Advances in Testing Software Product Lines

Hartmut Lackner
MES GmbH
Berlin, Germany
hartmut.lackner@model-engineers.com

Bernd-Holger Schlingloff
Humboldt-Universität and
Fraunhofer FOKUS, Berlin
hs@informatik.hu-berlin.de

Abstract

In this article, we describe some recent techniques and results in model-based testing of software product lines. Presently, more and more software-based products and services are available in many different variants to choose from. However, this brings about challenges for the software quality assurance processes. Since only few of all possible variants can be tested at the developer's site, several question arise. How shall the variability be described in order to make sure that all features are being tested? Is it better to test selected variants on a concrete level, or shall the whole software product line be tested abstractly? What is the quality of a test suite for a product line, anyway? If it is impossible to test all possible variants, which products should be selected for testing? Given a certain product, which test cases are appropriate for it, and given a test case, which products can be tested with it?

We address these questions from an empirical software engineering point of view. We sketch modeling formalisms for software product lines. Then, we compare domain-centered and application-centered approaches to software product line testing. We define mutation operators for assessing software product line test suites. Subsequently, we analyze methods for selecting product variants on the basis of a given test suite. Finally, we show how model-checking can be used to determine whether a certain test case is applicable for a certain product variant.

For all our methods we describe supporting tools and algorithms. Currently, we are integrating these in an integrated tool suite supporting several aspects of model-based testing for software product lines.

Keywords: Software Product Lines, Cyber Physical Systems, Model Based Testing, Test Generation, Variant Management, Feature Modeling, Domain Analysis, Fault Injection, Product Sampling, Test Case Assignment

1 Introduction

Due to increasing market diversification and customer demand, most industrial products today are available in many different variants. For example, when buying a new car, one can choose not only between different brands and makes.

Within a model, one has options on the type and power of the engine, on the navigation and entertainment system, on various driver assistance functions, etc. There is a common basis on which these different variants are built (chassis, doors, power train and other elements).

A *product line* (or *product family*) is a set of products offered by a producer to customers, which have the same base functionality and share a common set of base elements. The members of a product line differ in the *features* which they offer to the customer. Thus, the individual products in a product line have a similar "look-and-feel", however, they differ in that one product may offer more or other functionality than another one.

In software-based systems, these features are realized via software. The concept of a software product line originates in the work of D. Parnas [58]. It has gained much attention by the research and consultancy of the Carnegie Mellon University Software Engineering Institute [49]. The CMU SEI defines a *software product line* to be a "set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" [16].

Software product lines are abundant in today's cyber physical systems: most electronic control units, e.g., in cars or trains, are configurable or come in multiple variants, as well as software in consumer products like coffee machines, dishwashers, mobile phones, etc. Also software products itself like, e.g., the ordering system of a web shop, are available in different variants to fit varying customer needs.

A challenge common to the development of these systems is that the software is similar, but not identical for all products; there are slight differences according to the features exhibited by a particular product. Sources of variability include planned diversity for different user groups, evolution and enhancement of products, and re-use of modules from one product in another one. Software product line engineering addresses this challenge. The main goal of software product line development is the strategic re-use of software artifacts. There have been various approaches to re-use: by copy and paste, macros, subroutines, modules, objects, components and services. The common problem in all of these approaches is that re-use increases the probability of errors. Therefore, quality assurance for software product lines is of utmost importance.

In this paper, we summarize and extend some of our previous work on this topic. We address several questions related to the model-based testing of software product lines. First, we address the problem of how to *specify* a software product line. We describe feature models and other artifacts that arise in a model-based development of software product lines.

We then consider the question of how to *generate* test cases from models. Basically, there are two competing approaches: One can derive individual test suites for individual products from an appropriate instantiation of a domain model. Or, one can try to generate a generic test suite from the domain model and instantiate this generic suite for each individual product. We formalize and compare these two approaches and derive guidelines on when to use which one.

Given a test suite, an important issue is to asses its *error detection capability*: How likely will the test suite uncover an error, given a faulty system under test? We investigate this question for the automatically generated test suites from the previous section. We define mutation operators, an experiment setup, and a tool suite to determine the mutation score for a test suite.

For a large product line, there may be zillions of different variants; it is impossible to test all of them. The question is, which products should be selected for testing the product line? We suggest that the products should be sampled using the test suite generated from the domain model. Moreover, we suggest that in this sampling, certain optimization criteria should be taken into consideration, such as "the selected products should be as diverse as possible". We sketch an algorithm based on boolean optimization for this sampling, and evaluate the effectiveness of the approach with fault injection techniques.

