features. In contrast, our approach is not restricted to Boolean values or complete activation of features, but is also able to adapt feature details deep inside the behavioral specification. Olimpiew and Gomaa [OG05] deal with test generation from product lines and inter-

action diagrams. In contrast to that, we focus on UML state machines, OCL constraints, and class inheritance relationships. Pohl and Metzger [PM06] discuss the advantages of software product line testing and emphasize the preservation of variability in test artifacts. As we generate test cases from reused models automatically, this variability is preserved. Most interesting, Dziobek and Weiland [DW09] implemented a similar approach to ours for Simulink models one year after the publication of our approach [WSS08]. They also create 150% models and use external files to configure the single product variants, and they apply this approach to industrial projects.

Liskov [Lis88] defines the substitution principle for types and subtypes. We apply this approach by interpreting a state machine as a property of its context class and using it to describe the behavior of this class' subclasses. As a consequence, the state machine has to describe the behavior of all product variants, which makes it a 150% model. McGregor and Dyer present a note on inheritance and state machines in [MD93]. They show how to build a state machine for a class incrementally by assembling the state machines of the class' base classes. The authors claim that this approach reduces the modeling effort. Our approach differs in that we do not focus on the creation of the test model but on its use for automatic test generation for product lines. However, the approach can be combined with our approach to create the test models before generating test suites.

5.1.4 Conclusion, Discussion, and Future Work

Here, we present the conclusion of this section, a corresponding discussion, and possible future work.

Conclusion.

In this section, we proposed a new technique to reuse state machines for automatic model-based test generation. This reuse is enabled by inheriting state machines along inheritance relationships between classes: A state machine describes the behavior of a class. Behavior is a property of the class and can be inherited to the class' subclasses. The main idea is to create a state machine as a 150% model and use its possible context classes as configurations. The details of each configuration are expressed as elements that are referenced from the state machine, e.g. OCL pre-/postconditions of operations or default values of attributes of the context classes. We applied the approach to generate test suites for product lines. Each product variant can be described by a configuration class. Corresponding class hierarchies can also be created. We implemented the presented approach in ParTeG.

Discussion.

The presented approach supports the reuse of test models. Together with automatic test generation, a reasonable amount of test effort can be saved by using the presented method. Corresponding case studies still have to be carried out. Here, we present some additional points to discuss.

For instance, the approach of the OMG to inherit state machines is only focused on structural aspects of the state machine. It neglects the behavioral semantics of the state machine and, thus, Liskov's subsitution principle [Lis88]. In comparison to that, we reuse the same state machine and let details of it be influenced by settings of the selected context class. Theoretically, these aspects should also be conform to Liskov's principle. However, if they are not, our approach faces the same problem as the one of the OMG: Two state machines that describe totally different behaviors.

Another point to discuss is that the generated test suite for a particular context class is only applicable to this certain class. Thus, this test suite can only be used for one product variant. Our approach, however, aims at the reuse of test models instead of test suites. A whole new test suite can be generated automatically from the same state machine for each context class. Consequently, a set of test suites can be automatically generated for a set of context classes.

Another important issue is the derivation of class hierarchies from product lines. Is it possible to derive a class hierarchy similar to the hierarchy between features of a feature model? For the example in Figure 5.2, it is possible to define a basic class with all optional features deactivated. Therefore, two subclasses can be derived by activating either the feature *Wheel Control* or *Navigation System*. The definition of a class that activates both features seems to be an issue. For reasons of manageability, it is often undesirable to derive every class directly from the base class. Thus, the new class should inherit from the classes defined for the features *Wheel Control* and *Navigation System*. This leads to multiple inheritance, which has some disadvantages connected to the redefinition of class operations. Such problems are well known from C++. Another solution might be the use of aspect orientation [WFPW07], where each feature corresponds to a certain aspect.

Our approach includes the complete specification of all features in one state machine. The modeling effort for a state machine describing all product line features exceeds the effort for a state machine describing just one product variant. On the one hand, this means higher initial modeling effort because all dependencies between product variants are included. On the other hand, this is advantageous for two reasons: First, the inclusion of all features in one model forces the modeler to think about the SUT more thoroughly.

Second, the state machine is unchanged and reused for each product variant, which reduces the modeling effort significantly. We assume that this requires considerably less effort than modeling and maintaining an individual state machine for each product variant of a product line. One reason is that common features have to be modeled just once overall instead of once per product variant.

Testing product variants via models that are developed for just one product variant faces several problems. For instance, the accidental inclusion of a feature in the SUT cannot be discovered because the model contains no information about that feature. Pohl and Metzger [PM06] confirm this problem. Since our approach generates test cases for the whole state machine with all feature descriptions included, the generated test suites can discover corresponding failures.

Furthermore, the created 150% state machine can be quite large. The question arises whether it can still be manually created and understood by testers. As a solution, state machines can be nested, which provides the possibility to separate parts of the state machine in submachine state machines [Obj07]. In distributed development processes, these submachine state machines can then be worked on by different engineering teams. State machines can grow quite large, and using nested state machines is state of the art.

Finally, the presented explanations deal with each product variant as one class. In more complex systems, product variants are influenced by more than one class. Since our approach can be applied to each of these classes, however, this means no general restriction of our approach.

Future Work.

A possible future activity is to perform case studies to substantiate the advantages of our method. Another important aspect is the satisfaction of different coverage criteria that are based on state machines and class diagrams. How can coverage criteria be combined so that even product lines with large class diagrams can be tested with reasonable effort?

5.2 State Machines and Interaction Diagrams

In the previous section, we combined structural and behavioral diagrams. Here, we focus on combining two behavioral diagrams: state machines and interaction diagrams. Interaction diagrams are often used to describe use cases. Thus, they are close to requirements. In most cases, however, only a

small set of test cases can be derived from them. The proposed combination of state machines and interaction diagrams allows to automatically combine several interaction diagrams. We also present new coverage criteria that are focused on combined interaction diagrams.

5.2.1 Motivation

Model-driven engineering starts with requirements analysis. A requirements use case is often supplemented with an interaction diagram. Each interaction diagram can represent a few possible behavior interactions of the SUT, and there are usually only a few test cases for each use case. This supports traceability from requirements to test cases. For such reasons, interaction diagrams are popular to model test cases. One issue about interaction diagrams, however, is that they just consist of a sequence of interactions without any notion of state. Thus, determining the starting point for the execution of interaction diagrams is an issue [Nag04, Sok06b]. Furthermore, missing state information prevents the concatenation of interaction diagrams by, e.g., executing the described traces consecutively.

State machines are a more complex means to model behavior than interaction diagrams. In contrast to an interaction diagram, a state machine is used to describe a large and potentially infinite set of behavior traces. Coverage criteria are used as a stop criterion for test generation. Since the described behavior can be complex and the generated test cases are also determined by the used coverage criteria, the application of state machines makes traceability hard.

In this section, we present a technique to combine state machines and interaction diagrams in order to combine their advantages. In our approach, the behavior of interaction diagrams is retraced in state machines. The corresponding transition sequences are then combined based on state information. This approach can be seen as the test counterpart of sequence-based specification [PP03]. The contribution of this approach is the automatic concatenation of manually defined requirements specifications. There are further advantages:

First, by using state information of the state machine, this approach provides test oracles in the form of state invariants to test cases derived from interaction diagrams.

Second, the concatenation of interaction diagrams results in longer and possibly fewer test cases. In environments like embedded systems, the initialization of test cases causes higher costs than the test execution. Thus, the combination of many short test cases into a few long ones can save costs.

Third, the concatenation of interaction diagrams can result in the detection of faults that are undetected by the execution of single interaction diagrams. Concatenated interaction diagrams can be used to test the concatenated behavior of several sequences as well as their repeatability.

Finally, the presented possibility of interaction diagram concatenation allows to define new coverage criteria for the quality measurement of executing interaction diagrams. Until now, only the sole execution of an interaction diagram can be measured, e.g., with the coverage criterion *All-Paths Sequence Diagram Coverage* [UL06, page 122].

5.2.2 Interaction Diagram Concatenations

Interaction diagrams are often used to describe test cases manually (see e.g. UML Testing Profile [Obj05b] and TTCN-3 [SG03]) where a message to a given lifeline is considered as test input to the corresponding object under test. In this section, we describe how to concatenate interaction diagrams by tracing them as transition sequences in state machines and how to take advantage of this concatenation. For that, we define several states as follows.

Definition 37 (Initial State) *The* initial state *is the initial pseudo state of the state machine as defined in the UML specification [Obj07, page 521].*

Other states are used to refer to the state machine's possible start or the end of an interaction sequence described in an interaction diagram:

Definition 38 (Start State) *A* start state *of an interaction sequence is a state in the state machine from which it is allowed to start the execution of the sequence.*

Definition 39 (End State) *An* end state *of an interaction sequence is a state in the state machine at which the execution of an interaction sequence can end.*

This section contains the descriptions of how to concatenate interaction diagrams by concatenating corresponding transition sequences of a state machine. For that, we have to derive state machine transition sequences from interaction diagrams. We present a corresponding algorithm that produces transition sequences for each combination of interaction diagram and state machine. Afterwards, we show how to concatenate the transition sequences.

In [Sok06b], Sokenou describes a method to derive a set of possible start states for the execution of behavior defined in interaction diagrams. In contrast, this section is focused on deriving and comparing transition sequences

instead of single states. Thus, we present an algorithm to derive transition sequences from a state machine that reflect the described behavior of an interaction diagram. Figure 5.6 shows the algorithm of the corresponding function *findTransitionSequences*: For each possible start state of the state machine (line 06), the algorithm aims at executing the behavior of the interaction diagram (line 09). It returns a set of corresponding transition sequences (lines 21, 24). The pseudocode leaves out some aspects of transition matching such as transition guards, post conditions, or state invariants. We are, however, aware that information about the current system state (i.e. system attribute value assignment) is important to determine certain aspects of transition matching, e.g., the satisfaction of transition guards.

```
01 findTransitionSequences (InteractionDiagram id, SM sm) {
02   sequences = empty set; // return value
03   msg = first message of id;
04   startStates = all states of sm with outgoing transitions
05                 triggered by msg;
06   for each(s in startStates) {
07     tmpS = s;
08     transitionSequence = empty sequence;
09     for(i = 0; i < number of messages of id; ++i) {
10       msg = id.messages[i];
11       if(tmpS has outgoing transition t triggered by msg) {
12         tmpS = target state of t;
13         add t to transitionSequence;
14       }
15       else {
16         transitionSequence = empty sequence;
17         break;
18       }
19     }
20     if(transitionSequence is not empty) {
21       add transitionSequence to sequences;
22     }
23   }
24   return sequences;
25 }
```

Figure 5.6: Algorithm for detecting all state machine transition sequences corresponding to an interaction diagram.

For each interaction diagram, we retrace its described behavior as possible transition sequences in the state machine. Afterwards, we concatenate these transition sequences to combine the corresponding interaction diagrams: For two transition sequences ts_1, ts_2 with ts_1 assumed to be executed before ts_2, we consider four cases: 1) The transitions of ts_1 and ts_2 do not overlap, and

ts_1 does not include a transition whose target state is the start state of ts_2. In this case, both sequences cannot be concatenated. 2) The target state of the last transition in ts_1 is equal to the source state of the first transition in ts_2. 3) A transition subsequence ts_{1B} of ts_1 is equal to a transition subsequence ts_{2A} of ts_2 (e.g., see Figure 5.7). For these two cases, ts_2 can be executed after ts_1 (without executing the overlapping transitions in ts_{1B}/ts_{2A} twice). 4) Two subsequences of ts_1 and ts_2 are overlapping, and both transition sequences describe different behavior after the overlapping transitions. In this case, the combination algorithm can choose the transitions from one of both sequences. There are many more complicated ways of overlappings. In the considered case study, however, we were only confronted with rather simple overlappings as described for the cases 2 and 3.

Figure 5.7: Overlapping transition sequences.

Interaction sequences that can be combined using the proposed approach are called *adjacent*.

Definition 40 (Adjacent Interaction Sequences) *If two interaction sequences are overlapping and can be executed in succession, they are called* adjacent.

5.2.3 Coverage Criteria Definitions

In this section, we define new coverage criteria that describe the degree to which the possible concatenations of interaction diagrams are used. For instance, interaction diagrams can often be executed from several start states. Furthermore, interaction sequences can be combined to pairs or triples. In the following, we present several corresponding coverage criteria.

Definition 41 (All-Sequences) *The coverage criterion* All-Sequences *is satisfied iff all interaction sequences are executed at least once.*

In Figure 5.8, we formally define All-Sequences. The function *findTransitionSequences* is defined in Figure 5.6 on page 179. For each interaction diagram *id*, the function is used to derive all transition sequences of the state

machine that describe the behavior of *id*. Subsequently, all the transition sequences are described with trace patterns and combined into one complex test goal. The satisfaction of this complex test goal requires the test suite to traverse at least one described transition sequence.

```
P(TG) All-Sequences(SM sm, InteractionDiagrams ids) {
  testgoals = empty set;
  for all interaction diagrams id in ids {
    transitionSequences = findTransitionSequences (id, sm);
    CTG ctg = false; // new complex test goal - default: false
    for all transition sequences ts in transitionSequences {
      TP tracepattern = new TP(); // container for traced transitions
      for i=0 to ts.length-1 { // add all transitions
        tracepattern.add((?, ?, ?, {ts.get(i)}));
      }
      ATG atg1 = new ATG(tracepattern);
      ctg = ctg or atg1; // add atomic test goal atg to ctg
    }
    if(ctg is not empty)
      testgoals.add(ctg);
  }
  return testgoals;
}
```

Figure 5.8: Definition of All-Sequences.

Definition 42 (All-Context-Sequences) All-Context-Sequences *is a coverage criterion that is satisfied iff each transition sequence derived from an interaction diagram is executed from all of its start states.*

The coverage criterion All-Context-Sequences is similar to All-Sequences. The difference is that All-Context-Sequences requires to include all possible transition sequences that can be derived from a certain interaction diagram. Correspondingly, we define one atomic test goal for each transition sequence. Figure 5.9 shows the corresponding definition.

Definition 43 (All-Sequence-Pairs) *The coverage criterion* All-Sequence-Pairs *is satisfied iff all sequences of adjacent interaction sequences up to length 2 are executed at least once.*

Like presented for the coverage criteria in Section 2.1.5, "up to length 2" comprises all pair-wise combinations of interaction sequences as well as the

```
P(TG) All-Context-Sequences(SM sm, InteractionDiagrams ids) {
  testgoals = empty set;
  for all interaction diagrams id in ids {
    transitionSequences = findTransitionSequences (id, sm);
    for all transition sequences ts in transitionSequences {
      TP tracepattern = new TP(); // container for retraced transitions
      for i=0 to ts.length-1 { // add all transitions
        tracepattern.add((?, ?, ?, {ts.get(i)})); }
      testgoals.add(new ATG(tracepattern));
  } }
  return testgoals;}
```

Figure 5.9: Definition of All-Context-Sequences.

```
P(TG) All-Sequence-Pairs(SM sm, InteractionDiagrams ids) {
  testgoals = All-Sequences(sm, ids);
  for all interaction diagrams id1 in ids {
    transitionSequences1 = findTransitionSequences (id1, sm);
    for all interaction diagrams id2 in ids with id1 != id2 {
      transitionSequences2 = findTransitionSequences (id2, sm);
      if transitionSequences1 and transitionSequences2 overlap {
        for each overlapping pair of sequences seq1 and seq2 {
          CTG ctg = false; // new complex test goal - default: false
          tsA = the initial transition sequence of seq1;
          tsAB = the overlapping transition sequence of seq1 and seq2;
          tsB = the end transition sequence of seq2;
          TP tracepattern = new TP(); // list of retraced transitions
          for i=0 to tsA.length-1 { // add all initial transitions
            tracepattern.add((?, ?, ?, {tsA.get(i)})); }
          for i=0 to tsAB.length-1 { // add all overlapping transitions
            tracepattern.add((?, ?, ?, {tsAB.get(i)})); }
          for i=0 to tsB.length-1 { // add all end transitions
            tracepattern.add((?, ?, ?, {tsB.get(i)})); }
          ATG atg = new ATG(tracepattern);
        }
        ctg = ctg or atg; // add atomic test goal atg to ctg
  } } }
  if(ctg is not empty)
    testgoals.add(ctg); }
  return testgoals;}
```

Figure 5.10: Definition of All-Sequence-Pairs.

sole interaction sequences. Figure 5.10 shows the formal definition of All-Sequence-Pairs. We leave out the details of identifying the overlapping transitions because they make the pseudocode more complicated without helping to understand the defined coverage criterion.

Accordingly, we could also formally define coverage criteria that are focused on longer sequences like All-n-Sequences for sequences of length n and All-Sequence-Paths for combinations of arbitrary length.

Definition 44 (All-n-Sequences) *Similar to All-Sequence-Pairs, the coverage criterion* All-n-Sequences *is satisfied iff all sequences of adjacent interaction sequences up to length n are tested.*

Definition 45 (All-Sequence-Paths) *The new coverage criterion* All-Sequence-Paths *is satisfied iff all paths of interaction sequences are tested.*

Corresponding to these definitions, we sketch the definitions of the coverage criterion All-Context-Sequence-Pairs that is focused on combining all adjacent transition sequences for all their start states.

Definition 46 (All-Context-Sequence-Pairs) *The new coverage criterion* All-Context-Sequence-Pairs *is satisfied iff all sequences of interaction sequences up to length 2 are tested from each of its start states.*

Several additional definitions of such coverage criteria can be thought of. For instance, All-Context-Sequence-Pairs can be extended to sequences of length n, which results in the coverage criterion *All-Context-n-Sequences*. The presented coverage criteria are related by subsumption relations. Figure 5.11 shows the corresponding subsumption hierarchy.

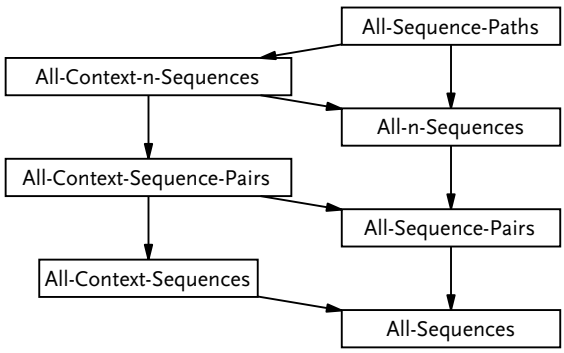


Figure 5.11: Subsumption hierarchy for interaction sequence combinations.

These new coverage criteria might be useful for evaluating the test suites derived from a state machine and a set of interaction diagrams. Their use, however, still has to be evaluated in case studies.

5.2.4 Case Study

In this section, we present an industrial scenario in which interaction diagrams were derived from requirements of an automated teller machine (ATM) and applied to create test cases for the ATM. Furthermore, these interaction diagrams are composed to a state machine. Originally, test cases were only derived directly from requirements use cases. In the following, we show how to use our approach to generate more complex test cases for such scenarios. In the following figures, AC stands for Account Check, and AR stands for Authorization System.

