Domain-Centered Product Line Testing

DISSERTATION

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

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

von
Dipl.-Inf. Hartmut Lackner

Präsident der Humboldt-Universität zu Berlin
Prof. Dr. Jan-Hendrik Olbertz

Dekan der Mathematisch-Naturwissenschaftlichen Fakultät
Prof. Dr. Elmar Kulke

Gutachter

1. Prof. Dr. Bernd-Holger Schlingloff
2. Prof. Dr. Ina Schäfer.
3. Prof. Dr. Alexander Knapp

Tag der Verteidigung: 13.07.2016

Abstract

Consumer expectations of (software-)products are growing continuously. They demand products that fit their exact needs, so they pay only for necessary functionalities. Producers react to those demands by offering more variants of a product. Product customization has reached a level where classically mass produced goods, like cars, can be configured to unique items. New paradigms facilitate the engineering of such variant-rich systems and reduce costs for development and production. While development and production became more efficient, quality assurance suffers from treating each variant as a distinct product. In particular, test effort is affected, since each variant must be tested sufficiently prior to production. For variant-rich systems this testing approach is not feasible anymore.

The methods for test design presented in this thesis overcome this issue by integrating variability into the test design process. The resulting test cases include requirements for variants, which must be fulfilled to execute the test successfully. Hence multiple variants may fulfill these requirements, each test case may be applicable to more than only one variant.

Having test cases with requirements enables sampling subsets of variants for the purpose of testing. Under the assumption that each test case must be executed once, variants can be sampled to meet predefined test goals, like testing a minimal or diverse subset of variants. In this thesis, five goals are defined and evaluated by assessing the tests for their fault detection potential. For this purpose, new criteria for assessing the fault detection capability of product line tests are established. These criteria enable quantitative as well as qualitative assessment of such test cases for the first time.

The results of the presented methods are compared with each other and furthermore with state of the art methods for product line testing. This comparison is carried out on four examples of different sizes, from small to industry-grade.

Zusammenfassung

Die Ansprüche von Kunden an neue (Software-)Produkte wachsen stetig. Produkte sollen genau auf die einzelnen Kundenwünsche zugeschnitten sein, sodass der Kunde genau die Funktionalitt erhält und bezahlt die er benötigt. Hersteller reagieren auf diese gestiegenen Ansprüche mit immer mehr Varianten in denen sie ihre Produkte ihren Kunden anbieten. Die Variantenvielfalt hat in solchem Maß zugenommen, dass selbst in Massen gefertigte Produkte heute als Unikate produziert werden können. Neue Methoden wie Produktlinienentwicklung unterstützen die Entwicklung solcher variantenreicher Systeme. Während der Aufwand für die Entwicklung neuer Varianten nun sinkt, profitiert die Qualitätssicherung nicht vom Effizienzgewinn der Entwicklung. Im Gegenteil: Insbesondere beim Test wird zunächst jede Variante wie ein einzelnes Produkt behandelt. Bei variantenreichen Systemen ist dies aufwandsbedingt jedoch nicht mehr möglich.

Die in dieser Arbeit vorgestellten Testentwurfsmethoden berücksichtigen die Variantenvielfalt in besonderem Maße. Bisher wurden, nach einer Stichprobenauswahl zur Reduktion des Testaufwands, die Testfälle auf Basis der konkreten Produkte entworfen. Statt nun auf Basis konkreter Produkte werden in dieser Arbeit zwei Ansätze vorgestellt, die die Phase des Testentwurfs auf die Produktlinienebene heben. Die bei Anwendung dieser Methoden entstehenden Testfälle enthalten, je nach Inhalt, Freiheitsgrade bzgl. ihrer Anforderungen an eine Variante, sodass ein Testfall auf ein oder mehrere Varianten angewendet werden.

Ausgehend von solchen Testfällen werden in dieser Arbeit neue Kriterien zur Stichprobenauswahl entwickelt. Mit diesen Kriterien kann der Umfang der Stichprobe, aber auch Eigenschaften der zu testenden Varianten bzgl. eines gegebenes Testziel optimiert werden. So ist es möglich, z.B. sehr wenige oder sehr unterschiedliche Varianten zum Test auszuwählen. Insgesamt werden in dieser Arbeit fünf Kriterien definiert und auf ihr Fehleraufdeckungspotenzial untersucht. Zu diesem Zweck werden neue Bewertungskriterien zur Fehleraufdeckungswahrscheinlichkeit von Produktlinientests etabliert. Somit ist erstmalig eine quantitative sowie qualitative Bewertung von Produktlinientests möglich.

Die Ergebnisse der vorgestellten Methoden und Auswahlkriterien werden sowohl untereinander evaluiert, als auch konventionellen Testmethoden für Produktliniensysteme gegenübergestellt. An vier Beispielen unterschiedlicher Grösse werden die in dieser Arbeit vorgestellten Methoden evaluiert.

Contents

1 Introduction **1**
1.1 Problem Statement . . . 3
1.2 Approach . . . 4
1.3 Assumptions . . . 5
1.4 Contributions . . . 5
1.5 Structure . . . 7

I Preliminaries **9**

2 Background **11**
2.1 Model-Based Product Line Engineering . . . 11
2.1.1 Feature-Oriented Design . . . 11
2.1.2 Variability Modeling . . . 14
2.1.3 A Basic Variability Language . . . 15
2.1.4 Summary . . . 16
2.2 Model-Based Testing . . . 16
2.2.1 Summary . . . 19
2.3 Test Assessment . . . 19
2.3.1 Mutation Analysis . . . 20
2.3.2 Error Design . . . 20
2.4 Examples . . . 21
2.4.1 Ticket Machine . . . 22
2.4.2 Alarm System . . . 24
2.4.3 eShop . . . 26
2.4.4 Body Comfort System . . . 26
2.4.5 Summary . . . 28

3 Assessment of Product Line Tests **33**
3.1 Potential Errors in Model-Based Product Line Engineering . . . 33
3.2 Product Line Test Assessment . . . 37
3.2.1 Mutation System for Product Lines . . . 37
3.2.2 Product Line Mutation Operators . . . 37

3.3 Evaluation . . . 41
3.3.1 Setup . . . 41
3.3.2 Results . . . 42
3.3.3 Threats to Validity . . . 45
3.4 Related Work . . . 45
3.5 Conclusions . . . 45

II Model-based Testing for Product Lines 47

4 Automated Test Design for Product Lines 49
4.1 Model-based Testing for Product Lines . . . 49
4.1.1 Application-Centered Test Design . . . 50
4.1.2 Domain-Centered Test Design . . . 51
4.2 Evaluation of both Approaches . . . 56
4.2.1 Tool Chain SPLTestbench . . . 56
4.2.2 Experiment Settings . . . 58
4.2.3 Results . . . 58
4.3 Related Work . . . 60
4.4 Conclusion, Discussion, and Future Work . . . 61

5 Test-driven Product Sampling 63
5.1 Reusable Test Cases . . . 64
5.2 Sampling Configurations from Reusable Test Cases . . . 64
5.2.1 General Sampling . . . 65
5.2.2 Targeted Sampling . . . 65
5.3 Example and Evaluation . . . 68
5.3.1 Tool Chain SPLTestbench . . . 68
5.3.2 Setup . . . 68
5.3.3 Results . . . 69
5.3.4 Discussion . . . 73
5.4 Related Work . . . 73
5.5 Conclusion . . . 74

6 Testing Product Boundaries 77
6.1 Testing Boundaries of Products . . . 77
6.1.1 Boundary Transitions . . . 78
6.1.2 Turning Open Boundaries into Test Goals . . . 79
6.2 Evaluation . . . 84
6.2.1 Setup . . . 84
6.2.2 Results . . . 84

6.2.3 Discussion 84
6.3 Related Work 85
6.4 Conclusion & Future Work 86

III Closure **87**

7 Conclusion **89**
7.1 Contributions 89
7.2 Impact 89
7.3 Future Work 91

Bibliography **93**

List of Figures **107**

List of Tables **109**

Chapter 1. Introduction

A trend of our time is the mass production of customized products to satisfy customers' individual needs. An illustrative example for this are the German car manufacturers who have recognized that customers enjoy to customize the purchased car to their needs. The manufacturers have therefore established platforms from which different variants of cars like sedans, station wagons, hatchbacks, or SUVs can be easily derived from. The idea of a platform is not limited to a car's body and extends to choices over any user-visible characteristic, like exterior and interior design, the engine, as well as countless comfort, infotainment, and assistance systems.

At the core of engineering such kind of platforms, lies planned reuse of (software) artifacts to efficiently derive new variants. A method established to particularly facilitate planned reuse is product line engineering (PLE) [PBL05]. In PLE, a platform is defined as a product line (PL), which specifies a core common to all variants (commonalities) and a managed set of features satisfying specific needs of a particular market (variabilities). PLE also prescribes the possible combinations of features to configure and subsequently derive new variants.

For the German car manufacturers, PLE has increased engineering capabilities dramatically (cf. Chap. 2 in [PBL05]). Modern mid-size car models can be configured to the degree that within one year any variant is not configured and bought twice, although hundred thousands of cars of this model are shipped. Furthermore, features are indeed shared across models of different classes. Hence the amount of offered models and variants was never greater than today.

The success of PLE is due to the separation of *domain* and *application* engineering as depicted in Figure 1.1. During domain engineering, the domain – by means of the common core, features, and their combinations – is analyzed, designed, and implemented. The analysis in this phase differs from traditional requirements analysis, since the result of this phase is not a static product but a platform to configure and build products from. PLE addresses this issue by considering the platforms variability through planned reuse in the early phases of the development life cycle. Design and implementation are also tailored to reflect the earlier defined variability.

Later in application engineering, requirements of a particular product are

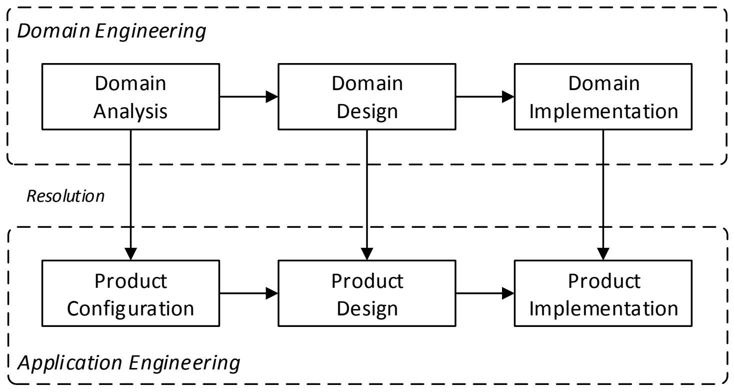


Figure 1.1.: Process of product line engineering (PLE)

specified by providing a configuration. Subsequently, the respective designs and implementations are derived according to the configuration from the domain-level artifacts created earlier. In PLE, the derivation of application artifacts is called *resolution*. This phase is sought to be highly automated, particularly in variant-rich systems development as we discuss them in this thesis.

With more capable engineering processes, also quality assurance has to adapt to achieve and maintain product and process quality in a PL environment. There are many well-known quality assurance techniques. In industry, testing is most wide-spread when it comes to measure product quality. For software systems, the costs for testing range between 30 and 50 percent of overall development costs and increase up to 80 percent when the system under test (SUT) has a safety-critical purpose [Jon91, Lin05]. To increase efficiency, testing becomes a more and more automated process. A first practical measure is the automation of test execution to avoid manual regression testing. Upon that, the next logical step is to automate test design.

Like many methods, also test design can be supported by models. Model-based testing (MBT) is considered one of the most mature automated test design techniques. Here, models are used to describe only the test-relevant aspects of the SUT, its environment, or both. There are several reports about the successful application of MBT in industrial practice [WS14, For12, LSW$^+$10].

Although MBT increases efficiency of testing, it does not scale for PLE under the premise that every variant must be tested individually as in static systems development. Hence, methods were proposed to systematically sample variants in order to select a subset for testing [OG05, ER11, OZML11, LSKL12]. Then for each variant test design is performed. This is an approach to which we refer as *application-centered* (AC) test design, since the test design is based on individual product models in the application engineering-level (cf. Fig 1.2).

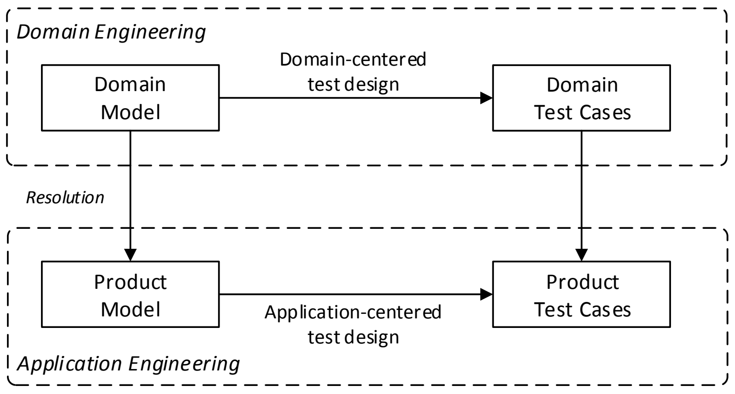


Figure 1.2.: Approaches to product line (PL) testing

In this thesis, we investigate methods for domain-centered (DC) test design which, in contrast, are based on models from the domain engineering-level. This approach should inherently reduce redundant test cases, while maintaining a high level of test quality. From the resulting domain test cases, products for testing are systematically sampled, resolved, and eventually tested. The methods presented in this thesis are supported by example software product lines (SPL) and compared to competing application-centered methods.

1.1. Problem Statement

Although test efficiency benefits from less redundancy, there is no test design method proposed on the domain-level that takes advantage of the information stored in domain artifacts. One reason for this is the inability of current testing techniques to deal with variability to create test cases for actual products. Models created during domain design are enhanced for variability: they may contain non-deterministic behavior due to multiple variants of the same component, duplicate declarations and initializations that are mapped to different variants, or other inconsistencies that are resolved at a later stage, e.g. resolution, to create valid product models.

For example, for most car models in Germany two engine options are offered: gasoline and diesel. On the domain engineering-level, models for both engines exist although only one of them can be deployed into a product model. In this case, conventional test methods will force the test designer to make a decision towards a particular variant. Of course, this decision must be made, if one of the engines is a test goal of the currently built test case. But if not, the decision on which engine to use for this test case can be postponed until test execution planning takes place.

The purpose of test execution planning is to assign test cases to products for later execution. The planning can be performed with several intents, like testing only a minimal amount of products for early smoke tests or testing a large number of products for building confidence, either for internal purposes or to convince authorities. Hence, the test effort for executing a particular test suite can be adapted to the current test intents.

Concluding, domain-centered test design touches upon several challenges:

- Levels of test design: On which levels of PLE can test design for variant-rich systems be performed? What are the main advantages of the individual methods? Is any method superior?
- Reusability of tests: Can reusability be expressed on the level of tests so that each test can be reused for testing more than one variant?
- Test planning for PL systems: For which subset of products should testing be performed? By which measure can products be systematically selected for testing if reusable test cases are provided?
- Fault detection capability of PL tests: Are there new kinds of faults to be made in PLE? Must tests be adapted to find such faults?
- Applicability of standard test generators: Can commercial off-the-shelf (COTS) test generators for non-variant systems be reused to design tests for variant-rich software systems within reasonable effort? Can they generate reusable tests?

1.2. Approach

In this thesis, we propose novel methods for designing, planning, and assessing tests in a PL context. A main goal is to define these methods on the domain engineering-level in PLE. The reason for this is as follows: During domain engineering, we find abstract representations of the application level to reduce the design's complexity. For example, instead of enumerating all valid configurations, feature models are used during domain engineering to represent the set of valid configurations. For the same reason, we want test artifacts to be domain artifacts until it is decided on which products should be tested. Hence, the key idea and hypothesis is:

> Leverage test design to the domain engineering-level.

Furthermore, we leverage established methods for assessing non-variant system tests to draw conclusions on how thoroughly the overall PL is tested. These methods are important to measure, compare, and discuss the quality of the presented results. We achieve this by leveraging established concepts and introducing traceability relations between test artifacts of the application and domain engineering-level.

1.3. Assumptions

One major improvement in the last fifteen years in software development is MDE (model-driven engineering). It closes the gap between requirements and implementation by refining the requirements until code can be derived manually or automatically. Further benefits are improved communication within and across developer teams and other stake holders, and many more [KWB03].

As many development methods, PLE can be supported by model-based abstractions [KCH$^+$]. Furthermore, automated test design can be supported by model-based approaches as well [UPL, BDG$^+$07]. We take up on this development and tailor the presented methods towards model-based PLE and test design. Hence, providing system and test models for the presented approaches is vital and considered a prerequisite.

1.4. Contributions

The contributions of this thesis are centered around test design, test planning, and test quality. Parts of the results of this thesis have been published in [LTWW14, LS14, LS15, Lac15, WWL15]. This thesis summarizes and extends these results. The work presented can be grouped into the following four categories.

Contribution 1: Domain-Centered Test Design

As pointed out, test design can be performed on different levels in PLE. This thesis investigates test design processes on varying levels. As a first contribution, we define a novel class of processes and propose two approaches for implementing such processes. The resulting tests carry on variability information and are thus reusable for testing one or more variants. We define test design processes and notions for reusable test cases, which provide the fundamentals for this thesis:

- *Test Design Processes.* We combine variability specifications by means of feature models and feature mapping models with behavioral specifications to enable automated model-based test design. The main contribution is the definition of a test generation approach on the domain-engineering level, i.e., it does not depend on resolving single product variants during the test design phase. We establish the two DC approaches *pre-configuration* and *step-by-step*. Both approaches are defined over feature models, mapping models, and software models. Furthermore, we present the main differences of established test design processes to the defined novel test design processes.

- *Reusable Tests.* As a result of DC test design methods, we receive reusable test cases. Each test case is reusable for a subset of variants in the PL. We provide notions for reusability of test cases in regard to both of the presented DC test design methods.

Contribution 2: Test Planning for Reusable Tests

Testing a PL is performed to fulfill prescribed test intents, e.g. system test, performance test or smoke tests, defined by the test plan as in any system testing process. However depending on the test goals defined by the chosen intent, choices for test design strategy, amount of tested variants and the variants' characteristics must be made. With the test design processes presented in this thesis, we enable standard test design strategies for generating reusable test cases.

Reusable test cases play a crucial role for planning tests in a PL context. The choice of variants for testing and planning their execution against sets of tests directly affects the test plan's adequacy to fulfill the test intents. Thus in this thesis, we propose and evaluate the combinations of the following aspects on selecting variants and assigning them to tests:

- *Probe Size.* The effort for testing variant-rich systems increases with the amount of variants selected for testing. Under the premise that every test case is executed once, test managers have a degree of freedom when it comes to choosing the variants for testing. A manager may choose a minimal set of variants to minimize test effort or on the other hand may choose a maximal set for increased confidence in the test's results.

- *Product Properties.* The selection of products can be driven by individual properties of the variants. We identify properties inherent to sets of variants, like the diversity of the variants' individual configurations or their size by means of the amount of selected and unselected features. In addition, we consider costs for building the variant and setting it up for testing.

- *Feature Combination.* The combinatorial selection of features was investigated intensively [LOGS12, PSK+10]. In this thesis, we analyze the effects of feature combinations on test quality in the context of test-based product selection.

Contribution 3: Product Line Test Quality

Quality of tests has many aspects. Since a test's purpose is to find faults, one major aspect is fault detection capability. The methods presented in this thesis enable test assessment of PL tests by raising mutation analysis to the domain engineering-level.

- *Mutation Process.* A mutation system for product lines was proposed in [LS14]. We present a mutation process for mutating mapping models and behavioral specifications. Test execution is performed on products and mutation scores are propagated to domain artifacts.

– *Faults in Variability Models.* Variability and domain engineering are split into different phases and models. Hence of new modeling languages used in PLE, *more* kinds of errors can be made on the model-level than in non-variable systems engineering. We observe new errors in mapping models. From these, we derive domain-based mutation operators.

Contribution 4: Tool Support and Experimental Results

All algorithms presented in this thesis are implemented in several software tools. In combination they make up the SPLTestbench which is integrated into the Eclipse Framework as plug-ins. In particular, the following tools were developed in the course of this thesis.

– *Valerie: Variability Injection.* is a set of model transformations that inject a variability specification into a behavioral specification, here software models. This transformation enables COTS software for DC test design. In particular, it provides support for the test generators *Conformiq Test Designer* and *RT Tester.*

– *Cosa: Configuration Sampler.* is a tool for sampling product configurations from incomplete configurations. The process of sampling is implemented as a constraint problem. For solving the problem it employs the Java Constraint Programming solver (JaCoP).

– *Muse: SPL Mutation System.* performs test assessment by means of fault detection capability for product line tests. It implements the presented mutation process and mutation operators.

– *Emol: Executable UML.* generates executable Java code from a given SPL model and a set of configurations.

Although all tools are implemented as proof of concept, experimental results demonstrate the general feasibility of the presented methods and the overall approach.

1.5. Structure

The remainder of this dissertation is organized as follows:

– In Chap. 2, we define our terminology and present the foundations of our work. We introduce model-based PLE and PL testing, as well as methods for test assessment, and examples used throughout this thesis.

– In Chap. 3, we survey potential faults specific to model-based PLE and means to evaluate the fault detection capability of PL test suites. The results from this chapter facilitate the evaluation in the remaining chapters.

– In Chap. 4, we present automated test design based on artifacts from the domain engineering-level. Furthermore, we compare the results to AC test design methods.

– In Chap. 5, we present our approach of sampling configurations for testing and planning test execution. We discuss possible sampling criteria, combinations thereof, and compare the results with AC test design methods.

– In Chap. 6, we show how the presented DC test design method can be improved for detecting PL-specific faults.

– We conclude and summarize this dissertation in Chap. 7.

Part I.

Preliminaries

Chapter 2. Background

In this chapter, we introduce and summarize basic concepts and recent findings in research used in the remainder of this thesis. In particular, we present the concepts of PLE (product line engineering) in Section 2.1 as well as MBT (model-based testing) and its application to PLE as discussed in literature so far in Section 2.2.

The evaluation of the methods for DC test design and product sampling presented in Part II of this thesis are facilitated by mutation analysis. Since the application of mutation analysis to PLE is a contribution of this thesis, we provide a general introduction to this point in Section 2.3 and present the contribution-specific parts in Chapter 3.

Furthermore, four examples are introduced to illustrate and support the findings of this thesis.

2.1. Model-Based Product Line Engineering

Individual customer expectations and the reuse of existing assets in a product's design are two driving factors for the emergence of PLE: increasing the number of product features while keeping system engineering costs at a reasonable level. In terms of software engineering, a SPL is a set of related software products that share a common core of software assets (commonalities), but can be distinguished (variabilities) [PBL05].

The definition and realization of commonalities and variabilities is the process of domain engineering. Actual products are built during application engineering. Here, products are built by reusing domain artifacts and exploiting the product line's variability.

2.1.1. Feature-Oriented Design

Like many methodologies, PLE can be supported by model-based abstractions such as feature models. Feature models offer a way to overcome the aforementioned challenges by facilitating the explicit design of global system variation points [KCH+].

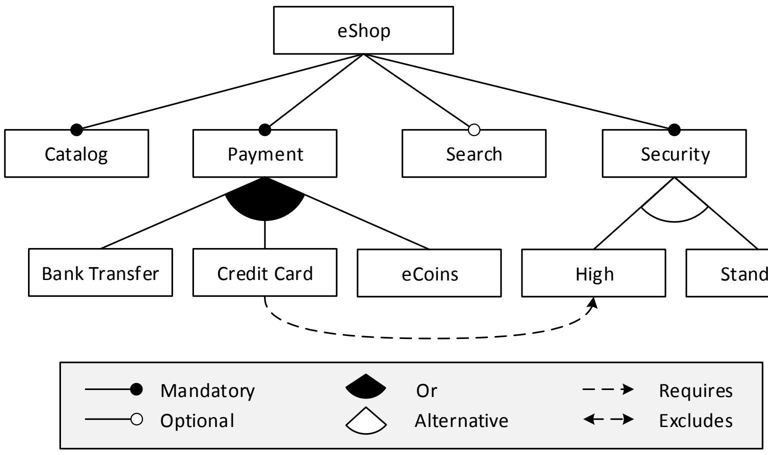


Figure 2.1.: A feature model for the eShop example.

A feature model specifies *valid* product configurations and has a tree structure in which a feature can be decomposed into sub-features. Figure 2.1 shows an example feature model that is reused later in this thesis. A parent feature can have the following relations to its sub-features: (a) *Mandatory*: child feature is required, (b) *Optional*: child feature is optional, (c) *Or*: at least one of the children features must be selected, and (d) *Alternative*: exactly one of the children features must be selected. Furthermore, one may specify additional (cross-tree) constraints between two features A and B: (i) A *requires* B: the selection of A implies the selection of B, and (ii) A *excludes* B: both features A and B must not be selected for the same product.

A feature model (FM) can also be represented as a propositional formula [Bat05]:

$$FM : (F \to \mathbb{B}) \to \mathbb{B}$$

where $F = f_1, ..., f_k$ is the set of features belonging to the PL. Since the empty configuration is considered correct under the above premises, we include the root feature as an additional conjoined literal to the formula.

For instance, the boolean formula for the eShop example in Figure 2.1 is:

$$\begin{aligned}
FM = {} & eShop \wedge (\neg eShop \vee Catalog) \\
& \wedge (\neg eShop \vee Payment) \wedge (\neg eShop \vee Security) \\
& \wedge (\neg Payment \vee CreditCard \\
& \quad \vee BankTransfer \vee eCoins) \\
& \wedge (\neg Security \vee (High \wedge \neg Standard) \\
& \quad \vee (Standard \wedge \neg High)) \\
& \wedge (\neg Catalog \vee eShop) \\
& \wedge (\neg Payment \vee eShop) \\
& \wedge (\neg CreditCard \vee Payment) \\
& \wedge (\neg BankTransfer \vee Payment) \\
& \wedge (\neg eCoins \vee Payment) \\
& \wedge (\neg Security \vee eShop) \\
& \wedge (\neg High \vee Security) \wedge (\neg Standard \vee Security) \\
& \wedge (\neg Search \vee eShop) \wedge (\neg CreditCard \vee High))
\end{aligned}$$

The assignment of a value to a literal indicates whether the corresponding feature is selected (*true*) or deselected (*false*). Any variable assignment that satisfies the formula is a valid product configuration *pc* for the PL:

$$pc = F \to \mathbb{B}$$

A product configuration is valid, iff $FM(pc) = true$ holds.

For instance, the following formula is a valid *pc* for the eShop feature model presented in Figure 2.1.

$$\begin{aligned}
P = {} & eShop, Catalog, Payment, BankTransfer, \\
& \neg CreditCard, \neg eCoins, \neg Search, Security, \\
& High, \neg Standard
\end{aligned}$$

We define the set of all valid product configurations specified by a feature model FM to be:

$$PC = \{pc : F \to \mathbb{B} | FM(pc) = true\}$$

To facilitate the design of such models, we provide a feature model language defined in Ecore with the Eclipse Modeling Framework (EMF) as depicted in Figure 2.2. The feature model language is similar to other standard feature languages [BAC04].