In many companies, especially when transitioning to a product line development process, there exists a large body of test cases. In order to be able to reuse these test cases for new products, we sketch a model-checking algorithm, which determines whether a certain test case is applicable for a certain product. We use a three-valued coloring scheme to distinguish between positive tests, negative tests, and tests which are insignificant for a certain product. We describe a prototypical implementation of our coloring algorithm and its application.

Throughout the paper, we use the example of a web shop system to illustrate our ideas.

This paper is structured as follows. After the introduction in section 1, we define modeling formalisms for software product lines in section 2. Subsequently, we show how to use domain models for test case generation in section 3. Then, in section 4, we describe a mutation system for assessing the error detection capability of a product line test suite. Section 5 deals with the problem of how to sample products for testing, and section 6 with the problem of how to reuse test cases for different products. Finally, in section 7 we discuss related literature, and, in section 8, some outlook on future work.

2 Specification of Variability

Two main driving factors for software product line engineering are the growing customer expectations for individualized products, and the potential to reuse existing software assets in the design of a product. The objective is to increase the number of product features while keeping the overall system engineering costs at a reasonable level. Product line development usually involves two engineering processes: *domain engineering* and *application engineering*. In domain engineering, reusable components are developed by a domain analysis and a domain reference architecture. In this process, commonalities and variabilities are analyzed and generic artifacts are developed. In application engineering, actual products serving different customer and market needs are derived from the domain artifacts. Products are built by instantiating and composing generic artifacts from the domain engineering process. Thus, domain artifacts are reused

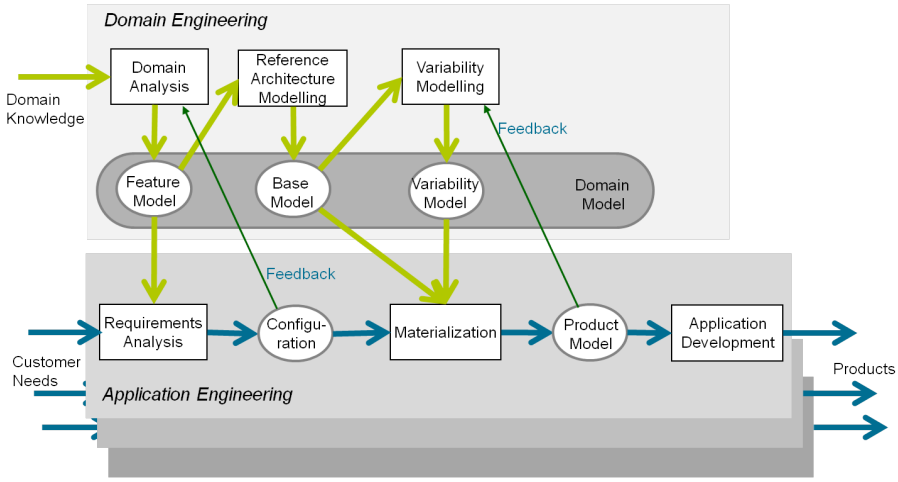

Figure 1: Various models in software product line engineering

to exploit the product line's commonalities. The instantiation and composition should be largely automatic. This way, many different products can be derived in an efficient way. There should be also feedback loops from application engineering to domain engineering, such that updates from different individual products can be generalized and adapted to the product line.

Model-based design is a particular form of software development, where a system model is continuously used as the central artifact throughout the whole engineering process. Initially, requirements are captured in an abstract model, e.g., in SysML, representing the system specification. This abstract model is refined and transformed into a concrete implementation model, e.g., in UML or Simulink®. From this implementation model, executable code is generated automatically by a suitable model compiler.

For model-based software product line engineering, the artifacts produced during domain engineering are mostly models. However, these models are generic, allowing an instantiation into different product models. During application engineering, the product models are refined and compiled as in "ordinary" model-based design.

Figure 1 depicts the model-based domain and application engineering work flows and their interrelations. The domain engineering process is generic, whereas the application engineering process exists in several instances, namely one for each product. Tasks are denoted by rectangles, and the models which are the results of tasks are given in circles.

2.1 Feature modeling

There have been various attempts to specify the variability of products belonging to a product line. Most approaches are based on a so-called *feature*

model. Feature models facilitate the explicit design of global system variation points [37].

A *feature* is the description of a designated functionality which is visible to the customer and forms an added value to the product. Each feature has a unique name and represents one characteristics of a product which is interesting for the stakeholder, e.g., a special added value. A *feature model* is a structuring concept for a set of features; it forms an explicit description of commonalities and differences of various products. Feature models are usually organized as and-or-trees, where each node is marked with the name of a feature. The root of the feature tree is the name of the product family. A parent feature can have the following relations to its sub-features:

- *Mandatory*: child feature is required,
- *Optional*: child feature is optional,
- *Or*: at least one of the children features must be selected, and
- *Alternative*: exactly one of the children features must be selected.