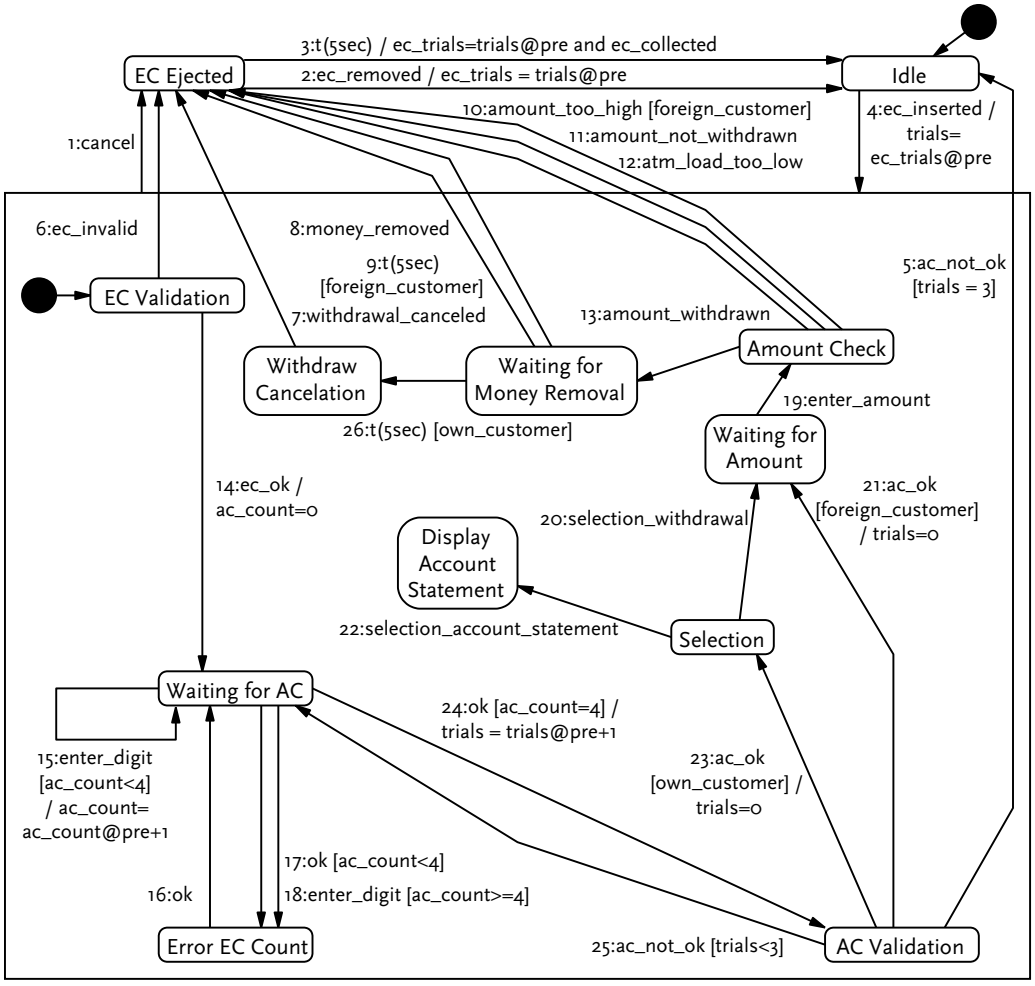


Figure 5.12: State Machine of ATM

The state machine is shown in Figure 5.12. All transitions of the state machine are numbered so it is easier to describe transition sequences in the

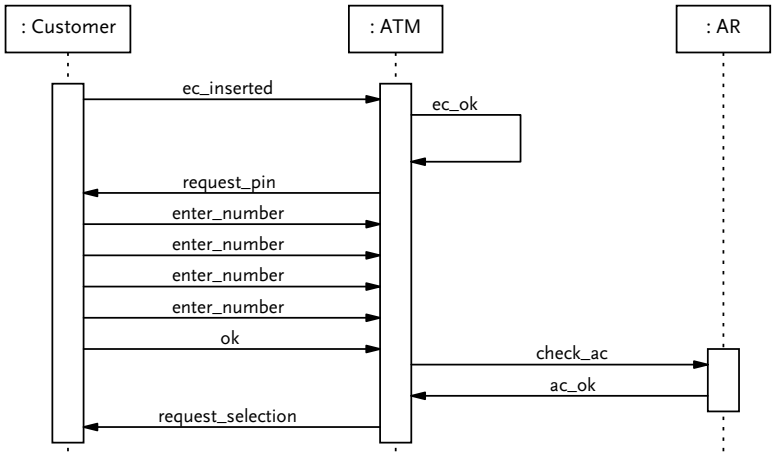


Figure 5.13: Sequence 1: Interaction diagram for inserting the EC card.

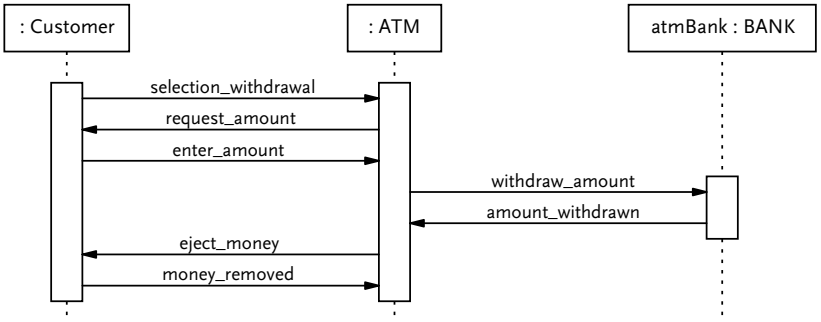


Figure 5.14: Sequence 2: Interaction diagram for withdrawing money.

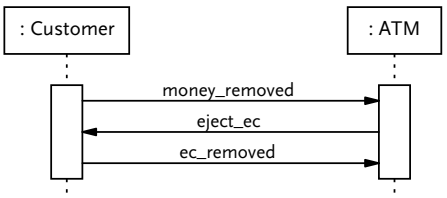


Figure 5.15: Sequence 3: Interaction diagram for removing money and EC card.

following. A possible behavior of a customer that enters his PIN (4 digits) and withdraws money is described for instance by the following transition sequence: (4, 14, 15, 15, 15, 15, 24, 23, 20, 19, 13, 8, 2). Note that the state machine describes just the behavior of the ATM. Thus, some messages of the interaction diagrams are not included in the state machine.

All interaction diagrams are directly derived from requirements use cases. Example sequences are shown for inserting the EC card and entering the PIN in Figure 5.13, for withdrawing money in Figure 5.14, and for removing EC card and money in Figure 5.15. Sequence 1 is an initializing sequence as it can be executed from the initial state of the state machine, state *Idle*. The message *ec_inserted* can only be executed in state *Idle*. Thus, the algorithm in Figure 5.6 starts with transition *4:ec_inserted* in the state machine. Following the algorithm, transition sequences for Sequence 1 are: $transseq_{s1.1}$ = (4, 14, 15, 15, 15, 15, 24, 23) and $transseq_{s1.2}$ = (4, 14, 15, 15, 15, 15, 24, 21). The last transitions of both sequences depend on the values of the attributes *own_customer* and *foreign_customer*. For Sequence 2, only one transition sequence can be found: $transseq_{s2} = (20, 19, 13, 8)$. There is also only one transition sequence for Sequence 3: $transseq_{s3} = (8, 2)$.

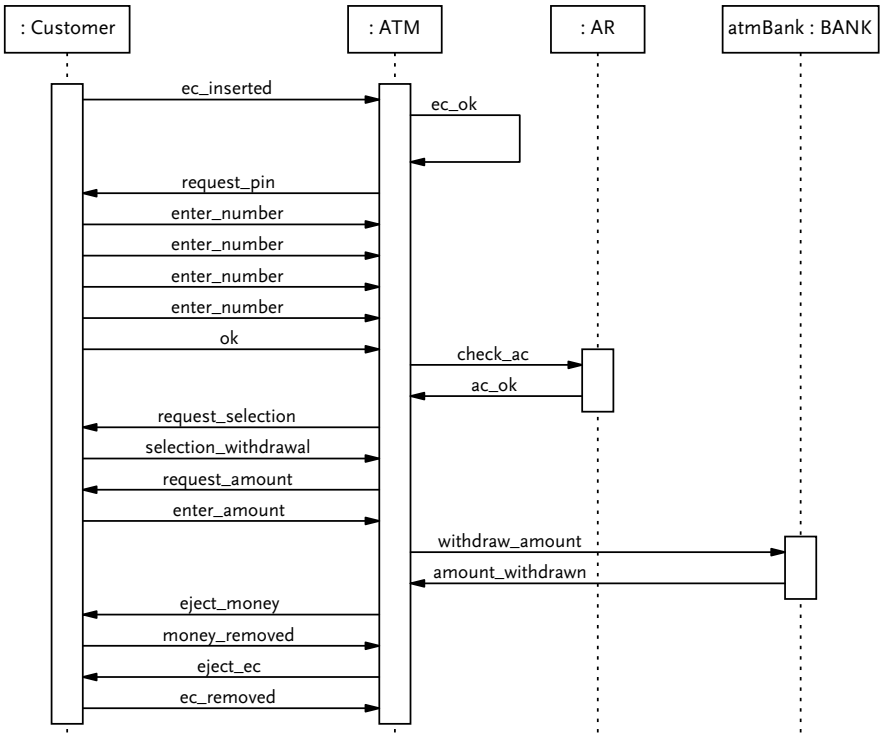


Figure 5.16: Concatenated interaction diagram.

The target state of the last transition of $transseq_{s1.1}$ is the same as the source state of first transition of $transseq_{s2}$. Thus, both can be concatenated to build a new scenario and a resulting new transition sequence $transseq_{s1.1,s2}$ = (4, 14, 15, 15, 15, 15, 24, 23, 20, 19, 13, 8). As the transition sequence $transseq_{s2}$ ends with the same transition as $transseq_{s3}$ starts, also Sequence 3 can be concatenated to the others. The resulting transition sequence is $transseq_{s1.1,s2,s3}$ = (4, 14, 15, 15, 15, 15, 24, 23, 20, 19, 13, 8, 2). Corresponding to the transition sequences, we concatenated the interaction diagrams and created longer test cases. Figure 5.16 shows the resulting interaction diagram.

As we expected, several faults can be detected by longer interaction diagrams. For instance, the combined sequence is the only sequence in the case study that contains a scenario from inserting EC card until money and EC removal. Furthermore, we found a sequence that was not covered by executing only the original interaction diagrams but can be derived by combining interaction diagrams: The original interaction diagrams only describe that a customer enters a wrong PIN three times without removing the EC card in between. However, there is no interaction diagram for a customer that inserts an EC card three times, enters the PIN incorrectly just once, and cancels the operation afterwards. With the repetition of several complete scenarios for interactions of customer and ATM, such scenarios can be covered. We found these improvements by manual inspection. Furthermore, the proposed coverage criteria are not applied, yet. It would be interesting to automate this approach to identify and evaluate further advantages.

5.2.5 Related Work

State machines and interaction diagrams of the UML [Obj07] are often used to model test cases. As one example, Nebut et al. [NFTJ03] derive test cases from contracts such as use cases and interaction diagrams. As another example, Abdurazik and Offutt provide an approach for automatic test generation from state machines [OA99]. In contrast to that, we aim at the combined use of both diagrams to generate test suites.

There has also been work about the combination of interaction diagrams and state machines. From the very beginning, it was clear that interaction diagrams can describe single transition sequences of a state machine. Bertolino et al. [BMM05] combine both diagrams to derive "reasonably" complete test models to achieve early results for partially modeled systems. Sokenou [Sok06b] and Nagy [Nag04] showed furthermore, that the state machine's start states to execute interaction diagrams can be important. We extend these approaches by identifying matching transition sequences instead

of start states. Additionally, we combine several interaction diagrams by matching start and end sequences of the corresponding transition sequences to build new and more complex sequences, and we defined coverage criteria based on interaction diagram combinations.

5.2.6 Conclusion, Discussion, and Future Work

This section completes the combination of interaction diagrams and state machines by providing conclusion, discussion, and future work.

Conclusion.

We presented an approach to combine interaction diagrams by retracing their described behavior as transition sequences in a state machine and concatenating these transition sequences. We presented an algorithm, listed several advantages of this approach, defined resulting coverage criteria, and showed the applicability of our approach to an industrial case study.

Discussion.

We identified some discussion points for our approach. For instance, the semantics of the used state machine is important for this approach. Input-enabled state machines can react to every sequence of events and, thus, all states would be start states for each event sequence. In this section, however, we focused just on explicitly modeled transitions like in protocol state machines [Obj07, page 529]. So, each event of the event sequence has to trigger at least one transition.

The combination of sequences can result in a lot of new and more complex test cases. They can reduce the test effort but increase the effort to identify the faults manually later on. To reduce complexity, the number of these test cases might be reduced using the proposed coverage criteria.

It might also be feasible to define limitations on the algorithm. It does not seem to make sense to combine interaction diagrams that overlap in all transitions except one. Or does it? What would be the maximum overlap that should be taken into consideration? Should it be defined absolutely or relatively to the sequences absolute length?

In the presented case study, the state machine was manually derived with the help of the interaction sequences. It would be interesting to identify ways to derive such state machines automatically. There are, however, several issues like the identification of state machine transition loops that are hard

to handle: Since each interaction sequence is finite – how can a corresponding algorithm be sure that a loop with a theoretically unlimited number of iterations is meant?

We also defined new coverage criteria and presented a corresponding subsumption hierarchy. This hierarchy has no connection to other coverage criteria. The reason is that we have no general knowledge about the covered behavior of the interaction diagrams. The only exception is the coverage criterion All-Paths, which subsumes each of the defined criteria anyway.

We used states to describe the retraced behavior of interaction sequences. In parallel state machines, these states would have to be replaced by state configurations. However, this poses no restriction to our approach.

Finally, interaction diagrams are used to manually describe typical scenarios. Thus, the effort to describe a sufficient test suite with interaction diagrams is high. They often do not define conditions for the initial state of execution and describe only parts of a scenario. Thus, the presented concatenation of interaction diagrams is a good way to create more complex test cases while avoiding to initialize each sequence separately before its execution. This concatenation can be automated and, thus, no additional manual effort is necessary.

Future Work.

In the future, we plan to implement the presented approach, e.g., as an extension of the tool ParTeG [Weib]. We want to use the tool to automatically create test cases for concatenated interaction diagrams. Here, we will take also the proposed new coverage criteria on interaction diagrams into account.

5.3 Conclusion

In this chapter, we presented two approaches to combine different test models for automatic test generation. We showed one example of combining behavioral and structural models and one example of combining two behavioral models. We showed advantages and application fields for each of these combinations. The first combination of test models is already implemented in ParTeG. For the latter combination, we proposed coverage criteria to measure the extent to which possible combinations are used.

In general, the combination of models is a helpful technique. Each time one model or two unconnected models are not sufficient, they might be combined somehow. There is other work besides the presented approaches to combine different models. For instance, Kösters et al. [KSW01] use re-

fined activity graphs to combine use cases and class diagrams. Schroeder et al. [SKAB03] present a technique to combine behavior and data models. Corresponding case studies indicate that the combination of models has advantages over the single application of models.

Chapter 6

Test Suite Efficiency

For automatic model-based test generation, coverage criteria can be applied to state machines. They return an unordered set of test-model-specific test goals (see Section 2.4). In this chapter, we investigate the effect of the test goal order on the test suite's efficiency. We consider this the fourth contribution of this thesis. All previously presented contributions are also focused on model-based test generation with coverage criteria. Thus, the results of this chapter can be combined with all the other contributions.

6.1 Introduction

This thesis is focused on coverage criteria that are applied to UML state machines for automatic model-based test generation. As explained above, a wide-spread approach to test suite generation is to use a coverage criterion to produce a set of test-model-specific test goals and to generate test cases that cover the corresponding trace patterns (see Section 2.4.2). Since each test case usually also covers trace patterns of other test goals, it satisfies the corresponding test goals, too. Consequently, the satisfaction of a certain test goal often results in the satisfaction of other test goals.

Most existing approaches to influence test execution efficiency focus on changing an existing test set for regression testing. Since the test suite is generated anew for each new version of the system, this is no adequate approach in model-based testing. Instead, one has to influence the test generation process. Our approach is focused on changing the order of test goals for test generation. The set of test goals returned by the coverage criterion is a priori unordered. This chapter describes an investigation of the impact of test goal order on test suite execution by defining and evaluating several test goal orders. The relations between test goals and the generated test suite

depend on many aspects of the test suite generation process. We focus on the following aspects: different kinds of satisfied coverage criteria, different kinds of test path search strategies, and the application of online or offline testing. Finding the best test goal order by permuting all test goals is infeasible. Consequently, we propose heuristic test goal prioritizations to generate test goal orders. We motivate all prioritizations with the small example of the freight elevator that is introduced in Section 3.2.2 and evaluate them based on the industrial test model presented in Section 3.2.5.

The chapter is structured as follows. The preliminaries are described in Section 6.2. Section 6.3 contains the description of test goal prioritizations. Their evaluation with the industrial test model of the train control is presented in Section 6.4. Section 6.5 contains the related work. The final section comprises conclusion, discussion, and future work.

6.2 Preliminaries

This section contains all preliminaries for the presented test goal prioritizations. The subsections contain descriptions of the idea of test goal prioritization and the applied search algorithm. Further subsections contain reflections of our understanding of online/offline testing.

6.2.1 Idea of Test Goal Prioritization

Each coverage criterion is a function that returns a set of test goals. For each test goal, a test case is created that covers the traces referenced by the test goal. The goal of our investigations is the deduction of general advantages of certain test goal orders for the efficiency of a test suite, i.e., the average effort necessary to detect a failure. Like presented in Section 3.3.1, we say that the test case satisfies the corresponding test goal *intentionally*. Each test case often covers several other state machine elements. We say that the test case satisfies the corresponding test goals *accidentally*. For each intentionally satisfied test goal, we are interested in the corresponding accidentally satisfied test goals. There can be several test cases to satisfy a test goal intentionally. We consider the applied search algorithm the most important aspect to determine the accidentally satisfied test goals of a test case.

6.2.2 Applied Search Algorithm

The basic search algorithm of our approach starts at the trace patterns referenced by the test goal to satisfy intentionally and searches for a way backward to the initial node. As described in Chapter 3, this approach has some advantages, e.g., the combination of structural (e.g. control-flow-based) and boundary-based coverage criteria [WS08a].

This chapter contains the description of a tool-independent approach. We neglect the details of the applied tool ParTeG and focus on common properties of test path search algorithms. Independent of the concrete search engine, each created test case has two important properties: the number of covered test goals and its length, e.g. given in lines of code. In this section, the focus is on the test case length. All test cases are roughly subdivided into comparatively short and comparatively long test cases. The applied search algorithm uses a corresponding short path strategy or a long path strategy to support the creation of rather short or long test cases.