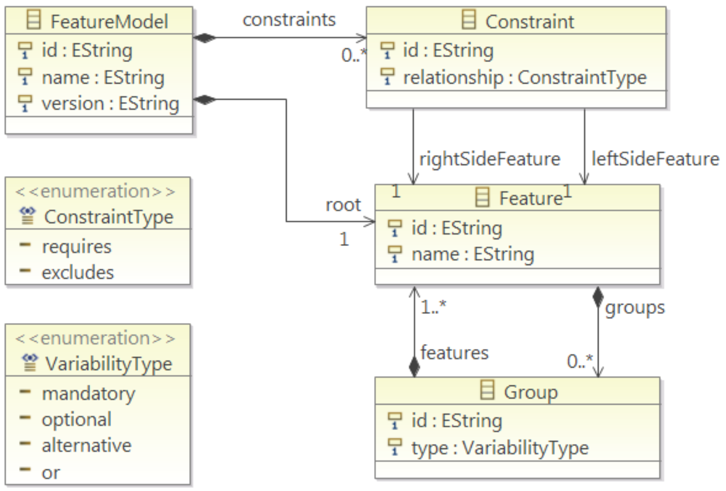


Figure 2.2.: Feature model language diagram.

It offers basic functionalities, for designing mandatory, optional, or, alternative, and binary cross-tree-constraints (CTC) between features. Extended functionalities, e.g. cardinalities or attributes, are not supported.

2.1.2. Variability Modeling

Although a feature model captures the system's variation points in a concise form its elements are only symbols [CA05]. Their semantics has to be provided by mapping them to a model with semantics: a base model. Such a mapping can be defined using an explicit mapping model or extending the base model's language [GV07].

An explicit mapping model consists of relations from feature model elements to base model elements. We refer to a domain model as the triple of feature model, mapping model, and base model. From such a domain model, product models or code can be resolved for a given *pc*. In the following, we introduce the three major paradigms for variability modeling as depicted in [GV07]:

Annotative Modeling In this case, the base model is designed in terms of a so called 150% model. A 150% model contains every element that is used in at least one product configuration and, thus, subsumes every possible product [GKPR08] (Fig. 2.3a). Subsequently, model elements are removed to resolve a valid variant.

Compositional Modeling In contrast to annotative languages, compositional languages start from a minimal core that contains features that are common to all possible products. From this starting point additional features will be added by a designer (Fig. 2.3b).

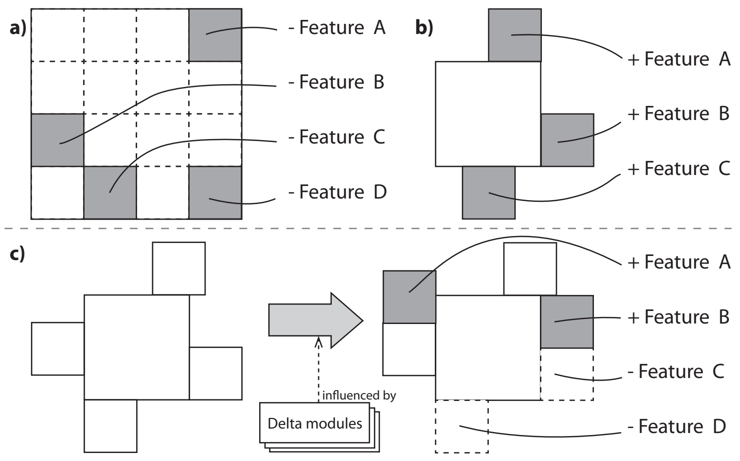


Figure 2.3.: Annotative (a), compositional (b) variability (based on [GV07]) and delta modeling (c)

Transformational Modeling Lastly, there are methods that combine compositional and annotative methods, where model elements can be removed and added to resolve a variant. A well-known approach for this is delta modeling (also delta-oriented programming) [Sch10]. Delta modeling consists of two parts: The first one is a core product comprised of a set of feature selections that represent a valid product. The second part is a set of delta modules which specify changes to the core module. These changes can either be the construction (add) or destruction (remove) of elements from the product model (Fig. 2.3c). Each delta module is associated to one or more features. Whenever a feature is selected or deselected the associated deltas are applied to the product model, resulting in a new product model.

2.1.3. A Basic Variability Language

For the examples applied in this thesis, we use a custom annotative modeling language. This language is a subset of the Common Variability Language (CVL). The standardization process of the CVL specification is still an ongoing process and the language specification might be subject to change [HMPO$^+$08]. For consistency over all experiments carried out during the period of this thesis, we apply only the here described language features. In particular, our language does only feature annotative modeling capabilities, while CVL also includes features transformational modeling.

A mapping model in our mapping language, consists of mappings, where each mapping maps a single feature to a set of elements in the base model. For designing base models, we employ the Unified Modeling Language (UML), which is commonly

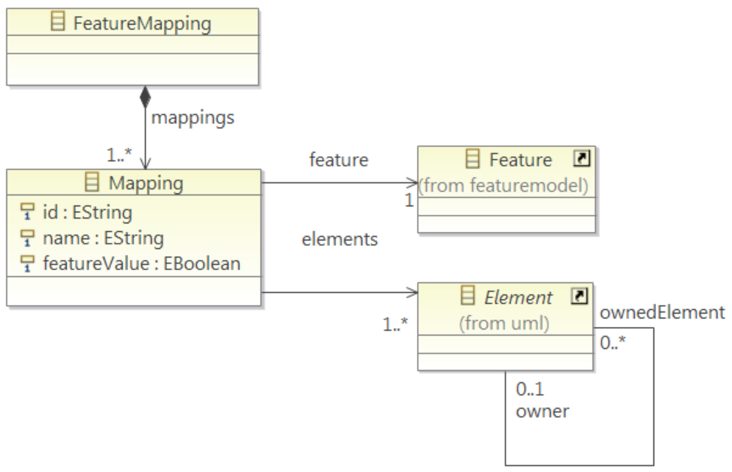


Figure 2.4.: Implementation of the mapping model language in Ecore.

applied in software engineering. Multiple features mapping to the same base model element are interpreted as a conjunction of features. Additionally, each mapping has a Boolean flag that indicates whether the mapped model elements are part of the product when the feature is selected (*true*) or unselected (*false*).

In Figure 2.4, we depict the mapping model language as implemented for this thesis with Ecore. The language refers to concepts of *elements* from the UML and to *features* from our feature model language. Figure 2.5 shows an excerpt of the eShop specification, where parts of the feature model are depicted in the upper half and parts of the state machine's payment process are shown in the lower half. In between, we find a mapping, denoted by a dotted edge, from feature *Credit Card* to the transition labeled as "`SelectCreditCard[]/`".

2.1.4. Summary

Separation of concerns is a key paradigm in PLE. There is a variety of languages and methods to support PLE in model-based environments. Here, we present a small portion of relevant languages. Although many standards are proposed, no common standard was established to this point, partly due to the diversity of available languages provided by academia and industry.

2.2. Model-Based Testing

Testing is a common approach to quality assurance. The idea is to systematically compare the observed system behavior with the expected one. The quality of tests

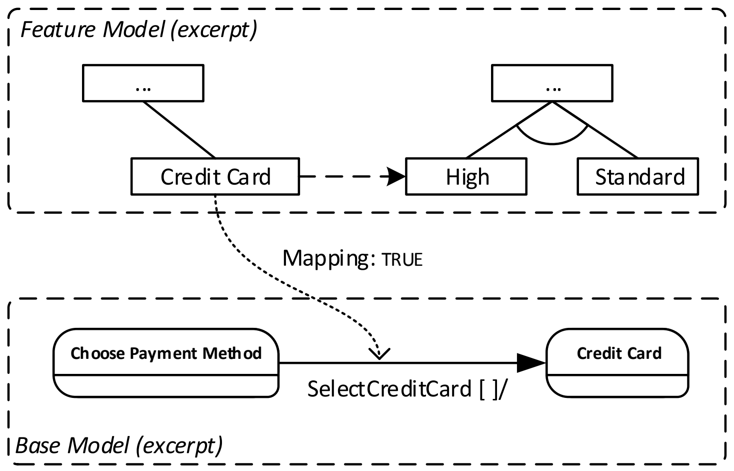


Figure 2.5.: SPL design with annotative variability.

is often measured in terms of requirements and code coverage. Satisfying a coverage criterion can result in big test suites with corresponding efforts in test design and test execution. There are various approaches and tools to automate the test execution in order to reduce testing costs. The biggest issue of this approach is change. A change of requirements or customer demands may result in far more effort for test design adaptation than for manual test execution. In the worst case, these test design adaptation costs outweigh the costs saved by automated test execution. In order to solve this issue, test design also needs to be automated.

There are a few techniques called MBT: some refer to test case models, like use cases, which are used as templates to facilitate test case implementation [HGB08]. Others generate test data from test data models, like CTE [GG93]. In this thesis, we interpret MBT as the automation of test design, where a test suite is generated from a dedicated test model. The test model contains test-relevant information about the intended behavior of the SUT and/or the behavior of its environment.

The resulting test suite is a set of test cases and possibly more test suites. Each test case consists of a sequence of stimuli and expected reactions to these stimuli. Especially test generators we use throughout this thesis for our state machine examples, we receive such sequences as outputs. For non-deterministic systems a test case may also be a tree or graph of stimuli and reactions. A test case may also pose prerequisites on the system's state or configuration that must be fulfilled, before the test case can be executed.

We depict a generic MBT process in Figure 2.6 as presented by Utting et al. in [UPL12]: First (1), the test model is created. Typically, it is created from textual requirements or formal specifications. Its level of abstraction depends on

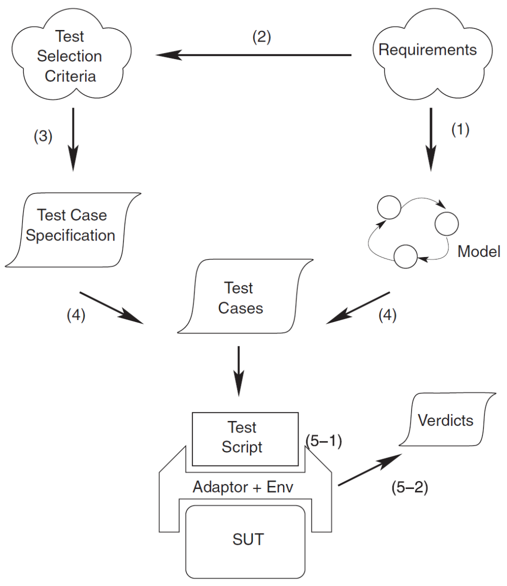


Figure 2.6.: MBT process by Utting et al. [UPL12]

the intended test level and test target. MBT is suitable for various kinds of test targets like full or partial functionality, quality of service, performance, and others.

Test models can be designed in many notations like abstract state machines, timed automata, and UML state machines [LS12], or Event-B [WBC14], Calculus of Communicating Systems (CCS) [BK05], communicating sequential processes (CSP) [MLL09], Markov Chains [Pro05], Petri Nets [Xu11], Z [LS05], and many others. Ultimately, the chosen notation must be suited to express the relevant test aspects and must be supported by the employed test generator.

Second (2), the test generator must be configured to select tests that achieve the test targets adequately. Hence, test selection criteria were defined to approximate the notion of a "good" test case. Typical selection criteria are coverage metrics based on model element and/or traced requirements. For transition-based notations, common coverage criteria are defined over the test model's control flow, data flow, conditional branching, use cases, requirements, or stochastic characterizations [CM94].

In a third step (3), test selection criteria are transformed into test case specifications. This is a necessary step, since test selection criteria must be formalized for the particular test generator to make them operational.

Test case generation can start whenever the model and the test case specification are defined (4). Before a test can be run by the test driver against the SUT, test scripts must be generated (5-1). Due to the diversity of test drivers, test generators are built to generate test cases that are agnostic of the intended test driver. Hence, the resulting test cases must be transformed into platform-specific test scripts. The remaining abstraction-gap between test model and SUT can be handled either by another transformation process or by an adaptor. Finally in (5-2), the test verdict is received and evaluated. In this step, an adaptor or transformation may again abstract the test verdict to the test model's level. From this on, further actions are planned as in any other test process.

2.2.1. Summary

In this thesis, we apply UML state machines as models for automated test design. UML state machines are a common modeling language to express behavioral test models and they are supported by commercial and academic tools: IBM ATG [IBM09], Conformiq TestDesigner [Con], JUMBL [Pro03], Sepp.med MBTSuite [SKK13], Verified RT-Tester [Pel], ParTeG [Wei09], CertifyIt Smartesting [BLPT05], and others. Typical test targets for state machines are criteria for control flow, data flow, conditional branching, use cases, and requirements [CM94]. Test designers create a test suite that fulfills a prescribed set of criteria to a certain degree. The resulting tests can then be adapted to the interface of the SUT and executed.

2.3. Test Assessment

In this thesis, we do not only propose new methods for testing product lines, we also evaluate them. This is achieved by assessing the generated test suites. Of course fault detection capability (FDC) is of major concern, when it comes to test assessment. The cause of a fault, as defined by the International Software Testing Qualifications Board (ISTQB), is an error made by humans during programming, which is due to a mistake, a misconception or misunderstanding. Only when the erroneous lines are executed, this error becomes a fault or defect in the system. A fault may cause the system to behave incorrectly, but does not necessarily have to. A failure is, due to a fault, the inability of the system to perform a required function.

In Section 2.2, we introduced measures for test selection by means of coverage criteria. The same measures are used to assess tests, i.e., test suites scoring higher coverage are more likely to detect faults. Also a test suite's code coverage on the SUT can be measured and used as a quality criterion [MSB12], although code coverage is only a necessary but not sufficient condition.

If available, a test suite can be assessed by calculating the ratio of detected faults to known faults. In late stages of development, also the ratio of faults reported during development and faults reported from field usage can be measured.

Beyond that, test assessment can be performed along various measures: execution time, costs, readability, debugability, maintainability, repeatability, and others.

Typically execution time, costs, and repeatability are easier to assess, while the other criteria are subjective to the individual testers.

2.3.1. Mutation Analysis

Mutation analysis (also mutation testing) as introduced by DeMillo et al. [DeM80] is a fault-based testing technique with the intended purpose to assess the quality of tests that will be applied to a system.

The process of mutation analysis introduces errors into software by creating multiple versions of the original software, where each created version contains one error. After that, existing test cases are used to execute the erroneous versions (*mutants*) with the goal to distinguish the faulty ones (*to kill a mutant*) from the original software. The ratio of killed mutants to generated mutants is called *mutation score*, that is computed after the execution of all test cases. The main goal of the test designer is to achieve a mutation score of 100 percent, that indicates that all mutants (i.e., all faults) have been detected [OU01, JH11].

Regarding to [JH09] we can distinguish multiple kinds of mutants that can be created. The simplest one are *first-order mutants* that, regarding to the mutation operators, only have one introduced error. Even if first-order mutants can be killed during the process of mutation testing, this does not guarantee that a combination of two (or even more) mutants will also be detected by the test suite. Such combined mutants are referred as *higher-order mutants*.

Though mutation operators are applied to fit in errors, there is the chance, that the resulting mutant offers the same behavior as the original. This type of mutants are referred as *hidden mutants*. Hidden mutants are not easily detectable (undecidable), but supposed to be killed if detected during the mutation analysis process [JH11].

2.3.2. Error Design

In mutation analysis, defective software versions are derived from a set of potential errors a human can make during software development. Potential errors are implemented as mutation operators, which are applied to the original software for introducing errors. The mutation operator's design affects the validity of the resulting mutation scores and the costs for testing by means of the amount of mutants to create and the number of tests to execute against them. Thus, we apply the following four guiding principles for creating mutation operators [BOY00, Woo93]:

1. Mutation categories should model potential error. It is important to recognize different types of error. In fact, each mutation operator is designed to model errors belonging to the corresponding error class.

2. Only simple, first-order mutants should be generated. These mutants are produced by making exactly one syntactic change to the original specification. This restriction is justified by the *coupling effect* hypothesis which says that the test sets that detect simple mutants will also detect more complex mutants [Off92].

3. Only syntactically and semantically correct mutants should be generated. Some mutations may result in an illegal expression, such as division by 0. Such mutants should not be generated.

4. Do not produce too many mutants. This includes some practical restrictions. For example, do not replace a relational connector with its opposite, if for other mutants a term negation operator is applied, since both mutants are semantically equivalent.

From other mutation systems presented in [BHP11, ALN13, FDMM94], we identify the following general categories for model-based mutation operators:

1. Model element deletion: a model designer forgets to add a model element, e.g. a feature, a mapping, or a transition.

2. Model element insertion: a model designer inserts a superfluous model element, e.g. a feature, a mapping, or a transition.

3. Property change: a model designer chooses a wrong value for a property of a model element, e.g. mandatory feature instead of optional, inverse value for a feature's status, or wrong transition target.

For each model element-type, like mappings, transitions, guards, etc., one can check for applicable categories and implement mutation operators accordingly.

2.4. Examples

To evaluate the methods presented in this thesis, we employ four model examples: An embedded ticket machine, an alarm system, an e-commerce webshop, and a car's body comfort system. All of the model examples conform to the previously presented requirements for applying SPLTestbench.

As presented in Section 2.1 we use a model-based PLE approach to specify the models. The domain analysis is performed with feature models, while for defining variability, we use annotative concepts as defined earlier by the presented mapping model language in Section 2.1.3. The base models are specified with UML, in particular state machines and classes. Each example consists of at least one state machine diagram. Each state machine has a context class, which holds attributes used in the state machine and ports for sending and receiving messages.

We apply syntax and semantics as defined in the UML standard. For sending messages from the SUT to the environment we use the following notation:

```
(1) <SignalType> <signalIdentifier>;
(2) <portname>.send(<signalIdentifier>);
```

First (1), we declare an identifier of the signal's type. Then (2), we call the send method of the outbound port from which we intend to send the signal. Furthermore, the signal's identifier is passed as a parameter to the port's send method.

In the following we present an example for sending a signal of type "`Text`" over a port named "`out`". We choose to declare the signal's identifier as "`letter`". If the signal has a property, e.g. "`content`" of type String, we may also set the value of this property before sending the signal. The corresponding effect is:

```
(1) Text letter;
( ) letter.content = "Hello World";
(2) out.send(letter);
```

2.4.1. Ticket Machine

The Ticket Machine is a simple case study and is adopted from Cichos et al. [CLOS12]. The functionality is as follows: a customer may select tickets, pay for them, receive the tickets, and collect change. The feature model has a root feature with three sub-features attached to it; all of them are optional. Depending on the selected features, the machine offers reduced tickets, accepts not only coins but also bills, and/or will dispense change.

The feature model is depicted in Figure 2.7, while the base model is shown in Figure 2.8. The features are mapped as follows:

- *Bills* to transitions labeled "`bill[paid=costs]/ paid+=5;`" at state "`Payment`" and "`bill[paid>=5]/ paid-=5;`", "`bill[paid>0 && paid<5]/ paid--;`" at state "`MoneyChange`".
- Not *Bills* to transition labeled "`coin[paid>0]paid--;`" at state "`MoneyChange`".
- *Change* to transition labeled "`change[tDay==0 && tShort==0 && tRed==0]`" from state "`TicketEjection`" to "`MoneyChange`".
- Not *Change* to transition labeled "`change[tDay==0 && tShort==0 && tRed==0]/ paid=0; costs=0; success o; Out.send(o);`" from state "`TicketEjection`" to "`MoneyChange`".

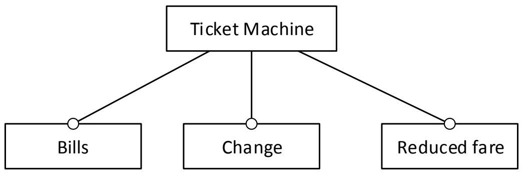


Figure 2.7.: Feature model of the Ticket Machine example.

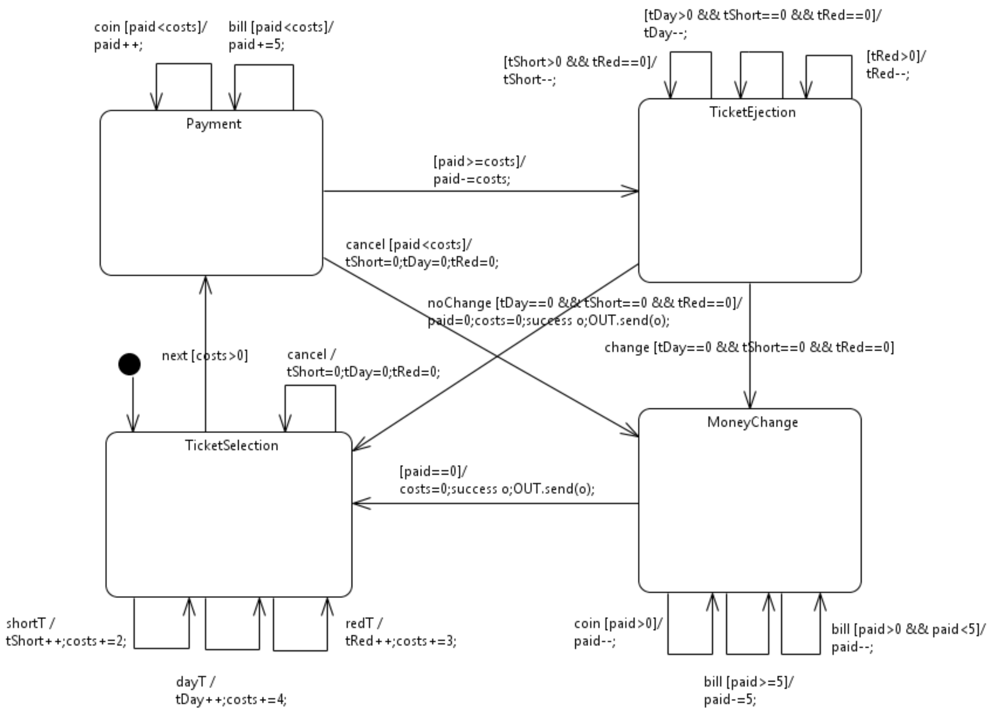


Figure 2.8.: Base model of the Ticket Machine example.

– *Reduced Fare* to transition labeled "`redT/ tRed++; costs+=3;`" at state "`TicketSelection`".

2.4.2. Alarm System

The Alarm System example is also adopted from Cichos et al. [CH11] and more complex. The alarm may be set off manually or automatically by a vibration detector. Both features are part of an or-group and, thus, at least one of the two features must be present in every product. In the event of an alarm, a siren or a warning light will indicate the security breach. When the vibration does not stop after a predefined period of time, the system optionally escalates the alarm by calling police authorities and/or sending photos of evidence. Additionally to its alarming functionality, the Alarm System SPL provides a feature for taking a photo of any operator that configures the system for security measures.

We adopted the Alarm System models by removing manual timers that were implemented as guard conditions. Furthermore, we added cross-tree-constraints (CTC) and more features to the feature model for exercising the SPLTestbench's functionalities more thoroughly.

The feature model is depicted in Figure 2.9, while the base model is shown in Figure 2.10. The features are mapped as follows:

– *User Interface*, Storage and *Online Access* are mandatory and need no mapping.

– *Camera* to state "`Camera`".

– *Photo* to transition labeled "`photo/ makePhoto o; Out.send(o); storePhoto p; Out.send(p); data = true;`" at state "`Camera`".

– *Video* to transition labeled "`video/ makeVideo o; Out.send(o); storeVideo p; Out.send(p); data = true;`" at state "`Camera`".

– *Manual Release* to transition labeled "`manStart/ m=true;`" from state "`StandBy`" to "`EnsureDanger`".

– *Vibration Detector* to transition labeled "`vibrating`" from state "`StandBy`" to "`EnsureDanger`".

– *Warning Light* to transition "`startWarning / flashFast o; Out.send(o);`" from state "`Ensure Danger`" to "`AlarmSignal`".

– *Siren* to transition "`startSiren / loudSound o; Out.send(o);`" from state "`Ensure Danger`" to "`AlarmSignal`".

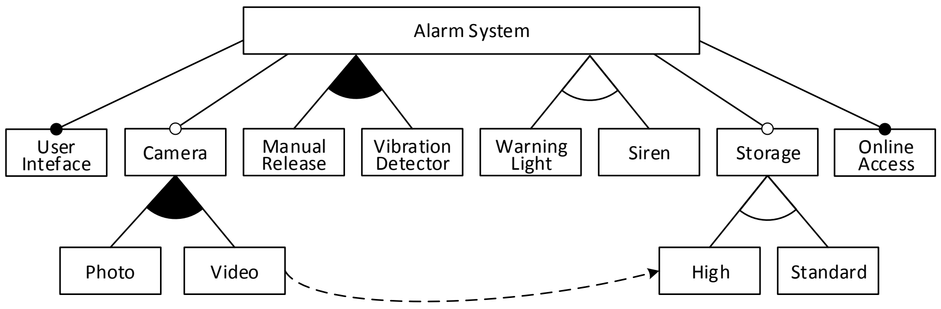


Figure 2.9.: Feature model of the Alarm System example.

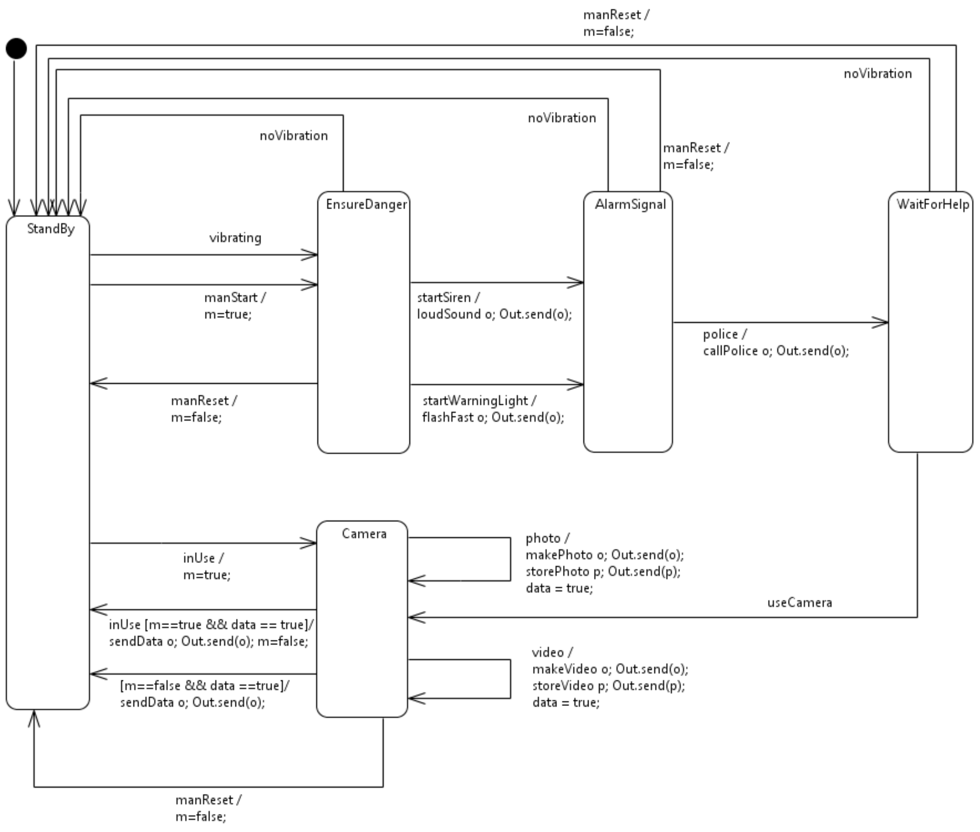

manReset / m=false;
noVibration
noVibration
noVibration
manReset / m=false;
StandBy
EnsureDanger
AlarmSignal
WaitForHelp
vibrating
manStart / m=true;
startSiren / loudSound o; Out.send(o);
police / callPolice o; Out.send(o);
manReset / m=false;
startWarningLight / flashFast o; Out.send(o);
Camera
photo / makePhoto o; Out.send(o); storePhoto p; Out.send(p); data = true;
inUse / m=true;
useCamera
inUse [m==true && data == true]/ sendData o; Out.send(o); m=false;
video / makeVideo o; Out.send(o); storeVideo p; Out.send(p); data = true;
[m==false && data ==true]/ sendData o; Out.send(o);
manReset / m=false;

Figure 2.10.: Base model of the Alarm System example.