Additionally, it is allowed to attach boolean constraints on features to the feature tree. For example, 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. Tools for maintaining feature trees include pure::variants®, BigLever Gears®, FeatureIDE [68], and EASy-Producer [65].

Figure 2 shows the feature model of our web shop example which is used throughout this article. The eShop is a fictional e-commerce web shop designed for this article; it covers only a small part of the "real" web application. It provides the following functionality: Each eShop contains a catalog in which the available goods are listed. A customer can browse the catalog of items and put orders in the cart. Once the customer is finished, he/she can checkout and may choose from up to three different payment options, depending on the eShop's configuration. Payment can be via bank transfer, credit card, or eCoins. Some variants may provide a search function within the catalog. The transactions are secured by either a standard or high security server. Credit card payment is only offered if the eShop also implements a high security server.

Feature models can also be represented in propositional logic: For each feature model, there is an equivalent boolean formula. The propositions in this formula are the feature names, and the logical connectives are derived from the tree structure (see, e.g., [4, 6]. For the eShop feature model in Figure 2, a boolean formula representing the feature tree is

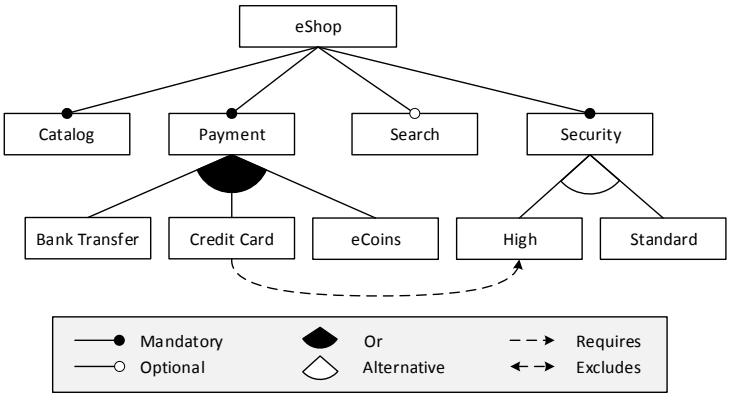

Figure 2: A feature model for the eShop example.

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

Since our feature model involves optional features, there are several potential products. Only a few of these will be materialized as actual products in the market. Given any feature model, a *resolution model* (or simply *resolution*) is an assignment of truth values to feature names. The assignment of a value to a proposition indicates whether the corresponding feature is selected (*true*) or deselected (*false*). A resolution is called *valid*, if the corresponding boolean formula evaluates to true. A valid resolution is also called a *product configuration*.

For instance, the following two assignments $P1$ and $P2$ are product configurations for the eShop feature model presented in Figure 2.

$$
\begin{aligned}
P1 \ &= \ eShop, Catalog, Payment, BankTransfer, \neg CreditCard, \neg eCoins, \\
&\quad \neg Search, Security, \neg High, Standard \\
P2 \ &= \ eShop, Catalog, Payment, BankTransfer, CreditCard, eCoins, \\
&\quad Search, Security, High, \neg Standard
\end{aligned}
$$

It can be seen that both configurations satisfy the formula FM above. $P2$ is a *maximal* configuration with respect to the number of features. For any feature

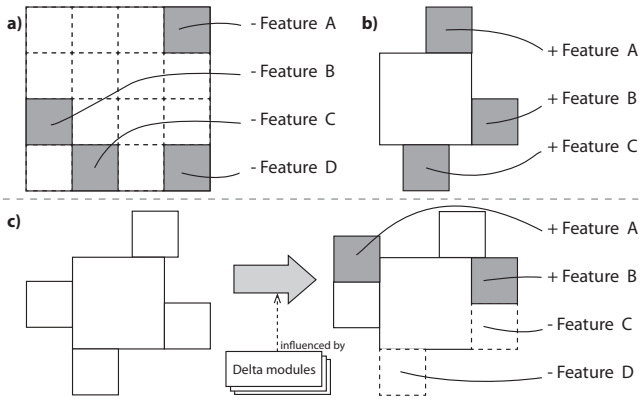

Figure 3: Annotative (a), compositional (b) variability (based on [27]) and delta modeling (c)

model with n features, there are between 0 and 2^n product configurations. In particular, the eShop feature model allows 20 different configurations.