The following terms are defined to clarify the notion of both strategies. Each state *s* has a graph-theoretic *distance*, which roughly describes the number of visited states on a shortest path from the initial node to *s*. The initial node and *s* can be connected by several paths with different lengths. Furthermore, composite states can contain complex behavior descriptions that can be executed or skipped if the composite state is left before completing its internal behavior. How should composite states influence the distance of a state? The following definitions take care for this issue: Each state has a *distance value*. The default distance value of each state (simple state, pseudostate, and composite state) is 1. The distance between two states along a path is the sum of all the states' distance values on the path between them.

Definition 47 (Minimal Distance) *The* minimal distance *of a state machine's state* s *is the minimum of all possible* distances *between the state machine's initial node and* s*. The* minimal distance *of a state machine's transition* t *is equal to the minimal distance of* t*'s source state.*

Definition 48 (Maximal Distance) *The* maximal distance *of a state machine's state* s *is the maximum of all possible* distances *between the initial node and* s *for all paths that contain both states without loops. The distance value of each composite state* cs *is redefined as the maximum of all maximal distances between* cs*'s initial state or entry point and an arbitrary state within* cs*'s submachine. The* maximal distance *of a state machine's transition* t *is equal to the maximal distance of* t*'s source state.*

One aim of our investigations is to create short and long test cases, i.e., investigate properties of test cases with minimal distances and maximal dis-

tances, respectively. We define two corresponding test path strategies. Both strategies differ in the selection of the next transition to traverse backward.

Definition 49 (Short Path Strategy (SPS)) *If more than one transition can be selected to be traversed backward, the* short path strategy *always prefers the transition with the smallest minimal distance.*

Definition 50 (Long Path Strategy (LPS)) *If more than one transition can be selected to be traversed backward, the* long path strategy *always prefers the transition with the largest maximal distance.*

The effects of both strategies are obvious: The application of SPS results in the creation of comparatively short test cases. Since short test cases satisfy a small number of test goals, SPS results in a large number of test cases. Contrary, the application of LPS results in a small number of long test cases. Both path strategies depend on the applied test generation technique, in this case, search-based test generation. The results of applying these strategies (the properties of generated test cases concerning their lengths) are tool- and technique-independent. The only exception are random approaches, which are not guided by the selected coverage criterion.

6.2.3 Online/Offline Testing

This section contains a short description of online/offline testing [UPL06]. In offline testing, the test generator is disconnected from the SUT and the generated test suite is executed on the SUT after complete creation. The whole test suite can be optimized after its creation. In online testing, the test generator and the SUT are connected and all commands are directly executed on the SUT. The test cases are usually generated and executed one after the other. For this reason, there is no post-optimization of the test suite. Online testing is also often used as explorative testing without a given test specification. In this case, however, we apply the given test model.

There are various reasons for performing online and offline testing, respectively. These reasons can be space limitations of the target environment. For instance, the test suite may be too large to fit into memory. As an example, the size of a test suite for road trials created in cooperation with DaimlerChrysler exceeded 10 gigabyte. Test efficiency may also be a reason: The process of test suite generation usually takes some time. If we are only interested in quickly detecting at least one fault (e.g. in smoke testing – see page 14), then online testing may detect faults faster because of the immediate execution of test cases. The selection of offline testing, however, results

in explicit test suites which can be repeatedly executed, managed in version control systems, and reused in regression testing.

All in all, there are pros and cons for both testing techniques. We investigate the impact of test goal prioritization for both of them.

6.3 Test Goal Prioritizations

This section contains the descriptions of the proposed test goal prioritizations. Each prioritization is adapted to a different aspect of the state machine's elements referenced by the test goals of the used coverage criteria: the distance to the state machine's initial state, the branching factor of the referenced elements, the size of guard conditions, and the number of positively evaluated atomic conditions in guard conditions. All of the used terms are explained in the following. For each mentioned aspect of the state machine's elements, two opposite test goal prioritizations are defined. The freight elevator example from Section 3.2.2 is used to provide a motivation for most prioritizations. The random test goal prioritization is used for comparison: Each advantageous prioritization should at least result in a better test suite execution evaluation than random prioritization.

6.3.1 Random Prioritization (RP)

Random prioritization results in random test goal order. It can be applied to all investigated coverage criteria. An effective prioritization should at least result in a better evaluation outcome than random prioritization.

6.3.2 Far Elements (FEF/FEL)

The prioritization *far elements first (FEF)* sorts test goals according to their referenced model element's distance in descending order. In Section 6.2.2, minimal and maximal distance are defined. The selected kind of distance corresponds to the chosen search strategy: Minimal distance is used for short path strategy (SPS) because SPS is focused on short paths. Since long path strategy (LPS) is focused on long paths, maximal distance is used for LPS. A test goal's distance describes the approximated length of the transition sequence to satisfy it. The intention of FEF is that paths from the initial node to a test model element with a high distance are comparably longer. Thus, they have a higher chance to satisfy more test goals accidentally. The opposite prioritization to FEF is *far elements last (FEL)*. FEL results in

comparatively short test cases. Both prioritizations can be applied to all of the examined coverage criteria.

In our example of a freight elevator control, FEF for All-States has the following consequences: A path that leads to the state *move slow* has to pass the states *idle*, *button pressed*, and *start moving*. As a consequence, the first generated test case satisfies already four of five test goals. The test case for the remaining test goal also satisfies four test goals, three of which are already satisfied by the first test case. Without postoptimization, FEL for All-States would result in separate test cases for all five test goals. Thus, the test execution effort might be considerably higher for FEL.

6.3.3 Branching Factor (HBFF/HBFL)

The prioritization *high branching factor first (HBFF)* sorts all test goals according to the branching factor of each test goal's referenced model element in descending order. For the opposed prioritization *high branching factor last (HBFL)*, the elements are sorted in ascending order. The branching factor of a state s is equal to the ratio of s's outgoing transitions to s's incoming transitions. The branching factor of a transition t is the ratio of the outgoing and incoming transitions of t's target state. The idea of HBFL is that an element e with a high branching factor is probably already accidentally satisfied by a longer test case for another test goal that contains e. Thus, the test case generation for elements with high branching factors should be delayed as far as possible in order to prevent the unnecessary creation of test cases. Both can be applied to all coverage criteria. We know no motivation for HBFF. Since we deal with heuristics, however, we evaluate both approaches.

In the freight elevator example, the states *move fast* and *move slow* have the lowest branching factor and – similar to FEF – should be used first for test case generation with HBFL. For test goals that reference transitions, all incoming transitions of the state *idle* have the lowest branching factor: *idle* has three outgoing and five incoming transitions. As a consequence, test cases for the two transitions from the states *move fast* and *move slow* are created first, which is also preferred with FEF.

6.3.4 Atomic Conditions (MACF/MACL)

The prioritization *many atomic conditions first (MACF)* sorts all test goals according to the size of the referenced guard condition in descending order. This results in the preferred creation of test cases with many atomic conditions to satisfy. The idea is that the satisfaction of many atomic guard

conditions results in the traversation of many transitions. Consequently, the created test case is longer, and more test goals are accidentally satisfied by it. The opposed prioritization of MACF is *many atomic conditions last (MACL)*. Since both prioritizations are focused on guard conditions, they can be applied to control-flow-based coverage criteria.

In the example of the elevator control, the transition triggered by *pressButton* has the highest number of atomic expressions. Since there are many test goals that need to traverse this transition, it does not seem to be a good idea to start with this transition. The transitions triggered by *reachFloor* should be used for test path generation first, but the referenced guard condition is empty. Thus, MACL seems to be the better choice for the example. Nevertheless, both prioritizations seem to be interesting and we include both prioritizations in our evaluation.

6.3.5 Positive Assignment Ratio (HPARF/ HPARL)

For several coverage criteria, the satisfaction depends on the value assignments of guards' atomic conditions. For each guard, we call the number of positive value assignments divided by the total number of atomic guard conditions the *positive assignment ratio*.

The prioritization *high positive assignment ratio last (HPARL)* sorts all test goals according to their positive assignment ratio in ascending order. The motivation for this prioritization is that the guards of all traversed transitions of a valid test case are satisfied. So, except for sneak path analysis, there are probably more satisfied guards than violated guards in a test case. Consequently, each test case probably satisfies many test goals that reference a satisfied transition guard. The corresponding opposed prioritization is *high positive assignment ratio first (HPARF)*. The idea of both prioritizations is similar to the distance estimation in [FW08a]. In our evaluation, both prioritizations are available for control-flow-based coverage criteria.

In the example of a freight elevator, the satisfaction of a guard condition often results in the violation of another guard condition. For instance, the satisfaction of $[actualWeight = minWeight]$ results in the violation of $[actualWeight > minWeight]$. As an exception, the guard of the transition triggered by *pressButton* is satisfied by other longer test cases, whereas it is not violated by others. It seems that the success of both proposed prioritizations also depends on the test model's structure.

6.4 Evaluation

In the previous sections, we used the freight elevator example to motivate the proposed test goal prioritizations. Here, we evaluate the prioritizations on the industrial test model presented in Section 3.2.5. The evaluation results of our experiments are influenced by the presented test goal prioritizations, the selected coverage criterion, the search strategy, and online/offline testing. The combination of these aspects results in 92 different experiment setups – we present only the results for a few representatives. For each setup, we created and executed a test suite 50 times with ParTeG. The evaluation values comprise arithmetic average, worst case, and standard deviation. We consider the coverage criteria All-States, Decision Coverage, and masking MC/DC. The evaluation of the case study provided prioritization-specific results as well as general results concerning test suite evaluation. First, we introduce the used effect measurement in the industrial case study. Then, we present the evaluation results for each selected coverage criterion and give a recommendation about when to use which prioritization. Finally, we show results concerning test suite evaluation and results about the influence of test generation aspects such as search strategy and online/offline testing.

6.4.1 Effect Measurement for Industrial Test Model

Test goal prioritization influences the test suite generation process as well as the execution of the generated test suite. The presented evaluation is focused on the latter one. To evaluate the fault detection capability and the efficiency of the test suites, we conduct mutation analysis on source code level as presented in Section 2.1.5. The application of the presented mutation operators results in 280 mutants, 268 of which show a different behavior than the original SUT. Note that these numbers are different from the extreme values of the efficient and the redundant SUT used in Section 4.1. The reason is simply that the used SUT is not an extreme SUT.

During test suite execution, we measured the number of test cases and function calls that were necessary to detect a failure. We call the measured value *test effort* and its inverse value *test suite efficiency*: a smaller test effort corresponds to a higher test suite efficiency, and means that a smaller number of executed test cases and function calls (lines of code) to detect faults. We also consider the absolute test suite's fault detection capability. Since test goal prioritization just sorts test goals but does not alter them, however, no prioritization is expected to have an impact on the test suite's fault detection capability.

For the presented case study with the German supplier of railway signaling solutions Thales, we generated test suites based on the UML state machine that is shown in Sections 3.2.5 and 4.1. Due to the large number of possible experimental setups, we present just the results for a few of them. The results of the other setups are similar, yet not as plain as the results of the selected setups.

6.4.2 All-States

The evaluation for All-States brought the following results: For offline testing, both distance-dependent prioritizations FEF and FEL depend on the selected path search strategy. The application of FEL for SPS and FEF for LPS result in efficient test suites. For online testing, the number of accidentally satisfied guards and the resulting test suite size have a greater impact than the path search strategy: Independent of the selected path strategy, the application of FEF results in a greater test suite efficiency in online testing. FEF and FEL are the only distance-dependent prioritizations. This might be a reason for that the path search strategy does not influence the other test goal prioritizations. For All-States, the application of HBFL results in a more efficient test suite execution than the application of HBFF.

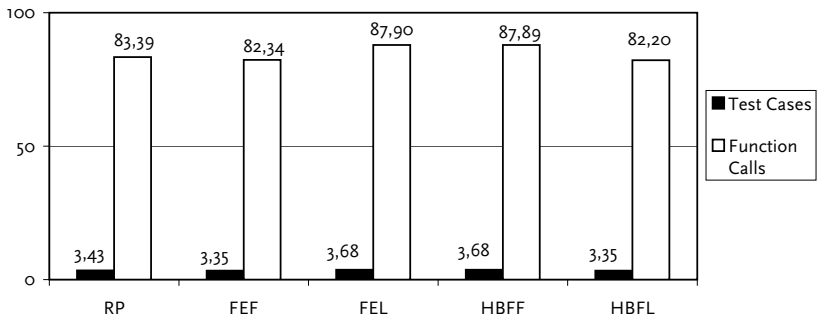


Figure 6.1: Average test effort for All-States with LPS and offline testing.

The application of FEF or HBFL results in the most efficient test suites, which seems to be caused by a large number of accidentally satisfied test goals and the resulting small test suite size. Figure 6.1 depicts the average test effort for LPS and offline testing as the average number of executed test cases and the number of function calls necessary to detect a mutant. The application of, e.g., FEF instead of FEL reduces the test effort from 87,9 function calls down to 82,34 (-6,3%) and from 3,68 test cases to 3,35 (-9%). Another result of the experiments are the differences of the standard deviations: the standard deviations for FEF and HBFL are smaller than the

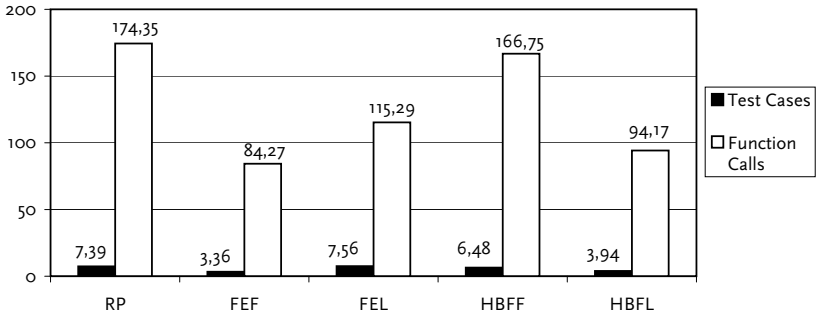


Figure 6.2: Worst-case test effort for All-States with LPS and online testing.

standard deviations for the other prioritizations. The lower average effort for HBFL and FEF is not crucial. However, we also analyzed the worst case scenario, which revealed much greater advantages for both prioritizations. In Figure 6.2, the worst absolute measured test effort for LPS and online testing is presented. The application of, e.g., FEF instead of RP decreases the worst-case test effort in terms of necessary test cases and function calls down to 45,5% and 48,3%, respectively.

6.4.3 Decision Coverage

Test goals for Decision Coverage differ from those for All-States. Likewise, the evaluation of test goal prioritization brought different results: For Decision Coverage, the application of FEF or HBFL results in an inefficient test suite. Instead, the application of MACF, HBFF, or HPARF results in an efficient test suite.

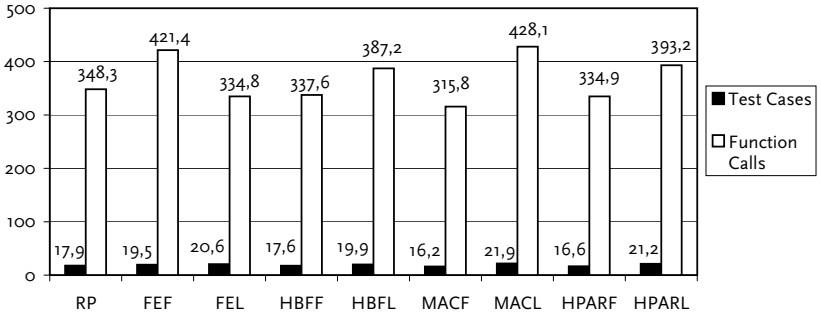


Figure 6.3: Average test effort for LPS and offline testing for Decision Coverage.

Figure 6.3 shows that the application of MACF instead of MACL decreases the average function calls to 73,7% and the test cases to 74,0% for

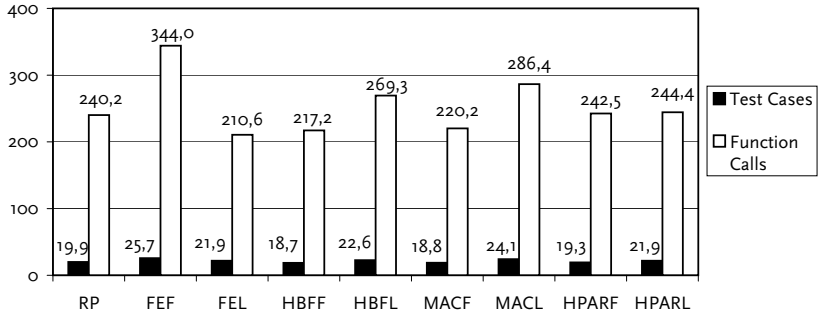


Figure 6.4: Average test effort for SPS and offline testing for Decision Coverage.

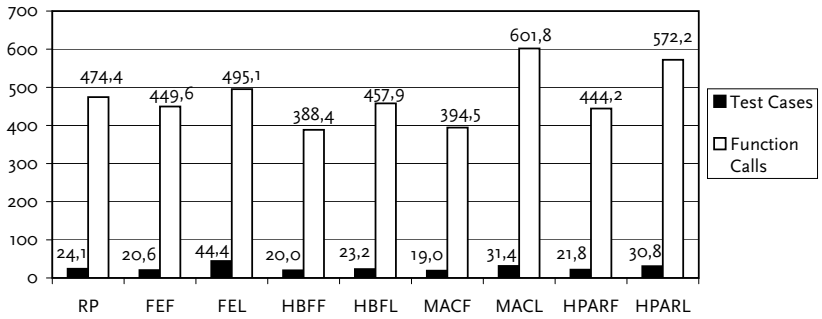


Figure 6.5: Worst test effort for LPS and online testing for Decision Coverage.

LPS and offline testing. In comparison to RP, MACF reduces the number of function calls to 90,7% and the number of test cases to 83,1%. Figure 6.4 shows similar results for SPS and offline testing. Here, the proper selection of a test goal prioritization can reduce the test effort. Applying FEL instead of FEF results in a reduction of the average number of function calls to 61,2% and of test cases to 85,2%. For HBFF instead of FEF, the number of function calls is reduced to 63,1% and the number of test cases to 72,8%. Applying FEL instead of RP results in a lower number of function calls but a higher number of test cases. Figure 6.5 depicts the test effort for the worst case. In contrast to All-States, the possible test effort reduction for the worst case is similar to the average test effort reduction: The selection of HBFF instead of MACL reduces the worst-case number of function calls to 64,5% and the number of executed test cases to 63,7% for LPS and online testing.

In comparison to All-States, the application of FEF still results in a large number of accidentally satisfied test goals. Moreover, inspections showed that the first executed test cases are capable of detecting a large number of mutants. However, since these sets of mutants are overlapping, many mutants

are detected later and the test effort for FEF is large (see Figure 6.4). Note that - just like for All-States - the application of FEL results in low test effort for SPS and offline testing.

6.4.4 Masking MC/DC

Like Decision Coverage, masking MC/DC is focused on the guard conditions of transitions. Thus, it is not surprising that the evaluation results for masking MC/DC are similar to the ones for Decision Coverage. The best test goal prioritizations are again MACF, HBFF, and HPARF.