– *High* and *Standard* are not mapped to model elements, since they have no influence on any resulting product model.

2.4.3. eShop

The eShop example is a fictional e-commerce Webshop designed for this thesis. A customer can browse the catalog of items, or if provided, use the search function. Once the customer put items into the cart, he can checkout and may choose from up to three different payment options, depending on the eShop's configuration. The transactions are secured by either a standard or high security server. A CTC ensures that credit card payment is only offered if the eShop also implements a high security server. While we show the base model in Figure 2.11, its feature model was already presented in Figure 2.1. The features are mapped to the state machine's elements as follows:

– *Catalog*, *Payment* and *Security* are mandatory and need no mapping.

– *Bank Transfer* to all transitions connected to state "`Bank Transfer`".

– *eCoins* to all transitions connected to state "`eCoins`".

– *Credit Card* to all transitions connected to state "`Credit Card`".

– *High* to attribute secureConnection with its default value set to true.

– *Standard* to attribute secureConnection with its default value set to false.

– *Search* to any transitions connected to state "`Search`".

2.4.4. Body Comfort System

This example is an actual case study created by Lity et al. and documented as technical report in [LLLS]. The case study stems from a cooperation between the Institute for Programming and Reactive Systems, Institute for Software Engineering and Automotive Informatics, Real-Time Systems Lab, and their partners from the German automotive industry. The system comprises the following (variable) functionality:

– *Power Windows* with *Finger Protection*

– *Exterior Mirrors* with *Heating*

– Controls (*Human Machines Interface*) with *Status LEDs*

– *Alarm System* with *Interior Monitoring*

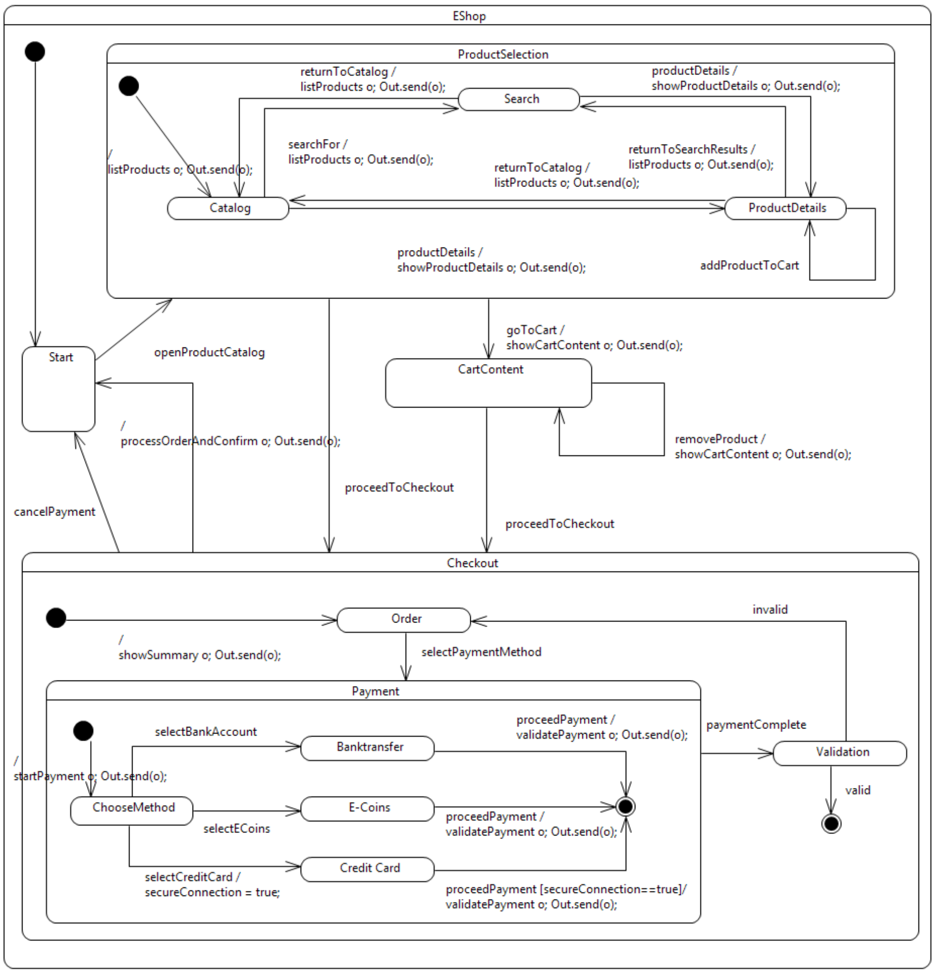

EShop
ProductSelection
returnToCatalog /
listProducts o; Out.send(o);
productDetails /
showProductDetails o; Out.send(o);
Search
searchFor /
listProducts o; Out.send(o);
returnToSearchResults /
listProducts o; Out.send(o);
returnToCatalog /
listProducts o; Out.send(o);
/
listProducts o; Out.send(o);
Catalog
ProductDetails
productDetails /
showProductDetails o; Out.send(o);
addProductToCart
Start
openProductCatalog
goToCart /
showCartContent o; Out.send(o);
CartContent
removeProduct /
showCartContent o; Out.send(o);
/
processOrderAndConfirm o; Out.send(o);
proceedToCheckout
proceedToCheckout
cancelPayment
Checkout
Order
invalid
/
showSummary o; Out.send(o);
selectPaymentMethod
Payment
selectBankAccount
Banktransfer
proceedPayment /
validatePayment o; Out.send(o);
paymentComplete
Validation
/
startPayment o; Out.send(o);
ChooseMethod
E-Coins
proceedPayment /
validatePayment o; Out.send(o);
selectECoins
valid
Credit Card
selectCreditCard /
secureConnection = true;
proceedPayment [secureConnection==true]/
validatePayment o; Out.send(o);

Figure 2.11.: Base model of the eShop example.

  - *Central Locking System* with *Automatic Locking*
  - *Remote Control Key* with *Safety Function* and controls for *Alarm System* and *Power Windows*.

The case study's feature model is depicted in Figure 2.12 and an overview of the system's compositions is shown in Figure 2.13. In the technical report, the base model consists of 21 state machine models and the variability is given by 15 delta-models.

For the purpose of the evaluations performed within this thesis, the models were transformed to mapping models and state machines with annotated variability. For annotation, we employ the mapping language as introduced in Section 2.1.3. As a result of this transformation we gain 21 state machines and 20 mappings to 357 state machine elements (each mapping can refer to multiple base model elements). Neither the state machine models nor the mapping model are presented individually in this thesis.

2.4.5. Summary

The presented examples vary in complexity of variability and behavior. Table 2.1 and 2.2 summarize the individual models on structural level. Besides of the Body Comfort System Case Study, the Alarm System example is the most variable SPL in this comparison by means of possible configurations (CNF), offering 42, followed by the eShop with 20, and the Ticket Machine with only 8 configurations. Although the Alarm System has only two features more than the eShop, it offers twice as many configurations. This is a typical effect observable in variable systems, where adding only a few features can drastically increase the amount of configurations. There are further metrics for feature models available to measure analyzability, changeability, and understandability, which we did not apply so far [BG11].

Similar to feature models, UML models are of different complexity by means of states, transitions, sub-machines, and signals. Here, the eShop case study is the most complex example. The Alarm System and the Ticket Machine are gradually less complex.

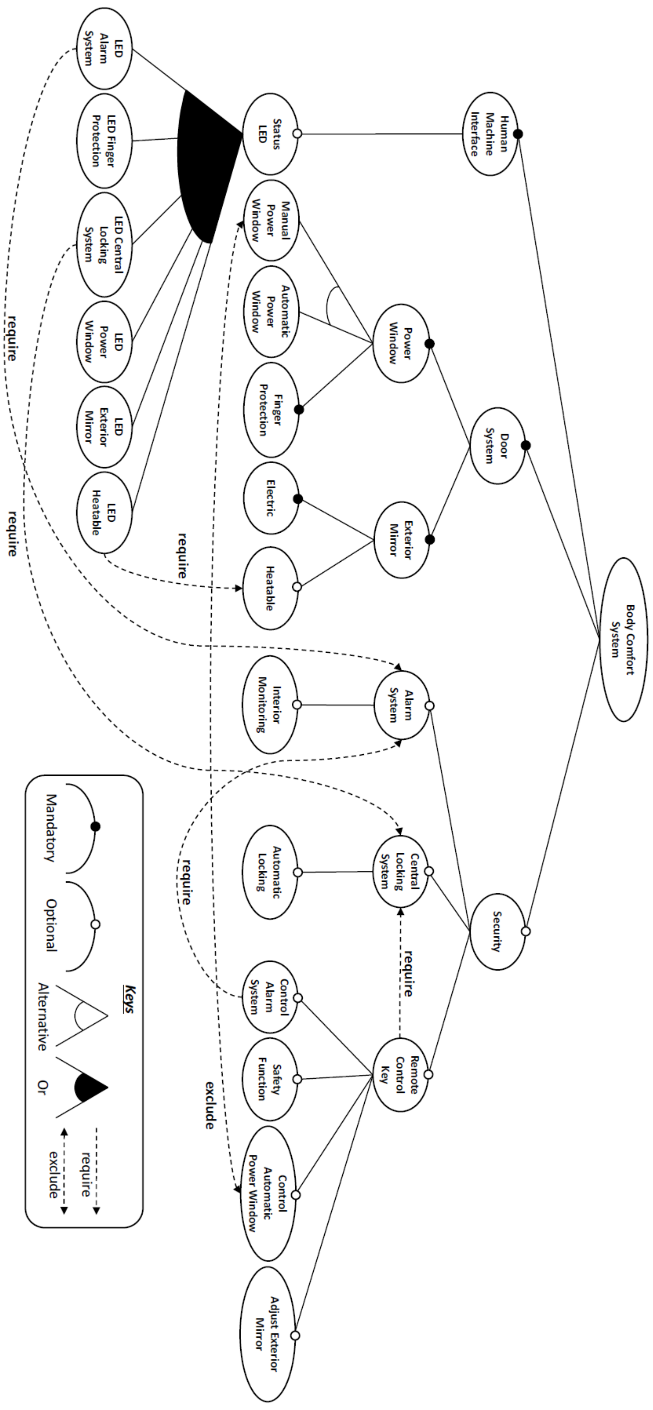


Figure 2.12.: Feature model of the Body Comfort System. [LLLS]

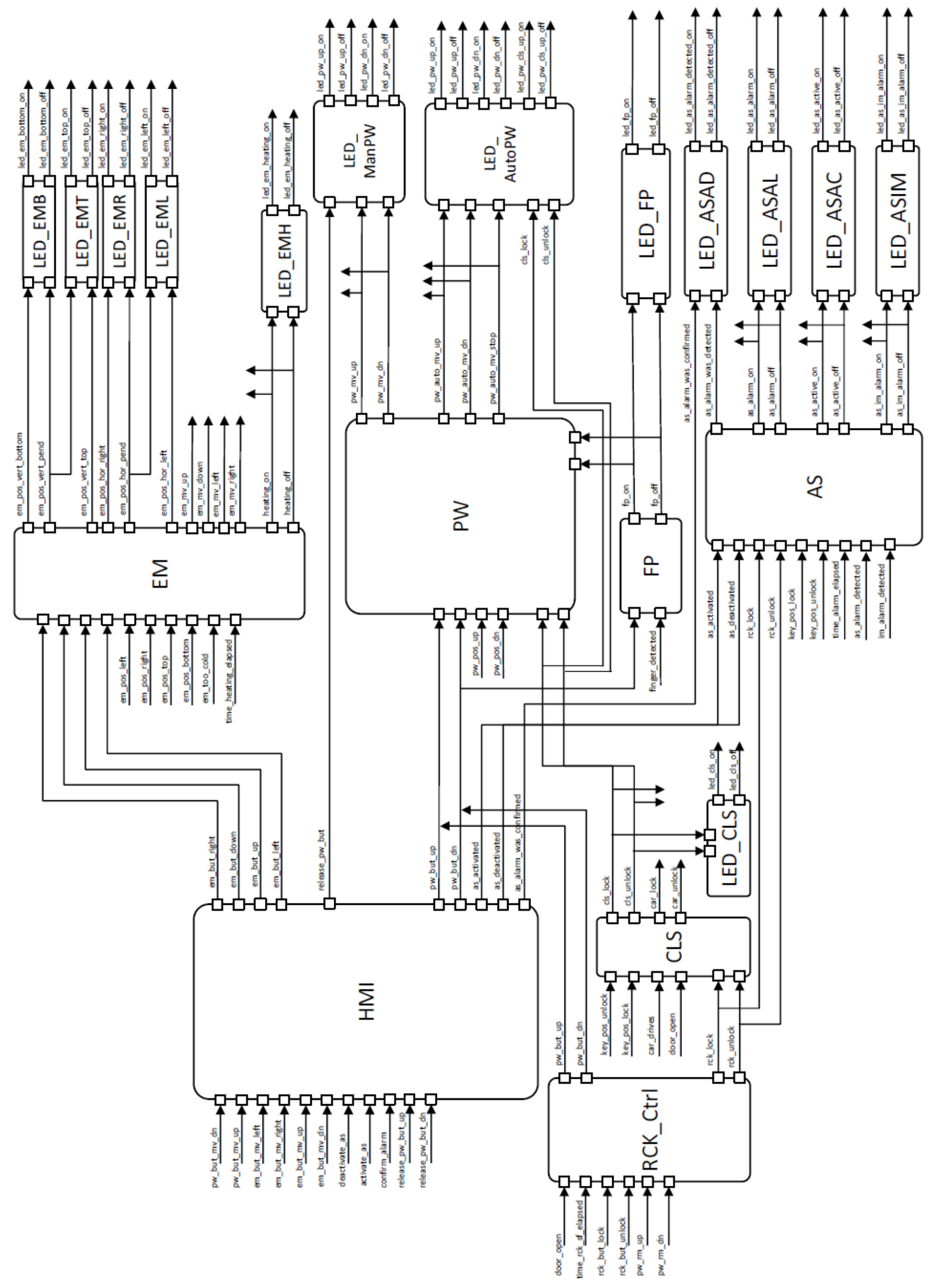


Figure 2.13.: Overview of the Body Comfort System's architecture. [LLLS]

Table 2.1.: Feature model summary for Ticket Machine (TM), Alarm System (AS), eShop (ES), and Body Comfort System (BCS).

Example	**TM**	**AS**	**ES**	**BCS**
Features	4	12	10	27
Core features	1	3	4	7
Grouped features	0	8	5	8
Cross-tree constraints	0	2	1	6
Configurations	8	42	20	11,616

Table 2.2.: Base model summary for Ticket Machine (TM), Alarm System (AS), eShop (ES), and Body Comfort System (BCS).

Example	**TM**	**AS**	**ES**	**BCS**
States	4	5	16	235
Transitions	19	19	28	369
Levels of Hierarchy	0	0	2	0
Sub-Machines	0	0	3	0
Signals	10	19	26	102

Chapter 3.

Assessment of Product Line Tests

Test assessment is an integral part for evaluation of the concepts presented in Part II of this thesis. This chapter builds the foundations for assessing the quality of PL test suites by means of fault detection capability (FDC). Though there are many methods proposed for testing a PL, until now, quality assessment of tests was limited to measuring code, model and/or requirements coverage [Mv03, CDS06]. Mutation analysis is a major approach to investigate a test suite's FDC. So far mutation analysis for PLE is constrained to mutating individual products of a PL. This approach has two major drawbacks: first, developers can introduce errors on all kinds of artifacts, not only on product models, leading to new kinds of faults. For better understanding, we analyze different design paradigms for model-based PLE as presented in Section 2.1 and errors that can occur during the respective design processes. From the results, we develop mutation operators for variability models and base models to mimic possible faults in these models.

Secondly, the selection of products and subsequently its mutations is biased by the products selected for testing. Therefore, mutation analysis assesses the quality of the tests for particular products, but not for the whole PL. In contrast, we define a mutation system and operators on the domain engineering-level. This enables us to assess the test quality independently from the tested products. Subsequently, the test suite's quality by means of FDC can be assessed for the complete PL.

The remainder of this chapter is structured as follows: In Section 3.1 we define and classify kinds of errors. We present our PL test assessment system and its evaluation with three of the four examples introduced in Section 3.2. Eventually, we show related work in Section 3.4 and conclude in Section 3.5.

3.1. Potential Errors in Model-Based Product Line Engineering

The feature mapping has a major impact on the outcome of the resolved products in a PL, however, the design is complex and error-prone. We identify potential errors in a systematic way by checking each modeling paradigm for possibilities to add superfluous or omit necessary elements or change the value of an element's attribute. For each potential error we discuss its effects onto the resolved products.

Annotative Variability

In the annotative variability paradigm, we identify the following model elements for potential errors from the feature mapping model: mappings, their attribute feature value, mapped feature, and the set of mapped elements. The errors which can be made on these model elements and their effects are as follows:

N1) *Omitted mapping:* a necessary mapping is left out by its entirety. Subsequently, mapped elements will be part of every product unless they are restricted by other features. As a result, some or all products unrelated to the particular feature will include superfluous behavior. Products including the mapped feature are not affected, since the behavior was enabled anyway.

N2) *Superfluous mapping:* a superfluous mapping is added, such that a previously unmapped feature is now mapped to some base model elements. This may also include adding a mapping for an already mapped feature, but with inverted feature value. Adding a mapping with feature value set to *true* results in the removal of elements from products unrelated to the mapped feature. Contrary, adding a mapping with feature value set to *false* removes elements from any product which the mapped feature is part of. In any case the behavior of at least some products is reduced.

N3) *Omitting a mapped element:* a mapped model element is missing from the set of mapped elements in a mapping. Subsequently, a previously mapped element will not only be available in products which the said feature is part of, but also in products unrelated to this feature. As a result, some products offer more behavior than they should or contain unreachable model elements.

N4) *Superfluously mapped element:* an element is mapped although it should not be related to the feature it is currently mapped to. As a result the element becomes unavailable in products which do not include the associated feature. The product's behavior is hence reduced.

N5) *Swapped feature:* the associated features of two mappings are mutually exchanged. Subsequently, behavior is exchanged among the two features and thus, affected products offer different behavior than expected. The result is the same as exchanging all mapped elements among two mappings.

N6) *Inverted feature status:* the binary-value of the feature value attribute is flipped. The mapped elements of the affected mapping become available to products where they should not be available. At the same time, the elements become unavailable in products where they should be. For example, if the feature value is true and is switched to false, the elements become unavailable to products with the associated feature and available to any product not including the said feature. Of course, other feature mappings to the same element(s) must still be considered.

Compositional Variability

In PL modeling with compositional variability, a mapping is a bijection between features and modules composed from domain elements. Potential errors in the feature mapping models can be made at: mappings, mapped feature, and mapped module. We identify the following potential errors:

P1) *Omitted mapping:* a necessary mapping is missing in its entirety. This appears to us to be an unrealistic scenario, since one can automatically check for all modules being mapped to some feature. But if we consider the case of a missing mapping, products with the associated feature would be missing the modules' functionality.

P2) *Superfluous mapping:* a superfluous mapping is added. Similar to the above, this is an unrealistic scenario for the same reason: all modules should be mapped exactly once. In a model-based environment, this check should be easily automatable. However, if adding a superfluous mapping is possible, more behavior becomes enabled in products containing the mapping's feature.

P3) *Swapped modules:* the associated modules of two mappings are mutually exchanged. As a result, all products containing one of the two features, but not the other, do not offer the expected behavior. Subsequently, all products containing none or both features behave as expected.

P4) *Swapped features:* the associated features of two mappings are mutually exchanged. The result is the same as above for swapped modules.

Transformational Variability

For other paradigms, like delta-modeling [Sch10], we make similar observations. In contrast to compositional variability models, delta-oriented variability models start from an actual core product, instead of a base module. From this on, only the differences from one product to another are defined by *deltas*. In delta-modeling, mapping multiple features to the same delta is allowed. A delta may add elements to and remove elements from the core product at the same time. As potential points of errors in delta-modeling we identify deltas, a delta's set of mapped features, its set of removed elements from the base product, and its set of added elements.

D1) *Omitted delta:* the domain model misses an entire delta definition. Products containing features of the missing delta may lack behavior or offer too much of it. This depends on whether the delta removes and/or adds elements from/to the base product.

D2) *Superfluous delta:* an unnecessary delta is added. As a result, products containing the associated feature(s) will offer additional behavior. Also, affected products might lack behavior if the delta removes elements.

D3) *Omitted feature:* a necessary feature from the set of mapped features is missing. If no feature is left, the delta is not mapped at all which can be statically verified. If otherwise the set still contains at least one feature, any product containing the current set of mapped features but not the missing feature, offers too much or too few behavior. In some cases, the set of mapped features and the affected elements may collide with another delta, which is again statically verifiable.

D4) *Superfluous feature:* an additional feature is added to a delta's already complete set of mapped features. As a result, the delta will be available in less products. If the added feature mutually excludes one of the already mapped features, the delta will be applicable to no product at all. A static check can be used to validate that a set of features is satisfiable by some product. Only products containing the correct set of features, but not the superfluous ones, are affected by this error. Affected products may offer more or less behavior.

D5) *Omitted base element:* a delta's set of base elements is missing an element. In consequence, too few elements are removed from the core product by this delta to match the product's model. Thus any product containing the features from this delta offers too much behavior.

D6) *Superfluous base element:* a delta's set of base elements contains additional elements. This will remove more elements than necessary from the products affected by this delta. Hence, these products offer too few behavior.

D7) *Omitted delta element:* an element from the set of delta elements in a delta is missing. As a result, all products containing the delta's features offer more *or* less behavior than specified - depending on whether the delta element adds or removes elements.

D8) *Superfluous delta element:* an element from a delta's set of delta elements is missing. In consequence, the products containing the delta's features offer more *or* less behavior than specified. Again, this depends on whether the delta element adds or removes elements.

3.2. Product Line Test Assessment

As laid out in section 3.1, new kinds of errors can be made in model-based PLE than in contrast to single systems engineering. Current test design methods and coverage criteria are not prepared to deal with these errors and resulting faults. We propose a mutation system for PL systems. It is specifically designed to assess test quality, by means of FDC, for the whole product line rather than for single systems.

Mutation systems for PL need novel mutation operators. The reason for this is the separation of concerns in PLE, where variability and domain engineering are split into different phases and models. Mutation operators defined for non-variant systems cannot infer mutants including modules from other products, since this information is only available during domain engineering. However, we expect a high-quality test suite to detect such faults. Hence, we also propose new mutation operators based on the potential errors, we identified in Section 3.1. For conciseness, we only consider potential errors from annotative variability modeling for implementation.

3.2.1. Mutation System for Product Lines

Performing mutation analysis on PL tests is different from non-variant system tests, since in contrast to conventional mutation systems, a mutated domain model is not executable per se. Thus, testing cannot be performed until a decision is made towards a set of products for testing. This decision depends on the PL test suite itself, since each test is applicable to just a subset of products.

In Figure 3.1, we depict a mutation process for assessing PL test suites, which addresses this issue. Independently from each other, we gain (a) a set of domain mutants by applying mutation operators to the domain model and identify (b) a set of configurations describing the applicable products for testing. We apply every configuration in (b) to every mutant in (a), which returns a new set of product model mutants. Any mutant structurally equivalent to the original product model is immediately removed and does not participate in the scoring. The mutant product models are easily resolved to product mutants and finally, tests are executed. Our mutation scores are based on the domain model mutants, hence we establish bidirectional traceability from any mutant domain model to all its associated product mutants and back again. If a product mutant is killed by a test, we backtrack its domain model mutant and flag it as killed. The final mutation score is then calculated from the killed and the overall number of domain models mutants.

3.2.2. Product Line Mutation Operators

Here, we present mutation operators for feature mapping models with annotative variability. Furthermore, we enrich the mutation system by standard state machine

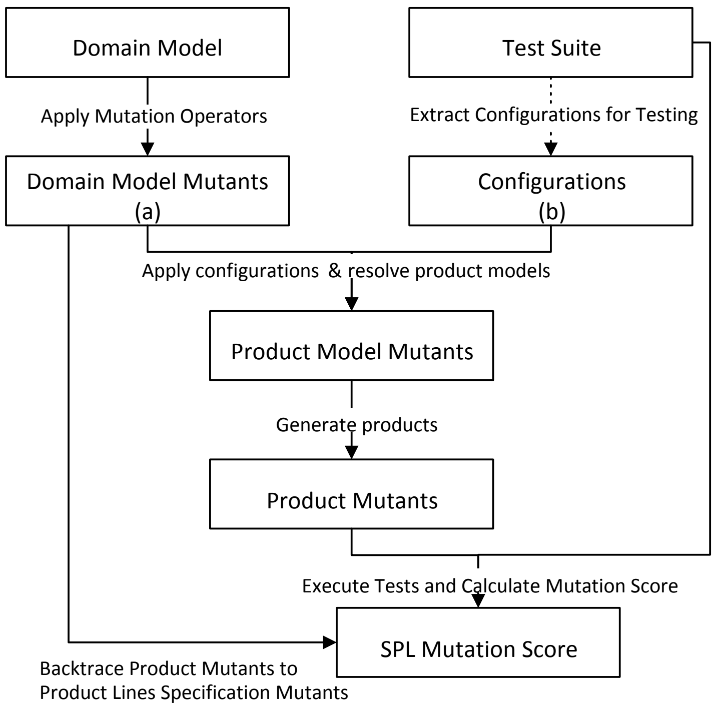

Domain Model
Test Suite
Apply Mutation Operators
Extract Configurations for Testing
Domain Model Mutants
(a)
Configurations
(b)
Apply configurations & resolve product models
Product Model Mutants
Generate products
Product Mutants
Execute Tests and Calculate Mutation Score
SPL Mutation Score
Backtrace Product Mutants to
Product Lines Specification Mutants

Figure 3.1.: Mutation process for PL systems

operators and apply them on domain-level as well. For systematic identification of mutation operators, we apply the guidelines presented in Chapter 2.3.2 and categorize each identified operator. Also, we discuss potentially invalid and hidden mutants resulting from each operator. Still there is no guarantee that the following list is complete.

Feature Mapping

We design the mutation operators according to the potential errors identified in Section 3.1. We do not consider inserting superfluous mappings as in this case it remains unclear which and how many UML elements should be selected for the mapping. We assume that this, if not carefully crafted, will lead to mostly invalid mutants.