2.2 Variability modeling

Although a feature model captures the system's variation points in a concise form, its elements are only propositional symbols [20]. Their semantics has to be provided by mapping them to other development artifacts. Such a mapping can be defined either explicitly, in a separate variability model, or implicitly, within the referenced artifacts. In the following, we introduce the three major paradigms for variability modeling:

Annotative Variability modeling A base model contains every element that is used in at least one product configuration and, thus, subsumes every possible product [28] (Fig. 3a). Subsequently, model elements are removed to resolve a valid variant.

Compositional Variability 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. 3b).

Transformational Variability modeling In transformational variability, compositional and annotative methods are combined. Model elements can be removed and added to resolve a variant. A well-known approach for this is so-called *delta modeling* (also delta-oriented programming) [64]. Delta modeling consists of two parts: The first one is the design of a core product comprised of a set of feature selections that represent a valid product. The second part

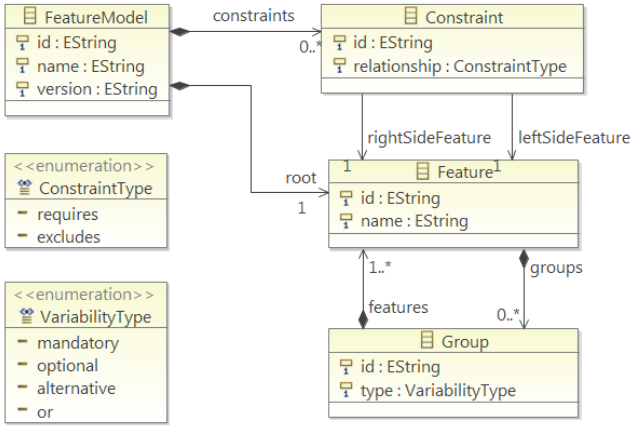

Figure 4: Meta-model for feature models.

is the specification of a set of so-called "delta modules" which describe changes ("'deltas") to the core module. Each delta is either the construction (add) or destruction (remove) of an element from the product model (Fig. 3c). A delta module then is associated to one or more features. Whenever a feature is selected or deselected the associated deltas are applied to the product model.

2.3 A Basic Variability Language

In this article, we describe a custom modeling language for annotative variability. This language is inspired by the "Common Variability Language" (CVL) standardization initiative [33].

As a concrete syntax for feature models, we provide a meta-model defined in Ecore with the Eclipse modeling framework EMF as depicted in Figure 4. This meta-model offers basic functionalities for designing mandatory, optional, or, alternative, and binary cross-tree-constraints between features.

We use an explicit *variability model* for mapping features to elements of a *base model*. That is, a base model is the artifact which implements the features of the product line. (The base model model is sometimes called "the 150% model", a terminology which we refrain to adopt.) Each feature corresponds to some parts of the base model.

Formally, a base model can be any model which is an instance of some MOF meta-model. In our work, a base model is a UML model consisting of state machine diagrams, class diagrams and OCL formulae. The base model describes realizations for *all* features; thus, if the feature model contains conflicting features, then the base model may not represent a possible product.

Part of the base model of the eShop example, the UML state machine diagram, is given in Figure 5. This state machine is designed from a user's perspective. Whenever a user starts the eShop, the system initializes and waits for