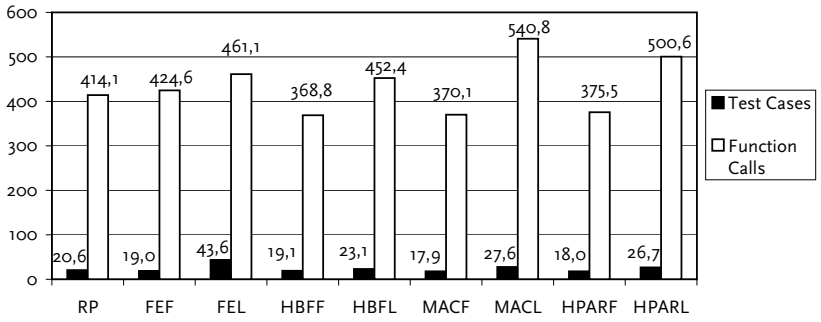


Figure 6.6: Average test effort for LPS and online testing for masking MC/DC.

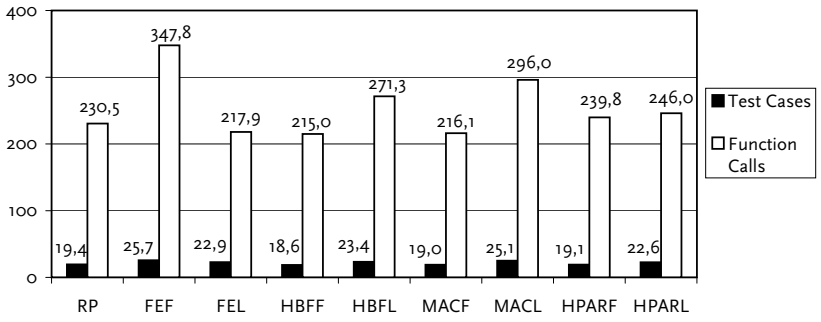


Figure 6.7: Average test effort for SPS and offline testing for masking MC/DC.

Figure 6.6 shows the average test effort for masking MC/DC with LPS and online testing. The selection of, e.g., HBFF instead of RP or MACL results in a decrease of the function calls down to 89,1% and 68,2%, respectively. The number of executed test cases is reduced to 92,7% and 69,2%, respectively. The results for SPS and offline testing in Figure 6.7 are similar: The test

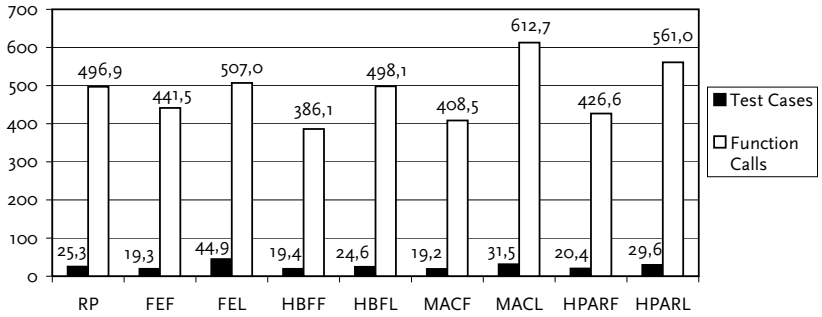


Figure 6.8: Worst test effort for LPS and online testing for masking MC/DC.

suites derived with MACF or HBFF have a low test effort, whereas RP or MACL results in a test suite with high test effort. For instance, the application of HBFF instead of MACL results in a function call decrease to 72,6% and a decrease of test cases to 74,1%. Again, the application of FEL for SPS results in low test effort. Figure 6.8 shows the worst case results for LPS and online testing: The application of MACF, HBFF, or HPARF results in efficient test suites. For instance, the selection of MACF instead of RP or MACL results in a function call reduction down to 82,2% and 66,7%, respectively, and a reduction of executed test cases to 75,9% and 61,0%, respectively.

6.4.5 Application Recommendation

In this section, we give an application recommendation of test goal prioritizations based on the results of the previous sections. All subsequent table rows list the recommended prioritizations corresponding to the resulting test suite's efficiency in descending order. The application of any of the recommended test goal prioritizations results in a good test suite efficiency with a comparatively small standard deviation. Note that all results are based on just one test model. Threats to generalization will be discussed in Section 6.6.

For All-States, we recommend in most cases to use FEF and HBFL (see Table 6.1). If the search strategy is SPS and offline testing is used, then we recommend to apply FEL instead of FEF. The recommendation of HBFL for All-States is not restricted to a certain path search strategy.

The results for Decision Coverage are completely different. As Table 6.2 shows, the application of MACF or HBFF results in test suites with high efficiency for all combinations of search strategy and online/offline testing. Additionally, FEL results in efficient test suites only for offline testing. The application of HPARF is only recommended for the search strategy LPS.

Search Strategy	Online/ Offline	Recommendation
LPS	Offline	FEF, HBFL
LPS	Online	FEF, HBFL
SPS	Offline	HBFL, FEL
SPS	Online	FEF, HBFL

Table 6.1: Recommendations of test goal prioritizations for All-States.

Search Strategy	Online/ Offline	Recommendation
LPS	Offline	MACF, HPARF, HBFF, FEL
LPS	Online	MACF, HBFF, HPARF
SPS	Offline	FEL, HBFF, MACF
SPS	Online	MACF, HBFF

Table 6.2: Test goal prioritizations recommendations for Decision Coverage.

Although masking MC/DC is considered more complex than Decision Coverage, the evaluation results for both coverage criteria are similar (see Table 6.3). This might be caused by the fact that both coverage criteria are focused on the control flow. There are only a few differences between the results for both coverage criteria. The standard deviation values of the evaluation results for masking MC/DC are greater than the ones for Decision Coverage. As a consequence, the worst-case evaluation values of the recommended prioritizations exceed or come close to the average value of RP (cf. Figures 6.6 and 6.8).

Search Strategy	Online/ Offline	Recommendation
LPS	Offline	MACF, HPARF, FEL, HBFF
LPS	Online	HBFF, MACF, HPARF
SPS	Offline	HBFF, MACF, FEL
SPS	Online	MACF, HBFF

Table 6.3: Test goal prioritization recommendations for masking MC/DC.

6.5 Related Work

There are already several publications about the prioritization of test cases. For instance, Jones and Harrold [JHS03] deal with the prioritization of test cases for the coverage criterion MC/DC. This topic is complex because, unlike for other coverage criteria, test goals for MC/DC can often only be satisfied by pairs of test cases. Elbaum et al. [EMR02] present several case studies about the prioritization of test cases for regression testing. They compare the

impact of several prioritization techniques. Bauer et al. [BSME08] present a risk-based approach for test case prioritization. The deciding aspect of the cited approaches is that the prioritized test cases already exist. Consequently, the prioritization is influenced by information of previous test runs such as a test case's fault detection capability or coverage criteria satisfaction. In contrast, our approach is aimed at prioritizing test goals without knowledge about the resulting test cases. Furthermore, our approach is applied to automatic test suite generation, whereas the presented related work is focused on regression testing with existing test suites.

To our knowledge, there is little work about the prioritization of test goals. As an exception, Fraser and Wotawa [FW08a] order test goals and create test suites with a model checker. They propose two different metrics for test goal ordering, and they measure the impact on the test suite creation time and on the number of generated test cases. The evaluation is done with experiments for several coverage criteria and several standard examples. The fundamental result is that the used test goal order matters less than expected. In contrast to this approach, we measure the impact of the test goal order on the test suite efficiency. Thus, we do not see our work contradicting the work of Fraser and Wotawa, but as a complement to it. We measure the fault detection effort instead of the test suite size. To measure this effort, we use mutation operators to inject faults in a correct SUT and count the number of executed test cases and function calls until the faults are detected. The result of our investigations is that test goal prioritization matters for test suite execution. Our approach is based on our prototype implementation ParTeG, yet it is generally applicable to all model-based test generators for state-based test models. As mentioned above, the only exception are random test generators.

As presented in Section 3.6, there are many commercial tools that support model-based test generation based on UML state machines and coverage criteria. Many of them follow the chosen approach of using a coverage criterion to generate a set of test goals. For instance, the Rhapsody ATG [IBM] creates test suites based on calculated test goals [IBM04]. Furthermore, the Smartesting Test Designer [Sma] supports All-Transitions and handles each transition as a target. For the presented case study, we use our prototype implementation ParTeG, which is based on the same approach. All mentioned tools could support test goal prioritization. In contrast to the mentioned commercial tools, however, ParTeG is the only tool that supports the prioritization of test goals.

6.6 Conclusion, Discussion, and Future Work

Here, we conclude and discuss the presented approach. We also sketch possible future work.

Conclusion.

In this chapter, we proposed several test goal prioritizations and evaluated their impact on test suite execution using a test model from an industrial case study. The results depend on many different factors like the applied path search strategy, the selected coverage criterion, and online/offline testing. The evaluation showed that the selection of a proper test goal prioritization has an impact on the test suite efficiency. For instance, the choice of test goal prioritization can decrease test effort below 70% for both considered control-flow-based coverage criteria. These results are encouraging. We suggest more case studies to determine the degree of the result's prioritization specificity and the impact of test model characteristics. Although unintended, we identified some results concerning test suite evaluation. For instance, in combination with a satisfied coverage criterion, the number of test cases and function calls (absolute – not per test case!) are often used as indicators for a test suite's efficiency. However, they can provide different results. For instance, Figure 6.3 shows that the average function calls for FEL are lower than for FEF but the average number of test cases is higher. The same phenomenon can be seen in Figure 6.6 for the prioritizations FEL and MACL. Thus, both measures of test suite efficiency should be used carefully.

As we expected, the selected search strategy influences the test suite size and the test suite efficiency. In contrast to SPS, the application of LPS results in test suites with a comparably small number of longer test cases. This directly influences the test suite efficiency. In general, test suites generated with LPS need less test cases but more function calls to detect a mutant than test suites generated with SPS (compare, e.g., Figures 6.3 and 6.4).

The selected prioritization has almost no impact on the mutation score. The test model contains linear ordered types and inequations and, thus, even the application of the strongest control-flow-based coverage criterion Multiple Condition Coverage (MCC) does not result in the detection of all mutants. The satisfaction of the boundary-based coverage criterion Multi-Dimensional (MD) and MCC as provided by ParTeG is the only combination (MDMCC) that results in killing all detectable mutants.

Furthermore, we realized that test goal prioritization is especially important for online testing. The main reason for this is the missing test suite post-optimization. Although we applied monitoring as a means to prevent test case generation for already satisfied test goals as an in-process-optimization, online testing results in considerably less efficient test suites than offline testing and is strongly influenced by test goal prioritization.

Discussion.

There is room for discussion. For instance, the proper selection of the examined coverage criterion is important. On the one hand, All-States is a bad choice because it is considered too weak and rarely used in industry whereas MC/DC is actually recommended in the standard RTCA/DO-178B [RTC92]. On the other hand, the comparison of both is a good means to evaluate the impact of the coverage criterion's complexity.

The presented evaluations are focused on test suite execution. We measured the quality of a test suite by the necessary fault detection effort. Since the sheer test suite size is also often used as an indicator of the test suite's quality, our research could be put into question. Our investigations showed, however, that there are differences between both measures. Although our experiments are not focused on the adequacy of the test suite size as a test suite efficiency indicator, they show that the test suite size is no indicator for the efficiency of a test suite in offline testing. This is quite obvious because a change of test case order has an impact on the efficiency of a test suite but almost no impact on its size in lines of code. For instance, Figure 6.4 shows different test efficiencies for test suites that are all of similar size.

We used the numbers of executed test cases and function calls to measure the test suite execution effort. An interesting result is that the two measured values sometimes indicate different results. For instance, both values for MACL and FEL in Figure 6.8 are diverse: The number of executed test cases for MACL is lower than for FEL. For the number of function calls, it is the other way round. Since the number of executed test cases is always connected to the length of each test case, we consider the absolute number of function calls a more appropriate means of test effort measurement. However, the importance of both means depends on the application domain. For instance, test case initialization is costly in embedded systems and, thus, a low number of executed test cases can be more important then a low number of function calls.

After introducing pairs of possible test goal prioritizations, we expected that always at least one of the two alternatives results in an efficient test suite. For Decision Coverage and masking MC/DC, however, we did not always rec-

ommend one prioritization of each pair because they provided worse results than random prioritization (e.g., HPARF/HPARL for SPS in Table 6.3).

Furthermore, our experiments were executed with our prototype implementation ParTeG. All the used aspects of test generation, e.g., the minimal/maximal path distance are independent of the tool and can be transferred to others as long as the approach of transforming coverage criteria into test goals is realized.

We used only one test model for the evaluation. Although other test models may indicate a different application recommendation, we showed that test goal prioritization has an impact on the efficiency of the generated test suite. With the results presented in Chapter 4, we identified some issues of the presented test goal prioritizations: There may be semantic-preserving test model transformations that have an influence on the test goal order. For instance, the insertion of nodes into transitions as presented in Section 4.2.2 on page 126 can influence the distance measures for FEF and FEL. Furthermore, complex guard conditions can be expressed in a more complicated way (e.g. $((a \wedge b) \vee (a \wedge \neg b))$ instead of a), which makes the single guards more complex and increases their weight, e.g. for MACF. The same holds for the positive assignment ratio when variables are negated. It would be interesting to repeat the test effort measurement for simulated coverage criteria on transformed test models.

Finally, the effort to sort the test goals took just a few milliseconds. Therefore, the gain of test goal prioritization outweighs its costs.

Future Work.

First of all, we plan to evaluate the impact of test goal prioritization for other test models. This will give more information about the validity of the presented application recommendation.

Next, we plan to investigate other means to improve the generated test suite. This can be achieved by, e.g., applying a new search strategy of the test case generation algorithm. The algorithm can be influenced, e.g., by already generated test cases and the correspondingly satisfied test goals. For instance, instead of generating short or long test cases, the path search strategy can be focused on satisfying unsatisfied test goals. Test suites for test models with several transition loops or for sneak path analysis often contain equal transition sequences that differ just in the last transition, which could be easily integrated in other test cases. Thus, this test generation approach would be especially useful for such scenarios.

Furthermore, we recognized a high standard deviation of the test suite efficiency resulting from the recommended prioritizations for masking MC/DC.

We plan to reduce this deviation, e.g., by combining different prioritizations. For instance, a second prioritization can be applied locally to all test goals with the same priority of the first prioritization.

Finally, all presented prioritizations are applied to test goals. Especially for control-flow-based coverage criteria, these goals can be satisfied by one of many different value assignments. It would be interesting to investigate the effects of prioritizing these value assignments, too.

Chapter 7

Conclusions

The presented thesis is focused on test models and coverage criteria for automatic model-based test generation with UML state machines. We describe improvements for the whole process of automatic model-based test generation: The first contribution is a new test generation algorithm that is a combination of guided depth-first graph search and backward abstract interpretation. It allows to combine control-flow-based, data-flow-based, or transition-based coverage criteria with boundary-based coverage criteria. The second contribution are test model transformations as a means to influence the impact of coverage criteria that are applied to test models. Most important, we present the simulated coverage criteria satisfaction as a means to satisfy a coverage criterion on the original test model by satisfying another (e.g. weaker or unrelated) coverage criterion on the transformed test model. We define coverage criteria combinations, define new coverage criteria, and also show how to use model transformations to implement both. Together with the first contribution, this allows to satisfy a combination of control-flow-based, transition-based, data-flow-based, and boundary-based coverage criteria. As the third contribution, we show how to combine test models to decrease test costs, to support better coverage of requirements, and to provide more information for the model-based test generation algorithm. The presented test generation algorithm can be applied to combined test models. The fourth contribution is an investigation of the impact of the test goal order on test suite efficiency. All contributions are focused on improvements for automatic model-based test generation. We showed that they can be used in conjunction to combine their advantages. The fifth and final contribution is the corresponding prototype implementation and the experiences from generating test suites for the presented standard examples, the academic test models, and the test model of the industrial cooperation.

Bibliography

[AADO00] Abdurazik, Aynur; Ammann, Paul; Ding, Wei; Offutt, Jeff: Evaluation of Three Specification-Based Testing Criteria. In: *IEEE International Conference on Engineering of Complex Computer Systems*, p. 179, 2000. doi:http://doi.ieeecomputersociety.org/10.1109/ICECCS.2000.873943.

[AB00] Ammann, Paul; Black, Paul E.: Test Generation and Recognition with Formal Methods. citeseer.ist.psu.edu/ammann00test.html, 2000.

[AB05] Artho, Cyrille; Biere, Armin: Combined Static and Dynamic Analysis. In: *AIOOL'05: Proceedings of the 1st International Workshop on Abstract Interpretation of Object-Oriented Languages*, ENTCS. Elsevier Science, Paris, France, 2005.

[ABDPL02] Antoniol, Giuliano; Briand, Lionel C.; Di Penta, Massimiliano; Labiche, Yvan: A Case Study Using the Round-Trip Strategy for State-Based Class Testing. In: *ISSRE'02: Proceedings of the 13th International Symposium on Software Reliability Engineering*, p. 269. IEEE Computer Society, Washington, DC, USA, 2002. ISBN 0-8186-1763-3.

[ABL05] Andrews, James H.; Briand, Lionel C.; Labiche, Yvan: Is Mutation an Appropriate Tool for Testing Experiments? In: *ICSE'05: Proceedings of the 27th International Conference on Software Engineering*, pp. 402–411. ACM, New York, NY, USA, 2005. ISBN 1-59593-963-2. doi:http://doi.acm.org/10.1145/1062455.1062530.

[ABLN06] Andrews, James H.; Briand, Lionel C.; Labiche, Yvan; Namin, Akbar S.: Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria. In: *IEEE Transactions on Software Engineering*, volume 32:pp. 608–624, 2006.

[ABM98] Ammann, Paul E.; Black, Paul E.; Majurski, William: Using Model Checking to Generate Tests from Specifications. In: *ICFEM'98: Proceedings of the Second IEEE International Conference on Formal Engineering Methods*, p. 46. IEEE Computer Society, Washington, DC, USA, 1998. ISBN 0-8186-9198-0.

[Abr07] Abrial, Jean-Raymond: Formal Methods: Theory Becoming Practice. In: *Journal of Universal Computer Science*, volume 13(5):pp. 619–628, 2007.

[AFGC03] Andrews, Anneliese Amschler; France, Robert B.; Ghosh, Sudipto; Craig, Gerald: Test Adequacy Criteria for UML Design Models. In: *Software Testing, Verification Reliability*, volume 13(2):pp. 95–127, 2003.

[AO00] Abdurazik, Aynur; Offutt, Jeff: Using UML Collaboration Diagrams for Static Checking and Test Generation. In: Evans, Andy; Kent, Stuart; Selic, Bran, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939, pp. 383–395. Springer, 2000.

[AO08] Ammann, Paul; Offutt, Jeff: *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 2008. ISBN 9780521880381.

[AOH03] Ammann, Paul; Offutt, Jeff; Huang, Hong: Coverage Criteria for Logical Expressions. In: *ISSRE'03: Proceedings of the 14th International Symposium on Software Reliability Engineering*, p. 99. IEEE Computer Society, Washington, DC, USA, 2003. ISBN 0-7695-2007-3.