Delete Mapping (DMP) The deletion of a mapping will permanently enable the mapped elements, if they are not associated to other features that constrain their enabledness otherwise. In our examples, no invalid mutants were created. However, for product lines that make heavy use of mutual exclusion (Xor and excludes) this does not apply. The reason for this are competing UML elements like transitions that would otherwise never be part of the same product. Multiple enabled and otherwise excluding transitions are possibly introducing non-determinism or at least unexpected behavior.

Some product mutants created with this operator might behave equivalent to an original product. This is the case for all products that include the feature for which the mapping was deleted. Since these mutants are structurally equivalent to the original product model, they are easy to detect.

Delete Mapped Element (DME) This operator deletes a UML element reference from a mapping in the feature mapping model. It resembles the case, where a modeler forgot to map a UML element that should have been mapped.

Similar to the delete mapping operator, this operator may yield non-deterministic models, where otherwise excluding transitions are concurrently enabled. Product mutants equivalent to the original product model can be derived, if the feature associated to the deleted UML reference is part of the product. Again, this is results in structural equivalence to the original product.

Insert Mapped Element (IME) This operator inserts a new UML element reference to the mapping. This is the contrary case to the operators defined before, where mappings and UML elements were removed. However, inserting additional elements is more difficult than deleting them, since a heuristic must be provided for creating such an additional element. We decided to copy the first UML element reference from the subsequent mapping. If there are no more mappings, we take the first mapping. This operator is not applicable if there is just one mapping in the feature mapping model.

Again, there is a chance of creating invalid mutants: If a UML element reference is copied from a mutually excluded mapping, the resulting model may be invalid due to non-determinism.

Also structurally equivalent mutants are created, when the features from the subsequent mapping, which acts as source for the copied element, and the target mapping are simultaneously activated in a product.

Swap Feature (SWP) Swapping features exchanges the mapped behavior among each other. This operator substitutes a mapping's feature by the following mapping's feature and vice versa. The last feature to swap is exchanged with the very first of the model.

Non-deterministic behavior and thus invalid models may be designed by this operator. This is due to the fact that the mutation operator may exchange a feature from a group of mutually exclusive features by an unrestricted feature. In consequence, the previously restricted feature is now independent, while the unrestricted feature joins the mutual exclusive group. This may concurrently enable transitions which results in non-deterministic behavior.

We gain structurally equivalent mutants, if the two swapped features are simultaneously activated.

Change Feature Value (CFV) This operator flips the feature value of a mapping. A modeler may have selected the wrong value for this boolean property of each mapping.

The operator must not be applied to a mapping, if there is a second mapping with the same feature, but different feature value. Otherwise, there will be two mappings for the same feature with the same feature value, which is not allowed for our feature mapping models.

This operator may yield invalid mutants, if it is applied to a mapping that excludes another feature. In that case, two otherwise excluding UML elements can be present at the same time, which may result in invalid models, e.g. two default values assigned to a single variable or concurrently enabled transitions.

UML State Machine

In the past 20 years, many mutation operators for transition-based systems were defined [FDMM94, OLAA03, BBW06, BH08]. Here, we limit ourselves to the design of operators based on transitions as these may have the strongest impact on the behavior of the SUT. We do not design operators that can be mimicked by the combination of two of them. In particular, we do not consider the exchange of an element by another, since this can easily be mimicked by removing and inserting the removed element at another point in the model.

We identified five targets for mutation: (i) remove the entire transition, change its (ii) target state, as well as mutating its (iii) triggers, (iv) guard, and (v) effect. The latter three can be mutated according to the three defined categories delete, add and change.

For all mutants created by the here presented operators, there is a chance of materializing mutants behaving equivalent to the original product. This is the case, when the mutated element is part of a disabled feature. Of course, hidden mutants – if detected – will be excluded from the scoring.

In general, we will not apply any class mutation to our UML state machines [KCM00]. The system's logic is designed in the state machine diagrams, while the classes are merely containers for variables and diagrams.

Delete Transition (DTR) Deletes a transition from a region in a UML state machine. This operator might create invalid UML models, if not enough transitions remain on a pseudo-state (fork, join, junction, and choice) [OMG, p.555].

Change Transition Target (CTT) Changes the target of a transition to another state of the target state's region. This operator is only applicable if the region has more than one state.

Delete Effect (DEF) Deletes the entire effect from a transition. We consider sending signals to the environment or other components to be part of a transition's effect, hence they are deleted as well.

Delete Trigger (DTI) Deletes a transition's trigger. Only a single trigger is deleted at a time and every trigger is deleted once.

Insert Trigger (ITG) Copies an additional trigger to a transition. The trigger is copied from another transition within the same region. This may lead to non-deterministic behavior if both transitions, the source transition of the trigger and the mutated transition, are outgoing transitions of the same state.

Delete Guard (DGD) Deletes the entire guard of a transition. This may lead to non-deterministic behavior of the state machine, if another transition is enabled simultaneously. Furthermore, in the case of transitions without triggers and where source and target are the same state, this operator leads to infinite looping of the state machine over the mutated transition. The reason for this behavior is UML's run-to-completion semantic, where an enabled transition without triggers is immediately traversed.

Change Guard (CGD) Changes a guard's term by exchanging operators or substituting boolean literals by their inverse. Our CGD operator supports 30 different arithmetic, relational, bitwise, compound assignment, and logic operators. Furthermore, literal "null" is exchanged by "this".

This may cause mutants with non-deterministic behavior, whenever two transition become concurrently enabled due to the manipulation of one of their guards.

3.3. Evaluation

3.3.1. Setup

We employ the three example product lines introduced in Section 2.4 for performing a mutation analysis on them. We designed the test suite for each example automatically by applying model-based testing techniques. We chose all-transitions coverage for selecting the tests. A test generator then automatically designed the tests and outputs XML-documents. From the tests, SPLTestbench selected variants for testing and resolved mutated product models from the mutated domain model.

Since our examples lack implementations, we decided to generate code from the mutated product models and run the tests against them. Therefore, we developed and employed a code generator for transforming individual product models into Java. Another transformator generates executable JUnit code from the tests which we gained from the test generator. The mutation system then collects all the code artifacts, executes the tests against the product code, and finally reports the mutation scores for all tests and for every operator individually. All of the transformations above and the mutation system are integrated into SPLTestbench.

Generating code *and* tests from the same basis for testing the code is not feasible in productive environments, since errors propagate from the basis to code *and* tests. However in our case, tests are executed against code derived from mutated artifacts, while the test cases were generated from the original model.

3.3.2. Results

We were able to assess the test quality for all three test suites derived from the examples. Here, we present the results. For each mutation operator we measured the amount of detected mutants based on the domain model. In addition, we assessed accumulated mutation scores for each example over all mutation operators and vice versa, the accumulated results for each mutation operator over all examples. The detailed results for feature mapping operators can be read from Table 3.1 and for UML operators from Table 3.2.

For every example we tracked the number of original products selected for testing, generated domain model mutants, and resolved product mutants. Test-wise we counted tests, test steps by means of stimuli and expected reactions in all tests, tests executed against all product mutants, and the number of failed tests during test execution.

For the eShop example, SPLTestbench selected four products for testing. Independent from this, the mutation system generated 30 domain model mutants and 96 product mutants for the mapping mutation operators. For the state machine mutation operators it generated 122 domain model mutants and 478 product mutants. Every test case from the 13 test cases for this example was executed against every suitable mutant. This results in 302 test executions for the mutants created by the mapping mutation operator and 1,553 test executions for state machine mutation operators. Ultimately, 20 tests for mapping operators and 283 tests for state machine operators failed, killing 69.67% and 36.67% of the mutants.

For the Ticket Machine example, we executed 1332 tests for state machine operators, executed them against 296 product mutants, and eventually 272 tests failed, killing 66.89% of the PL mutants. Analog to the eShop, we executed less tests and generated less product mutants for the feature mapping operators: 252 tests were executed against 56 product mutants, which were resolved from 28 domain

Table 3.1.: Mapping operator scores per mutation operator in % and Accumlated Scores (Acc)

Op.	eShop	Ticket Machine	Alarm System	Acc
DMP	0.00 (4)	0.00 (5)	0.00 (8)	0.00
DME	0.00 (14)	0.00 (8)	0.00 (21)	0.00
IME	75.00 (4)	40.00 (5)	50.00 (8)	52.94
SWP	100.00 (4)	60.00 (5)	62.5 (8)	70.59
CFV	100.00 (4)	100.00 (5)	87.50 (8)	94.12
Acc	36.67 (30)	35.71 (28)	30.19 (53)	33.33

Table 3.2.: UML operator scores per mutation operator in % and Accumlated Scores (Acc)

Op.	eShop	Ticket Machine	Alarm System	Acc
DTR	89.29 (28)	84.21 (19)	63.16 (19)	80.30
CTT	64.29 (28)	63.16 (19)	36.84 (19)	56.06
DEF	100.00 (16)	82.35 (17)	61.54 (13)	82.61
DTI	82.61 (23)	100.00 (13)	94.12 (17)	90.57
ITG	20.83 (24)	27.78 (18)	16.67 (18)	21.67
DGD	0.00 (1)	42.86 (14)	50.00 (2)	41.18
CGD	100.00 (2)	68.75 (48)	90.00 (10)	73.33
Acc	69.67 (122)	66.89 (148)	57.17 (98)	65.21

model mutants. The tests yield an even lower mutation score of 35.71% than for the eShop case study.

In case of the Alarm System, we executed 537 tests against 278 product mutants created by the mapping mutation operators and 1,168 tests against 585 product mutants created by the state machine mutation operators. Eventually, 37 and 123 tests failed, killing 30.19% and 57.17% of the mutants, respectively. The results are summarized in Table 3.3 and 3.4.

The table shows some interesting effects. For most of the operators we gain mutation scores between 60% and 100%. This is in the expected range for test suites generated with the all-transitions coverage on random mutants [Wei09, LS12]. However, there are some notable exceptions: DMP (0%) and DME (0%) on feature mappings, ITG (21.67%) and DGD (41.18%) on base models. The reason is that these operators add superfluous behavior to the product. Errors consisting of additional (unspecified) behavior are notoriously hard to find by specification-based testing methods. Specification-based testing checks whether the specified behavior is

Table 3.3.: Summarized Results for Mapping Operators

	eShop	Ticket Machine	Alarm System
Products for testing	4	4	6
Product line mutants	30	28	53
Product mutants	96	56	278
Test cases	13	9	12
Test steps	103	68	62
Tests executed	302	252	537
Failed tests	20	30	37

Table 3.4.: Mutation Results for State Machine Operators

	eShop	Ticket Machine	Alarm System
Products for testing	4	4	6
Product line mutants	122	148	98
Product mutants	478	296	585
Test cases	13	9	12
Test steps	103	68	62
Tests executed	1,553	1,332	1,168
Failed tests	283	272	123

implemented; it cannot find unspecified program behaviors (e.g. viruses). Program-code-based testing checks whether the implemented behavior is correct with respect to the specification; it cannot find unimplemented requirements (e.g. missing features).

To solve this problem, we must not only check whether a required feature is implemented, but also whether a deselected feature is really absent. One possibility for this is to add *product boundary tests*, which are automatically generated from the domain model. An detailed elaboration of this idea can be found in [WWL15]. Another possibility is to admit so-called “negative tests”, which are constructed manually and test whether a certain behaviour is absent. This idea will be followed in section 6. Another method for detecting superfluous behavior is code coverage. Metrics, such as branch coverage, will easily detect superfluous behavior in separate modules. Discovering superfluous behavior hidden within production code is much harder, since this would require high efforts for test generation, e.g. to achieve metrics such as multiple condition/decision coverage (MC/DC).

3.3.3. Threats to Validity

A general concern regarding mutation analysis is the likeliness of the mutation operators representing actual errors a developer can make. Hence, we decided to design the operators in a systematic way and followed the general guidelines for designing errors in section 2.3.2. Due to the systematic way, we disregard the likelihood of an error being made. In consequence, errors that are unrealistic in practice and hard to discover by the tests yield bad mutation scores, although they are negligible due to their unlikeliness to appear. This could lead to investing into improved fault detection capability, where the impact might be low, when the efforts could be spend more efficiently elsewhere.

3.4. Related Work

Mutation analysis for PL systems seems to be a rather new topic. To our knowledge, there is no publication dealing with mutation operators on all model artifacts of a domain model. Though, Henard et al. proposed two mutation operators for feature models based on propositional formulas in [HPP+13]. They employ their mutation system for showing the effectiveness of dissimilar tests, in contrast to similar tests. For calculating dissimilarity, the authors provide a distance metric to evaluate the degree of similarity between two given products.

In contrast, mutation analysis for behavioral system models, e.g. finite state machines, is established since two decades. Fabbri et al. introduced mutation operators for finite state machines in [FDMM94]. In addition to our operators, they also consider adding states and the exchange of elements (event, guard, effect) by another. Belli and Hollmann provide mutation operators for multiple formalisms: directed graphs, event sequence graphs [BBW06], finite state machines [OLAA03], and basic state charts [BH08]. They conclude, that there are two basic operations from which most operations can be derived: omission and insertion. Also for timed automata, mutation operators can be found in [ALN13].

In [SZCM04] Stephenson et al. propose the use of mutation testing for prioritizing test cases from a test suite in a PL environment. Unfortunately, the authors provide no evaluation of their approach.

3.5. Conclusions

In this contribution, we lifted mutation analysis to the domain engineering-level. We defined and investigated mutation operators for feature models, mapping models, and UML state machine models. As opposed to product-based mutation analysis, our mutation operators are based on the domain model. This allows us to mimic realistic errors made by humans during modeling a PL. To our knowledge, this is the first step towards a qualitative evaluation of PL tests, which is based on the domain model.

Our results for the three examples are as expected for most of the mutation operators. As predicted, mutation operators contributing superfluous behavior are hard to detect for conformance tests. Such mutations are DMP (0%) and DME (0%) on feature mappings and ITG (21.67%) on base models. For most of the other operators we gain scores above 70%, which is in the expected range for all-transitions coverage [Wei09]. For DGD and CTT mutations the tests score surprisingly low results. Here, further investigations seem necessary.

In conclusion, we identified a lack of fault detection capability in standard test procedures for PL systems. Even simple faults are not detectable, neither by all-transitions, MC/DC as for safety-critical systems, nor any other conformance test procedure. As indicated, the results are applicable to at least the here surveyed PLE paradigms with annotative, compositional, and transformational variability. We assume, other paradigms suffer from this lack as well. Unfortunately, current procedures for negative testing, which could potentially detect such faults are still not enabled for PL systems. Thus, future work will proceed to enable negative testing procedures for PL systems.

Part II.

Model-based Testing for Product Lines

Chapter 4.

Automated Test Design for Product Lines

Testing is a costly and repetitive task, therefore automation is implemented where possible. We have already introduced the principles of automated test design by model-based testing (MBT) in Section 2.2 in Part I of this thesis. In this contribution, we deal with test design automation for PL systems. We present the intuitive approach of resolving a representative set of products from a PL for the purpose of testing, introduce an improved approach that works on the domain engineering-level, and compare them. Both approaches are implemented in the prototype implementation *SPLTestbench* and evaluated using several examples.

This contribution is structured as follows: In Section 4.1, we define both mentioned automated test design approaches for product lines. We present the evaluation and the comparison of both approaches in Section 4.2. Eventually, we show the related work in Section 4.3 and conclude in Section 4.4.

4.1. Model-based Testing for Product Lines

Tests for PL systems face two challenges: covering a significant subset of products and covering a significant subset of the test focus on the domain-level. As the products share commonalities, some test cases may be applicable to more than one product.

Domain models consist of a base model, in our case UML state machines, a feature model explicitly expressing the PL's variation points, and a feature mapping model that connects the two former. This is essentially the view of domain engineering, where commonalities and variabilities are specified. Of course, this view is also applicable from the point of testing. From the domain artifacts, application artifacts can be derived, i.e. products and tests. Based on this, we define two approaches to automated test design for PL systems as depicted in Figure 4.1: *application-centered* (AC) and *domain-centered* (DC).

The AC approach consists of first selecting a representative set of products (test models) and second generating test cases from each of these models. This approach is focused on satisfying a defined coverage on each test model, which also leads to an overlap of the resulting test cases. In contrast, the DC approach directly

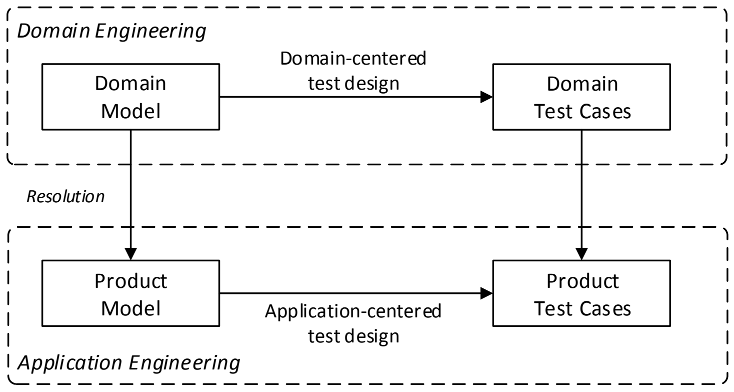


Figure 4.1.: Product line testing.

applies the domain model for designing tests. This second approach is focused on the behavior defined at the domain-level and does not focus on covering single products. Instead, there is still variability in the choice of the concrete products for which the test cases will be executed. Both approaches are investigated in more detail in the following paragraphs.

4.1.1. Application-Centered Test Design

We call any test design method that binds the variability by selecting products before the test design phase AC (see Figure 4.2). In this approach, product models are selected from the domain model according to a predefined *feature coverage criterion* first. Since testing all products individually is usually not feasible, these criteria are applied to gain a representative set of configurations from the PL for testing. Criteria like n-wise are presented in [OZML11, PSK$^+$10, JHF12].

From the selected product models, tests are designed according to a given *model coverage criterion*. In our example, this may be any coverage criterion applicable to UML state machines.

Since each test case is generated only for a single configuration, the resulting test suite will be specific to its respective product. Therefore, the test generation will result in *application-specific test suites*. Due to the commonalities of the products of a PL, a test case that was generated for a single configuration may be applicable to other configurations as well. Consequently, test cases that aim for the same goals are executed over and over again. Still, test cases must be generated for all selected configurations and then a model coverage criterion is applied to all of their corresponding models.

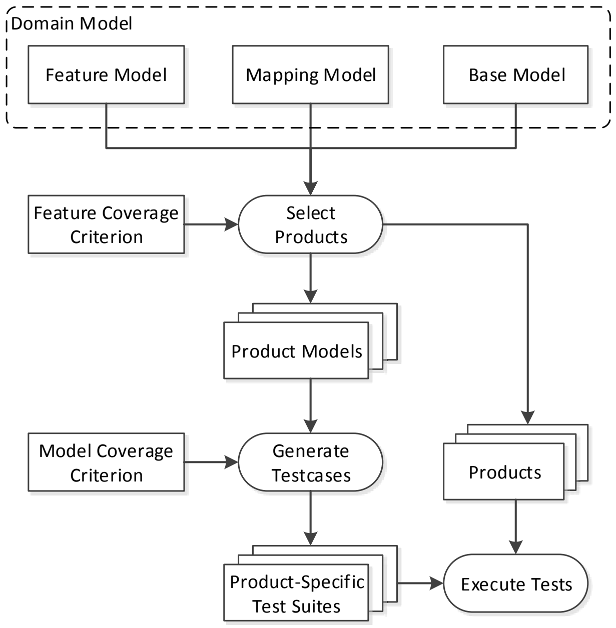


Figure 4.2.: Detailed application-centered (AC) test design process.

4.1.2. Domain-Centered Test Design

In this approach, we use only domain engineering level artifacts for test generation. DC test design preserves variability until a product under test has been selected for test execution.

A major advantage of this approach is the focus on the test aspects of the domain model without deriving single products first. This allows to focus only on the test aspects without the overlap of the results from independently generating tests for similar products. This approach maximizes the coverage of the test targets and thus should lead to high-quality test cases, while at the same time it classifies the products into sets that are most test-worthy due to their diverse behavior.

Though, using the base model as the only input for test design is not sufficient as it lacks information about the model element's associations to the features and the features itself, since they impose additional constraints on the system's behavior. Also, the base model may include contradictory requirements and syntactical violations. Thus, the main challenge of this approach is to merge the domain model, consisting of a feature model, mapping model and a base model, into a single model artifact that a standard test generator will accept as valid input (Fig. 4.3). We identified two solutions to this problem: 1) the *step-by-step* approach: sequentially excluding non-conforming configurations during test design-time, and 2) the *pre-*

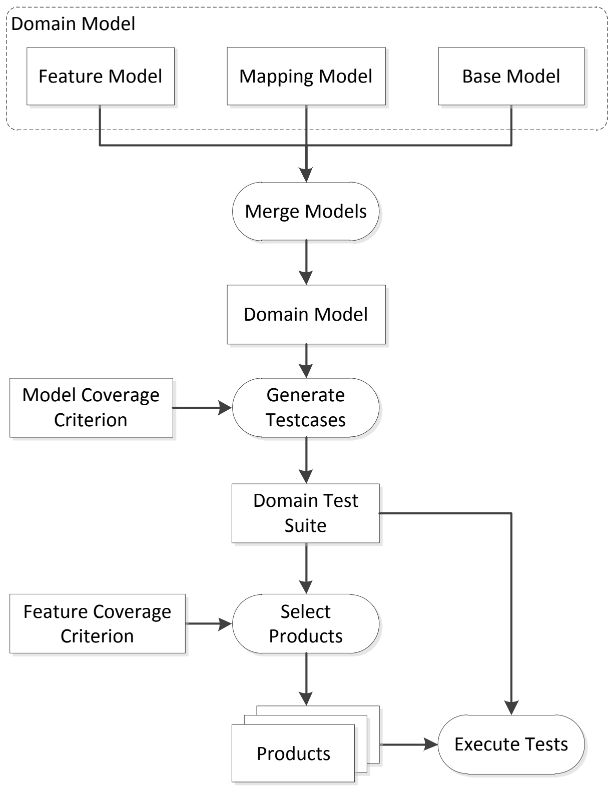


Figure 4.3.: Detailed domain-centered (DC) test design process.

configuration approach: choosing a valid configuration before designing individual test cases.

The Step-by-Step Approach

The key idea of the step-by-step approach is to sustain the variability until it becomes necessary to bind it. Therefore, at the beginning of each test case design the assumption is made that the test case is applicable to any valid product of the PL. Since not necessarily all valid paths in the base model are applicable to all products, the test designer must take account of test steps that bind variability. A test step must bind variability if not all products do conform. Subsequently, the set of valid products for this particular test case must be reduced by the set of non-conforming products. Hence, each test case is valid for any of the remaining products that do conform.

We implemented the step-by-step approach for state machines as follows. The tracking of the excluded products can be achieved by introducing a Boolean variable into the system class for each feature that is not a core feature (*feature variable*). This variable is set whenever a transition added to a test case forces the mapped feature to be present (*true*) or prohibits its presence (*false*). For preventing repeated assigning to such a feature variable, an additional control variable is necessary. Therefore, another Boolean variable is added for each non-core feature to the system class (*control variable*) and must be initialized with *false*. Each of these variables tracks whether the corresponding feature has not yet been set and is thus free (*false*) or was already set (*true*). In the latter case, no further assignments to the feature variable are allowed as the feature is bound to the value of the corresponding feature variable.

The guards and effects on the transitions of the respective state machine can then be instrumented with these variables to include variability information in the state machine. For each feature f_i that is mapped by a mapping $m_{f_i,t}$ to a transition t its partial feature formula $pf_{f_i,t}$ is derived. Since we have now derived all features that have to be accounted for before taking transition t, we collect them in a single conjunction:

$$G_t = \bigwedge_{i=1}^{n} pf_{f_i,t}$$

We still have to incorporate the protection against repeated writings by substituting each feature literal in G_t with the following expression: $(\neg f_c \vee (f_v == m_{f,v}))$, where f_c is the control variable of feature f, f_v is the feature variable of f, and $m_{f,t}$ is the value of the feature mapping's flag associated with transition t. The resulting expression can safely be conjoined with t's original guard.

Finally, t's effect must bind the variability of all associated features. This is possible by setting the control variable f_c to true and the feature variable f_v to the value of its mapping's flag for each feature that appeared in G_t. Thus, for each feature f in G_t we append the following code to the effect of transition t:

```
if $\neg f_c$ then
    $f_c \leftarrow \top$
    $f_v \leftarrow m_{f,t}$
```

Once the test generator executes this code, the feature is bound and it is not possible to change the binding for this test case anymore. Figure 4.4 shows the result of merging the domain model into a single UML state machine for the excerpt of the eShop introduced in Figure 2.5.

After test generation has finished, the valid configurations for a particular test case can be read from the feature variables in each test case. Since the test cases may contain variability we obtain an *incomplete configuration* from each test case. An incomplete configuration is a configuration that supports a three-valued

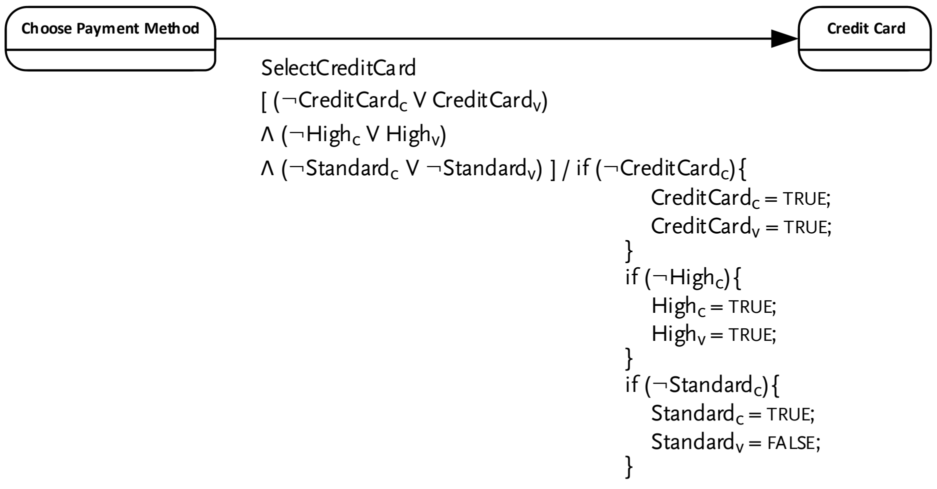


Figure 4.4.: Excerpt of the merged domain model with step-by-step approach applied.

semantics for features instead of two values. The first two values are the same as in normal configurations (selected/unselected), the third stands for *undecided*. An undecided feature expresses variability by making no premise on the presence of the feature. Hence, each of the resulting test cases is *reusable* for any product of the PL that conforms to the following: For each control variable that is evaluated to true, the corresponding feature variable evaluation indicates whether this feature must be selected or unselected in the product. Features for which the respective control variable evaluates to *false* are yet undecided and thus not evaluated.

The Pre-Configuration Approach

In the pre-configuration approach, test goals are selected from the domain model and also the test design is performed on this model similar to the step-by-step approach. However, during the design of an individual test case, the product configuration is fixed from the beginning of each test case and must not change before a new test case is created. Consequently, within a test case the test designer is limited to target for test goals that are specific to the selected product configuration. Thus, satisfying all domain model test goals is a matter of finding the sufficient configurations.