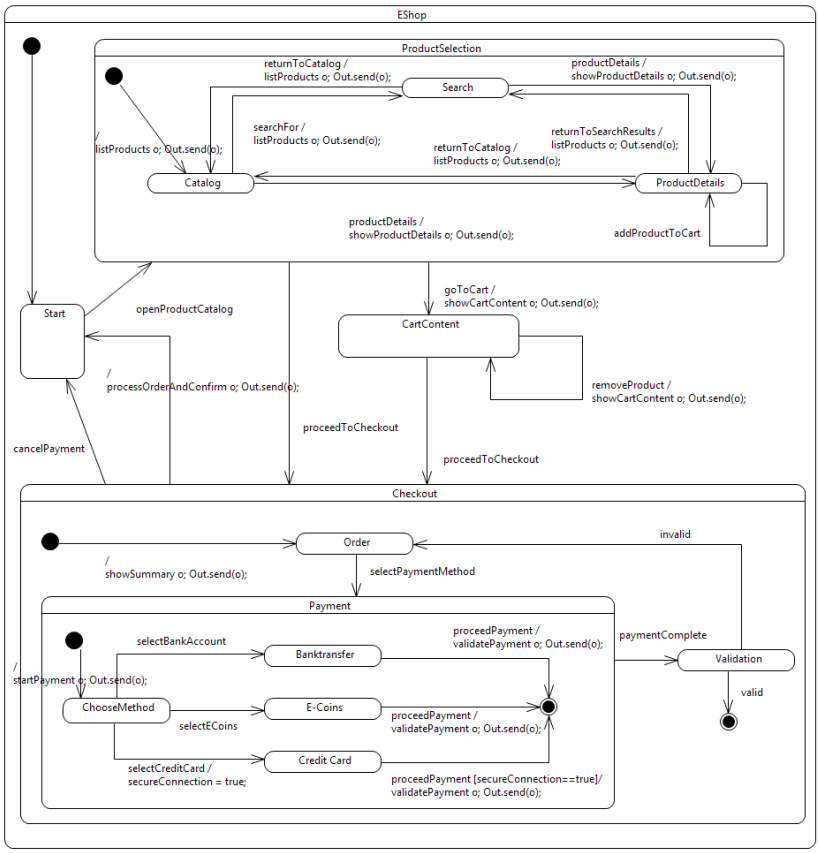

Figure 5: Base model of the eShop example.

the user in state `Start` to open the product catalog. Upon receit of the signal `openProductCatalog`, it invokes the sub-machine `ProductSelection`, which initially outputs a list of product items and then waits for further user input in state `Catalog`. The system sends outputs to the user or known banks by instantiating the output message, e.g., `listProducts o;` and sending it via calling the method `send` of its outport object. The eShop example has exactly one outport, called `Out`, hence, a message object `o` is send by calling `Out.send(o)`. The user can display the product details by sending the message `productDetails` to the system. Not shown is the necessary parameter id of the product for which the details shall be displayed. When the system is in state `ProductDetails`, the user may add the displayed product to his cart and/or return to the catalog. If enabled, the user can search for a particular product in state `Catalog` by sending signal `searchFor`, which will return a list of products that match the search term entered by the user. Again, the user may view the product details, and add the product. It is also possible to return to the search or the catalog.

Anytime state `ProductSelection` is active, the user may view the cart's contents (`goToCart`) or proceed to checkout (`proceedToCheckout`). While the user displays the contents of the cart, items can be removed (`removeProducts`) before proceeding to the checkout (`proceedToCheckout`). When the user proceeds to the checkout the eShop invokes the submachine `Checkout`. Here, the user is presented with a final summary, before selecting a payment method. Depending on the configuration of the eShop, the user may select from up to three payment methods. The internal workings of the individual payment methods are not modeled. When a customer has entered his payment data, s/he may proceed to the validation (`proceedValidation`). Credit card payments will not be validated unless the `secureConnection` flag is set to true. This flag is raised whenever a high security eShop product is built. The eShop will send the payment data to the corresponding bank for validation (`validatePayment`). The bank acknowledges the receiving of the data by sending a `paymentComplete` signal to the eShop. If the data provided by the customer is correct and the transaction is carried out, the bank sends the message `valid`, and otherwise `invalid`. If a `valid` signal is received by the eShop it will continue to process the order and lead the user back to the state `Start`. Otherwise, if the signal `invalid` is received, the user is prompted again to choose the payment method and data. The user can abort the checkout process at any time by sending a `cancelPayment` signal to the eShop.

The *variability model* maps features to elements of the base model. In Figure 6, we give a meta-model for the variability modeling language as implemented with Ecore. The language refers to concepts of *elements* from the UML and to *features* from our feature modeling language. 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*).

Figure 7 shows an excerpt of the eShop variability model, where parts of the feature model are depicted in the upper half and parts of the state machine's

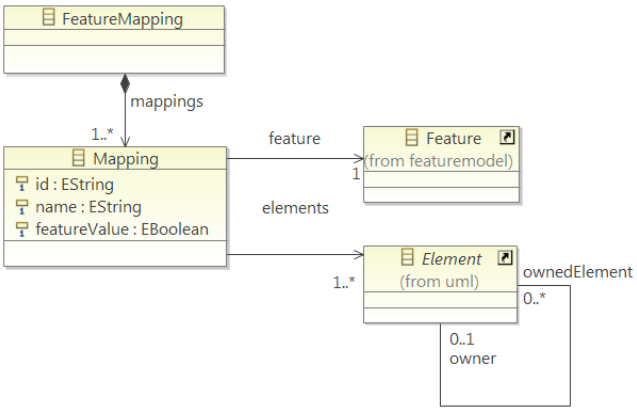

Figure 6: Ecore meta-model of variability models

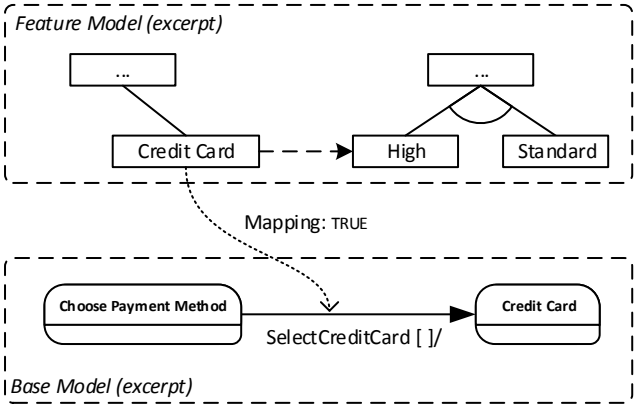

Figure 7: Software product line design with annotative variability.

payment process are shown in the lower half. In between is the mapping, denoted by a dotted edge, from feature *Credit Card* to the transition labeled as `SelectCreditCard[]/`.