[Arc08] Arcuri, Andrea: On the Automation of Fixing Software Bugs. In: *ICSE Companion'08: Companion of the 30th International Conference on Software Engineering*, pp. 1003–1006. ACM, New York, NY, USA, 2008. ISBN 978-1-60558-079-1. doi: http://doi.acm.org/10.1145/1370175.1370223.

[AS05] Aichernig, Bernhard K.; Salas, Percy Antonio Pari: Test Case Generation by OCL Mutation and Constraint Solving. In: *International Conference on Quality Software*, pp. 64–71,

2005. ISSN 1550-6002. doi:http://doi.ieeecomputersociety.org/10.1109/QSIC.2005.63.

[ATF09] Afzal, Wasif; Torkar, Richard; Feldt, Robert: A Systematic Review of Search-based Testing for Non-functional System Properties. In: *Information and Software Technology*, volume 51(6):pp. 957–976, 2009. ISSN 0950-5849. doi:http://dx.doi.org/10.1016/j.infsof.2008.12.005.

[ATP+07] Alekseev, Sergej; Tollkühn, P.; Palaga, P.; Dai, Zhen R.; Hoffmann, A.; Rennoch, Axel; Schieferdecker, Ina: Reuse of Classification Tree Models for Complex Software Projects. In: *Conference on Quality Engineering in Software Technology (CONQUEST)*. 2007.

[AUW08] Aydal, Emine G.; Utting, Mark; Woodcock, Jim: A Comparison of State-Based Modelling Tools for Model Validation. In: *TOOLS (46)*, pp. 278–296. 2008.

[Bar03] Baral, Chitta: *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003. ISBN 9780521818025.

[BB00] Basanieri, Francesca; Bertolino, Antonia: A Practical Approach to UML-based Derivation of Integration Tests. In: *4th International Software Quality Week Europe*. November 2000.

[BBH02] Benattou, Mohammed; Bruel, Jean-Michel; Hameurlain, Nabil: Generating Test Data from OCL Specification. In: *ECOOP'2002 - Workshop on Integration and Transformation of UML models (WITUML'02)*. 2002.

[BBM] Basanieri, Francesca; Bertolino, Antonia; Marchetti, Eda: CoWTeSt: A Cost Weighted Test Strategy. citeseer.ist.psu.edu/basanieri01cowtest.html.

[BBM+01] Basanieri, Francesca; Bertolino, Antonia; Marchetti, Eda; Ribolini, Alberto; Lombardi, Gaetano; Nucera, Giovanni: An Automated Test Strategy Based on UML Diagrams. In: *Proceeding of the Ericsson Rational User Conference*, October 2001.

[BBvB+01] Beck, Kent; Beedle, Mike; van Bennekum, Arie; Cockburn, Alistair; Cunningham, Ward; Fowler, Martin; Grenning, James;

Highsmith, Jim; Hunt, Andrew; Jeffries, Ron; Kern, Jon; Marick, Brian; Martin, Robert C.; Mellor, Steve; Schwaber, Ken; Sutherland, Jeff; Thomas, Dave: Manifesto for agile software development. http://www.agilemanifesto.org/, 2001.

[BDAR97] Bourhfir, C.; Dssouli, R.; Aboulhamid, E.; Rico, N.: Automatic Executable Test Case Generation for Extended Finite State Machine Protocols. In: *International Workshop on Testing Communicating Systems (IWTCS'97)*, pp. 75–90. 1997.

[BDG+07] Baker, Paul; Dai, Zhen Ru; Grabowski, Jens; Haugen, Øystein; Schieferdecker, Ina; Williams, Clay: *Model-Driven Testing: Using the UML Testing Profile*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007. ISBN 3540725628.

[BDL05] Basin, David; Doser, Jürgen; Lodderstedt, Torsten: Model Driven Security. In: Broy, Manfred; Grünbauer, Johannes; Harel, David; Hoare, Tony, editors, *Engineering Theories of Software Intensive Systems*, pp. 353–398. Springer, 2005.

[BDOS08] Barrett, Clark; Deters, Morgan; Oliveras, Albert; Stump, Aaron: Design and results of the 3rd annual satisfiability modulo theories competition (SMT-COMP 2007). In: *International Journal on Artificial Intelligence Tools*, volume 17(4):pp. 569–606, 2008.

[Bec00] Beck, Kent: *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000. ISBN 978-0201616415.

[Bec02] Beck, Kent: *Test Driven Development: By Example*. Addison-Wesley Professional, November 2002. ISBN 0321146530.

[Bei90] Beizer, Boris: *Software Testing Techniques*. John Wiley & Sons, Inc., New York, NY, USA, 1990. ISBN 0442245920.

[Bel91] Bell Labs: SPIN Model Checker. http://www.spinroot.com/, 1991.

[BEN04] Berkelaar, Michel; Eikland, Kjell; Notebaert, Peter: lp_solve 5.1. http://lpsolve.sourceforge.net/5.5/, 2004.

[Ber00] Bertolino, A.: Knowledge Area Description of Software Testing SWEBOK. http://www.swebok.org, 2000.

[BFG00] Bozga, Marius; Fernandez, Jean-Claude; Ghirvu, Lucian: Using Static Analysis to Improve Automatic Test Generation. In: *TACAS'00: Proceedings of the 6th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pp. 235–250. Springer-Verlag, London, UK, 2000. ISBN 3-540-67282-6.

[BFJT02] Baudry, Benoit; Fleurey, Franck; Jezequel, Jean-Marc; Traon, Yves Le: Automatic Test Cases Optimization Using a Bacteriological Adaptation Model: Application to .NET Components. In: *Proceedings of ASE'02: Automated Software Engineering, Edinburgh*. 2002.

[BFPT06] Badban, Bahareh; Fränzle, Martin; Peleska, Jan; Teige, Tino: Test automation for hybrid systems. In: *Proceedings of the 3rd international workshop on Software quality assurance*, SOQUA '06, pp. 14–21. ACM, New York, NY, USA, 2006. ISBN 1-59593-584-3. doi:http://doi.acm.org/10.1145/1188895.1188902. URL `http://doi.acm.org/10.1145/1188895.1188902`.

[BFS+06] Brottier, Erwan; Fleurey, Franck; Steel, Jim; Baudry, Benoit; Traon, Yves Le: Metamodel-based Test Generation for Model Transformations: an Algorithm and a Tool. In: *ISSRE'06: Proceedings of the 17th International Symposium on Software Reliability Engineering*, pp. 85–94. IEEE Computer Society, Washington, DC, USA, 2006. ISBN 0-7695-2684-5. doi:http://dx.doi.org/10.1109/ISSRE.2006.27.

[BGM91] Bernot, Gilles; Gaudel, Marie Claude; Marre, Bruno: Software Testing Based on Formal Specifications: A Theory and a Tool. In: *Software Engineering Journal*, volume 6(6):pp. 387–405, 1991. ISSN 0268-6961.

[BGN+03] Barnett, Michael; Grieskamp, Wolfgang; Nachmanson, Lev; Schulte, Wolfram; Tillmann, Nikolai; Veanes, Margus: Towards a Tool Environment for Model-Based Testing with AsmL. In: *Proceedings of Formal Approaches to Testing of Software (FATES)*, pp. 252–266. 2003.

[BHvMW09] Biere, Armin; Heule, Marijn; van Maaren, Hans; Walsh, Toby, editors: *Handbook of Satisfiability*, volume 185 of *Frontiers in*

Artificial Intelligence and Applications. IOS Press, 2009. ISBN 978-1-58603-929-5.

[Bin99] Binder, Robert V.: *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0-201-80938-9.

[BJK05] Broy, Manfred; Jonsson, Bengt; Katoen, Joost P.: *Model-Based Testing of Reactive Systems: Advanced Lectures (Lecture Notes in Computer Science)*. Springer, August 2005. ISBN 3540262784. doi:http://dx.doi.org/http://dx.doi.org/10.1007/b137241.

[BLC05] Briand, Lionel C.; Labiche, Yvan; Cui, Jim: Automated Support for Deriving Test Requirements from UML Statecharts. In: *Software and Systems Modeling*, volume V4(4):pp. 399–423, November 2005. doi:10.1007/s10270-005-0090-5.

[BLL05] Briand, Lionel C.; Labiche, Yvan; Lin, Qing: Improving Statechart Testing Criteria Using Data Flow Information. In: *ISSRE'05: Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering*, pp. 95–104. IEEE Computer Society, Washington, DC, USA, 2005. ISBN 0-7695-2482-6. doi:http://dx.doi.org/10.1109/ISSRE.2005.24.

[BM83] Bird, David L; Munoz, Carlos Urias: Automatic Generation of Random Self-Checking Test Cases. In: *IBM Systems Journal*, volume 22(3):pp. 229–245, 1983. ISSN 0018-8670.

[BMM05] Bertolino, Antonia; Marchetti, Eda; Muccini, Henry: Introducing a Reasonably Complete and Coherent Approach for Model-based Testing. In: *Electronic Notes in Theoretical Computer Science*, volume 116:pp. 85–97, 2005.

[Bow88] Bowser, John H.: Reference Manual for Ada Mutant Operators. Technical Report GIT-SERC-88/02, Georgia Institute of Technology, 1988.

[BOY00] Black, Paul E.; Okun, Vadim; Yesha, Yaacov: Mutation Operators for Specifications. In: *ASE'00: Proceedings of the 15th IEEE International Conference on Automated Software Engineering*, p. 81. IEEE Computer Society, Washington, DC, USA, 2000. ISBN 0-7695-0710-7.

[BRJ98] Booch, Grady; Rumbaugh, Jim; Jacobson, Ivar: *The Unified Modeling Language User Guide*. Addison-Wesley, 1998. ISBN 0-201-57168-4.

[Bro87] Brooks, Frederick P., Jr.: No Silver Bullet: Essence and Accidents of Software Engineering. In: *Computer Journal*, volume 20(4):pp. 10–19, 1987.

[BSME08] Bauer, Thomas; Stallbaum, Heiko; Metzger, Andreas; Eschbach, Robert: Risikobasierte Ableitung und Priorisierung von Testfällen für den modellbasierten Systemtest. In: Herrmann, Korbinian; Brügge, Bernd, editors, *Software Engineering*, volume 121 of *Lecture Notes in Informatics*, pp. 99–111. GI, 2008. ISBN 978-3-88579-215-4.

[BSST09] Barrett, Clark W.; Sebastiani, Roberto; Seshia, Sanjit A.; Tinelli, Cesare: Satisfiability modulo theories. In: Biere et al. [BHvMW09], pp. 825–885.

[BSV08] Budnik, Christof J.; Subramanyan, Rajesh; Vieira, Marlon: Peer-to-Peer Comparison of Model-Based Test Tools. In: Hegering, Heinz-Gerd; Lehmann, Axel; Ohlbach, Hans Jürgen; Scheideler, Christian, editors, *GI Jahrestagung (1)*, volume 133 of *Lecture Notes in Informatics*, pp. 223–226. GI, 2008. ISBN 978-3-88579-227-7.

[Bur01] Burton, Simon: *Automated Generation of High Integrity Tests from Graphical Specifications*. Ph.D. thesis, University of York, 2001.

[CC92] Cousot, Patrick; Cousot, Radhia: Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation, invited paper. In: Bruynooghe, M.; Wirsing, M., editors, *Proceedings of the International Workshop Programming Language Implementation and Logic Programming (PLILP '92)*, Leuven, Belgium, 13–17 August 1992, Lecture Notes in Computer Science 631, pp. 269–295. Springer-Verlag, Berlin, Germany, 1992.

[CC04] Cousot, Patrick; Cousot, Radhia: *Basic Concepts of Abstract Interpretation*, pp. 359–366. Kluwer Academic Publishers, 2004.

[CCF+03] Cousot, Patrick; Cousot, Radhia; Feret, Jérôme; Mauborgne, Laurent; Miné, Antoine; Rival, Xavier: ASTRÉE Static Analyzer. http://www.astree.ens.fr/, 2003.

[CCL98] Canfora, Gerardo; Cimitile, Aniello; Lucia, Andrea De: Conditioned Program Slicing. In: *Information & Software Technology*, volume 40(11-12):pp. 595–607, 1998.

[Cer01] Certification Authorities Software Team: Position Paper-6: Rationale for Accepting Masking MC/DC in Certification Projects, 2001.

[CGP00] Clarke, Edmund M.; Grumberg, Orna; Peled, Doron A.: *Model Checking*. MIT Press, 2000. ISBN 0-262-03270-8.

[CH03] Czarnecki, Krzysztof; Helsen, Simon: Classification of Model Transformation Approaches. In: *OOPSLA 2003 Workshop on Generative Techniques in the context of Model Driven Architecture*. October 2003.

[Chi01] Chilenski, John Joseph: MCDC Forms (Unique-Cause, Masking) versus Error Sensitivity. In: *white paper submitted to NASA Langley Research Center under contract NAS1-20341*. January 2001.

[Cho95] Chow, Tsun S.: Testing Software Design Modeled by Finite-State Machines. In: *Conformance testing methodologies and architectures for OSI protocols*, pp. 391–400, 1995.

[CHR82] Clarke, Lori A.; Hassell, Johnette; Richardson, Debra J.: A Close Look at Domain Testing. In: *IEEE Transactions on Software Engineering*, volume 8(4):pp. 380–390, 1982. ISSN 0098-5589. doi:http://doi.ieeecomputersociety.org/10.1109/TSE.1982.235572.

[CIvdPS05] Calame, Jens R.; Ioustinova, Natalia; van de Pol, Jaco; Sidorova, Natalia: Data Abstraction and Constraint Solving for Conformance Testing. In: *APSEC '05: Proceedings of the 12th Asia-Pacific Software Engineering Conference*, pp. 541–548. IEEE Computer Society, Washington, DC, USA, 2005. ISBN 0-7695-2465-6. doi:http://dx.doi.org/10.1109/APSEC.2005.57.

[CK93] Cheng, Kwang Ting; Krishnakumar, A. S.: Automatic Functional Test Generation Using the Extended Finite State Machine Model. In: *DAC'93: Proceedings of the 30th International Conference on Design Automation*, pp. 86–91. ACM Press, New York, NY, USA, 1993. ISBN 0-89791-577-1. doi: http://doi.acm.org/10.1145/157485.164585.

[CKM+02] Cook, Steve; Kleppe, Anneke; Mitchell, Richard; Rumpe, Bernhard; Warmer, Jos; Wills, Alan: The Amsterdam Manifesto on OCL. In: *Object Modeling with the OCL, The Rationale behind the Object Constraint Language*, pp. 115–149. Springer-Verlag, London, UK, 2002. ISBN 3-540-43169-1.

[CLOM06] Ciupa, Ilinca; Leitner, Andreas; Oriol, Manuel; Meyer, Bertrand: Object Distance and Its Application to Adaptive Random Testing of Object-Oriented Programs. In: *RT'06: Proceedings of the 1st International Workshop on Random Testing*, pp. 55–63. ACM Press, New York, NY, USA, 2006. ISBN 1-59593-457-X. doi:http://doi.acm.org/10.1145/1145735.1145744.

[CLOM07] Ciupa, Ilinca; Leitner, Andreas; Oriol, Manuel; Meyer, Bertrand: Experimental Assessment of Random Testing for Object-Oriented Software. In: *ISSTA'07: Proceedings of the International Symposium on Software Testing and Analysis 2007*, pp. 84–94. 2007.

[CM94] Chilenski, John Joseph; Miller, Steven P.: Applicability of Modified Condition/Decision Coverage to Software Testing. In: *Software Engineering Journal, Issue*, volume 9, pp. 193–200. September 1994.

[CN00] Cavalcanti, Ana; Naumann, David A.: A Weakest Precondition Semantics for Refinement of Object-Oriented Programs. In: *IEEE Transactions on Software Engineering*, volume 26(8):pp. 713–728, 2000. ISSN 0098-5589. doi:http://doi.ieeecomputersociety.org/10.1109/32.879810.

[Con] Conformiq: Qtronic. http://www.conformiq.com/.

[Con09] Consortium, OW2: Sat4j 2.1. http://www.sat4j.org/, 2009.

[Cou00] Cousot, Patrick: Abstract Interpretation Based Program Testing. In: *In Proc. SSGRR 2000 Computer & eBusiness International Conference, Compact disk paper 248 and electronic proceedings http://www.ssgrr.it/en/ ssgrr2000/proceedings.htm, 2000. Scuola Superiore G. Reiss Romoli*. 2000.

[Cou03] Cousot, Patrick: Automatic Verification by Abstract Interpretation. In: *VMCAI'03: Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation*, pp. 20–24. Springer-Verlag, London, UK, 2003. ISBN 3-540-00348-7.

[CPL+08] Ciupa, Ilinca; Pretschner, Alexander; Leitner, Andreas; Oriol, Manuel; Meyer, Bertrand: On the Predictability of Random Tests for Object-Oriented Software. In: *ICST'08: Proceedings of the First International Conference on Software Testing, Verification and Validation*. April 2008.

[CPRZ85] Clarke, Lori A.; Podgurski, Andy; Richardson, Debra J.; Zeil, Steven J.: A Comparison of Data Flow Path Selection Criteria. In: *ICSE'85: Proceedings of the 8th International Conference on Software Engineering*, pp. 244–251. IEEE Computer Society Press, Los Alamitos, CA, USA, 1985. ISBN 0-8186-0620-7.

[CS05] Csallner, Christoph; Smaragdakis, Yannis: Check 'n' Crash: Combining Static Checking and Testing. In: *ICSE'05: Proceedings of the 27th International Conference on Software Engineering*, pp. 422–431. ACM, New York, NY, USA, 2005. ISBN 1-59593-963-2. doi:http://doi.acm.org/10.1145/1062455.1062533.

[CSE96] Callahan, John; Schneider, Francis; Easterbrook, Steve: Automated Software Testing Using Model-Checking. In: *Proceedings 1996 SPIN Workshop*. August 1996. Also WVU Technical Report NASA-IVV-96-022.

[CTF02] Chevalley, Philippe; Thevenod-Fosse, Pascale: A Mutation Analysis Tool for Java Programs. In: *STTT: International Journal on Software Tools for Technology Transfer*, volume 5:pp. 90–103, 2002.

[DDB+05] Dai, Zhen Ru; Deussen, Peter H.; Busch, Maik; Lacmene, Laurette Pianta; Ngwangwen, Titus; Herrmann, Jens; Schmidt,

Michael: Automatic Test Data Generation for TTCN-3 using CTE. In: *International Conference Software and Systems Engineering and their Applications (ICSSEA)*. December 2005.

[DEFT09] Drechsler, Rolf; Eggersglüß, Stephan; Fey, Görschwin; Tille, Daniel: *Test Pattern Generation using Boolean Proof Engines*. Springer, 2009. ISBN 978-90-481-2359-9.

[DF93] Dick, Jeremy; Faivre, Alain: Automating the Generation and Sequencing of Test Cases from Model-Based Specifications. In: *FME'93: Proceedings of the First International Symposium of Formal Methods Europe on Industrial-Strength Formal Methods*, pp. 268–284. Springer-Verlag, London, UK, 1993. ISBN 3-540-56662-7.