We implemented the pre-configuration approach by adding a signal to the very beginning of the base model for configuring the model. Therefore, we introduce a new state to the state machine, redirect all transitions leaving the initial state to leaving this new state, and add a transition between the initial state and the new state. Due to the UML specification the redirected transitions must not have

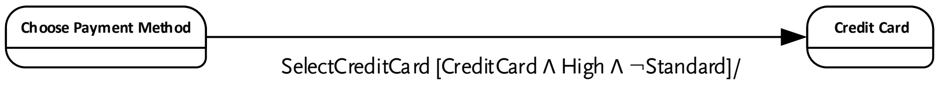


Figure 4.5.: Excerpt of the merged domain model with the pre-configuration approach applied.

a trigger, which is why we can add a trigger for configuration purposes to each of them. The trigger listens to a configuration signal that carries a valuation for all non-core features. The guard of these transitions must protect the state machine to be configured with invalid configurations and thus contains the propositional formula corresponding to the PL's feature model. Since any configuration that is provided by the signal must satisfy the guard's condition, only valid configurations are accepted.

After validating the configuration, the parameter values of the signal will be assigned to system class variables by the transition's effect. Hence, for each non-core feature a boolean variable indicating whether the feature is selected or not is added to the system class. Again, transitions specific to a set of products are protected by these variables, like in the step-by-step approach to limit the base model behavior to a behavior an actual product can conform to. However, control variables need not to be checked during test design, since the configuration is fixed and valid from the beginning of each test case. Therefore, it is sufficient to derive the partial feature formulas *pf* for all features f_n that are mapped to a transition t by a mapping $m_{f_i,t}$ and construct a conjunction from these formulas:

$$G_t = \bigwedge_{i=1}^{n} pf_{f_i,t}.$$

For conjoining G_t with t's guard, of course, the feature literals must be exchanged by the corresponding feature variables from the class. Figure 4.5 depicts the resulting merged domain model for this approach. As a result, no product can conform to any test case's first step, since it was used to set the configuration and presents not the real system's behavior. In a simple post-processing action this configuration step must be removed from the test cases before testing can be performed.

With these transformations made to the base model, a test designer can already create test cases for the PL. However, each test case will be specific to one configuration. As pointed out, a generalization is possible, though is not yet implemented it. The necessary model transformation for this generalization is sketched as follows: The additional transformation steps consist of adding Boolean control variables for each non-core feature to the system class and initializing them with *false* and effects on transitions for setting these variables to *true* when traversed by the test generator. More precisely, for every transition t that is mapped to a feature f by a mapping $m_{f,t}$, the following code needs to be appended to t's effect for every mapped feature f.

```
if $\neg f_c$ then
    $f_c \leftarrow \top$
```

A test generator will set every control variable for all features associated with that transition, when this transition is added as a step to a test case. Hence, each control feature that is still *false* at the end of a test case indicates a free variation point. This result can be captured in a reusable test case for a subsequent selection of variants for testing.

4.2. Evaluation of both Approaches

In this section, we present the implementation of both DC test design approaches, step-by-step and pre-configuration. Furthermore, we apply them to the three examples introduced in Section 2.4 to compare these approaches to AC test design methods.

4.2.1. Tool Chain SPLTestbench

The SPLTestbench is composed of five major components: (i) the feature injector to merge domain models as introduced in section 4.1.2, (ii) a model printer that exports the model to a test generator-specific format, (iii) a third-party test generator - in this case Conformiq, (iv) a configuration extractor that collects incomplete configurations from the generated test cases, and (v) domain-specific languages [Voe13] (DSLs) that facilitate design and data processing of a) feature models, b) feature mapping models, and c) configuration models. Figure 4.6 depicts the SPLTestbench workflow.

As defined in 4.1.2, the first step towards a domain-level test suite is to merge the individual models of a domain model into one. This task is performed by the *feature injector* by limiting the behavior in the base model according to feature and mapping model. We created two libraries for this purpose: the first library generates propositional formulas for a given feature model and for individual features as discussed in the preliminaries in Section 2.1. The latter library consists of typical transformations on UML models for facilitating the creation and manipulation of states, transitions, guards, triggers, effects, and signals. Eventually, the newly gained model must be exported into a format for a particular test generator. Therefore, we implemented a *model printer* that reuses the UML model transformations library to prepare the model to be exported and then prints it into the target file format. This means, no adaptations to test generators are necessary, since the output format of this transformation complies to the expected input of the test generator.

When coupled with Conformiq, SPLTestbench currently supports UML models

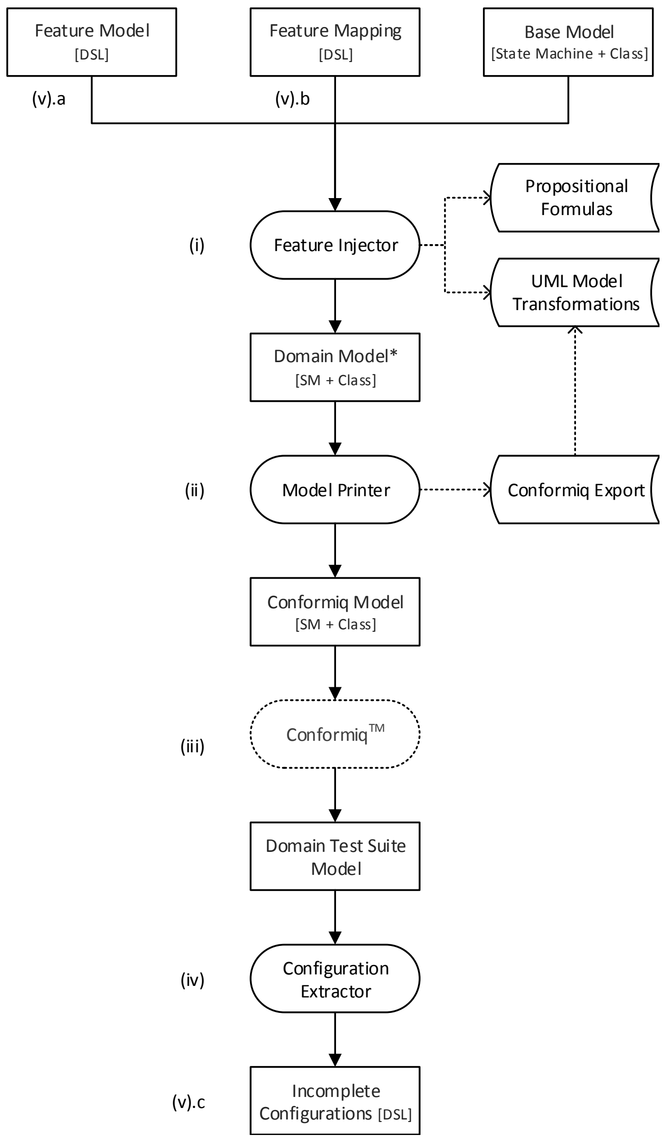


Figure 4.6.: Workflow of the SPLTestbench.

that have a single class and one or more state machines that specify the class' behavior. The class has to provide at least two ports, one for receiving signals from and the other for sending signals to the environment. One state machine must be selected as the class' *classifierBehavior* and must own at most one region, while each region must own an initial state. A transition may own a trigger, a guard, and/or an effect. This far, only SignalEvents are supported. Signals, SignalEvents,

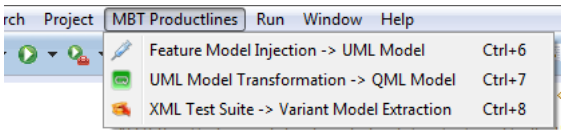

rch
Project
MBT Productlines
Run
Window
Help
Feature Model Injection -> UML Model Ctrl+6
UML Model Transformation -> QML Model Ctrl+7
XML Test Suite -> Variant Model Extraction Ctrl+8

Figure 4.7.: Menu options in the SPLTestbench.

and PrimitiveTypes are stored in the same Package as the class.

Conformiq delivers a test suite in XML format as a results of the test generation. As these test cases are reusable test cases, we can derive incomplete configurations from them to sample valid configurations in a later stage.

SPLTestbench is designed as Eclipse plug-in. Figure 4.7 shows how the components (i), (ii), and (iv) integrate into the IDE. Each of the three menu items starts an individual wizard that guides the user through the details of the respective process.

4.2.2. Experiment Settings

For our experiment we generated tests according to both presented approaches, AC and DC test design. For AC test design we selected two different feature model coverage criteria for individual comparison: all-features-included-excluded and all-feature-pairs [POS+12].

For the actual test case design we employed Conformiq Designer for both approaches. Conformiq Designer supports control-flow, branching, requirements, and path coverage criteria. For the individual state machine models as well as for the merged state machine models we applied all-transition coverage [WS10]. Though, there are many other more sophisticated metrics for state machines to choose from [GMP03, CLMG+10, GVP13].

Due to Conformiq Designer's state machines semantics it is not possible to apply the Body Comfort System case study directly. Although further model transformations can be used to adapt the model, those additional transformations lead to growth of the model's state space. In the case of the BCS, memory consumption of Conformiq Designer was then too high to be processed by available workstations. For this reason, Real-Time Tester (RTT) by Verfied and University of Bremen is employed to handle the BCS case study [Pel13]. As in Conformiq Designer, we apply all transitions coverage with RTT as well.

4.2.3. Results

We were able to generate test suites for both approaches with all the aforementioned parameters for all examples. Here, we present the first results. We counted the

Table 4.1.: Test cases, test steps, and configuration for each of the presented approaches.

	Example	Approach			
		AC-IX	**AC-PW**	**DC-Pre**	**DC-Step**
Tests	Ticket Machine	13	50	9	9
	Alarm System	21	57	12	12
	eShop	20	71	13	13
Steps	Ticket Machine	33	56	48	39
	Alarm System	55	165	62	50
	eShop	135	486	48	39
Conf.	Ticket Machine	2	6	5	1
	Alarm System	3	9	6	2
	eShop	2	7	4	2

number of test cases and test steps that were generated by the test generator. Of course, the amount of configurations that is necessary to execute the test cases is of equal interest.

The results for these measures are shown in Table 4.1 for each individual approach: AC with all-features-included-excluded (AC-IX) as well as with pair-wise (AC-PW) coverage and DC with pre-configuration (DC-Pre) as well as with step-by-step (DC-Step). The AC-PW approach scores the highest values for all measures since it applies the strongest feature coverage criterion and thus covers a maximum of configurations. Consequently, more test cases and test steps are generated than for any other approach. In contrast, the DC-Step yields the lowest scores for any measure except for tests steps on the Ticket Machine, while at the same time — as stated in section 4.1.2 — it is focused on covering every reachable transition. We take this as an indicator for DC test design to scale better than AC approaches.

For the BCS case study, no final results are available yet. So far, RTT generated 13 tests and covered 284 of 369 transitions. The researchers at University of Bremen are confident that RTT is able to cover the remaining transitions, although this will need some optimization of RTT's configuration.

Concluding, DC test design produces test suites with a significantly lower number of tests and test steps than AC test design. Thus, test execution efforts are much lower for these test suites. As we will see in Chapter 5, this does not necessarily lead to a lower error detection capability.

4.3. Related Work

Testing is one of the most important quality assurance techniques in industry. Since testing often consumes a high percentage of project budget, there are approaches to automate repeating activities like, e.g., regression tests. Some of these approaches are data-driven testing, keyword-driven testing, and model-based testing. There are many books that provide surveys of conventional standard testing [AO08, Bin99, MSB12] and model-based testing [BJK$^+$05, UL06, ZSM11]. Here, we use MBT techniques and apply them to PL systems. Modeling languages like the UML have been often used to create test models. We apply UML state machines.

Feature models are commonly used to describe the variation points in PL systems. There are several approaches to apply feature models in quality assurance. For instance, Olimpiew and Gomaa [OG05] deal with test generation from PL systems and sequence diagrams. However, we focus on UML state machines and describe different approaches for combining both. In contrast to sequence diagrams, state machines are commonly used to describe a higher number of possible behaviors, which makes the combination with feature models more complex than combining feature models and sequence diagrams. As another example, McGregor [McG] shows the importance of a well defined SPL testing process. Just like McGregor, the focus of this contribution is to investigate the process of creating tests rather than defining the structurally possible relations of feature models and state machines. Pohl and Metzger [PM06] emphasize the preservation of variability in test artifacts of SPL testing. As we derive test case design from models automatically, this variability is preserved. Lochau et al. [LSKL12] also focus on test design with feature models. In contrast to our work, they focus on defining and evaluating coverage criteria that can be applied to feature models. In the presented AC approaches, we strive for using such coverage criteria on feature models for the automation of test design. Furthermore, Lochau et al. recently proposed an extension for AC testing to decrease its efforts by omitting common test goals that were already exercised for other products of the PL [LOGS12].

The contribution that is most relatable to the DC approach is presented by Cichos et al. in [COLS11]. They merge feature models into the behavioral model in a different way than presented here. Then they select a set of configurations for testing, e.g. by applying pair-wise selection. One after another, each configuration is passed to the test generator until all test targets are covered. Consequently, each resulting test case is applicable to the configuration it was generated for. No post-processing, e.g. model-checking, takes place to identify features the test case is agnostic to. In conclusion, they use domain-centered infrastructure (domain-model) to generate test cases in an application-centered manner. In contrast, DC test design as presented above, needs no predefined set of configurations and yields reusable test cases from which configurations can be sampled. Furthermore Cichos et al. state, no standard test generator can be applied for their method due their approach to merge domain models. In contrast, DC test design is easily integrated with commercial off the shelf test generators and existing tool chains - as demonstrated with Conformiq Designer and Real-time Tester.

4.4. Conclusion, Discussion, and Future Work

In this contribution, we presented different approaches to the automatic test design for PL systems. We presented two approaches to use feature models and mapping models for automatic test design from base models. Our main contributions are the definition and evaluation of test design approaches using three examples.

Methodically, the main difference of both presented approaches lies in the order in which test targets by means of coverage criteria are applied. In the AC approaches, first a feature model coverage criterion and *then* a test model coverage criterion is applied. In contrast, for DC approaches only a test model coverage criterion is applied. As our results show, this leads to very different test suites for both approaches. Test suites created with DC test design approaches have significantly fewer test steps, but also cover all transitions of the model. We conclude that DC test design scales better w.r.t. system size than AC methods by means of test steps and the amount of products to test. Although we have not yet carried out a mature performance analysis, the same argument should hold for test generation and execution time, as tests are generated only for a single model and executed on less products.

Currently, the approaches are limited to mapping features to transitions. In larger systems, not only transitions, but also variables, default values, classes, whole components, and other behavioral or structural elements are potential targets for mappings. This need further investigation as one cannot simply apply our approach to these kinds of elements. For example, to allow a variable to have multiple default values is a non-trivial problem, not only because most UML editors will not allow to create a second default value for a single variable. But also test generators will not accept a model with two variables that have the same name. Hence, efficient solutions must be found to deal with such syntactically incorrect base models.

Furthermore, we showed that DC test design opens the fields for new feature coverage criteria. Since this test design method preserves variability throughout the test design process, products can be selected with other intentions, e.g., the fewest configurations for executing all tests, or the largest configurations by means of activated features. This kind of criteria is investigated in the next chapter.

Chapter 5.

Test-driven Product Sampling

With the introduction of domain-centered (DC) test design in Chapter 4, product configurations can be sampled from the PL's test cases. So far, the only criterion for which sampling was performed is minimizing the amount of configurations. A valid reason for employing this criterion is to minimize the amount of tested products, and subsequently reduce test effort. However, its fault detection capability (FDC) has not yet been assessed.

In this contribution, we address the question whether coverage criteria for sampling configurations from test cases affect the FDC of a test. We set up our experiment to reuse the same test suite for every sampling, thus we can measure the effects of different sampling criteria in isolation. The assessment of fault detection capability is facilitated by the PL mutation framework as presented in Chapter 3. We do expect the FDC to vary by altering the following conditions:

a) Sampling as much configurations as possible.

b) Sampling large products by means of activated features.

c) Sampling diverse products.

The reasoning behind this is the following: Testing many products decreases the chance to miss some fault which is specific for a particular combination of features. A selection of large products for testing should expose faults when some or many features combined do not interact as planned. Finally, research indicates that testing diverse products is beneficial to FDC than rather testing similar products [HPP+13].

The remainder of the contribution is structured as follows: In Section 5.1 we formally define the concept of reusable test cases as introduced in Section 4.1.2 in the previous chapter. Section 5.2 presents sampling methods and Section 5.3 shows the experimental setup and initial results achieved so far. Section 5.4 covers related work and finally Section 5.5 concludes this contribution.

5.1. Reusable Test Cases

Testing a PL faces two major challenges: first, the behavioral test goals must be sufficiently covered and secondly, a meaningful subset of products should be sampled for testing. MBT of PL systems allows the application of behavioral coverage criteria as well as the usage of structural coverage criteria like t-wise coverage of features. As presented in Chapter 4, we distinguish testing processes for PL systems into *application-centered* (AC) and *domain-centered* (DC) testing processes.

In the first process, a set of configurations is selected by a structural criterion for the purpose of testing, then corresponding products are resolved from the domain model and tests are designed from the individual product models. In contrast, in the latter process tests are designed from the domain model in the first place. Hence products in a PL share commonalities, PL test cases are not necessarily limited to be assigned to a single product anymore. Instead, a test case is reusable for a set of products.

This is achieved by keeping track of the features that must be selected or deselected for each test case. However, for some features such a decision is unnecessary, if a particular test case is agnostic to said feature. In this case, we mark the feature for this test as *undecided*. Therefore, we introduce *incomplete* product configurations, which extend the concept of product configurations by a third value to be assigned to a feature, here X:

$$ic = F \to \mathbb{B} \cup \{X\}$$

Now, a test designer can create an incomplete product configuration and store it with the test case.

After test creation, product configurations are sampled from the incomplete configurations by a coverage criterion. We can sample a product configuration from an incomplete product configuration by making decisions for undecided features whether to select or deselect them. This must be done for every undecided feature in every incomplete configuration, until no feature is assigned undecided anymore. From the resulting product configurations, products can be resolved and finally the tests can be executed against their associated product. In the next section, we present coverage criteria for sampling configurations from such reusable test cases.

5.2. Sampling Configurations from Reusable Test Cases

A main challenge in DC testing, is to sample product configurations from the test cases' incomplete configurations such that every test case can be executed at least once. The product configurations are sampled with the target to maximize the likelihood of detecting faults in the PL during testing, while keeping the test effort reasonably low. Such sampling of product configurations is facilitated by coverage criteria. In the following, we present novel coverage criteria for sampling product configurations so that each test case can be executed once.

5.2.1. General Sampling

Due to the nature of feature models being representable as propositional formulas, the problem of sampling configurations can be viewed as boolean satisfiability problem. Hence we search for an optimal solution to a coverage criterion, we present the individual coverage criteria as constraint problems. As a first step, we model the problem of sampling a product configuration from an incomplete configuration. On this basis, we define mini-/maximization criteria to sample large, small, few, many, and diverse variants from a set of incomplete configurations.

Problem 1 Complete a given incomplete configuration.

Solution The first step is to declare variables for each feature in F and their domains. The domain varies depending on the feature's assignment:

- $f = true$ then the corresponding variable's domain is $\{1\}$
- $f = undecided$ then the corresponding variable's domain is $\{0, 1\}$
- $f = false$ then the corresponding variable's domain is $\{0\}$

Finally, we define the propositional formula of the feature model as constraints for the variables. A constraint solver is now able to make assignments to undecided features and check its solution for validity against the propositional formula.

The solution of Problem 1 can easily be extended to sample product configurations for all test cases in a test suite.

Problem 2 Complete all incomplete configurations in a test suite with m test cases.

Solution The method to solve problem 1, can be repeated individually for every incomplete configuration in the given test suite. Eventually, all incomplete configurations of the test suite are complete.

5.2.2. Targeted Sampling

In the following, we define coverage criteria for sampling product configurations from test cases. We denote the coverage criteria as optimization problems, so that an optimization engine can automatically find optimal solutions for a given sampling target.

Problem 3 Optimize the set of m-test cases for constraints. In particular:

a) Few/many configurations not exceeding m,

b) Small/large configurations by means of selected features,

c) Diverse configurations,

d) Combinations thereof

Solutions

a) Optimizing the Amount of Distinct Configurations. The aim is to select either few or many products to execute every test case in the given test suite at least once. The optimization problem here is to achieve a minimal or maximal number of distinct product configurations. We model the constraint problem as follows: A product configuration can be interpreted as a binary number b, when we interpret *selected* features as binary value "1" and *deselected* features as "0" respectively. Hence a product configuration pc_n with features $F_n : f_1, \dots, f_k$ is interpreted as the number:

$$b_n = (f_1 f_2 \dots f_{k-1} f_k)_2$$

For a test suite with m test cases, we derive b_n for every product configuration pc_n, where $1 \leq n \leq m$. We collect all b_i in the set Z:

$$Z = \{b_1, b_2, \dots, b_{m-1}, b_m\}$$

For receiving a minimal set of concrete configurations we have to minimize the cardinality of Z. Vice versa, we maximize the cardinality of Z, if we want to maximize the number of configurations for testing:

$$\max / \min cost_a = |Z|$$

In terms of optimization, we refer to the cardinality of Z as costs, here $cost_a$. The expected costs for the criteria of maximizing or minimizing the amount of configurations are in the range of 1 to the number of test cases m. The selection of the same product for all test cases results in costs of 1. The upper limit is m, since we require each test case to be assigned to only one product.

b) Optimizing the Size of all Configurations. We define the size of a configuration as the sum of all *selected* features. For constraint solving, we interpret *selected* as numerical value "1" and *deselected* as "0" respectively. Therefore, we can define the size of a product configuration pc_n as follows:

$$s_n = \sum_{i=1}^{k} f_i$$

When we accumulate sizes of all product configurations, we can optimize towards either a minimal or maximal overall size:

$$\max / \min cost_b = \textstyle\sum_{n=1}^{m} s_n$$

where maximization achieves large product configurations and minimization small product configurations. The costs of the smallest solution is $2 \times m$ where the root feature and only one other feature is enabled (hence 2) and multiply these by the amount of test cases m. The highest cost for solution is $k \times m$, where every feature k is selected in every test case m, which is the result of assigning a single product with all features activated to all test cases.

c) Optimizing the Diversity of Configurations We define diversity over a set of m test cases and k features. First we establish a relation between a single feature i over all configurations. The goal is to have each feature as often selected as deselected, hence we gain most different assignments.

We achieve this by calculating the diversity d_i of each feature $f_{n,i}$, where $1 \leq n \leq m$ and $1 \leq i \leq k$:

$$d_i = \textstyle\sum_{n=1}^{m} f_{n,i}$$

Next, we calculate the deviation from optimal diversity, which is $m/2$, because we want a feature to be equally often selected and deselected over all n configurations. Subsequently, the deviation of a feature f_i from its optimal diversity is calculated by $|d_i - (m/2)|$. Finally, we achieve maximal diversity by minimizing the sum of all deviations:

$$\min cost_c = \textstyle\sum_{i=1}^{k} |d_i - (m/2)|$$

The minimal costs for a solution to this problem is 0 with product configurations being maximally diversified. The highest cost are $(m/2) \times k$, where the same configuration is sampled for every test case.

We note that this approach does not maximize the amount of sampled product configurations, but their diversity. Inherently, this approach leads to solutions with fewer unique product configurations, if the calculated diversity is higher than for another solutions with more product configurations and less diversity. An approach to increase the amount of product configurations is the combination of the two criteria diversity and maximization of the amount of product configurations.

d) Combinations In general, all combinations of the previously defined constraints are valid with the exception of:

– few with many product configurations,

– small with large product configurations.

Any other combination is valid, e.g. many with large and diverse configurations. For making a preference towards one or more criteria, weights can be added to the costs.

Of course, costs cannot be summed up directly if the optimization targets are opposing, e.g. if large and diverse should be combined, the targets are minimization and maximization. In this case, a decision for an overall optimization target must be made (min *or* max) and the costs of the criterion not fitting that target must be inverted. Costs are inverted by subtracting the solution's costs from the expected maximal costs. The result of this subtraction are the inverted costs.

5.3. Example and Evaluation

In this section, we assess the fault detection capability of an example test suite in respect to sampled configurations. First, we introduce the example and setup, then we present the results.

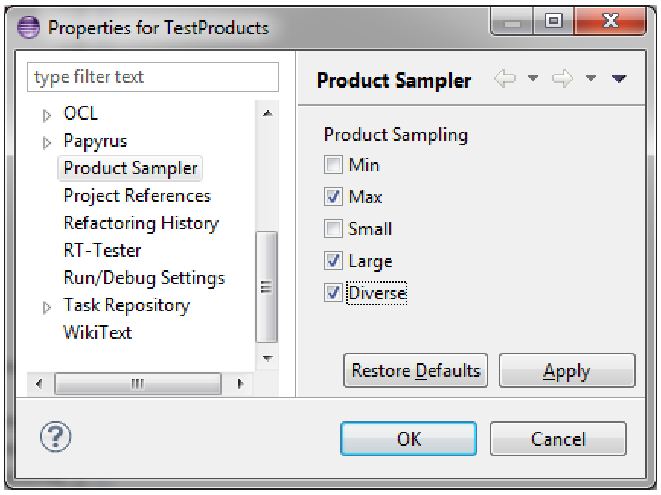
5.3.1. Tool Chain SPLTestbench

For the purpose of evaluating the here proposed methods, we extended SPLTestbench to support coverage-driven product sampling from incomplete product configurations. We implemented the coverage criteria as presented in this contribution by constraint programming. As constraint programming environment JaCoP is employed [KS13]. The sampling process is supported by Eclipse plug-ins to configure sampling and start the sampling process as depicted in Figure 5.1.

The whole process domain model to sampling of product configurations is depicted in Figure 5.2. This includes the test generation from a domain model for given coverage criteria. As discussed in Chapter 4, the result of the test generation process is a suite of reusable test cases. From these test cases, the feature model, and a given set of sampling criteria, product configurations will be sampled. As a result of the sampling, product configurations are mapped to test cases. Then test execution can take place.

5.3.2. Setup

In the previous Chapter 4, we have already designed test cases for the three examples eShop, Ticket Machine, and Alarm System with AC as well as DC test approaches. Here, we reuse these test cases generated with the step-by-step method to measure the effects of sampling for different criteria on the test case's FDC and compare them to include/exclude-all-features and pair-wise combinatorial testing from AC


Properties for TestProducts

type filter text
- OCL
- Papyrus
- Product Sampler
- Project References
- Refactoring History
- RT-Tester
- Run/Debug Settings
- Task Repository
- WikiText

Product Sampler

Product Sampling
- [ ] Min
- [✓] Max
- [ ] Small
- [✓] Large
- [✓] Diverse

Figure 5.1.: Project options in SPLTestbench for sampling products.

test design. However, the Ticket Machine example is not assessed here, since its variance is too low and the base model is too small for providing credible results.

For assessing the FDC, we employ the PL mutation framework presented in Chapter 3. For the current experiment, we apply both of the supported types of mutation operators: behavioral operators, which mutate the state machine model, and variability operators, which mutate the feature mapping model. However, we do not apply the following operators which in most cases add behavior: delete mapping, delete mapped element, and copy trigger. The reason for this is the test cases' inefficiency to detect additional behavior, since we perform conformance-testing when we apply all-transitions coverage. Still, we apply the excluded operators in Chapter 6 when testing for enabled features.