The complete mapping in the variability model is as follows:

- *Catalog*, *Payment* and *Security* are mandatory and need no mapping.
- *Bank Transfer* maps to all transitions connected to state `Bank Transfer`.
- *eCoins* maps to all transitions connected to state `eCoins`.
- *Credit Card* maps to all transitions connected to state `Credit Card`.
- *High* maps to attribute secureConnection with its default value set to true.

- *Standard* maps to attribute secureConnection with its default value set to false.
- *Search* maps to all transitions connected to state `Search`.

We refer to the triple of feature model, base model and variability model as the *domain model*. From such a domain model, product models can be resolved for a given product configuration. In a model-based design process, implementation code or test cases can be derived automatically or semi-automatically from these product models.

3 Model-based Testing for Product Lines

Systematic testing is the most common method for quality assurance of complex software. A *system under test* (SUT) is executed with a systematically constructed set of test cases, the *test suite*, and the observed system behavior is compared with the expected one. The quality of a test suite is often measured in terms of requirements and/or code coverage. Constructing a test suite which satisfies given coverage criteria by hand may be a tedious exercise. The term *model-based testing* subsumes various techniques in which test suites are constructed from a given model [50].

There are several such techniques: in some approaches test case models, e.g., use case diagrams, are used for test case description and test implementation [32]. In other approaches, e.g., the classification tree method, test data is generated from test data models [26]. In this article, we consider 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 system under test and/or the behavior of its environment. Each generated test case is a sequence of test stimuli and expected system reactions. A test case may also pose preconditions on the system's state or configuration that must be fulfilled before the test case can be executed. Additionally, it may specify postconditions which must hold after the execution of the test case.

For software product lines, there are various possibilities to define model-based testing processes. One approach is to resolve a representative set of products from a product line for the purpose of testing, and then to generate test cases for these products. Another possibility is to generate tests from domain-level artifacts, and to specialize this generic test suite to specific products. Subsequently, we elaborate both approaches, and compare them. This material is based on the forthcoming dissertation of the first author [43]. We have implemented and evaluated both approaches in a prototypical tool "SPLTestbench"; we report on the results using several examples.

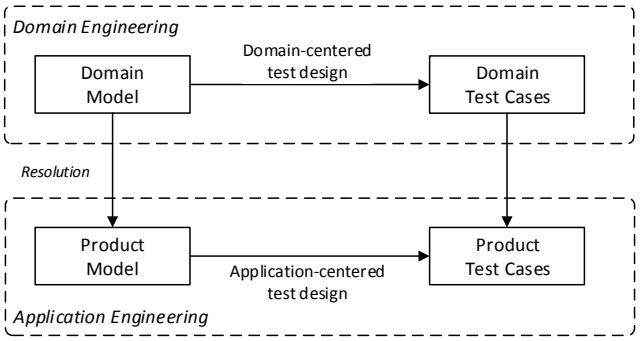

Figure 8: Product line testing.

3.1 Application-centered and Product-centered Test Generation

A test generation method for software product lines faces two challenges: covering a significant subset of products, and covering a significant subset of the test focus on the domain-level. As the products in a product line share commonalities, there may be significant overlap in test suites, and test cases may be applicable to more than one product.

As described in the previous section, a domain model consists of a base model (e.g., a UML state machine), a feature model explicitly expressing the product line's variation points (e.g., a feature tree), and a variability model connecting these two. From domain artifacts, application artifacts (products and tests) can be derived. Based on this, we define two approaches to automated test design for software product lines. These are depicted in Figure 8: *application-centered*(AC) and *domain-centered*(DC) testing.

In the AC approach, first a representative set of products for testing is selected. Then, test cases are generated from each of these models. The focus of the test generation process is on satisfying a defined coverage for each test model. This may lead to an overlap in the resulting test suites.

In contrast to this, in the DC approach the domain model is directly used for test design. In this approach, the focus is on the behavior defined at the domain-level; coverage criteria of single products are not considered. 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.