[Dij76] Dijkstra, Edsger W.: *A Discipline of Programming*. Prentice-Hall, 1976.

[dL01] de Lucia, Andrea: Program Slicing: Methods and Applications. In: *First IEEE International Workshop on Source Code Analysis and Manipulation*, pp. 142–149. IEEE Computer Society Press, Los Alamitos, California, USA, November 2001.

[DLS78] DeMillo, Richard A.; Lipton, Richard J.; Sayward, Fred G.: Hints on Test Data Selection: Help for the Practicing Programmer. In: *Computer Journal*, volume 11(4):pp. 34–41, 1978. ISSN 0018-9162. doi:http://dx.doi.org/10.1109/C-M.1978.218136.

[DM96] Delamaro, Márcio E.; Maldonado, José C.: Proteum - A Tool for the Assessment of Test Adequacy for C Programs. In: *PCS96: Conference on Performability in Computing Systems*, pp. 79 – 95. July 1996.

[DV09] Dahl, Joachim; Vandenberghe, Lieven: CVXOPT 1.1.1. http://abel.ee.ucla.edu/cvxopt/, 2009.

[DVB+09] Denecker, Marc; Vennekens, Joost; Bond, Stephen; Gebser, Martin; Truszczyński, Miroslaw: The Second Answer Set Programming Competition. In: Erdem, E.; Lin, F.; Schaub, T., editors, *Proceedings of the Tenth International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'09)*, volume 5753 of *Lecture Notes in Artificial Intelligence*, pp. 637–654. Springer-Verlag, 2009.

[DW09] Dziobek, Christian; Weiland, Jens: Variantenmodellierung und -konfiguration eingebetteter automotive Software mit Simulink. In: *MBEES'09: Model-Based Development of Embedded Systems*. April 2009.

[Ecl05] Eclipse: Eclipse Object Constraint Language (OCL) Plugin. http://www.eclipse.org/modeling/mdt/?project=ocl#ocl, 2005.

[Ecl07a] Eclipse: Eclipse Modeling Framework Project (EMF). http://www.eclipse.org/emf/, 2007.

[Ecl07b] Eclipse: Model Development Tools (MDT) - UML2. www.eclipse.org/uml2/, 2007.

[Ecl09] Eclipse: Atlas Transformation Language (ATL) 3.0.0. http://www.eclipse.org/m2m/atl/, 2009.

[EFYvB02] El-Fakih, Khaled; Yevtushenko, Nina; von Bochmann, Gregor: FSM-based Re-Testing Methods. In: *TestCom'02: Proceedings of the IFIP 14th International Conference on Testing Communicating Systems XIV*, pp. 373–390. Kluwer, B.V., Deventer, The Netherlands, 2002. ISBN 0-7923-7695-1.

[EG06] Erich Gamma, Kent Beck: JUnit 4.1 - A Testing Framework for Java. http://www.junit.org, 2006.

[Elm73] Elmendorf, W. R.: Cause-Effect Graphs in Functional Testing, 1973. doi:TR-00.2487.

[EM04] Engler, Dawson; Musuvathi, Madanlal: Static Analysis Versus Software Model Checking for Bug Finding. citeseer.ist.psu.edu/engler04static.html, 2004.

[EMR02] Elbaum, Sebastian; Malishevsky, Alexey; Rothermel, Gregg: Test Case Prioritization: A Family of Empirical Studies. In: *IEEE Transactions on Software Engineering*, volume 28:pp. 159–182, 2002.

[Ern03] Ernst, Michael D.: Static and dynamic analysis: Synergy and duality. In: *WODA'03: ICSE Workshop on Dynamic Analysis*, pp. 24–27. Portland, Oregon, USA, May 9, 2003.

[Eve09] Eve Software Utilities: Emma 1.0. http://www.eveutilities.com/products/emma, 2009.

[EW00] Eshuis, Rik; Wieringa, Roel: Requirements-Level Semantics for UML Statecharts. In: *Fourth International Conference on Formal Methods for Open Object-based Distributed Systems*, pp. 121–140. Kluwer Academic Publishers, Norwell, MA, USA, 2000. ISBN 0-7923-7923-3.

[Fav] Favre, Jean-Marie: Meta-Model and Model Co-evolution within the 3D Software Space. citeseer.ist.psu.edu/favre03metamodel.html.

[FDMM94] Fabbri, S.C. Pinto Ferraz; Delamaro, Marcio Eduardo; Maldonado, Jose Carlos; Masiero, P.C.: Mutation Analysis Testing for Finite State Machines. In: *Proceedings of the 5th International Symposium on Software Reliability Engineering*. 1994.

[FHH+01] Fox, Chris; Harman, Mark; Hierons, Rob; Ph, Ub; Danicic, Sebastian: Backward Conditioning: A new Program Specialisation Technique and its Application to Program Comprehension. citeseer.ist.psu.edu/fox01backward.html, 2001.

[Fin00] Finger, Frank: Design and Implementation of a Modular OCL Compiler. Diploma Thesis, Dresden University of Technology, Germany, 2000.

[FS07] Friske, Mario; Schlingloff, Holger: Improving Test Coverage for UML State Machines Using Transition Instrumentation. In: Saglietti, Francesca; Oster, Norbert, editors, *SAFECOMP'07: The International Conference on Computer Safety, Reliability and Security*, volume 4680 of *Lecture Notes in Computer Science*, pp. 301–314. Springer, 2007. ISBN 978-3-540-75100-7.

[FSW08] Friske, Mario; Schlingloff, Holger; Weißleder, Stephan: Composition of Model-based Test Coverage Criteria. In: *MBEES'08: Model-Based Development of Embedded Systems*. 2008.

[FW88] Frankl, Phyllis G.; Weyuker, Elaine J.: An Applicable Family of Data Flow Testing Criteria. In: *IEEE Transactions on Software Engineering*, volume 14(10):pp. 1483–1498, 1988. ISSN 0098-5589. doi:http://dx.doi.org/10.1109/32.6194.

[FW08a] Fraser, Gordon; Wotawa, Franz: Ordering Coverage Goals in Model Checker Based Testing. In: *ICSTW'08: Proceedings of the 2008 IEEE International Conference on Software Testing*

Verification and Validation Workshop, pp. 31–40, 2008. doi: http://doi.ieeecomputersociety.org/10.1109/ICSTW.2008.31.

[FW08b] Fraser, Gordon; Wotawa, Franz: Using Model-Checkers to Generate and Analyze Property Relevant Test-Cases. In: *Software Quality Journal, 16 (2)*, pp. 161–183, 2008.

[Gel08] Gelfond, Michael: Answer Sets. In: Lifschitz, Vladimir; van Hermelen, Frank; Porter, Bruce, editors, *Handbook of Knowledge Representation*, chapter 7. Elsevier, 2008.

[GG93] Grochtmann, Matthias; Grimm, Klaus: Classification Trees for Partition Testing. In: *STVR: Software Testing, Verification and Reliability*, volume 3(2):pp. 63–82, 1993.

[GH99] Gargantini, Angelo; Heitmeyer, Constance: Using Model Checking to Generate Tests from Requirements Specifications. In: *ACM SIGSOFT Software Engineering Notes*, volume 24(6):pp. 146–162, 1999. ISSN 0163-5948. doi:http://doi.acm.org/10.1145/318774.318939.

[Gim85] Gimpel Software: PC-Lint for C/C++. http://www.gimpel.com/, 1985.

[GJK+09] Gent, Ian; Jefferson, Chris; Kotthoff, Lars; Miguel, Ian; Moore, Neil; Nightingale, Peter; Petrie, Karen; Rendl, Andreas: MINION 0.9. http://minion.sourceforge.net/, 2009.

[GKPR08] Grönninger, Hans; Krahn, Holger; Pinkernell, Claas; Rumpe, Bernhard: Modeling Variants of Automotive Systems using Views. In: *Modellierung08*. Humboldt-University of Berlin, 2008.

[GLRW04] Geppert, Birgit; Li, J. Jenny; Rößler, Frank; Weiss, David M.: Towards Generating Acceptance Tests for Product Lines. In: *ICSR: International Conference on Software Reuse: Methods, Techniques and Tools*, pp. 35–48. 2004.

[GMS98] Gupta, Neelam; Mathur, Aditya P.; Soffa, Mary Lou: Automated Test Data Generation Using an Iterative Relaxation Method. In: *SIGSOFT'98/FSE-6: Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 231–244. ACM, New York, NY, USA,

1998. ISBN 1-58113-108-9. doi:http://doi.acm.org/10.1145/288195.288321.

[GMS99] Gupta, Neelam; Mathur, Aditya P.; Soffa, Mary Lou: UNA Based Iterative Test Data Generation and its Evaluation. In: *ASE'99: Proceedings of the 14th IEEE International Conference on Automated Software Engineering*, p. 224. IEEE Computer Society, Washington, DC, USA, 1999. ISBN 0-7695-0415-9.

[GNRS09] Götz, Helmut; Nickolaus, Markus; Roßner, Thomas; Salomon, Knut: iX-Studie: Modellbasiertes Testen. http://www.heise.de/kiosk/special/ixstudie/09/01/, 2009.

[Gom04] Gomaa, Hassan: *Designing Software Product Lines with UML: From Use Cases to Pattern-based Software Architectures.* Addison-Wesley, 2004. ISBN 978-0201775952.

[Gri81] Gries, David: *The science of programming.* Springer, 1981.

[Gut99] Gutjahr, Walter J.: Partition Testing vs. Random Testing: The Influence of Uncertainty. In: *IEEE Transactions on Software Engineering*, volume 25(5):pp. 661–674, 1999. ISSN 0098-5589. doi:http://dx.doi.org/10.1109/32.815325.

[GW85] Girgis, Moheb R.; Woodward, Martin R.: An Integrated System for Program Testing Using Weak Mutation and Data Flow Analysis. In: *Proceedings of the 8th International Conference on Software Engineering.* 1985.

[GWZ94] Goldberg, Allen; Wang, Tie-Chen; Zimmermann, David: Applications of Feasible Path Analysis to Program Testing. In: *ISSTA'94: International Symposium on Software Testing and Analysis*, pp. 80–94. 1994.

[HAA+06] Holt, Nina E.; Anda, Bente C. D.; Asskildt, Knut; Briand, Lionel C.; Endresen, Jan; Frøystein, Sverre: Experiences with Precise State Modeling in an Industrial Safety Critical System. In: *CSDUML'06: Critical Systems Development Using Modeling Languages*, pp. 68–77. Springer, 2006. ISBN 0809-1021.

[Ham77] Hamlet, Richard G.: Testing Programs with the Aid of a Compiler. In: *IEEE Transactions on Software Engineering*, volume 3(4):pp. 279–290, 1977. ISSN 0098-5589. doi:http://dx.doi.org/10.1109/TSE.1977.231145.

[Har87] Harel, David: Statecharts: A Visual Formalism for Complex Systems. In: *Science of Computer Programming*, volume 8(3):pp. 231–274, 1987. ISSN 0167-6423. doi:http://dx.doi.org/10.1016/0167-6423(87)90035-9.

[HCH+99] Hamie, Ali; Civello, Franco; Howse, John; Kent, Stuart J. H.; Mitchell, Richard: Reflections on the Object Constraint Language. In: Muller, Pierre-Alain; Bézivin, Jean, editors, *Proceedings of the International Conference on the Unified Modelling Language (UML) 1998, Mulhouse, France*, 1618, pp. 162–172. Springer-Verlag, 1999.

[HD95] Harman, Mark; Danicic, Sebastian: Using Program Slicing to Simplify Testing. In: *Software Testing, Verification & Reliability*, volume 5(3):pp. 143–162, 1995.

[HFH+02] Harman, Mark; Fox, Chris; Hierons, Rob; Hu, Lin; Danicic, Sebastian; Wegener, Joachim: VADA: A Transformation-Based System for Variable Dependence Analysis. In: *IEEE International Workshop on Source Code Analysis and Manipulation*, p. 55, 2002. doi:http://doi.ieeecomputersociety.org/10.1109/SCAM.2002.1134105.

[HG97] Harel, David; Gery, Eran: Executable Object Modeling with Statecharts. In: *IEEE Computer Journal*, volume 30:pp. 31–42, 1997.

[HGK06] Hammer, C.; Grimme, M.; Krinke, J.: Dynamic path conditions in dependence graphs. In: *Workshop on Partial Evaluatian and Program Manipulation*. 2006.

[HHF+02] Hierons, Robert M.; Harman, Mark; Fox, Chris; Ouarbya, Lahcen; Daoudi, Mohammed: Conditioned Slicing Supports Partition Testing. In: *Software Testing, Verification and Reliability*. 2002.

[HHH+04] Harman, Mark; Hu, Lin; Hierons, Rob; Wegener, Joachim; Sthamer, Harmen; Baresel, André; Roper, Marc: Testability Transformation. In: *IEEE Transactions on Software Engineering*, volume 30(1):pp. 3–16, 2004. ISSN 0098-5589. doi:http://dx.doi.org/10.1109/TSE.2004.1265732.

[HHL+07] Harman, Mark; Hassoun, Youssef; Lakhotia, Kiran; McMinn, Phil; Wegener, Joachim: The Impact of Input Domain Reduction on Search-Based Test Data Generation. In: *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*, pp. 155–164. ACM, New York, NY, USA, 2007. ISBN 978-1-59593-811-4. doi:http://doi.acm.org/10.1145/1287624.1287647.

[HHS03] Hierons, Robert M.; Harman, Mark; Singh, Harbhajan: Automatically Generating Information from a Z Specification to Support the Classification Tree Method. In: Bert, Didier; Bowen, Jonathan P.; King, Steve; Waldén, Marina A., editors, *ZB'03: Formal Specification and Development in Z and B, Third International Conference of B and Z Users, Turku, Finland, June 4-6, 2003, Proceedings*, volume 2651 of *Lecture Notes in Computer Science*, pp. 388–407. Springer, 2003. ISBN 3-540-40253-5.

[HLL94] Horgan, Joseph R.; London, Saul; Lyu, Michael R.: Achieving Software Quality with Testing Coverage Measures. In: *Computer Journal*, volume 27(9):pp. 60–69, 1994. ISSN 0018-9162. doi:http://dx.doi.org/10.1109/2.312032.

[HLSC01] Hong, Hyoung; Lee, Insup; Sokolsky, Oleg; Cha, Sung: Automatic Test Generation from Statecharts Using Model Checking. In: *In Proceedings of FATES'01 Workshop on Formal Approaches to Testing of Software, volume NS-01-4 of BRICS Notes Series*. 2001.

[HM90] Horgan, Joseph R.; Mathur, Aditya P.: Weak Mutation is Probably Strong Mutation. Technical Report SERC-TR-83-P, Software Engineering Research Center, Purdue University, 1990.

[HM07] Harman, Mark; McMinn, Phil: A Theoretical & Empirical Analysis of Evolutionary Testing and Hill Climbing for Structural Test Data Generation. In: *ISSTA'07: Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pp. 73–83. ACM, New York, NY, USA, 2007. ISBN 978-1-59593-734-6. doi:http://doi.acm.org/10.1145/1273463.1273475.

[HN96] Harel, David; Naamad, Amnon: The STATEMATE Semantics of Statecharts. In: *ACM Transactions on Software Engineering and Methodology*, volume 5(4):pp. 293–333, 1996. ISSN 1049-331X. doi:http://doi.acm.org/10.1145/235321.235322.

[HN04] Hartman, Alan; Nagin, Kenneth: The AGEDIS Tools for Model Based Testing. In: *ACM SIGSOFT Software Engineering Notes*, volume 29(4):pp. 129–132, 2004. ISSN 0163-5948. doi:http://doi.acm.org/10.1145/1013886.1007529.

[Hon01] Hong, Hyoung S. and Lee, Insup: Automatic Test Generation from Specifications for Control-flow and Data-flow Coverage Criteria. In: *Monterey Workshop, California: Naval Postgraduate School*, pp. 230–246. June 2001.

[Hop47] Hopper, Grace Murray: The First Computer Bug was a Moth. http://www-history.mcs.st-andrews.ac.uk/Biographies/Hopper.html, 1947.

[How76] Howden, William E.: Reliability of the Path Analysis Testing Strategy. In: *IEEE Transactions on Software Engineering*, volume 2(3):pp. 208–214, September 1976.

[How82] Howden, William E.: Weak Mutation Testing and Completeness of Test Sets. In: *IEEE Transactions on Software Engineering*, volume 8(4):pp. 371–379, 1982.

[HT90] Hamlet, Dick; Taylor, Ross: Partition Testing Does Not Inspire Confidence (Program Testing). In: *IEEE Transactions on Software Engineering*, volume 16(12):pp. 1402–1411, 1990. ISSN 0098-5589. doi:http://dx.doi.org/10.1109/32.62448.

[HVL+99] Havelund, Klaus; Visser, Willem; Lerda, Flavio; Pasareanu, Corina; Penix, John; Mansouri-Samani, Masoud; O'Malley, Owen; Giannakopoulou, Dimitra; Mehlitz, Peter; Dillinger, Peter: Java PathFinder. http://javapathfinder.sourceforge.net/, 1999.

[IBM] IBM (Telelogic): Rhapsody Automated Test Generation. http://www.telelogic.com/products/rhapsody.

[IBM04] IBM (Telelogic) I-Logix: *Rhapsody Automatic Test Generator, Release 2.3, User Guide*, 2004.

[IKV] IKV++ Technologies AG: medini QVT. http://projects.ikv.de/qvt.

[IPT+07] Irvine, Sean A.; Pavlinic, Tin; Trigg, Leonard; Cleary, John Gerald; Inglis, Stuart J.; Utting, Mark: Jumble Java Byte Code to Measure the Effectiveness of Unit Tests. In: *TAICPART-MUTATION: Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*. 2007.

[ITC98] ITC-IRST and Carnegie Mellon University and University of Genoa and University of Trento: SMV. http://www.cs.cmu.edu/ modelcheck/smv.html, 1998.

[ITC99] ITC-IRST and Carnegie Mellon University and University of Genoa and University of Trento: NuSMV. http://nusmv.fbk.eu/, 1999.

[JBW+94] Jasper, Robert; Brennan, Mike; Williamson, Keith; Currier, Bill; Zimmerman, David: Test Data Generation and Feasible Path Analysis. In: *ISSTA'94: Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 95–107. ACM, New York, NY, USA, 1994. ISBN 0-89791-683-2. doi:http://doi.acm.org/10.1145/186258.187150.

[Jet00] JetBrains: IntelliJ IDEA. http://www.jetbrains.com/, 2000.

[JHS03] Jones, James A.; Harrold, Mary Jean; Society, Ieee Computer: Test-suite Reduction and Prioritization for Modified Condition/Decision Coverage. In: *IEEE Transactions on Software Engineering*, volume 29:pp. 92–101, 2003.

[JM99] Jéron, Thierry; Morel, Pierre: Test Generation Derived from Model-Checking. In: Halbwachs, N.; Peled, D., editors, *CAV'99: Proceedings of the 11th International Conference on Computer Aided Verification*, volume 1633 of *LNCS*, pp. 108–122. Springer-Verlag, London, UK, July 1999. ISBN 3-540-66202-2.