5.3.3. Results

We performed mutation analysis for all examples and their test suites with different sampling criteria and their combinations. Since the test suite stays the same for all samplings, this procedure assesses the impact of the different sampling criteria on the test suite's FDC. For a first experiment, we sampled configurations for all sampling criteria in isolation and in combination as shown Table 5.1. To reduce the amount of samplings, we chose four representative combinations: Few+Small+Div, Few+Large+Div, Many+Small+Div, and Many+Large+Div. Furthermore, we see the configurations sampled for AC test design in the last two rows. There, we show the configurations for include/exclude-all-features and two-pair combinatorial sampling [LSKL12, PSK+10].

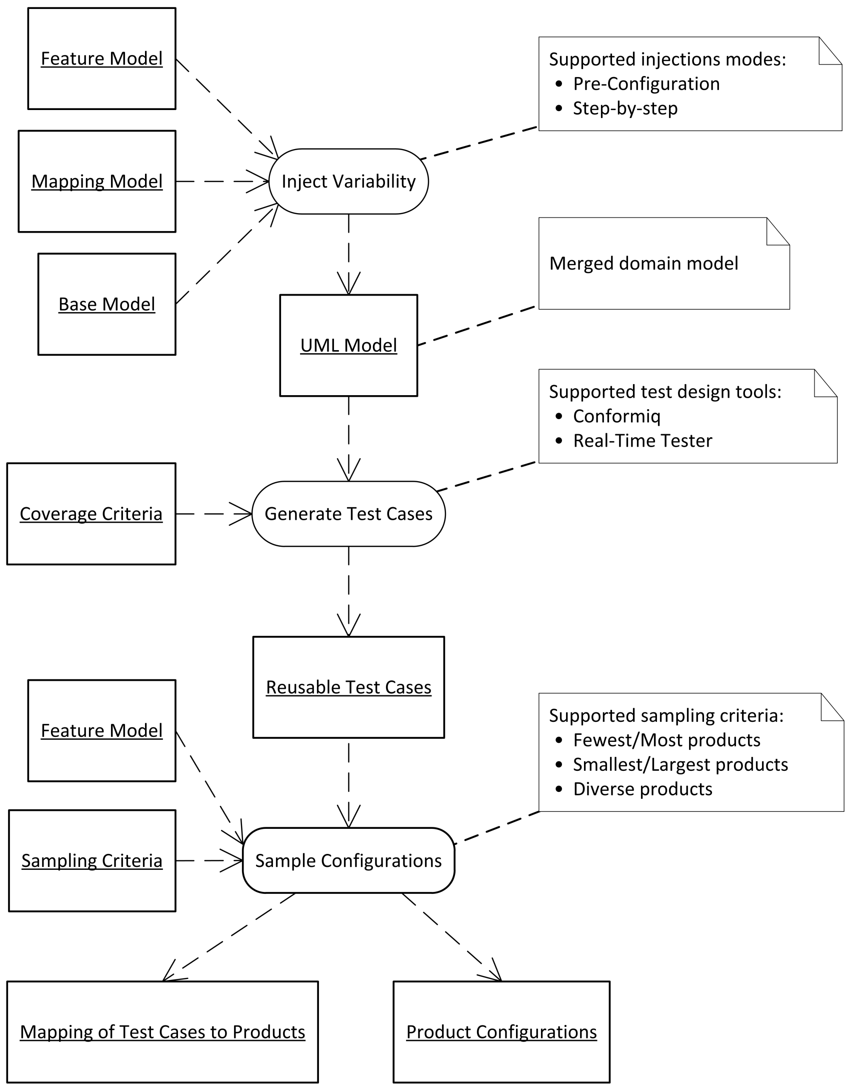


Figure 5.2.: Sampling process in SPLTestbench.

Additionally, we present necessary test executions for achieving selected sampling criteria in Figure 5.2. As discussed earlier, every test case for a given DC sampling is executed only once in this evaluation. Hence we reuse the same test suite for different samplings, the overall amount of test executions is the same for any DC sampling regardless of being an isolated or a combined one. In contrast, for AC test design we generated the test cases individually for each variant. From this, we gain more test cases and subsequently more test executions. Due to the setup, AC test cases contain redundancy, since the common core and repeatedly activated

Table 5.1.: Distinct configurations sampled.

Isolated DC samplings	eShop	Alarm System
Few	1	3
Many	13	12
Small	4	7
Large	1	3
Div	7	8
Combined DC samplings	**eShop**	**Alarm System**
Few+Small+Div	4	7
Few+Large+Div	4	7
Many+Small+Div	10	11
Many+Large+Div	10	11
AC samplings	**eShop**	**Alarm System**
Include/exclude-all	2	3
Two-pair	7	9

Table 5.2.: Test executions

	eShop	Alarm System
Any DC sampling	13	9
Include/exclude-all	20	19
Two-pair	71	54

features are tested over and over again for each tested variant.

Next, we proceeded with the results of our mutation analysis. The assessment process is as follows: During sampling, we received an assignment of product configurations to test cases. The mutation system generates mutants and derives product code from the mutated domain model. Then it executes the test cases for every mutated product according to the assignment of the original products. We present the mutation scores in two parts. First, we assess the sampling criterion's effect on the FDC for base models. The results of this first assessment are presented in Table 5.3. In the second part, we assess the FDC for mapping models and present the results in Table 5.4.

Table 5.3.: Mutation scores for detected faults in base models in %.

Isolated DC samplings	eShop	Alarm System
Few	77.6	66.3
Many	76.5	65.0
Small	76.5	65.0
Large	77.6	66.3
Div	77.6	65.0
Combined DC samplings	**eShop**	**Alarm System**
Few+Small+Div	76.5	65.0
Few+Large+Div	77.5	66.3
Many+Small+Div	76.5	65.0
Many+Large+Div	77.5	65.0
AC samplings	**eShop**	**Alarm System**
Include/exclude-all	76.5	57.5
Two-pair	77.5	71.3

Table 5.4.: Mutation scores for detected faults in mapping models in %.

Isolated DC samplings	eShop	Alarm System
Few	33.3	54.2
Many	91.7	66.7
Small	91.7	66.7
Large	33.3	43.5
Div	66.7	66.7
Combined DC samplings	**eShop**	**Alarm System**
Few+Small+Div	91.7	70.8
Few+Large+Div	66.7	62.5
Many+Small+Div	91.7	70.8
Many+Large+Div	66.7	70.8
AC samplings	**eShop**	**Alarm System**
Include/exclude-all	66.7	54.2
Two-pair	100.0	79.2

5.3.4. Discussion

The FDC of the test suite varies with the applied sampling criterion. However, the mutation scores for the base models exposes only marginal variance. This is due to the DC test design approach where all-transitions as test selection is applied to the base model in the first place and product sampling afterwards. Hence we do not further discuss FDC of test cases towards faults in the base model.

More variance is found in the FDC of the test cases for errors in the mapping models: the highest FDC of isolated criteria are achieved by maximized amount of products (Many) and small products (Small). However, we expected larger products to contain more components and thus be more likely to expose faulty behavior due to more possible interactions. Instead, large products led to testing too few products for having a positive impact on fault detection. Though Many and Small have the same FDC in this example, Small is more efficient when we evaluate the test effort based on the amount of tested products. Small yields always less products for testing than Many (cf. Table 5.1).

For combined sampling criteria the variance of the mutation scores declines. The highest scores in the group of combined criteria are achieved by the following two combinations: maximized amount with small and diverse products (Max+Small+Div) and by minimized amount with small and diverse products (Min+Small+Div). In this case, fault detection efficiency is higher for Min+Small+Div than for Max+Small+Div, since less products are sampled for testing. In general, combined criteria scored equal or lower to the top scoring isolated criteria, but never better.

In comparison to the AC testing methods, DC lies in between the two presented AC criteria. DC testing achieves at least include/exclude all features in all cases, while at the same time it needs less test executions. In comparison to two-pair combinatorial testing, DC testing is only as good as in one case and otherwise 5 to 8.4 points behind. This advantage over DC testing comes with a price: two-pair testing needs at least five times more test executions than any DC test in this evaluation.

Summing up, application-centered and domain-centered testing differs only if errors in the variability model are of concern. Within different DC methods, sampling few, small and diverse products seems to be most advantageous.

5.4. Related Work

Sampling product configurations for testing is an ongoing challenge. Most work is focused on structural coverage criteria for feature models and hence is agnostic to the interactions in behavioral models [PSK+10, OZML11]. Still, the test effort is high, since feature interactions are selected for testing where no behavioral interaction is present.

Lochau et al. present incremental test design methods to subsequently test every specified behavior [LSKL12]. Here, configurations are sampled as needed to achieve the next test goal. The result is a set of test cases where each is limited to a single product configuration. In contrast, the here presented coverage criteria for sampling configurations rely on reusable test cases.

Similar to the notion of incremental test design Beohar et al. propose spinal test suites [BM14]. A spinal test suite allows one to test the common features of a PL once and for all, and subsequently, only focus on the specific features when moving from one product configuration to another. This is different from the notion of reusable test cases, where there is no such thing as progressing through product configurations.

Baller et al. present optimization criteria and a supplementing formal framework for optimizing product line tests in [BLLS14]. In general, their approach is based on product-centered test design, where redundant test cases were designed. By applying additional efforts for optimizing the test suite, they eliminate some redundant test cases and subsequently reduce efforts for test execution.

Under constrained budgets for testing, it is useful to prioritize the products selected for testing. Al-Hajjaji et al. present such an approach in [AHTM^{+}14] that can be applied after variants are sampled for testing.

5.5. Conclusion

In this contribution, we presented five novel coverage criteria for sampling product configurations from a test suite consisting of reusable test cases. This is the first assessment of a single test suite with varying product configurations so that every test case can be executed exactly once. We assessed the coverage criteria in isolation and additionally in combination with each other. Our experiment was conducted on three PL examples. The FDC of the test suites was assessed by mutation analysis. Faults were injected into the behavioral models as well as into variability models of the examples.

We found that testing many products (Many) or rather small products by means of enabled features (Small) increases the test's FDC in our case. This is particularly true for faults located in the variability model. Fault detection for faults in the behavioral model remained almost equal over all sampling criteria. We also assessed combinations of the defined sampling criteria, which increases the test suites' FDC further.

In future, further experiments based on larger examples and industrial case studies are scheduled. This will provide more confidence on the current findings. Furthermore, the current sampling approach can be extended to allow multiple test executions. The reasoning behind this is the following: since all sampled configurations are built anyways, the effort for executing all compatible test cases, rather than only the assigned products, might not be much higher after all. The efficiency of this approach is dependent on the costs for building products in respect to additional effort for test executions and the setup/tear down phases. Also, this extension might close the gap between DC and two-pair's FDC.

Chapter 6.

Testing Product Boundaries

So far, the contributions of this thesis focused on covering the selected behavior of a product, i.e., they check if everything that should be in the product is implemented. In this contribution, we focus on covering the deselected behavior by checking whether everything that should not be in the product variant is actually not implemented. Common test design approaches cannot be used for this, because they are focused on cutting away all deselected behavior for a variant, and thus the model for the variant does not contain deselected behavior, anymore. To overcome this issue, we introduce model transformations that create new model elements describing the non-existence of deselected behavior.

The contribution is structured as follows. In Section 6.1 we describe the approach. The implementation and experiments are described in Section 6.2. Section 6.3 contains an analysis of the related work. In Section 6.4, we conclude and discuss the presented approach including threats to validity.

6.1. Testing Boundaries of Products

MBT is focused on creating test cases from behavioral models. In MBT, test cases are typically designed for performing positive testing by means of checking the SUT for conformance to the test models. For testing a PL, the information about features and parts being explicitly excluded, however, is valuable too. A test designer can make use of this information by creating tests that actively try to invoke excluded behavior. We think of this as an attempt of breaching the boundaries of a product under test (PUT), where the boundary is predefined by the PUT's configuration. A boundary is overcome if an excluded behavior is invoked and executed as specified in the domain model.

In this contribution, we refer to the Ticket Machine as running example. In Figure 6.1, we depict an excerpt of the Ticket Machine's domain model, consisting of a feature model and a UML state machine as base model. In this excerpt, the system waits for coins or bills to be inserted until the costs for the selected tickets are covered. The dotted arrow maps the feature *Bills* to the transition t_6 in the state machine: If the Ticket Machine's configuration includes the feature *Bills*, then the

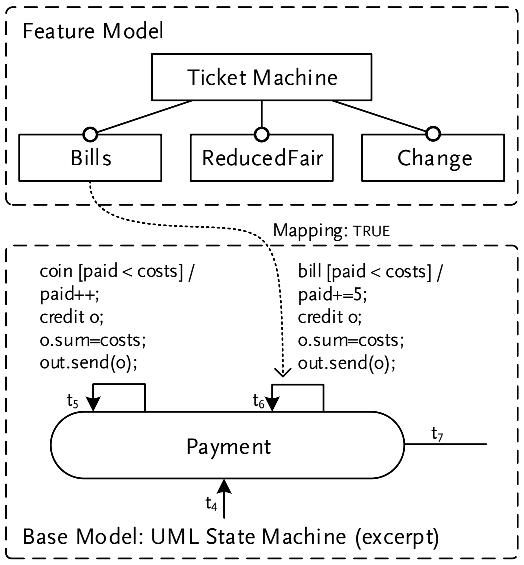

Feature Model
Ticket Machine
Bills
ReducedFair
Change
Mapping: TRUE
coin [paid < costs] / paid++; credit o; o.sum=costs; out.send(o);
bill [paid < costs] / paid+=5; credit o; o.sum=costs; out.send(o);
t5
t6
t7
t4
Payment
Base Model: UML State Machine (excerpt)

Figure 6.1.: Excerpt of the product line model for the Ticket Machine.

mapped transition t_6 must be present in the corresponding product. If the feature is not selected, this transition is not part of the corresponding product and hence leaving the customer's only payment option to be coins as denoted in transition t_5.

6.1.1. Boundary Transitions

Inside the PUT's boundaries is the PL's core and all included features declared by the configuration. Outside its boundaries lie the excluded features. Figure 6.2 depicts an excerpt of a Ticket Machine product, in which the feature *Bills* is deactivated. Here, the state "Payment" and the transitions t_4, t_5, t_7 lie within the boundaries of the product. Transition t_6 as shown in the excerpt of Figure 6.1 is not part of this product. We overcome this boundary, if we make the product process a bill in this state as defined in the domain model.

More formally speaking, we define a product's boundary by *boundary transitions* over UML state machines. We define a boundary transition bt, where S is the set of states and T is the set of transitions in a base model and $t(s, s')$ is a transition from state s to s' as:

$$\begin{aligned} bt(s, s') \in T \Leftrightarrow s, s' \in S \land s \in \mathit{productmodel} \\ \land bt \notin \mathit{productmodel} \end{aligned}$$

Hence, a boundary transition is not part of the particular product. We call a product

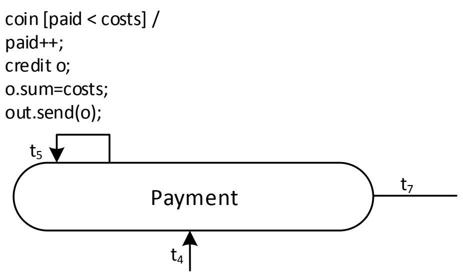


Figure 6.2.: Ticket Machine product w/o feature *Bills*.

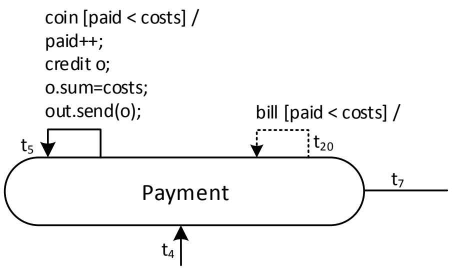


Figure 6.3.: Same product with complementary transition added.

to have an *open boundary*, if behavior from an excluded feature can be invoked at some point of the PUT's execution.

In general, it is possible to detect open boundaries by stimulating the PUT with unexpected events in every state. This resembles sneak-path-analysis and is costly [HTBH12]. Here, we propose a method to reduce test effort by stimulating the PUT with unexpected events only if its active state has at least one boundary transition. In particular, we stimulate the PUT with only those events that could possibly trigger one of its boundary transitions attached to this state.

6.1.2. Turning Open Boundaries into Test Goals

We choose transition-based coverage criteria for selecting test goals. Our approach comprises introducing a transition for each boundary transition to which we refer to as *complementary transition*. The intention of this is to create transitions specifying that the PUT should stay in its current state and with events that are not expected to trigger product behavior. Hence, for every boundary transition, we add a complementary transition with its source state as target and source. For the presented Ticket Machine product without feature *Bills*, Figure 6.3 shows the same excerpt of the product as in Figure 6.2, but with the additional complementary transition t_{20}, which complements boundary transition t_6 of this product. The complementary transition must have no effect, since in the state “Payment” no reaction is expected for any product that does not include feature *Bills*. However, we should not add a complementary transition, if there is an explicitly specified behavior for processing the signal event when feature *Bills* is excluded as in state “Selection”.

So far, we defined boundary transitions for a given product and outlined how to add complementary transitions. For DC test design we must raise these concepts to the domain engineering-level, in order to set complementary transitions as test goals. Particularly, we define a transformation for adding complementary transitions to

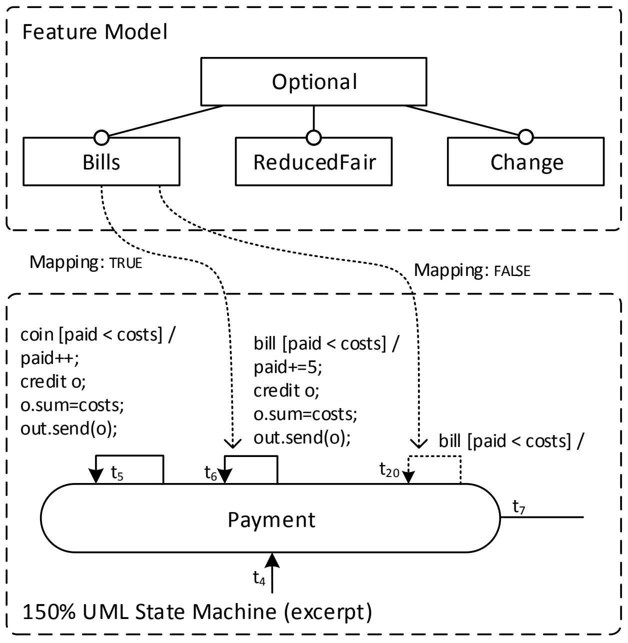


Figure 6.4.: PL Model Example: Ticket Machine with Complementary Transition.

the domain model whenever there is a boundary transition of any product available. This enables DC test design methods to consider complementary transition as test goals during test design. Also, AC test design methods benefit from this approach, since the complementary transitions are propagated during the resolution process to the application-level.

In Figure 6.4, we depict the desired outcome of the transformation: we added a complementary transition t_{20} to state *Payment* for transition t_6, which is a boundary transition for any product not including the feature *Bills*. Hence, the complementary transition is mapped to feature *Bills* with the mapping's flag set to *false*, denoting the transition is only to be included when the feature *Bills* is deselected.

We present the pseudo code to achieve the result shown in Figure 6.4 in Algorithm 1. Let $SM(S,T)$ be a state machine, where S is the set of states and T the set of transitions. For each transition $t \in T$ we define:

- source(t) as the source state of t,
- target(t) as the target state of t,
- triggers(t) as the triggers of t,

- triggers $*(t)$ as the triggers from all transitions leaving target(t), if triggers(t) is empty, and triggers(t) otherwise. Since this is a recursive definition, triggers $*(t)$ must stop once all $t \in T$ are traversed.

- features(e) as the set of feature selections mapped to a UML element $e \in SM$. A feature selection states whether a feature must be selected or deselected to include e.

- $concurrentGuards(t)$ as a conjunction of guard conditions. The conditions are collected from transitions that can be concurrently enabled with t.

First a set of transitions for storing complementary transitions during this procedure is initialized. Then for all transitions of the state machine the following actions are performed: the algorithm checks in lines 4–7 if current transition b is a boundary transition for some product. This is achieved by checking whether b has different feature mapping selections than its source state. The selections from b, which are not shared by its source state are stored in *difference*. When *difference* is not empty, b is a boundary transition and creation of a complementary transition begins. Otherwise, the for-loop continues with the next b.

From line 8 to 12, the complementary transition c is added to C and is initialized with source(b) as target *and* source state, and triggers $*(b)$ as triggers. The complementary transition's guard is built from the original boundary transition's guard and, to prevent non-deterministic behavior, conjoined with the negated guard conditions of concurrently enabled transitions. Lastly in this if-block, c is mapped to the negated difference of feature selections unified with the selections of b's source state, so c is included in every product when b's source state is, but b is not. Line 14 concludes the procedure by adding the set of complementary transitions C to the state machine's set of transitions T.

The outcome of this procedure when applied to the Ticket Machine's domain model is depicted in Figure 6.5. We denote the mappings from the feature model by feature formulas in the transition's guards analog to Featured Transition System (FTS) introduced by Classen [CHSL11]. We use the following acronyms: B for *Bills*, C for *Change*, and R for *ReducedFare*. The complementary transitions added by our transformation procedure are denoted by dotted arcs (transitions t_{19}–t_{22}). Beginning from the initial state, we find the first state with at least one boundary transition to be "Selection". The boundary transition here is t_3, which is enabled when the feature *ReducedFare* is part of a product. Hence, t_{19} is added to the state machine for serving as an additional test goal to any product not including *ReducedFare*. To achieve all-transition coverage, a test case must include sending the signal event "reducedTicket" when the feature *ReducedFare* is disabled while the state machine is supposed to stay in state "Selection". Analog to this, transition t_{20} is added for boundary transition t_6 in state "Payment".

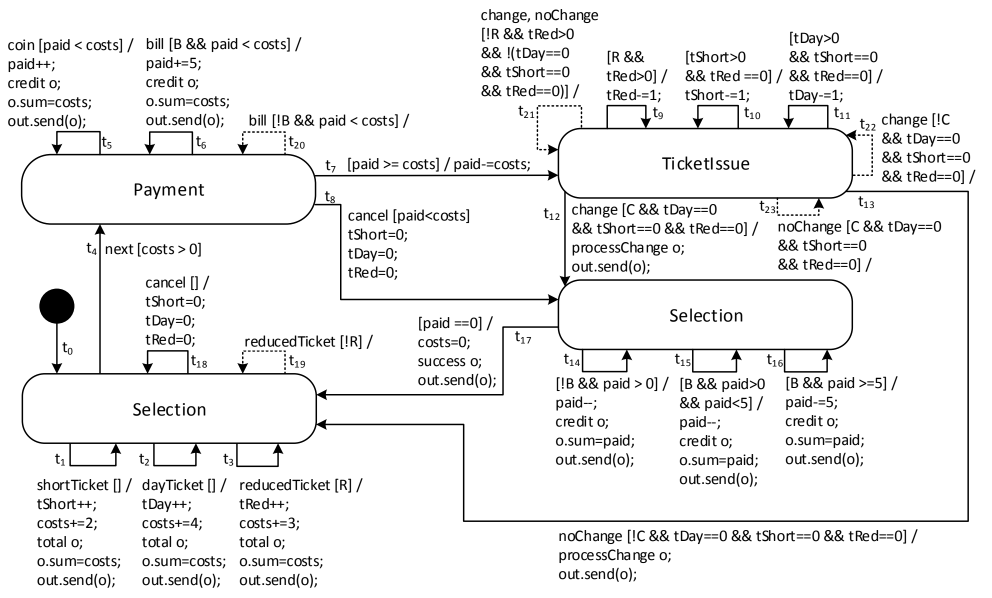

coin [paid < costs] / paid++; credit o; o.sum=costs; out.send(o);
t5
bill [B && paid < costs] / paid+=5; credit o; o.sum=costs; out.send(o);
t6
bill [!B && paid < costs] /
t20
Payment
t7 [paid >= costs] / paid-=costs;
t8
cancel [paid<costs] tShort=0; tDay=0; tRed=0;
change, noChange [!R && tRed>0 && !(tDay==0 && tShort==0 && tRed==0)] /
t21
[R && tRed>0] / tRed-=1;
t9
[tShort>0 && tRed ==0] / tShort-=1;
t10
[tDay>0 && tShort==0 && tRed==0] / tDay-=1;
t11
t22
change [!C && tDay==0 && tShort==0 && tRed==0] /
TicketIssue
t12
change [C && tDay==0 && tShort==0 && tRed==0] / processChange o; out.send(o);
t23
noChange [C && tDay==0 && tShort==0 && tRed==0] /
t13
Selection
[paid ==0] / costs=0; success o; out.send(o);
t17
t14
t15
t16
[!B && paid > 0] / paid--; credit o; o.sum=paid; out.send(o);
[B && paid>0 && paid<5] / paid--; credit o; o.sum=paid; out.send(o);
[B && paid >=5] / paid-=5; credit o; o.sum=paid; out.send(o);
t4 next [costs > 0]
cancel [] / tShort=0; tDay=0; tRed=0;
t0
t18
reducedTicket [!R] /
t19
Selection
t1
t2
t3
shortTicket [] / tShort++; costs+=2; total o; o.sum=costs; out.send(o);
dayTicket [] / tDay++; costs+=4; total o; o.sum=costs; out.send(o);
reducedTicket [R] / tRed++; costs+=3; total o; o.sum=costs; out.send(o);
noChange [!C && tDay==0 && tShort==0 && tRed==0] / processChange o; out.send(o);

Figure 6.5.: Domain Model: Ticket Machine with added feature formulas and complementary transitions.

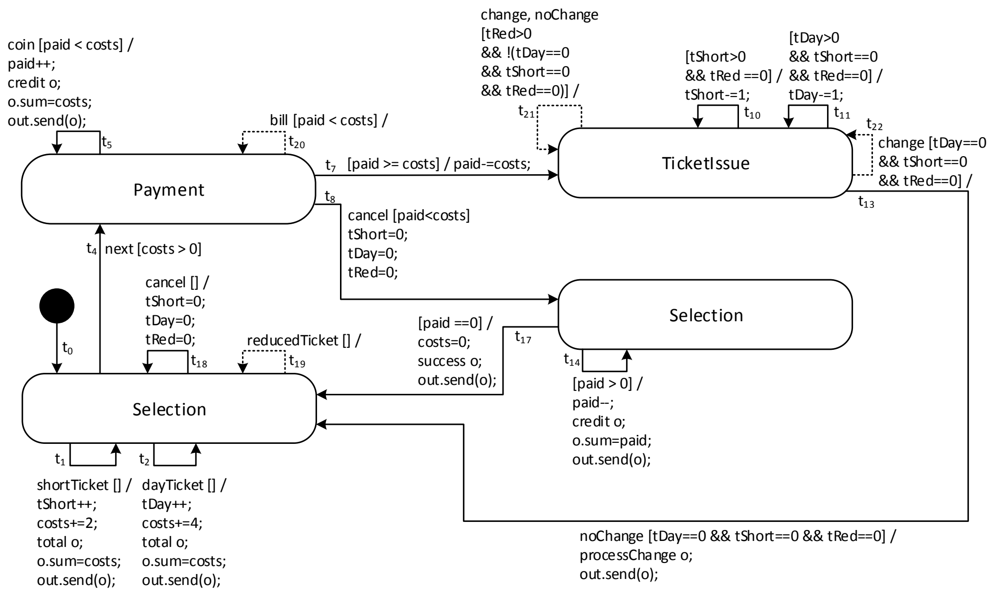

coin [paid < costs] / paid++; credit o; o.sum=costs; out.send(o);
t5
bill [paid < costs] /
t20
Payment
t7 [paid >= costs] / paid-=costs;
t8
cancel [paid<costs] tShort=0; tDay=0; tRed=0;
change, noChange [tRed>0 && !(tDay==0 && tShort==0 && tRed==0)] /
t21
[tShort>0 && tRed ==0] / tShort-=1;
t10
[tDay>0 && tShort==0 && tRed==0] / tDay-=1;
t11
t22
change [tDay==0 && tShort==0 && tRed==0] /
TicketIssue
t13
Selection
[paid ==0] / costs=0; success o; out.send(o);
t17
t14
[paid > 0] / paid--; credit o; o.sum=paid; out.send(o);
t4 next [costs > 0]
cancel [] / tShort=0; tDay=0; tRed=0;
t0
t18
reducedTicket [] /
t19
Selection
t1
t2
shortTicket [] / tShort++; costs+=2; total o; o.sum=costs; out.send(o);
dayTicket [] / tDay++; costs+=4; total o; o.sum=costs; out.send(o);
noChange [tDay==0 && tShort==0 && tRed==0] / processChange o; out.send(o);

Figure 6.6.: Product: State machine model of a Ticket Machine without *Bills*, *Change*, and *ReducedFare*.