3.1.1 Application-Centered Test Design

Any test design method that binds the variability by selecting products before the test design phase can be called application-centric (see Figure 9). Usually, it is not feasible to build and test *all* possible products of a product line. Therefore, in application-centered test design, product models to be tested are selected

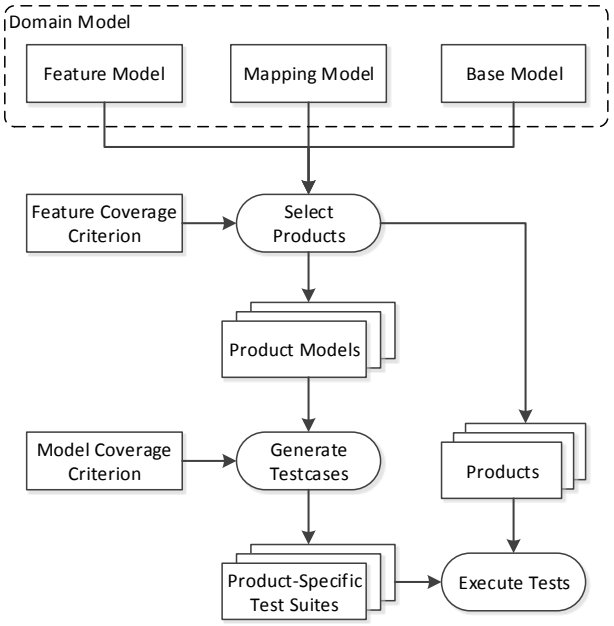

Figure 9: Application-centered test design process.

from the domain model before the test design starts. The selection is done according to a predefined *feature coverage criterion*. For example, pairwise feature coverage (see [47, 61]) is achieved when for each tuple of features all valid combinations of feature presence and absence are represented by at least one product model. Tests are performed only on the products corresponding to the selected product models.

Using conventional test generation methods, test suites can be automatically obtained from the selected product models. The test generator can be configured such that is satisfies a certain *model coverage criterion*. In our example, this may be any coverage criterion applicable to UML state machines, e.g., state or transition coverage.

Since each test case is generated for one configuration, the resulting test suite is specific to its respective product. Therefore, the test generation yields *application specific test suites*. Due to the commonalities of the products of a product line, a test case that was generated for one configuration may be applicable to other configurations as well. Consequently, test cases that aim for the same goals are executed over and over again. Therefore, there may be significant redundancies in application-centered test suites.

3.1.2 Domain-Centered Test Design

In order to avoid these potential drawbacks, in domain-centered test design domain engineering level artifacts are used for test generation. This approach preserves the variability until a product has been selected for test execution.

A major advantage of this approach is that one can focus on testing aspects of the domain model, without having to derive products first. Thus, the overlap of the results from independently generating tests for similar products can be avoided. The coverage of test targets can be maximized, which leads to high-quality test cases.

For domain-centered test design, all domain artifacts should be taken into consideration. Using a base model as the only source of information is not adequate. The base model lacks information about the features, and about the associations of model elements to features. However, this information is important for a test generator: Features impose additional constraints on the behavior of the system. Since the base model contains implementation information for *all* features, it may include contradictory requirements. It might not even be a correct model according to the UML syntax. Thus, a challenge of this approach is to merge a domain model, consisting of a feature model, variability model and a base model, into a single model artifact that a standard test generator will accept as valid input (Fig. 10). Subsequently, we describe two solutions to this problem: 1) the *step-by-step* approach: sequentially excluding non-conforming configurations during test design-time, and 2) the *pre-configuration* approach: choosing a valid configuration before the design of 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 test case is applicable to any valid product of the product line. 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

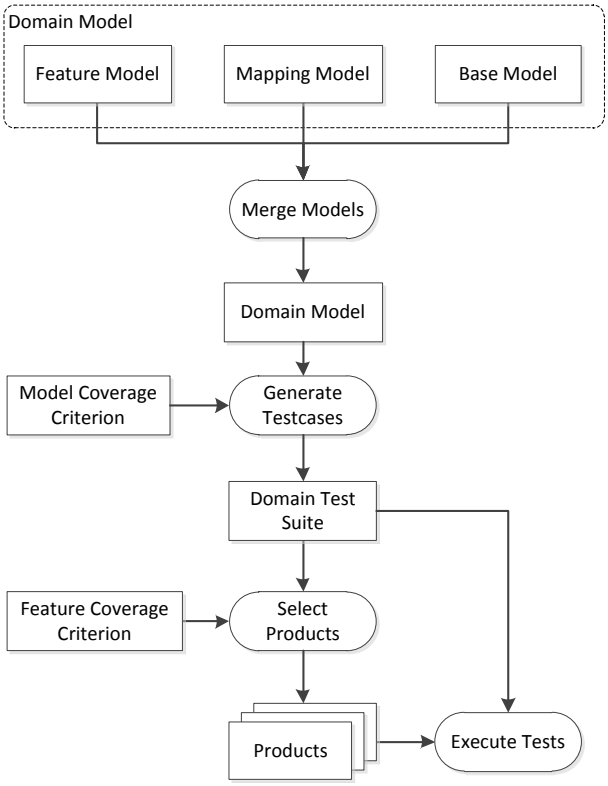

Figure 10: Domain-centered test design process.