[Jos99] Jos Warmer and Anneke Kleppe: *The Object Constraint Language : Precise Modeling with UML*. Addison Wesley Longman, Inc., 1999. ISBN 978-0201379402.

[JVSJ06] Jalote, Pankaj; Vangala, Vipindeep; Singh, Taranbir; Jain, Prateek: Program Partitioning: A Framework for Combining Static and Dynamic Analysis. In: *WODA'06: Proceedings of the 2006 International Workshop on Dynamic Systems Analysis*, pp. 11–16. ACM Press, New York, NY, USA, 2006. ISBN 1-59593-400-6. doi:http://doi.acm.org/10.1145/1138912.1138916.

[JW01] Joachim Wegener, André Baresel, Harmen Sthamer: Application Fields for Evolutionary Testing. In: *Eurostar: Proceedings of the 9th European International Conference on Software Testing Analysis & Review*, 2001.

[KFN99] Kaner, Cem; Falk, Jack; Nguyen, Hung Quoc: *Testing Computer Software, 2nd Ed.* John Wiley and Sons, Inc., New York, USA, 1999. ISBN 0-471-35846-0.

[KG04] Khor, Susan; Grogono, Peter: Using a Genetic Algorithm and Formal Concept Analysis to Generate Branch Coverage Test Data Automatically. In: *ASE'04: Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, pp. 346–349. IEEE Computer Society, Washington, DC, USA, 2004. ISBN 0-7695-2131-2. doi:http://dx.doi.org/10.1109/ASE.2004.71.

[KHBC99] Kim, Y.; Hong, H.; Bae, D.; Cha, S.: Test Cases Generation from UML State Diagrams. citeseer.ist.psu.edu/kim99test.html, 1999.

[KLPU04] Kosmatov, Nikolai; Legeard, Bruno; Peureux, Fabien; Utting, Mark: Boundary Coverage Criteria for Test Generation from Formal Models. In: *ISSRE'04: Proceedings of the 15th International Symposium on Software Reliability Engineering*, pp. 139–150. IEEE Computer Society, Washington, DC, USA, 2004. ISBN 0-7695-2215-7. doi:http://dx.doi.org/10.1109/ISSRE.2004.12.

[KN04] Kishi, Tomoji; Noda, Natsuko: Design Testing for Product Line Development based on Test Scenarios. In: *SPLiT: Software Product Line Testing Workshop*. Boston, MA, 2004.

[Kol03] Kolb, Ronny: A Risk-Driven Approach for Efficiently Testing Software Product Lines. citeseer.ist.psu.edu/630355.html, 2003.

[Kor90] Korel, Bogdan: Automated Software Test Data Generation. In: *IEEE Transactions on Software Engineering*, volume 16(8):pp. 870–879, 1990. ISSN 0098-5589. doi:http://dx.doi.org/10.1109/32.57624.

[KRS08] Kahsai, Temesghen; Roggenbach, Markus; Schlingloff, Holger: Specification-Based Testing for Software Product Lines. In: *SEFM'08: Proceedings of the 2008 Sixth IEEE International Conference on Software Engineering and Formal Methods*, pp. 149–158. IEEE Computer Society, Washington, DC, USA, 2008. ISBN 978-0-7695-3437-4. doi:http://dx.doi.org/10.1109/SEFM.2008.38.

[Küs06] Küster, Jochen M.: Definition and Validation of Model Transformations. In: *Software and Systems Modeling*, volume V5(3):pp. 233–259, 2006. doi:10.1007/s10270-006-0018-8.

[KSW01] Kösters, Georg; Six, Hans-Werner; Winter, Mario: Coupling Use Cases and Class Models as a Means for Validation and Verification of Requirements Specifications. citeseer.ist.psu.edu/ksters01coupling.html, 2001.

[LBE+05] Lamberg, K.; Beine, M.; Eschmann, M.; Otterbach, R.; Conrad, M.; Fey, I.: Model-based Testing of Embedded Automotive Software using MTest, July 2005.

[LCI03] LCI: Object Constraint Language Environment 2.0. http://lci.cs.ubbcluj.ro/ocle/, 2003.

[LHM08] Lakhotia, Kiran; Harman, Mark; McMinn, Phil: Handling Dynamic Data Structures in Search Based Testing. In: *GECCO'08: Proceedings of the 10th annual conference on Genetic and Evolutionary Computation*, pp. 1759–1766. ACM, New York, NY, USA, 2008. ISBN 978-1-60558-130-9. doi:http://doi.acm.org/10.1145/1389095.1389435.

[Lis88] Liskov, Barbara: Keynote Address - Data Abstraction and Hierarchy. In: *ACM SIGPLAN Notices*, volume 23(5):pp. 17–34, 1988. ISSN 0362-1340. doi:http://doi.acm.org/10.1145/62139.62141.

[LJX+04] Linzhang, Wang; Jiesong, Yuan; Xiaofeng, Yu; Jun, Hu; Xuandong, Li; Guoliang, Zheng: Generating Test Cases from UML

Activity Diagram based on Gray-Box Method. In: *Asia-Pacific Software Engineering Conference*, pp. 284–291, 2004. ISSN 1530-1362. doi:http://doi.ieeecomputersociety.org/10.1109/APSEC.2004.55.

[LKL02] Lee, Kwanwoo; Kang, Kyo Chul; Lee, Jaejoon: Concepts and Guidelines of Feature Modeling for Product Line Software Engineering. In: *ICSR'07: Proceedings of the 7th International Conference on Software Reuse*, pp. 62–77. Springer-Verlag, London, UK, 2002. ISBN 3-540-43483-6.

[LMM99] Latella, Diego; Majzik, Istvan; Massink, Mieke: Towards a Formal Operational Semantics of UML Statechart Diagrams. In: *FMOODS: Proceedings of the IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems*, p. 465. Kluwer, B.V., Deventer, The Netherlands, The Netherlands, 1999. ISBN 0-7923-8429-6.

[LPU02] Legeard, Bruno; Peureux, Fabrice; Utting, Mark: Automated Boundary Testing from Z and B. In: *Formal Methods Europe*, pp. 21–40. Springer Verlag LNCS 2391, 2002.

[lT06] le Traon, Yves: Design by Contract to Improve Software Vigilance. In: *IEEE Transactions on Software Engineering*, volume 32(8):pp. 571–586, 2006. ISSN 0098-5589. doi:http://dx.doi.org/10.1109/TSE.2006.79. Member-Benoit Baudry and Member-Jean-Marc Jezequel.

[Mar91] Marick, Brian: The Weak Mutation Hypothesis. In: *TAV4: Proceedings of the symposium on Testing, Analysis, and Verification*, pp. 190–199. ACM, New York, NY, USA, 1991. ISBN 0-89791-449-X. doi:http://doi.acm.org/10.1145/120807.120825.

[May05] Mayer, Johannes: On Testing Image Processing Applications with Statistical Methods. In: Liggesmeyer, Peter; Pohl, Klaus; Goedicke, Michael, editors, *Software Engineering*, volume 64 of *Lecture Notes in Informatics*, pp. 69–78. GI, 2005. ISBN 3-88579-393-8.

[McG01] McGregor, John D.: Testing a Software Product Line. In: *Technical Report CMU/SEI-2001-TR-022*, 2001.

[McG05] McGregor, John D.: Reasoning about the Testability of Product Line Components. In: *SPLiT: Proceedings of the International Workshop on Software Product Line Testing*, pp. 1–7. September 2005.

[McM04] McMinn, Phil: Search-based Software Test Data Generation: A Survey: Research Articles. In: *STVR: Software Testing, Verification and Reliability*, volume 14(2):pp. 105–156, 2004. ISSN 0960-0833. doi:http://dx.doi.org/10.1002/stvr.v14:2.

[MD93] McGregor, John D.; Dyer, Douglas M.: A Note on Inheritance and State Machines. In: *ACM SIGSOFT Software Engineering Notes*, volume 18(4):pp. 61–69, 1993. ISSN 0163-5948. doi:http://doi.acm.org/10.1145/163626.163635.

[Mic09] Microsoft Research: SpecExplorer. http://research.microsoft.com/en-us/projects/SpecExplorer/, 2009.

[MMS97] Mcgraw, Gary; Michael, Christoph; Schatz, Michael: Generating Software Test Data by Evolution. In: *IEEE Transactions on Software Engineering*, volume 27:pp. 1085–1110, 1997.

[MMSC98] MacColl, Ian; Murray, Leesa; Strooper, Paul A.; Carrington, David A.: Specification-Based Class Testing: A Case Study. In: *International Conference on Formal Engineering Methods*, pp. 222–. 1998.

[MMWW09] Martin, Robert C.; Martin, Micah D.; Wilson-Welsh, Patrick: FitNesse User Guide. http://fitnesse.org, 2009.

[Mor83] Morell, Larry Joe: *A Theory of Error-based Testing*. Ph.D. thesis, University of Maryland at College Park, College Park, MD, USA, 1983.

[Mor90] Morell, Larry Joe: A Theory of Fault-Based Testing. In: *IEEE Transactions on Software Engineering*, volume 16(8):pp. 844–857, 1990. ISSN 0098-5589. doi:http://dx.doi.org/10.1109/32.57623.

[MS04] Mansour, Nashat; Salame, Miran: Data Generation for Path Testing. In: *Software Quality Control*, volume 12(2):pp. 121–136, 2004. ISSN 0963-9314. doi:http://dx.doi.org/10.1023/B:SQJO.0000024059.72478.4e.

[MS06] Mayer, Johannes; Schneckenburger, Christoph: An Empirical Analysis and Comparison of Random Testing Techniques. In: *ISESE'06: Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, pp. 105–114. ACM Press, New York, NY, USA, 2006. ISBN 1-59593-218-6. doi:http://doi.acm.org/10.1145/1159733.1159751.

[Mye79] Myers, Glenford J.: *Art of Software Testing*. John Wiley & Sons, Inc., New York, NY, USA, 1979. ISBN 0471043281.

[Nag04] Nagy, Roman: Bedeutung von Ausgangszuständen beim Testen von objektorientierter Software. In: *CoMaTech'04*. Trnava, Slowakei, 2004.

[NAM08] Namin, Akbar Siami; Andrews, James H.; Murdoch, Duncan J.: Sufficient Mutation Operators for Measuring Test Effectiveness. In: *ICSE'08: Proceedings of the 30th International Conference on Software Engineering*, pp. 351–360. ACM, New York, NY, USA, 2008. ISBN 978-1-60558-079-1. doi: http://doi.acm.org/10.1145/1368088.1368136.

[NF06] Nebut, Clémentine; Fleurey, Franck: Automatic Test Generation: A Use Case Driven Approach. In: *IEEE Transactions on Software Engineering*, volume 32(3):pp. 140–155, 2006. ISSN 0098-5589. doi:http://dx.doi.org/10.1109/TSE.2006.22. Member-Yves Le Traon and Member-Jean-Marc Jezequel.

[NFTJ03] Nebut, Clémentine; Fleurey, Franck; Traon, Yves Le; Jézéquel, Jean-Marc: Requirements by Contracts allow Automated System Testing. In: *ISSRE'03: Proceedings of the 14th. IEEE International Symposium on Software Reliability Engineering*, pp. 17–21. 2003.

[Nta01] Ntafos, Simeon C.: On Comparisons of Random, Partition, and Proportional Partition Testing. In: *IEEE Transactions on Software Engineering*, volume 27(10):pp. 949–960, 2001. ISSN 0098-5589. doi:http://doi.ieeecomputersociety.org/10.1109/32.962563.

[OA99] Offutt, Jeff; Abdurazik, Aynur: Generating Tests from UML Specifications. In: France, Robert; Rumpe, Bernhard, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA,*

October 28-30. 1999, Proceedings, volume 1723, pp. 416–429. Springer, 1999.

[OB88] Ostrand, Thomas J.; Balcer, Marc J.: The Category-Partition Method for Specifying and Generating Fuctional Tests. In: *Communications of the ACM*, volume 31(6):pp. 676–686, 1988. ISSN 0001-0782. doi:http://doi.acm.org/10.1145/62959.62964.

[OB03] Okun, Vadim; Black, Paul E.: Issues in Software Testing with Model Checkers. citeseer.ist.psu.edu/okun03issues.html, 2003.

[OB04] Oliver Bühler, Joachim Wegener: Automatic Testing of an Autonomous Parking System using Evolutionary Computation, 2004.

[Obj05a] Object Management Group: Object Constraint Language (OCL), version 2.0. http://www.uml.org, 2005.

[Obj05b] Object Management Group: UML Testing Profile. http://www.omg.org/technology/documents/formal/test_profile.htm, 2005.

[Obj06] Object Management Group: MetaObject Facility (MOF) 2.0. http://www.omg.org/mof/, 2006.

[Obj07] Object Management Group: Unified Modeling Language (UML), version 2.1. http://www.uml.org, 2007.

[ODC06] Owen, David; Desovski, Dejan; Cukic, Bojan: Random Testing of Formal Software Models and Induced Coverage. In: *Random Testing*, pp. 20–27. 2006.

[Off88] Offutt, Andrew Jefferson, VI: *Automatic Test Data Generation*. Ph.D. thesis, Georgia Institute of Technology, Atlanta, GA, USA, 1988. Director: Demillo, R. A.

[Off92] Offutt, A. Jefferson: Investigations of the Software Testing Coupling Effect. In: *ACM Transactions on Software Engineering and Methodology*, volume 1(1):pp. 5–20, 1992. ISSN 1049-331X. doi:http://doi.acm.org/10.1145/125489.125473.

[OG05] Olimpiew, Erika Mir; Gomaa, Hassan: Model-Based Testing for Applications Derived from Software Product Lines. In: *ACM SIGSOFT Software Engineering Notes*, volume 30(4):pp.

1–7, 2005. ISSN 0163-5948. doi:http://doi.acm.org/10.1145/1082983.1083279.

[OK87] Offutt, A. Jefferson, VI; King, Kim N.: A Fortran 77 Interpreter for Mutation Analysis. In: *SIGPLAN'87: Papers of the Symposium on Interpreters and Interpretive Techniques*, pp. 177–188. ACM, New York, NY, USA, 1987. ISBN 0-89791-235-7. doi:http://doi.acm.org/10.1145/29650.29669.

[OL91] Offutt, A. Jefferson; Lee, Stephen D.: How Strong is Weak Mutation? In: *Proceedings of the Symposium on Testing, Analysis, and Verification*. 1991.

[OL94] Offutt, A. Jefferson; Lee, Stephen D.: An Empirical Evaluation of Weak Mutation. In: *IEEE Transactions on Software Engineering*, volume 20(5):pp. 337–344, 1994.

[OLR+96] Offutt, A. Jefferson; Lee, Ammei; Rothermel, Gregg; Untch, Roland H.; Zapf, Christian: An Experimental Determination of Sufficient Mutant Operators. In: *ACM Transactions on Software Engineering and Methodology*, volume 5(2):pp. 99–118, 1996.

[Ope09] Open Source: TopCased UML Editor 3.0. http://www.topcased.org/, 2009.

[Opt07] Optimization Department of Cybernetic Institute: OpenOpt. http://openopt.org/, 2007.

[OVP96] Offutt, A. Jefferson; Voas, Jeff; Payn, Jeff: Mutation Operators for Ada. Technical report, Information and Software Systems Engineering, George Mason University, 1996.

[OWB04] Ostrand, Thomas; Weyuker, Elaine J.; Bell, Robert: Using Static Analysis to Determine Where to Focus Dynamic Testing Effort. In: *WODA'04: Workshop on Dynamic Analysis*. May 2004.

[Par81] Park, David: Concurrency and automata on infinite sequences. In: *Proceedings of the 5th GI-Conference on Theoretical Computer Science*, pp. 167–183. Springer-Verlag, London, UK, 1981. ISBN 3-540-10576-X.

[Par05] Paradkar, Amit: Case Studies on Fault Detection Effectiveness of Model Based Test Generation Techniques. In: *A-MOST'05: Proceedings of the 1st International Workshop on Advances in Model-Based Testing*, pp. 1–7. ACM Press, New York, NY, USA, 2005. ISBN 1-59593-115-5. doi:http://doi.acm.org/10.1145/1083274.1083286.

[PE05] Pacheco, Carlos; Ernst, Michael D.: Eclat: Automatic Generation and Classification of Test Inputs. In: *ECOOP'05 — Object-Oriented Programming, 19th European Conference*, pp. 504–527. Glasgow, Scotland, July 27–29, 2005.

[PHP99] Pargas, Roy P.; Harrold, Mary Jean; Peck, Robert R.: Test-Data Generation Using Genetic Algorithms. In: *Software Testing, Verification And Reliability*, volume 9:pp. 263–282, 1999.

[Pik09] PikeTec: TPT (Time Partition Testing) Version 3.2. http://www.piketec.com/, 2009.

[PIS02] PISATEL LAB: http://www1.isti.cnr.it/ERI/special.htm, 2002.

[PLK07] Peleska, Jan; Löding, Helge; Kotas, Tatiana: Test Automation Meets Static Analysis. In: Koschke, Rainer; Herzog, Otthein; Rödiger, Karl-Heinz; Ronthaler, Marc, editors, *GI Jahrestagung (2)*, volume 110 of *Lecture Notes in Informatics*, pp. 280–290. GI, 2007. ISBN 978-3-88579-204-8.

[PM06] Pohl, Klaus; Metzger, Andreas: Software Product Line Testing. In: *Communications of the ACM*, volume 49(12):pp. 78–81, 2006. ISSN 0001-0782. doi:http://doi.acm.org/10.1145/1183236.1183271.

[PML08] Peleska, Jan; Möller, Oliver; Löding, Helge: A Formal Introduction to Model-Based Testing. http://www.informatik.uni-bremen.de/agbs/jp/papers/peleska_ictac2008_tutorial.html, 2008.

[PP03] Prowell, Stacy J.; Poore, Jesse H.: Foundations of sequence-based software specification. In: *IEEE Trans. Softw. Eng.*, volume 29(5):pp. 417–429, 2003. ISSN 0098-5589. doi:http://dx.doi.org/10.1109/TSE.2003.1199071.

[PPW+05] Pretschner, Alexander; Prenninger, Wolfgang; Wagner, Stefan; Kühnel, Christian; Baumgartner, Martin; Sostawa, Bernd; Zölch, Rüdiger; Stauner, Thomas: One Evaluation of Model-based Testing and its Automation. In: *ICSE'05: Proceedings of the 27th International Conference on Software Engineering*, pp. 392–401. 2005. ISBN 1-59593-963-2. doi:http://doi.acm.org/10.1145/1062455.1062529.

[Pre03] Pretschner, Alexander: Compositional Generation of MC/DC Integration Test Suites. In: *Electronic Notes in Theoretical Computer Science*, volume 82(6):pp. 1–11, 2003.

[Pre06] Pretschner, Alexander: Zur Kosteneffektivität des modellbasierten Testens. In: *MBEES'06: Modellbasierte Entwicklung eingebetteter Systeme*, pp. 85–94. 2006.