Algorithm 1 Adds Complementary Transitions to a Region

```
1: procedure addComplementaryTransitions
2:     $C \leftarrow \emptyset$
3:     for all $b \in T$ do
4:         incoming $\leftarrow \bigcup$ features$(s \in S | s =$ source$(b))$
5:         difference $\leftarrow$ features$(b) -$ incoming
6:         if difference $\neq \emptyset$ then
7:             $C \leftarrow C \cup c$
8:             source$(c) \leftarrow$ source$(b)$
9:             target$(c) \leftarrow$ source$(b)$
10:            guard$(c) \leftarrow$ guard$(b) \wedge \neg(\mathit{concurrentGuards}(b))$
11:            triggers$(c) \leftarrow$ triggers $*(b)$
12:            features$(c) \leftarrow \mathit{incoming} \wedge \neg$ difference
13:    $T \leftarrow T \cup C$
```

State “TicketIssue” shows three boundary transitions t_9, t_{12}, and t_{13}. Transition t_9 has no trigger, hence its target state must be checked for outgoing transitions with triggers. The transformation’s check for further transitions in t_9’s target state delivers t_9 to t_{13}. Since t_9 is currently under investigation it will not be checked for triggers again. Transitions t_{10} and t_{11} are untriggered and thus their target state must be evaluated for further triggers. Since their target state is also “TicketIssue”, for which this check is currently performed, there are no further checks at this point. For each of the triggered transitions t_{12} and t_{13} one self-loop must be created. Each of them includes the copied trigger, negated feature constrained for the currently investigated feature *ReducedFare* and its guard constraint, the copied feature mapping (C) from the transition at the target state, and its negated guard constraint:

$$\begin{aligned} t_{12} : \mathit{change}\big[\neg R \wedge \mathit{tRed} > 0 \wedge C \\ \wedge\neg\big(\mathit{tDay} == 0 \wedge \mathit{tShort} == 0 \wedge \mathit{tRed} == 0\big)\big]/ \end{aligned}$$

$$\begin{aligned} t_{13} : \mathit{noChange}\big[\neg R \wedge \mathit{tRed} > 0 \wedge \neg C \\ \wedge\neg\big(\mathit{tDay} == 0 \wedge \mathit{tShort} == 0 \wedge \mathit{tRed} == 0\big)\big]/ \end{aligned}$$

We combine both transitions to create t_{21} with both triggers and reduced guards, where constraint C and $\neg C$ cancel each other out. Unfortunately, t_{21} is unreachable, since the condition $\mathit{tRed} > 0$ never holds for any product that does not include t_3. Transitions t_{22} and t_{23} are added accordingly. Finally, no further boundary transitions exists and therefore the procedure ends here.

6.2. Evaluation

In this section, we present the evaluation of the product line's test suites, with and without the presented model transformations. We assess all tests by means of fault detection capability (FDC). First, we introduce the test setup and then, present the results.

6.2.1. Setup

We assess two test suites for each of the examples presented in Section 2.4 in Chapter 2. As a first test suite we use the test suite generated by DC test design as presented in Chapter 4. In particular, we apply the step-by-step method. For the second, we apply our transformations as presented in this contribution and generate the test cases with the step-by-step test design method. For test generation, Real-Time Tester by Verified is employed [Pel13]. For all test suites, we applied all-transition coverage for test selection.

From the test cases, SPLTestbench sampled configurations for testing few products. The assessment of the test cases' FDC was carried out by mutation analysis. Here, we applied all mutation operators for mapping models as defined in Chapter 3.

6.2.2. Results

In Table 6.1, we show the test assessment results of test suites, that were designed with the original models. In each row, we show the mutation results for all examples in the form of killed mutants/all mutants. As supposed, mutations with behavior that is not described by the test model (DME, DMP) are not detected. For the other two mutation types which alter specified behavior (IME, CFV), we receive mixed results in the range of 40% to 100%.

In contrast, Table 6.2 depicts the assessment results for the test suites that were created from our transformed models. Again in each row, we show the mutation results for all examples in the form of killed mutants/all mutants. We observe increased scores for every mutation operator on any of our examples. In the last row of each table, we show the overall results for each example. Furthermore, in the last column we present the accumulated scores of every mutation operator over all examples.

6.2.3. Discussion

Our results support the recommendation of Binder [Bin99] and the conclusions drawn by Mouchawrab et al. [MBLD11] and Holt et al. [HTBH12]: Testing sneak paths (in our case as boundaries of product line variants) is an essential component of

Table 6.1.: Mutation Scores for Regular Tests

Op.	TM	eShop	AS	p.Op.
DMP	20.0	0.0	0.0	5.9
DME	12.5	0.0	0.0	2.3
IME	40.0	25.0	25.0	29.4
CFV	100.0	100.0	75.0	88.2
SWP	60.0	50.0	37.5	47.1
per Ex.	42.8	23.3	20.8	

Table 6.2.: Mutation Scores for Tests with Transformations

Op.	TM	eShop	AS	p.Op.
DMP	60.0	100.0	62.5	70.6
DME	37.5	28.6	38.1	34.9
IME	60.0	100.0	25.0	52.9
CFV	100.0	100.0	87.5	94.1
SWP	80.0	100.0	50.0	70.6
per Ex.	64.3	66.7	49.1	

state-based testing and drastically increases FDC. Furthermore the results indicate that sneak path testing is a necessary step in state-based testing due to the same observations made by Holt et al. [HTBH12]: 1) The proportion of sneak paths in the collected fault data was high (61,5 %), and 2) the presence of sneak paths is undetectable by conformance testing.

We were able to increase the amount of killed mutants by a significant amount through our model transformations but were not able to kill all mutants. Especially the mutation score for the DME operator is still below 50% of killed mutants. This is partly the result of unreachable behavior, e.g. in the case when a UML element (e.g. a transition) that was mapped to a feature (and thus is now permanently enabled) has preceding elements mapped to the same feature. In that case, the element is always enabled but only reachable if its preceding elements are present, which is only true if the feature is present. A fundamental question here is if this indicates an issue of the test design or an unrealistic mutation operator.

This leads to the consideration of the threats to validity. The first point was already mentioned: The introduced mutation operators are new and depend on a model-based PLE. Further analysis with well-known mutation operators need to be done. Concerning the validity of our examples: we are aware that the used examples are rather small. A larger case study with realistic background would be necessary to underline the advantages of our approach and also the assumed conditions like, e.g., the application of feature models.

6.3. Related Work

In recent years MBT emerged as an efficient test design method that yields a number of improvements compared to conventional test design such as higher test coverage or earlier defect detection. There are several surveys on the effectiveness of MBT in general [ZSM11, BJK+05, UL06] and MBT of SPL systems [OWES11]. In contrast to this, we combine the application of model-based software product line testing with a product line-specific sneak path analysis. To our knowledge, this combination has not been covered before.

An early evaluation of the mutation scores suggests that our generated test suites satisfying all-transitions coverage are capable of detecting many seeded faults except unspecified behavior, so-called sneak paths [Bin99]. In safety-critical systems, an unintentional sneak path may have catastrophic consequences. Sneak path testing aims at verifying the absence of sneak paths and at showing that the software under test handles them in a correct way. Several studies showed that sneak path testing improves the fault detection capabilities [BDL04, MBLD11, HTBH12]. However, the effort spent for sneak path testing is considerably high. Here, we present a novel, more efficient approach for detecting unspecified behavior in product line engineering: We define boundary transitions that stimulate the product under test with only those events that could possibly trigger a transition that would invoke excluded behavior. To our knowledge, this approach has not been applied in the context of software product line engineering before.

6.4. Conclusion & Future Work

In this contribution, we combined MBT for product lines with boundary transition analysis. We extended our previous work on DC test design with model transformations that increase the FDC of the generated test suites.

We were able to increase the mutation score in each of our three examples and for each of the proposed mutation operators significantly by using the proposed model transformation. The scores increased for the eShop by 43%, for the Ticket Machine by 24% and for the Alarm System by 29%. As for the operators the numbers increased by 63% for the DMP operator, by 33% for the DME operator, by 24% for the IME operator, and by 6% for the CFV operator (which were already very high), and for the SWP operator by 23%.

Part III.

Closure

Chapter 7.

Conclusion

This chapter concludes this thesis. Herein, we reflect on the scientific achievements contributed by this thesis and look at the achievement's impact on the scientific community. Furthermore, we discuss more general future work derived from the open leads in this thesis.

7.1. Contributions

In this thesis, we proved the Hypothesis initially stated in Section 1.2:

> Leverage test design to the domain engineering-level.

For this, we provided methods and concepts as solutions. In particular, we make the following contributions to leverage testing to the domain-level:

1. Enable automated test design on domain-level by model transformations.
2. Product sampling from reusable test cases.
3. Adapt established test coverage criteria for PL systems.
4. Enable test assessment on domain-level.

The contributions are implemented as proof of concept prototypes within the SPLTestbench.

7.2. Impact

The contributions are published as multiple scientific articles and were presented on international conferences and workshops. All articles are peer-reviewed by leading scientists and experts in the field of software testing and PLE. In the following, we list the contributing article's title, authors, venue, and where it contributes to within this thesis:

- *Model-based Product Line Testing: Sampling Configurations for Optimal Fault Detection.* Hartmut Lackner. SDL Forum 2015: Chapter 5 Test-driven Product Sampling
- *Automated Test Design for Boundaries of Product Line Variants.* Stephan Weißleder, Florian Wartenberg, and Hartmut Lackner. ICTSS 2015. Chapter 6 Testing Product Boundaries
- *Potential Errors and Test Assessment in Software Product Line Engineering.* Hartmut Lackner and Martin Schmidt. MBT 2015. Chapter 3 Assessment of Product Line Tests
- *Model-Based Test Design of Product Lines: Raising Test Design to the Product Line Level.* Hartmut Lackner, Martin Thomas, Florian Wartenberg, and Stephan Weißleder. ICST 2014. Chapter 4 Automated Test Design for Product Lines
- *Assessment of Software Product Line Tests.* Hartmut Lackner and Martin Schmidt. SPLat 2014. Chapter 3 Assessment of Product Line Tests
- *Top-Down and Bottom-Up Approach for Model-Based Testing of Product Lines.* Stephan Weißleder and Hartmut Lackner. MBT 2013. Chapter 4 Automated Test Design for Product Lines

Our concept of domain-level test design and reusable test cases attracted attention of both, the community of MBT and PLE. Notably, Aitor Arrieta et al. present a test architecture for configurable cyber-physical systems [ASE14b, ASE14a]. The authors identify test efficiency as one of the major challenges and derive from the concepts presented in this thesis a domain-level test design and product sampling process for Simulink environments. For this purpose, the authors introduce test feature models to support reusable test cases throughout the test process.

Our approaches for test design and fault detection are also recognized by Devroey et al. [DPL+15]. Their most recent vision is to have dissimilar test case selection for testing PL systems. For measuring fault detection capability they also research in the fields of mutation analysis. Since they have chosen other notations to model variability and behavior, the results in this thesis are not directly transferable.

A general confirmation of the relevance of the discussed topics in this thesis is the fact that Østein Haugen, who is a leading expert on the standardization of the Common Variability Language, is also researching on solutions for sampling products from reusable test cases [SH15]. So far, the findings presented by him and his colleagues are limited to sampling configurations for testing a minimal amount of products. In contrast, in Chapter 5 of this thesis, we presented five different sampling criteria - sampling for minimal amount of products being one of them -

and additionally measured the individual criterion's fault detection capability and discussed their implications.

7.3. Future Work

Although we found answers in this thesis for leveraging testing to the domain-level, there are still open questions and we raised new ones.

The examples in this thesis employed for carrying out the evaluations are typical for small and isolated systems. However in the industry, we can find more complex systems not only by size in states, transitions, signals, or features. The Body Comfort System case study, which is currently under investigation, could provide more confidence in the findings.

Other aspects of complexity, which we have not dealt with in this thesis but are widely spread, are distributed system or cyber-physical systems or internet of things. A large-scale study with these kinds of systems would significantly improve confidence in the feasibility and reliability of the current findings.

Furthermore, a case study of a real system that is already in use could supply known errors from test or field reports. With this knowledge the FDC of versions exposing the documented errors could be tested with domain-level tests. The aim is to measure the tests accuracy to find such known errors. Whether this kind of test assessment is independent from the employed modeling languages, is still arguable, since each language may promote different kinds of errors.

In industry, we can also observe other notations for model-based PLE for both, variability and behavior modeling. In Chapter 3, we showed that errors in different variability notations, like annotative, compositional, or transformational variability modeling effectively lead to similar faults at runtime.

Bibliography

[AHTM$^+$14] Al-Hajjaji, Mustafa; Thüm, Thomas; Meinicke, Jens; Lochau, Malte; Saake, Gunter: Similarity-based prioritization in software product-line testing. In: Gnesi, Stefania; Fantechi, Alessandro, editors, *the 18th International Software Product Line Conference*, pp. 197–206. 2014. doi:10.1145/2648511.2648532.

[ALN13] Aichernig, Bernhard K.; Lorber, Florian; Ničković, Dejan: Time for Mutants — Model-Based Mutation Testing with Timed Automata. In: Hutchison, David; Kanade, Takeo; Kittler, Josef; Kleinberg, Jon M.; Mattern, Friedemann; Mitchell, John C.; Naor, Moni; Nierstrasz, Oscar; Pandu Rangan, C.; Steffen, Bernhard; Sudan, Madhu; Terzopoulos, Demetri; Tygar, Doug; Vardi, Moshe Y.; Weikum, Gerhard; Veanes, Margus; Viganò, Luca, editors, *Tests and Proofs*, volume 7942 of *Lecture Notes in Computer Science*, pp. 20–38. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013. ISBN 978-3-642-38915-3. doi: 10.1007/978-3-642-38916-0{\textunderscore}2.

[AO08] Ammann, Paul E.; Offutt, Jeff: *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 2008. ISBN 9780521880381.

[ASE14a] Arrieta, Aitor; Sagardui, Goiuria; Etxeberria, Leire: A configurable test architecture for the automatic validation of variability-intensive cyber-physical systems. In: Mannaert, Herwig, editor, *ICSEA 2014*. Curran, Red Hook, NY, 2014. ISBN 9781634394697.

[ASE14b] Arrieta, Aitor; Sagardui, Goiuria; Etxeberria, Leire: A model-based testing methodology for the systematic validation of highly configurable cyber-physical systems. In: Kanstrén, Teemu, editor, *VALID 2014*, pp. 66–73. Curran, Red Hook, NY, 2014. ISBN 9781634394727.

[BAC04] Burke, Michael G.; Antkiewicz, Michal; Czarnecki, Krzysztof: Feature-Plugin. In: Vlissides, John M.; Schmidt, Douglas C., editors, *OOPSLA 2004*, volume v. 39, no. 10 (Oct. 2004) of *ACM SIGPLAN notices*, pp. 67–72. Association for Computing Machinery, New York, NY, 2004. ISBN 1-58113-831-8.

[Bat05] Batory, Don: Feature models, grammars, and propositional formulas. In: *Proceedings of the 9th international conference on Software Product Lines*, SPLC'05, pp. 7–20. Springer-Verlag, Berlin, Heidelberg, 2005. ISBN 3-540-28936-4. doi:10.1007/11554844{\textunderscore}3. URL http://dx.doi.org/10.1007/11554844_3.

[BBW06] Belli, Fevzi; Budnik, Christof J.; White, Lee: Event-based modelling, analysis and testing of user interactions: approach and case study. In: *Software Testing, Verification and Reliability*, volume 16(1):pp. 3–32, 2006. ISSN 0960-0833. doi:10.1002/stvr.335.

[BDG$^+$07] Baker, Paul; Dai, Zhen Ru; Grabowski, Jens; Haugen, Øystein; Schieferdecker, Ina; Williams, Clay: *Model-Driven Testing: Using the UML Testing Profile*. Springer, Berlin, 1st edition, 2007.

[BDL04] Briand, Lionel C.; Di Penta, M.; Labiche, Yvan: Assessing and improving state-based class testing: a series of experiments. In: *Software Engineering, IEEE Transactions on*, volume 30(11):pp. 770–783, 2004. doi:10.1109/TSE.2004.79.

[BG11] Bagheri, Ebrahim; Gasevic, Dragan: Assessing the maintainability of software product line feature models using structural metrics. In: *Software Quality Journal*, volume 19(3):pp. 579–612, 2011. ISSN 0963-9314. doi:10.1007/s11219-010-9127-2.

[BH08] Belli, Fevzi; Hollmann, Axel: Test generation and minimization with basic statecharts. In: Delp, Edward J.; Wong, Ping Wah, editors, *the 2008 ACM symposium*, volume vol. 5681, p. 718. SPIE and IS&T, Bellingham, Wash and Springfield, Va, 2008. ISBN 978-1-59593-753-7. doi:10.1145/1363686.1363856.

[BHP11] Belli, Fevzi; Hollmann, Axel; Padberg, Sascha: Model-Based Integration Testing with Communication Sequence Graphs. In: Zander, Justyna; Schieferdecker, Ina; Mosterman, Pieter J., editors, *Model-based testing for embedded systems*, Computational analysis, synthesis, and design of dynamic systems. CRC Press, Boca Raton, 2011. ISBN 1439818452.

[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.

[BJK$^+$05] Broy, Manfred; Jonsson, Bengt; Katoen, Joost-Pieter; Leucker, Martin; Pretschner, Alexander, editors: *Model-based testing of reactive systems:*

Advanced lectures, volume 3472 of *Lecture Notes in Computer Science*. Springer, Berlin, 2005. ISBN 978-3-540-26278-7. doi:10.1007/b137241.

[BK05] Berkenkötter, Kirsten; Kirner, Raimund: Real-Time and Hybrid Systems Testing. In: Broy, Manfred; Jonsson, Bengt; Katoen, Joost-Pieter; Leucker, Martin; Pretschner, Alexander, editors, *Model-based testing of reactive systems*, volume 3472 of *Lecture Notes in Computer Science*, pp. 355–387. Springer, Berlin, 2005. ISBN 978-3-540-26278-7. doi:10.1007/11498490{\textunderscore}16.

[BLLS14] Baller, Hauke; Lity, Sascha; Lochau, Malte; Schaefer, Ina: Multi-objective Test Suite Optimization for Incremental Product Family Testing. In: *ICST' 14: International Conference on Software Testing, Verification, and Validation*. IEEE, 2014. ISBN 9781479957910.

[BLPT05] Bouquet, Fabrice; Legeard, Bruno; Peureux, Fabien; Torreborre, Eric: Mastering Test Generation from Smart Card Software Formal Models. In: Barthe, Gilles; Burdy, Lilian; Huisman, Marieke; Lanet, Jean-Louis; Muntean, Traian, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3362 of *Lecture Notes in Computer Science*, pp. 70–85. Springer Berlin Heidelberg, 2005. ISBN 978-3-540-24287-1. doi:10.1007/978-3-540-30569-9{\textunderscore}4. URL http://dx.doi.org/10.1007/978-3-540-30569-9_4.

[BM14] Beohar, Harsh; Mousavi, Mohammad Reza: Spinal Test Suites for Software Product Lines France, 6 April 2014. In: Schlingloff, Holger; Petrenko, Alexander K., editors, *Proceedings Ninth Workshop on Model-Based Testing, MBT 2014, Grenoble, France, 6 April 2014*, volume 141 of *EPTCS*, pp. 44–55. 2014. doi:10.4204/EPTCS.141.4. URL http://dx.doi.org/10.4204/EPTCS.141.4,TitelanhanddieserDOIinCitavi-Projektbernehmen.

[BOY00] Black, Paul E.; Okun, Vadim; Yesha, Y.: Mutation operators for specifications. In: *ASE 2000 15th IEEE International Automated Software Engineering Conference*, pp. 81–88. 2000.

[CA05] Czarnecki, Krzysztof; Antkiewicz, Michal: Mapping Features to Models: A Template Approach Based on Superimposed Variants. In: Glück, Robert, editor, *Generative programming and component engineering*, volume 3676 of *Lecture Notes in Computer Science*, pp. 422–437. Springer, Berlin [u.a.], 2005. ISBN 3-540-29138-5. doi:10.1007/11561347{\textunderscore}28,. URL http://dblp.uni-trier.de/db/conf/gpce/gpce2005.html#CzarneckiA05.

[CDS06] Cohen, Myra B.; Dwyer, Matthew B.; Shi, Jiangfan: Coverage and adequacy in software product line testing. In: *ROSATEA 2006*, pp. 53–63. Association for Computing Machinery, Inc., New York, NY, 2006. ISBN 1-59593-459-6. doi:10.1145/1147249.1147257. URL http://portal.acm.org/citation.cfm?id=1147249.1147257&coll=Portal&dl=GUIDE&CFID=65839091&CFTOKEN=21681387.

[CH11] Cichos, Harald; Heinze, Thomas S.: Efficient Reduction of Model-Based Generated Test Suites Through Test Case Pair Prioritization. In: *Proceedings of the 7th International Workshop on Model-Driven Engineering, Verification and Validation (MoDeVVa 10)*, pp. 37–42. IEEE Computer Society Press, Los Alamitos, 2011.

[CHSL11] Classen, Andreas; Heymans, Patrick; Schobbens, Pierre-Yves; Legay, Axel: Symbolic Model Checking of Software Product Lines. In: *33rd International Conference on Software Engineering, ICSE 2011, May 21-28, 2011, Waikiki, Honolulu, Hawaii, Proceedings*, pp. 321–330. ACM, 2011. ISBN 978-1-4419-4888-5. URL http://2011.icse-conferences.org/.

[CLMG$^+$10] Cruz-Lemus, José A.; Maes, Ann; Genero, Marcela; Poels, Geert; Piattini, Mario: The impact of structural complexity on the understandability of UML statechart diagrams. In: *Information Sciences*, volume 180(11):pp. 2209–2220, 2010. ISSN 00200255. doi:10.1016/j.ins.2010.01.026.

[CLOS12] Cichos, Harald; Lochau, Malte; Oster, Sebastian; Schürr, Andy: Reduktion von Testsuiten für Software-Produktlinien. In: Jähnichen, Stefan; Küpper, Axel; Albayrak, Sahin, editors, *Software Engineering 2012: Fachtagung des GI-Fachbereichs Softwaretechnik, 27. Februar - 2. März 2012 in Berlin*, volume 198 of *LNI*, pp. 143–154. GI, 2012. ISBN 978-3-88579-292-5.

[CM94] Chilenski, John Joseph; Miller, Steven P.: Applicability of modified condition/decision coverage to software testing. In: *Software Engineering Journal*, volume 9(5):p. 193, 1994. ISSN 02686961. doi:10.1049/sej.1994.0025.

[COLS11] Cichos, Harald; Oster, Sebastian; Lochau, Malte; Schürr, Andy: Model-Based Coverage-Driven Test Suite Generation for Software Product Lines. In: Hutchison, David; Kanade, Takeo; Kittler, Josef; Kleinberg, Jon M.; Mattern, Friedemann; Mitchell, John C.; Naor, Moni; Nierstrasz, Oscar; Pandu Rangan, C.; Steffen, Bernhard; Sudan, Madhu; Terzopoulos, Demetri; Tygar, Doug; Vardi, Moshe Y.; Weikum, Gerhard; Whittle, Jon; Clark, Tony; Kühne, Thomas, editors, *Model Driven Engineering Languages and Systems*, volume 6981 of *Lecture Notes in Computer Science*, pp. 425–439. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-24484-1. doi: 10.1007/978-3-642-24485-8{\textunderscore}31.

[Con] Conformiq Qtronic: Semantics and Algorithms for Test Generation: a Conformiq Software Whitepaper. URL http://www.verifysoft.com/ConformiqQtronicSemanticsAndAlgorithms-3.

[DeM80] DeMillo, Richard A.: Mutation Analysis as a Tool for Software Testing. In: *4th IEEE Computer Software and Application Conference (COMPSAC)*, pp. 390–393. IEEE Press, New York, 1980.

[DPL$^+$15] Devroey, Xavier; Perrouin, Gilles; Legay, Axel; Schobbens, Pierre-Yves; Heymans, Patrick: Covering SPL Behaviour with Sampled Configurations. In: Schmid, Klaus; Haugen, Øystein; Müller, Johannes, editors, *the Ninth International Workshop*, pp. 59–66. 2015. doi: 10.1145/2701319.2701325.

[ER11] Engström, Emelie; Runeson, Per: Software product line testing – A systematic mapping study. In: *Information and Software Technology*, volume 53(1):pp. 2–13, 2011. ISSN 09505849. doi:10.1016/j.infsof.2010.05.011.

[FDMM94] Fabbri, Sandra C. P. F.; Delamaro, M. E.; Maldonado, J. C.; Masiero, P. C.: Mutation analysis testing for finite state machines. In: *1994 IEEE International Symposium on Software Reliability Engineering*, pp. 220–229. 1994.

[For12] Forrester Research Inc.: The Total Economic Impact of Conformiq Tool Suite, 2012.

[GG93] Grochtmann, Matthias; Grimm, Klaus: Classification trees for partition testing. In: *Software Testing, Verification and Reliability*, volume 3(2):pp. 63–82, 1993. ISSN 1099-1689. doi:10.1002/stvr.4370030203. URL http://dx.doi.org/10.1002/stvr.4370030203.

[GKPR08] Grönniger, Hans; Krahn, Holger; Pinkernell, Claas; Rumpe, Bernhard: Modeling Variants of Automotive Systems using Views. In: Kühne, Thomas; Reisig, Wolfgang; Steimann, Friedrich, editors, *Tagungsband* *zur Modellierung 2008 (Berlin-Adlershof, Deutschland, 12-14. März 2008)*, LNI. Gesellschaft für Informatik, Bonn, 2008.

[GMP03] Genero, Marcela; Miranda, David; Piattini, Mario: Defining Metrics for UML Statechart Diagrams in a Methodological Way. In: Goos, Gerhard; Hartmanis, Juris; Leeuwen, Jan; Jeusfeld, Manfred A.; Pastor, Óscar, editors, *Conceptual Modeling for Novel Application Domains*, volume 2814 of *Lecture Notes in Computer Science*, pp. 118–128. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003. ISBN 978-3-540-20257-8. doi:10.1007/978-3-540-39597-3{\textunderscore}12.

[GV07] Groher, Iris; Voelter, Markus: Expressing Feature-Based Variability in Structural Models. In: *Workshop on Managing Variability for Software Product Lines*. 2007.

[GVP13] Guo, Liangpeng; Vincentelli, Alberto Sangiovanni; Pinto, Alessandro: A complexity metric for concurrent finite state machine based embedded software. In: *2013 8th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pp. 189–195. 2013.

[HGB08] Hasling, Bill; Goetz, Helmut; Beetz, Klaus: Model Based Testing of System Requirements using UML Use Case Models. In: *1st International Conference on Software Testing, Verification and Validation, 2008*, pp. 367–376. IEEE, Piscataway, NJ, 2008. ISBN 978-0-7695-3127-4. doi:10.1109/ICST.2008.9.

[HMPO$^+$08] Haugen, Øystein; Møller-Pedersen, Birger; Oldevik, Jon; Olsen, Gøran K.; Svendsen, Andreas: Adding Standardized Variability to Domain Specific Languages. In: *SPLC 2008*, pp. 139–148. IEEE Computer Society, Los Alamitos, Calif, 2008. ISBN 978-0-7695-3303-2. doi:10.1109/SPLC.2008.25.

[HPP$^+$13] Henard, Christopher; Papadakis, Mike; Perrouin, Gilles; Klein, Jacques; Le Traon, Yves: Assessing Software Product Line Testing Via Model-Based Mutation: An Application to Similarity Testing. In: *ICSTW '13: IEEE 6th International Conference On Software Testing, Verification and Validation Workshops 2013*, pp. 188–197. 2013. ISBN 978-1-4799-1324-4.

[HTBH12] Holt, Nina Elisabeth; Torkar, Richard; Briand, Lionel C.; Hansen, Kai: State-Based Testing: Industrial Evaluation of the Cost-Effectiveness of Round-Trip Path and Sneak-Path Strategies ISSRE 2012, Dallas, TX, USA, November 27-30, 2012. In: *23rd IEEE International Symposium* *on Software Reliability Engineering, ISSRE 2012, Dallas, TX, USA, November 27-30, 2012*, pp. 321–330. IEEE, 2012. ISBN 978-1-4673-4638-2. doi:10.1109/ISSRE.2012.17. URL http://dx.doi.org/10.1109/ISSRE.2012.17.

[IBM09] IBM: Telelogic Rhapsody (IBM) – Official homepage, 2009. URL http://www.telelogic.com/products/rhapsody/index.cfm.

[JH09] Jia, Yue; Harman, Mark: Higher Order Mutation Testing. In: *Inf. Softw. Technol.*, volume 51(10):pp. 1379–1393, 2009. ISSN 0950-5849. doi:10.1016/j.infsof.2009.04.016. URL http://dx.doi.org/10.1016/j.infsof.2009.04.016, TitelanhanddieserDOIinCitavi-Projektbernehmen.

[JH11] Jia, Yue; Harman, Mark: An Analysis and Survey of the Development of Mutation Testing. In: *IEEE Transactions on Software Engineering*, volume 37(5):pp. 649–678, 2011. ISSN 0098-5589. doi:10.1109/TSE.2010.62.

[JHF12] Johansen, Martin Fagereng; Haugen, Øystein; Fleurey, Franck: An algorithm for generating t-wise covering arrays from large feature models. In: Santana de Almeida, Eduardo; Schwanninger, Christa; Benavides, David, editors, *the 16th International Software Product Line Conference*, pp. 46–55. 2012. ISBN 978-1-4503-1094-9. doi:10.1145/2362536.2362547.

[Jon91] Jones, Capers: *Applied software measurement: assuring productivity and quality*. McGraw-Hill, Inc, New York, NY, USA, 1991. ISBN 0-07-032813-7.

[KCH^{+}] Kang, K. C.; Cohen, S. G.; Hess, J. A.; Novak, W. E.; Peterson, A. S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. URL http://www.sei.cmu.edu/library/abstracts/reports/90tr021.cfm.

[KCM00] Kim, S.; Clark, John A.; Mcdermid, J. A.: Class Mutation: Mutation Testing for Object-Oriented Programs. In: *FMES*, pp. 9–12. 2000.

[KS13] Kuchcinski, Krzysztof; Szymanek, Radoslaw: JaCoP - Java Constraint Programming Solver. In: , 2013. URL http://lup.lub.lu.se/record/4092008/file/4092009.pdf.

[KWB03] Kleppe, Anneke G.; Warmer, Jos B.; Bast, Wim: *MDA explained: The model driven architecture : practice and promise*. The Addison-

Wesley object technology series. Addison-Wesley, Boston, 2003. ISBN 978-0321194428.

[Lac15] Lackner, Hartmut: Model-Based Product Line Testing: Sampling Configurations for Optimal Fault Detection. In: Fischer, Joachim; Scheidgen, Markus; Schieferdecker, Ina; Reed, Rick, editors, *SDL 2015: Model-Driven Engineering for Smart Cities*, volume 9369, pp. 238–251. Springer International Publishing, Cham, 2015. ISBN 978-3-319-24911-7. doi:10.1007/978-3-319-24912-4{\textunderscore}17.

[Lin05] Linz, Andreas: *Basiswissen Softwaretest: Aus- und Weiterbildung zum Certified Tester - Foundation Level nach ISTQB-Standard*. Dpunkt Verlag, 2005. ISBN 3898643581.

[LLLS] Lity, Sascha; Lachmann, R.; Lochau, Malte; Schaefer, Ina: Delta-oriented Software Product Line Test Models: The Body Comfort System Case Study.

[LOGS12] Lochau, Malte; Oster, Sebastian; Goltz, Ursula; Schürr, Andy: Model-based pairwise testing for feature interaction coverage in software product line engineering. In: *Software Quality Journal*, volume 20(3-4):pp. 567–604, 2012. ISSN 0963-9314. doi:10.1007/s11219-011-9165-4.

[LS05] Lúcio, Levi; Samer, Marko: Technology of Test-Case Generation. In: Broy, Manfred; Jonsson, Bengt; Katoen, Joost-Pieter; Leucker, Martin; Pretschner, Alexander, editors, *Model-based testing of reactive systems*, volume 3472 of *Lecture Notes in Computer Science*, pp. 323–354. Springer, Berlin, 2005. ISBN 978-3-540-26278-7. doi:10.1007/11498490{\textunderscore}15.

[LS12] Lackner, Hartmut; Schlingloff, Holger: Modeling for automated test generation - a comparison. In: Giese, Holger; Huhn, Michaela; Phillips, Jan; Schätz, Bernhard, editors, *Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme VIII, Schloss Dagstuhl, Germany, 2012, Tagungsband Modellbasierte Entwicklung eingebetteter Systeme*, pp. 57–70. fortiss GmbH, München, 2012.

[LS14] Lackner, Hartmut; Schmidt, Martin: Towards the assessment of software product line tests: a mutation system for variable systems. In: Gnesi, Stefania; Fantechi, Alessandro; Maurice H. ter Beek; Botterweck, Goetz; Becker, Martin, editors, *18th International Software Product Lines Conference - Companion Volume for Workshop, Tools and Demo papers, SPLC '14, Florence, Italy, September 15-19, 2014*, pp. 62–69.

ACM, 2014. ISBN 978-1-4503-2739-8. doi:10.1145/2647908.2655968. URL http://doi.acm.org/10.1145/2647908.2655968.

[LS15] Lackner, Hartmut; Schmidt, Martin: Potential Errors and Test Assessment in Software Product Line Engineering. In: Pakulin, Nikolay V.; Petrenko, Alexander K.; Schlingloff, Bernd-Holger, editors, *Proceedings Tenth Workshop on Model Based Testing, MBT 2015, London, UK, 18th April 2015*, volume 180 of *EPTCS*, pp. 57–72. 2015. doi:10.4204/EPTCS.180.4. URL http://dx.doi.org/10.4204/EPTCS.180.4.

[LSKL12] Lochau, Malte; Schaefer, Ina; Kamischke, Jochen; Lity, Sascha: Incremental Model-Based Testing of Delta-Oriented Software Product Lines. In: Hutchison, David; Kanade, Takeo; Kittler, Josef; Kleinberg, Jon M.; Mattern, Friedemann; Mitchell, John C.; Naor, Moni; Nierstrasz, Oscar; Pandu Rangan, C.; Steffen, Bernhard; Sudan, Madhu; Terzopoulos, Demetri; Tygar, Doug; Vardi, Moshe Y.; Weikum, Gerhard; Brucker, Achim D.; Julliand, Jacques, editors, *Tests and Proofs*, volume 7305 of *Lecture Notes in Computer Science*, pp. 67–82. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-30472-9. doi:10.1007/978-3-642-30473-6{\textunderscore}7.

[LSW$^+$10] Lackner, Hartmut; Svacina, Jaroslav; Weißleder, Stephan; Aigner, Mirko; Kresse, Marina: Introducing Model-Based Testing in Industrial Context - An Experience Report. In: *MoTiP'10: Workshop on Model-Based Testing in Practice*. 2010.

[LTWW14] Lackner, Hartmut; Thomas, Martin; Wartenberg, Florian; Weißleder, Stephan: Model-Based Test Design of Product Lines: Raising Test Design to the Product Line Level. In: *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA*, pp. 51–60. IEEE Computer Society, 2014. ISBN 978-0-7695-5185-2. doi:10.1109/ICST.2014.16. URL http://dx.doi.org/10.1109/ICST.2014.16.

[MBLD11] Mouchawrab, Samar; Briand, Lionel C.; Labiche, Yvan; Di Penta, Massimiliano: Assessing, Comparing, and Combining State Machine-Based Testing and Structural Testing: A Series of Experiments. In: *IEEE Trans. Softw. Eng.*, volume 37(2):pp. 161–187, 2011. ISSN 0098-5589. doi:10.1109/TSE.2010.32.

[McG] McGregor, John D.: Testing a Software Product Line. URL http://www.bibsonomy.org/api/users/ist_spl/posts/aec593ff707ee5e036253d36633a414e.

[MLL09] Malik, Qaisar A.; Lilius, Johan; Laibinis, Linas: Model-Based Testing Using Scenarios and Event-B Refinements. In: Butler, Michael; Jones, Cliff; Romanovsky, Alexander; Troubitsyna, Elena, editors, *Methods, Models and Tools for Fault Tolerance*, volume 5454 of *Lecture Notes in Computer Science*, pp. 177–195. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-642-00866-5. doi: 10.1007/978-3-642-00867-2{\textunderscore}9.

[MSB12] Myers, Glenford J.; Sandler, Corey; Badgett, Tom: *The art of software testing*. John Wiley & Sons, Hoboken, N.J, 3rd edition, 2012. ISBN 1118133153.

[Mv03] Muccini, H.; van der Hoek, A.: Towards Testing Product Line Architectures. In: *Electronic Notes in Theoretical Computer Science*, volume 82(6):pp. 99–109, 2003. ISSN 15710661. doi:10.1016/S1571-0661(04)81029-6.

[Off92] Offutt, A. Jefferson: Investigations of the Software Testing Coupling Effect. In: *ACM Trans. Softw. Eng. Methodol.*, volume 1(1):pp. 5–20, 1992. ISSN 1049-331X. doi:10.1145/125489.125473. URL `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:10.1145/1082983.1083279.

[OLAA03] Offutt, Jeff; Liu, Shaoying; Abdurazik, Aynur; Ammann, Paul: Generating test data from state-based specifications. In: *The Journal of Software Testing, Verification and Reliability*, volume 13:pp. 25–53, 2003.

[OMG] OMG: UML 2.4.1 Superstructure Specification.

[OU01] Offutt, A. Jefferson; Untch, Roland H.: Mutation 2000: Uniting the Orthogonal. In: Wong, W. Eric, editor, *Mutation Testing for the New Century*, pp. 34–44. Springer US, Boston, MA, 2001. ISBN 978-1-4419-4888-5. doi:10.1007/978-1-4757-5939-6{\textunderscore}7.

[OWES11] Oster, Sebastian; Wubbeke, Andreas; Engels, Gregor; Schürr, Andy: A Survey of Model-Based Software Product Lines Testing. In: Zander, Justyna; Schieferdecker, Ina; Mosterman, Pieter J., editors, *Model-based testing for embedded systems*, Computational analysis, synthesis, and design of dynamic systems, pp. 339–384. CRC Press, Boca Raton, 2011. ISBN 1439818452.

[OZML11] Oster, Sebastian; Zorcic, Ivan; Markert, Florian; Lochau, Malte: MoSoPoLiTe: tool support for pairwise and model-based software product line testing. In: *VaMoS '11*, pp. 79–82. 2011.

[PBL05] Pohl, Klaus; Böckle, Günter; Linden, Frank J. van der: *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc, Secaucus, NJ, USA, 2005. ISBN 3540243720.

[Pel] Peleska, Jan: RT-Tester Model-Based Test Case and Test Data Generator: User Manual: Version 9.0-1.0.0.

[Pel13] Peleska, Jan: Industrial-Strength Model-Based Testing - State of the Art and Current Challenges. In: *Electronic Proceedings in Theoretical Computer Science*, volume 111:pp. 3–28, 2013. ISSN 2075-2180. doi: 10.4204/EPTCS.111.1.

[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:10.1145/1183236.1183271.

[POS$^+$12] Perrouin, Gilles; Oster, Sebastian; Sen, Sagar; Klein, Jacques; Baudry, Benoit; Traon, Yves: Pairwise testing for software product lines: comparison of two approaches. In: *Software Quality Journal*, volume 20(3-4):pp. 605–643, 2012. ISSN 0963-9314. doi:10.1007/s11219-011-9160-9.

[Pro03] Prowell, S. J.: JUMBL: a tool for model-based statistical testing. In: *36th Annual Hawaii International Conference on System Sciences*. 2003. doi:10.1109/HICSS.2003.1174916.

[Pro05] Prowell, S. J.: Using Markov Chain Usage Models to Test Complex Systems. In: Sprague, Ralph H., editor, *Proceedings of the 38th Annual Hawaii International Conference on System Sciences*, p. 318c. IEEE Computer Society Press, Los Alamitos, Calif, 2005. ISBN 0-7695-2268-8. doi:10.1109/HICSS.2005.663.

[PSK$^+$10] Perrouin, Gilles; Sen, Sagar; Klein, J.; Baudry, B.; Le Traon, Yves: Automated and Scalable T-wise Test Case Generation Strategies for Software Product Lines. In: *ICST '10: International Conference on Software Testing, Verification and Validation*, pp. 459–468. IEEE Computer Society and IEEE, Los Alamitos, Calif and Piscataway, N.J,

2010. ISBN 978-1-4244-6435-7. doi:10.1109/ICST.2010.43. URL http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=5477055.

[Sch10] Schaefer, Ina: Variability Modelling for Model-Driven Development of Software Product Lines Systems, Linz, Austria, January 27-29, 2010. Proceedings. In: Benavides, David; Batory, D.; Grünbacher, Paul, editors, *Fourth International Workshop on Variability Modelling of Software-Intensive Systems, Linz, Austria, January 27-29, 2010. Proceedings*, volume 37 of *ICB-Research Report*, pp. 85–92. Universität Duisburg-Essen, 2010.

[SH15] Shimbara, Daisuke; Haugen, Øystein: Generating Configurations for System Testing with Common Variability Language. In: Fischer, Joachim; Scheidgen, Markus; Schieferdecker, Ina; Reed, Rick, editors, *SDL 2015: Model-Driven Engineering for Smart Cities*, volume 9369 of *Lecture Notes in Computer Science*, pp. 221–237. Springer International Publishing, Cham, 2015. ISBN 978-3-319-24911-7. doi:10.1007/978-3-319-24912-4{\textunderscore}16.

[SKK13] Schöffler, Jennifer; Kramer, Anne; Kastner, Norbert: Parameterized Generation of Process Variants and Project-Specific Operating Procedures from Business Process Models. In: Woronowicz, Tanja; Rout, Terry; O'Connor, RoryV.; Dorling, Alec, editors, *Software Process Improvement and Capability Determination*, volume 349 of *Communications in Computer and Information Science*, pp. 261–266. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-38832-3. doi:10.1007/978-3-642-38833-0{\textunderscore}27. URL http://dx.doi.org/10.1007/978-3-642-38833-0_27.

[SZCM04] Stephenson, Zoë; Zhan, Yuan; Clark, John; McDermid, John: Test Data Generation for Product Lines - A Mutation Testing Approach. In: Geppert, Birgit; Krueger, Charles; Li, Jenny, editors, *SPLiT '04: Proceedings of the International Workshop on Software Product Line Testing*, pp. 13–18. Boston, MA, 2004.

[UL06] Utting, Mark; Legeard, Bruno: *Practical model-based testing: A tools approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2006. ISBN 0123725011.

[UPL] Utting, Mark; Pretschner, Alexander; Legeard, Bruno: A taxonomy of model-based testing. URL http://www.cs.waikato.ac.nz/pubs/wp/2006/uow-cs-wp-2006-04.pdf.

[UPL12] Utting, Mark; Pretschner, Alexander; Legeard, Bruno: A Taxonomy of Model-Based Testing Approaches. In: *Softw. Test. Verif. Reliab.*, volume 22(5):pp. 297–312, 2012. ISSN 0960-0833. doi:10.1002/stvr.456. URL http://dx.doi.org/10.1002/stvr.456.

[Voe13] Voelter, Markus: *DSL engineering: Designing, implementing and using domain-specific languages*. CreateSpace Independent Publishing Platform, 2013. ISBN 1481218581.

[WBC14] Wilkinson, Toby; Butler, Michael; Colley, John: A Systematic Approach to Requirements Driven Test Generation for Safety Critical Systems. In: Ortmeier, Frank; Rauzy, Antoine, editors, *Model-Based Safety and Assessment*, volume 8822 of *Lecture Notes in Computer Science*, pp. 43–56. Springer International Publishing, Cham, 2014. ISBN 978-3-319-12213-7. doi:10.1007/978-3-319-12214-4{\textunderscore}4.

[Wei09] Weißleder, Stephan: Influencing Factors in Model-Based Testing with UML State Machines: Report on an Industrial Cooperation. In: Hutchison, David; Kanade, Takeo; Kittler, Josef; Kleinberg, Jon M.; Mattern, Friedemann; Mitchell, John C.; Naor, Moni; Nierstrasz, Oscar; Pandu Rangan, C.; Steffen, Bernhard; Sudan, Madhu; Terzopoulos, Demetri; Tygar, Doug; Vardi, Moshe Y.; Weikum, Gerhard; Schürr, Andy; Selic, Bran, editors, *Model Driven Engineering Languages and Systems*, volume 5795 of *Lecture Notes in Computer Science*, pp. 211–225. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-642-04424-3. doi:10.1007/978-3-642-04425-0{\textunderscore}16.

[Woo93] Woodward, M. R.: Errors in algebraic specifications and an experimental mutation testing tool. In: *Software Engineering Journal*, volume 8(4):p. 211, 1993. ISSN 02686961. doi:10.1049/sej.1993.0027.

[WS10] Weißleder, Stephan; Sokenou, Dehla: ParTeG - A Model-Based Testing Tool. In: *Softwaretechnik-Trends*, volume 30(2), 2010.

[WS14] Weißleder, Stephan; Schlingloff, Holger: An Evaluation of Model-Based Testing in Embedded Applications. In: *ICST' 14: International Conference on Software Testing, Verification, and Validation*. IEEE, 2014. ISBN 9781479957910.

[WWL15] Weißleder, Stephan; Wartenberg, Florian; Lackner, Hartmut: Automated Test Design for Boundaries of Product Line Variants. In: El-Fakih, Khaled; Barlas, Gerassimos; Yevtushenko, Nina, editors, *Testing Software and Systems*, volume 9447, pp. 86–101. Springer

International Publishing, Cham, 2015. ISBN 978-3-319-25944-4. doi: 10.1007/978-3-319-25945-1{\textunderscore}6.

[Xu11] Xu, Dianxiang: A Tool for Automated Test Code Generation from High-Level Petri Nets. In: Hutchison, David; Kanade, Takeo; Kittler, Josef; Kleinberg, Jon M.; Mattern, Friedemann; Mitchell, John C.; Naor, Moni; Nierstrasz, Oscar; Pandu Rangan, C.; Steffen, Bernhard; Sudan, Madhu; Terzopoulos, Demetri; Tygar, Doug; Vardi, Moshe Y.; Weikum, Gerhard; Kristensen, Lars M.; Petrucci, Laure, editors, *Applications and Theory of Petri Nets*, volume 6709 of *Lecture Notes in Computer Science*, pp. 308–317. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-21833-0. doi: 10.1007/978-3-642-21834-7{\textunderscore}17.

[ZSM11] Zander, Justyna; Schieferdecker, Ina; Mosterman, Pieter J.: A Taxonomy of Model-Based Testing for Embedded Systems from Multiple Industry Domains. In: Zander, Justyna; Schieferdecker, Ina; Mosterman, Pieter J., editors, *Model-based testing for embedded systems*, Computational analysis, synthesis, and design of dynamic systems. CRC Press, Boca Raton, 2011. ISBN 1439818452.

List of Figures

1.1 Process of product line engineering (PLE) 2
1.2 Approaches to product line (PL) testing 3

2.1 A feature model for the eShop example. 12
2.2 Feature model language diagram. 14
2.3 Annotative (a), compositional (b) variability (based on [GV07]) and delta modeling (c) 15
2.4 Implementation of the mapping model language in Ecore. 16
2.5 SPL design with annotative variability. 17
2.6 MBT process by Utting et al. [UPL12] 18
2.7 Feature model of the Ticket Machine example. 23
2.8 Base model of the Ticket Machine example. 23
2.9 Feature model of the Alarm System example. 25
2.10 Base model of the Alarm System example. 25
2.11 Base model of the eShop example. 27
2.12 Feature model of the Body Comfort System. [LLLS] 29
2.13 Overview of the Body Comfort System's architecture. [LLLS] 30

3.1 Mutation process for PL systems 38

4.1 Product line testing. 50
4.2 Detailed application-centered (AC) test design process. 51
4.3 Detailed domain-centered (DC) test design process. 52
4.4 Excerpt of the merged domain model with step-by-step approach applied. 54
4.5 Excerpt of the merged domain model with the pre-configuration approach applied. 55
4.6 Workflow of the SPLTestbench. 57
4.7 Menu options in the SPLTestbench. 58

5.1 Project options in SPLTestbench for sampling products. 69
5.2 Sampling process in SPLTestbench. 70

6.1 Excerpt of the product line model for the Ticket Machine. 78

6.2 Ticket Machine product w/o feature *Bills*. 79
6.3 Same product with complementary transition added. 79
6.4 PL Model Example: Ticket Machine with Complementary Transition. 80
6.5 Domain Model: Ticket Machine with added feature formulas and complementary transitions. 82
6.6 Product: State machine model of a Ticket Machine without *Bills*, *Change*, and *ReducedFare*. 82

List of Tables

2.1 Feature model summary for Ticket Machine (TM), Alarm System (AS), eShop (ES), and Body Comfort System (BCS). 31
2.2 Base model summary for Ticket Machine (TM), Alarm System (AS), eShop (ES), and Body Comfort System (BCS). 31

3.1 Mapping operator scores per mutation operator in % and Accumlated Scores (Acc) 43
3.2 UML operator scores per mutation operator in % and Accumlated Scores (Acc) 43
3.3 Summarized Results for Mapping Operators 44
3.4 Mutation Results for State Machine Operators 44

4.1 Test cases, test steps, and configuration for each of the presented approaches. 59

5.1 Distinct configurations sampled. 71
5.2 Test executions 71
5.3 Mutation scores for detected faults in base models in %. 72
5.4 Mutation scores for detected faults in mapping models in %. 72

6.1 Mutation Scores for Regular Tests 85
6.2 Mutation Scores for Tests with Transformations 85

List of Publications

Articles

2015 **Model-Based Product Line Testing: Sampling Configurations for Optimal Fault Detection**, *Hartmut Lackner*, 17th International SDL Forum: Model-Driven Engineering for Smart Cities.

Potential Errors and Test Assessment in Software Product Line Engineering, *Hartmut Lackner and Martin Schmidt*, Proceedings of the Tenth Workshop on Model Based Testing (MBT).

Automated Test Design for Boundaries of Product Line Variants, *Stephan Weißleder, Florian Wartenberg, and Hartmut Lackner*, Proceedings of the 27th International Conference on Testing Software and Systems (ICTSS).

2014 **Towards the assessment of software product line tests: a mutation system for variable systems**, *Hartmut Lackner and Martin Schmidt*, Proceedings of the 18th International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools (SPLC).

Model-Based Test Design of Product Lines: Raising Test Design to the Product Line Level, *Hartmut Lackner, Martin Thomas, Florian Wartenberg, Stephan Weißleder*, IEEE Seventh International Conf. on Software Testing, Verification and Validation (ICST).

2013 **Top-Down and Bottom-Up Approach for Model-Based Testing of Product Lines**, *Stephan Weißleder and Hartmut Lackner*, Proceedings of the Eighth Workshop on Model-Based Testing (MBT).

Zwei Ansätze zur automatischen modellbasierten Generierung von Testfällen fÃijr variantenreiche Systeme, *Stephan Weißleder and Hartmut Lackner*, in Softwaretechnik-Trends 33.

2012 **Modeling for automated test generation - a comparison**, *Hartmut Lackner and Holger Schlingloff*, 8th Dagstuhl-Workshop on Modellbasierte Entwicklung eingebetteter Systeme (MBEES).

2010 **Introducing Model-Based Testing in Industrial Context**, *Hartmut Lackner, Jaroslav Svacina, Stephan Weißleder et al.*, 3rd Workshop on Model-based Testing in Practice (MoTiP).

System Models vs. Test Models -Distinguishing the Undistinguishable?, *Stephan Weißleder and Hartmut Lackner*, GI Jahrestagung (LNI).

2009 **Test Case Generation from workflow-based Requirement Specifications**, *Hartmut Lackner, Jaroslav Svacina, and Holger Schlingloff*, Workshop on Concurrency, Specification and Programming (CSP).

Book Chapters

2012 **Software Platform Embedded Systems 2020: Application and Evaluation in the Healthcare Domain**, *Hendrik Heinze, Khalid Kallow, Hartmut Lackner et al.*, in Model-Based Engineering of Embedded Systems (ISBN 978-3642-346149).

Magazines

2011 **Nicht kapitulieren, sondern automatisieren**, *Hartmut Lackner and Salko Tahirbegovic*, in medizin & technik Ausgabe 5/11.

Selbständigkeitserklärung

Hiermit erkläre ich, Hartmut Lackner, geboren am 06.12.1982 in Berlin, dass

- ich die vorliegende Dissertationsschrift “Domain-Centered Product Line Testing” selbstständig und ohne unerlaubte Hilfe angefertigt sowie nur die angegebene Literatur verwendet 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