[Pro04] Prowell, Stacy: State of the Art of Model-Based Testing with Markov Chain Usage Models. http://www.andrew.cmu.edu/user/sprowell/pubs/prowell2004 soa_pres.pdf, 2004.

[PS96] Price, Christopher J.; Snooke, Price: Automated Sneak Identification. In: *Engineering Applications of Artificial Intelligence*, volume 9:pp. 423–427, 1996.

[PS07] Pinte, Florin; Saglietti, Francesca: UnITeD - Unterstützung Inkrementeller TestDaten. http://www11.informatik.uni-erlangen.de/Forschung/Projekte/United/ index.html, 2007.

[PTV97] Paradkar, Amit; Tai, K. C.; Vouk, M. A.: Specification-Based Testing Using Cause-Effect Graphs. In: *Annals of Software Engineering*, volume 4:pp. 133–157, 1997.

[PW02] Pierro, Alessandra Di; Wiklicky, Herbert: Probabilistic Abstract Interpretation and Statistical Testing. In: *PAPM-PROBMIV '02: Proceedings of the Second Joint International Workshop on Process Algebra and Probabilistic Methods, Performance Modeling and Verification*, pp. 211–212. Springer-Verlag, London, UK, 2002. ISBN 3-540-43913-7.

[PZ07] Peleska, Jan; Zahlten, Cornelia: Integrated Automated Test Case Generation and Static Analysis. In: *QA+Test 2007: International Conference on QA+Testing Embedded Systems*. 2007.

[Ran03] Ranville, Scott: MCDC Test Vectors From Matlab Models – Automatically. In: *Embedded Systems Conference*. San Francisco, USA, March 2003.

[Rat97] Rat der IT-Beauftragten: V-Modell. http://www.cio.bund.de, 1997.

[Rav08] Ravindran, A. Ravi, editor: *Operations Research and Management Science Handbook*. CRC Press, 2008. ISBN 978-0849397219.

[Rea09] Reactive Systems Inc.: Reactis. http://www.reactive-systems.com/, 2009.

[REG76] Rankin, John P.; Engels, Gary J.; Godoy, Sylvia G.: Software Sneak Circuit Analysis. Technical report, AFNL-TR-75-254. Boeing Aerospace Co., Houston, Texas, 1976.

[RG98] Richters, Mark; Gogolla, Martin: On Formalizing the UML Object Constraint Language OCL. In: Ling, Tok-Wang; Ram, Sudha; Lee, Mong Li, editors, *Proceedings of 17th International Conference on Conceptual Modeling (ER)*, volume 1507, pp. 449–464. Springer-Verlag, 1998.

[Rob06] Robinson, Harry: Model-Based Testing. model.based.testing.googlepages.com/starwest-2006-mbt-tutorial.pdf, 2006.

[RTC92] RTCA Inc.: RTCA/DO-178B, Software Considerations in Airborne Systems and Equipment Certification, December 1992.

[RvBW06] Rossi, Francesca; van Beek, Peter; Walsh, Toby, editors: *Handbook of Constraint Programming*. Elsevier, 2006. ISBN 978-0-444-52726-4.

[RWH08] Rajan, Ajitha; Whalen, Michael W.; Heimdahl, Mats P.E.: The Effect of Program and Model Structure on MC/DC Test Adequacy Coverage. In: *ICSE'08: Proceedings of the 30th International Conference on Software Engineering*, pp. 161–170. ACM, New York, NY, USA, 2008. ISBN 978-1-60558-079-1. doi:http://doi.acm.org/10.1145/1368088.1368111.

[SBT+09] Sharygina, Natasha; Bruttomesso, Robertoe; Tsitovich, Aliaksei; Rollini, Simone; Tonetta, Stefano;

Braghin, Chiara; Barone-Adesi, Katerina: OpenSMT. http://verify.inf.unisi.ch/opensmt, 2009.

[SBW01] Sthamer, Harmen; Baresel, André; Wegener, Joachim: Evolutionary Testing of Embedded Systems. In: *QW'01: Proceedings of the 14th International Internet & Software Quality Week*, pp. 1–34. 2001.

[SEBC09] Schläpfer, Michael; Egea, Marina; Basin, David; Clavel, Manuel: Automatic Generation of Security-Aware GUI Models. In: Bagnato, A., editor, *SECMDA'09: European Workshop on Security in Model Driven Architecture 2009, Enschede (The Netherlands), June 24, 2009. Proceedings.*, CTIT proceedings WP09-06, pp. 42–56. Centre for Telematics and Information Technology, University of Twente, June 2009. ISBN 0920-0672. ISSN 978-90-365-2857-3.

[SG03] Schieferdecker, Ina; Grabowski, Jens: The Graphical Format of TTCN-3 in the Context of MSC and UML. In: *SAM'02: Telecommunications and beyond: The BroaderApplicability of SDL and MSC*, volume 2599 of *LNCS*. Springer, Rosslyn, VA, USA, 2003.

[SHS03] Seifert, Dirk; Helke, Steffen; Santen, Thomas: Test Case Generation for UML Statecharts. In: Broy, Manfred; Zamulin, Alexandre V., editors, *PSI'03: Perspectives of System Informatics*, volume 2890 of *Lecture Notes in Computer Science*, pp. 462–468. Springer-Verlag, 2003.

[SK00] Schroeder, Patrick J.; Korel, Bogdan: Black-box Test Reduction Using Input-Output Analysis. In: *ISSTA'00: Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 173–177. ACM Press, New York, NY, USA, 2000. ISBN 1-58113-266-2. doi: http://doi.acm.org/10.1145/347324.349042.

[SKAB03] Schroeder, Patrick J.; Kim, Eok; Arshem, Jerry; Bolaki, Pankaj: Combining Behavior and Data Modeling in Automated Test Case Generation. In: *QSIC'03: Proceedings of the 3rd International Conference on Quality Software*, p. 247. IEEE Computer Society, Washington, DC, USA, 2003. ISBN 0-7695-2015-4.

[Sma] Smartesting: Test Designer. http://www.smartesting.com.

[Sok06a] Sokenou, Dehla: Generating Test Sequences from UML Sequence Diagrams and State Diagrams. In: *INFORMATIK 2006: Informatik für Menschen - Band 2, GI-Edition: Lecture Notes in Informatics (LNI), P-94*, pp. 236–240. Gesellschaft für Informatik, 2006.

[Sok06b] Sokenou, Dehla: *UML-basierter Klassen- und Integrationstest objektorientierter Programme*. Ph.D. thesis, Technische Universität Berlin, Germany, 2006.

[Som01] Sommerville, Ian: *Software Engineering*. Addison-Wesley, New York, USA, 2001.

[Sou08] Sourceforge: CppUnit 1.12 – Unit Tests for C++. http://sourceforge.net/projects/cppunit, 2008.

[Spi92] Spivey, Mike: *The Z Notation: A Reference Manual*. Prentice-Hall International Series in Computer Science, 1992. ISBN 0139785299.

[Sun95] Sun Microsystems: Java. http://java.sun.com/, 1995.

[SVG+08] Santiago, Valdivino; Vijaykumar, N. L.; Guimar Danielle, aes; Amaral, Ana Silvia; Ferreira, Érica: An Environment for Automated Test Case Generation from Statechart-Based and Finite State Machine-based Behavioral Models. In: *ICSTW'08: Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop*, pp. 63–72. IEEE Computer Society, Washington, DC, USA, 2008. ISBN 978-0-7695-3388-9. doi:http://dx.doi.org/10.1109/ICSTW.2008.7.

[SW96] Stumptner, Markus; Wotawa, Franz: Model-Based Program Debugging and Repair. In: *IEA/AIE'96: 9th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, pp. 155–160. 1996.

[SW07] Smith, Benjamin Hatfield; Williams, Laurie: An Empirical Evaluation of the MuJava Mutation Operators. In: *TAICPART-MUTATION: Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*. 2007.

[SW08] Sadilek, Daniel A.; Weißleder, Stephan: Testing Metamodels. In: Ina Schieferdecker, Alan Hartman, editor, *ECMDA'08: European Conference on Model Driven Architecture*. Springer, June 2008. ISBN 978-3-540-69095-5.

[SW09] Smith, Ben H.; Williams, Laurie: Should Software Testers Use Mutation Analysis to Augment a Test Set? In: *Journal of Systems and Software*, volume 82(11):pp. 1819–1832, 2009. ISSN 0164-1212. doi:http://dx.doi.org/10.1016/j.jss.2009.06.031.

[TCFR96] Tip, Frank; Choi, Jong-Deok; Field, John; Ramalingam, G.: Slicing Class Hierarchies in C++. In: *OOPSLA'96: Proceedings of the 11th ACM SIGPLAN conference on Object-Oriented Orogramming, Systems, Languages, and Applications*, pp. 179–197. ACM Press, New York, NY, USA, 1996. ISBN 0-89791-788-X. doi:http://doi.acm.org/10.1145/236337.236355.

[Tel] Telcordia Technologies: The AETG System: An Approach to Testing Based on Combinatorial Design. http://aetgweb.argreenhouse.com.

[The94] The Mathworks Inc.: Polyspace Embedded Software Verification. http://www.mathworks.com/products/polyspace/index.html, 1994.

[The09] The Choco Team: Choco Solver 2.1.0. http://choco.emn.fr/, 2009.

[Tre04] Tretmans, Jan: Model-Based Testing: Property Checking for Real. Keynote Address at the International Workshop for Construction and Analysis of Safe Secure, and Interoperable Smart Devices. http://www-sop.inria.fr/everest/events/cassis04, 2004.

[UL06] Utting, Mark; Legeard, Bruno: *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006. ISBN 0123725011.

[UPL06] Utting, Mark; Pretschner, Alexander; Legeard, Bruno: A Taxonomy of Model-Based Testing. Technical Report 04/2006, Department of Computer Science, The Universiy of Waikato (New Zealand), 2006.

[UTC+07] Utting, Mark; Trigg, Len; Cleary, John G.; Irvine, Archmage; Pavlinic, Tin: Jumble. http://jumble.sourceforge.net/, 2007.

[Utt08] Utting, Mark: The Role of Model-Based Testing. In: *Verified Software: Theories, Tools, Experiments: First IFIP TC 2/WG 2.3 Conference, VSTTE 2005, Zurich, Switzerland, October 10-13, 2005, Revised Selected Papers and Discussions*, pp. 510–517, 2008. doi:http://dx.doi.org/10.1007/978-3-540-69149-5_56.

[vMF09] van Maaren, Hans; Franco, John: The international sat competitions web page. http://www.satcompetition.org/, 2009.

[Wah03] Wah, K. S. How Tai: An Analysis of the Coupling Effect I: Single Test Data. In: *Science of Computer Programming*, volume 48(2-3):pp. 119–161, 2003. ISSN 0167-6423. doi:http://dx.doi.org/10.1016/S0167-6423(03)00022-4.

[WC80] White, Lee J.; Cohen, Edward I.: A Domain Strategy for Computer Program Testing. In: *IEEE Transactions on Software Engineering*, volume 6(3):pp. 247–257, 1980. ISSN 0098-5589. doi:http://dx.doi.org/10.1109/TSE.1980.234486.

[Weia] Weißleder, Stephan: Coverage Simulator (Transforming test models to simulate coverage criteria). http://covsim.sourceforge.net.

[Weib] Weißleder, Stephan: ParTeG (Partition Test Generator). http://parteg.sourceforge.net.

[Wei79] Weiser, Mark David: *Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method*. Ph.D. thesis, University of Michigan, Ann Arbor, MI, USA, 1979.

[Wei89] Weiss, Steward N.: Comparing Test Data Adequacy Criteria. In: *ACM SIGSOFT Software Engineering Notes*, volume 14(6):pp. 42–49, 1989. ISSN 0163-5948. doi:http://doi.acm.org/10.1145/70739.70742.

[Wei08] Weißleder, Stephan: Partition-Oriented Test Generation. In: Hegering, Heinz-Gerd; Lehmann, Axel; Ohlbach, Hans Jürgen; Scheideler, Christian, editors, *3rd Workshop about Model-based Testing (MoTes) in conjunction with the Annual Congress of*

the Gesellschaft für Informatik (GI), volume 133 of *Lecture Notes in Informatics*, pp. 199–204. GI, 2008. ISBN 978-3-88579-227-7.

[Wei09a] Weißleder, Stephan: Influencing Factors in Model-Based Testing with UML State Machines: Report on an Industrial Cooperation. In: *Models'09: 12th International Conference on Model Driven Engineering Languages and Systems*. October 2009.

[Wei09b] Weißleder, Stephan: Semantic-Preserving Test Model Transformations for Interchangeable Coverage Criteria. In: *MBEES'09: Model-Based Development of Embedded Systems*. April 2009.

[Wei10] Weißleder, Stephan: Simulated Satisfaction of Coverage Criteria on UML State Machines. In: *International Conference on Software Testing, Verification, and Validation (ICST)*. April 2010.

[Wey93] Weyuker, Elaine J.: More Experience with Data Flow Testing. In: *IEEE Transactions on Software Engineering*, volume 19(9):pp. 912–919, 1993. ISSN 0098-5589. doi:http://dx.doi.org/10.1109/32.241773.

[WFPW07] Wehrmeister, Marco A.; Freitas, Edison P.; Pereira, Carlos E.; Wagner, Flávio R.: Applying Aspect-Orientation Concepts in the Model-driven Design of Distributed Embedded Real-Time Systems. In: *ISORC'07 - 10th IEEE Symposium on Object-Oriented Real-Time Distributed Computing*. may 2007.

[WH88] Woodward, Martin R.; Halewood, K.: From Weak to Strong, Dead or Alive? An Analysis of Some Mutation Testing Issues. In: *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*. 1988.

[Whi91] Whitty, Robin W.: An exercise in weakest preconditions. In: *Software Testing, Verification & Reliability*, volume 1(1):pp. 39–43, 1991.

[Whi02] Whittaker, James A.: *How to Break Software: A Practical Guide to Testing with CDrom*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. ISBN 0201796198.

[Win93] Winskel, Glynn: *The Formal Semantics of Programming Languages*. The MIT Press, 1993.

[WJ91] Weyuker, Elaine J.; Jeng, Bingchiang: Analyzing Partition Testing Strategies. In: *IEEE Transactions on Software Engineering*, volume 17(7):pp. 703–711, 1991. ISSN 0098-5589. doi:http://dx.doi.org/10.1109/32.83906.

[WK03] Warmer, Jos; Kleppe, Anneke: *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison Wesley, 2nd edition, 2003.

[WKC06] Wang, Junhua; Kim, Soon-Kyeong; Carrington, David: Verifying Metamodel Coverage of Model Transformations. In: *ASWEC'06: Proceedings of the Australian Software Engineering Conference (ASWEC'06)*, pp. 270–282. IEEE Computer Society, Washington, DC, USA, 2006. ISBN 0-7695-2551-2. doi:http://dx.doi.org/10.1109/ASWEC.2006.55.

[WL05] Wappler, Stefan; Lammermann, Frank: Using Evolutionary Algorithms for the Unit Testing of Object-Oriented Software. In: *GECCO'05: Proceedings of the conference on Genetic and Evolutionary Computation*, pp. 1053–1060. ACM Press, New York, NY, USA, 2005. ISBN 1-59593-010-8. doi:http://doi.acm.org/10.1145/1068009.1068187.

[WS07] Wappler, Stefan; Schieferdecker, Ina: Improving Evolutionary Class Testing in the Presence of Non-Public Methods. In: *ASE'07: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pp. 381–384. ACM, New York, NY, USA, 2007. ISBN 978-1-59593-882-4. doi:http://doi.acm.org/10.1145/1321631.1321689.

[WS08a] Weißleder, Stephan; Schlingloff, Holger: Quality of Automatically Generated Test Cases based on OCL Expressions. In: *ICST'08: International Conference on Software Testing, Verification, and Validation*, pp. 517–520. IEEE Computer Society, 2008.

[WS08b] Weißleder, Stephan; Sokenou, Dehla: Cause-Effect Graphs for Test Models Based on UML and OCL. In: *27. Treffen der GI-Fachgruppen TAV und RE*. June 2008.

[WSS08] Weißleder, Stephan; Sokenou, Dehla; Schlingloff, Holger: Reusing State Machines for Automatic Test Generation in Product Lines. In: Thomas Bauer, Axel Rennoch, Hajo Eichler, editor, *Model-Based Testing in Practice (MoTiP)*. Fraunhofer IRB Verlag, June 2008. ISBN 978-3-8167-7624-6.

[ZG03] Ziemann, Paul; Gogolla, Martin: Validating OCL Specifications with the USE Tool — An Example Based on the BART Case Study. In: *FMICS'03: Formal Methods for Industrial Critical Systems*. 2003.

List of Figures

List of Tables

Danksagung

Die Arbeit an meiner Dissertation mitsamt der Themenfindung, den Veröffentlichungen und dem Aufschreiben war für mich eine sehr wichtige und interessante Erfahrung. Ich möchte mich bei den Menschen bedanken, die diese Arbeit möglich gemacht haben. Dazu gehören in erster Linie meine Familie und insbesondere Barbara Weißleder, Henriette Barbe und Tobias Weißleder, die mir in den schwersten Phasen der Arbeit immer wieder Kraft gegeben haben. Dazu gehört ebenso mein Betreuer Holger Schlingloff, dem ich insbesondere für die Hilfestellung bei der ersten Themenfindung sowie für die kurzfristigen und hilfreichen Kommentare so kurz vor der nächsten Einreichungsfrist danke. Weiterhin möchte ich auch allen Beteiligten danken, die das DFG-Graduiertenkolleg 1324 (METRIK) möglich gemacht haben. In diesem Zuge möchte ich Joachim Fischer und die Deutsche Forschungsgemeinschaft hervorheben. Weiterhin gilt mein Dank insbesondere Daniel Sadilek für die fruchtbare Zusammenarbeit und Markus Scheidgen für das angenehme Arbeitsklima. Ebenso hervorzuheben sind die Diskussionen mit Guido Wachsmuth, Dirk Fahland, Stefan Brüning und Siamak Haschemi. Außerhalb des Graduiertenkollegs möchte ich Roman Nagy danken, der mich erstmals für das Testen interessiert hat. Mein besonderer Dank gilt Dehla Sokenou für ihre konstruktive Zusammenarbeit und Martin Gebser für seine umfangreichen Kommentare und Verbesserungsvorschläge. Schließlich möchte ich mich noch bei Alexander Pretschner und Mario Friske für ihre Anregungen und Diskussionen bedanken.

Selbstständigkeitserklärung

Hiermit erkläre ich, Stephan Weißleder, geboren am 18.04.1979 in Berlin, dass

- ich die vorliegende Dissertationsschrift "Test Models and Coverage Criteria for Automatic Model-Based Test Generation with UML State Machines" selbstständig und ohne unerlaubte Hilfe angefertigt habe,
- ich mich nicht bereits anderwärts um einen Doktorgrad beworben habe oder einen solchen besitze und
- mir die Promotionsordnung der Mathematisch-Naturwissenschaftlichen Fakultät II der Humboldt-Universität zu Berlin bekannt ist gemäß des Amtlichen Mitteilungsblattes Nr. 34/2006.

Berlin, den