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: A partial feature formula for a particular feature f is constructed as a conjunction of (i) the feature f itself, (ii) f's *parents* p, and (iii) every feature r that is *required* by f. Depending on the structure of the feature model, not all features related to f are captured by this approach. Any parent or required feature may require additional features. Therefore, steps (ii) and (iii) are repeated for every p and r until the formula is stable. So far, the formula contains features that must be selected for f, but since some of the selected features may require the absence of other features, we combine the formula with another conjunction of (iv) features *excluded* by already selected features and (v) *alternative* features. Finally, the formula can be reduced by removing all core features, because they are a mandatory part of every product. For instance, the partial feature formula for the CreditCard feature from our eShop example is:

$$F = CreditCard \wedge High \wedge \neg Standard$$

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 11

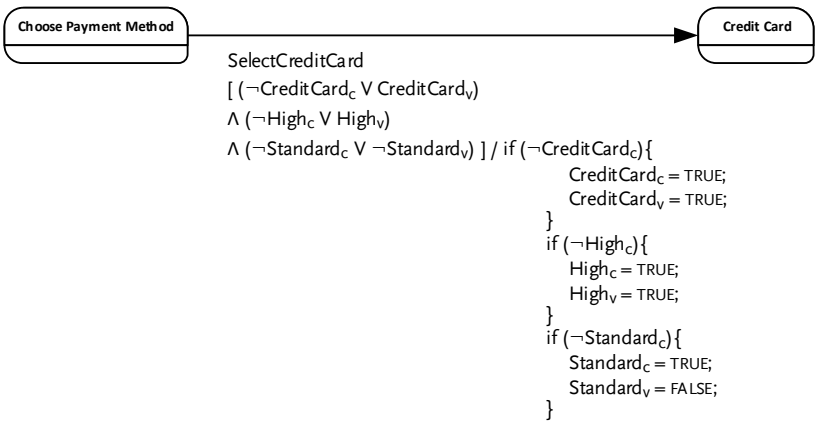

Figure 11: Excerpt of the merged domain model with step-by-step approach applied.

shows the result of merging the domain model into a single UML state machine for the excerpt of the eShop introduced in Figure 7.

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 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 generic for any product of the product line 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 test goals that are specific to the selected product configuration. Thus, satisfying all domain model test goals is a matter of finding the appropriate configurations.

  We implemented the pre-configuration approach by adding a configuration

Figure 12: Excerpt of the merged domain model in the pre-configuration approach.

signal to the very beginning of the base model. To this end, 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 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 product line'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. This is similar to the step-by-step approach, where the base model behavior is limited to a potential behavior of an actual product. 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, the feature literals must be exchanged by the corresponding feature variables from the class. Figure 12 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 does not present the real system's behavior. In a simple post-processing action, this configuration step can be removed from the test cases before testing is performed.

With these transformations to the base model, a test designer can create test cases for the product line. However, each test case will be specific to

one configuration. In order to create generic test cases which are applicable to more than one product, we can apply a model transformation.

The additional transformation steps consist of adding Boolean control variables for each non-core feature to the system class. These control variables are initialized with *false*. Effects on transitions are added which change them 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 the effect of t 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.

3.2 An Evaluation of Product Line Test Generation

In this section, we present an implementation and evaluation of the two DC test design approaches described above, and compare them to AC test design methods.

3.2.1 A Prototypical Tool Chain

Our prototypical SPLTestbench is depicted in Figure...

It consists of five major components:

(i) a feature injector to merge domain models as introduced in section 3.1.2,

(ii) a model printer that exports the model to a test generator-specific format,

(iii) an adapter for third-party test generators,

(iv) a configuration extractor that collects incomplete configurations from the generated test cases, and

(v) domain-specific languages [71] that facilitate design and data processing of feature models, variability models, and configuration models.

An important component in this workbench is the model-based test generator (iii). This is a tool which takes a test model (e.g., a UML state machine), and produces a set of test cases from it. Since this paper deals with test generation for product lines, we do not describe and compare different algorithms for model-based test generation. The interested reader is referred to the literature [13]. There are several test generators available, both from industrial and

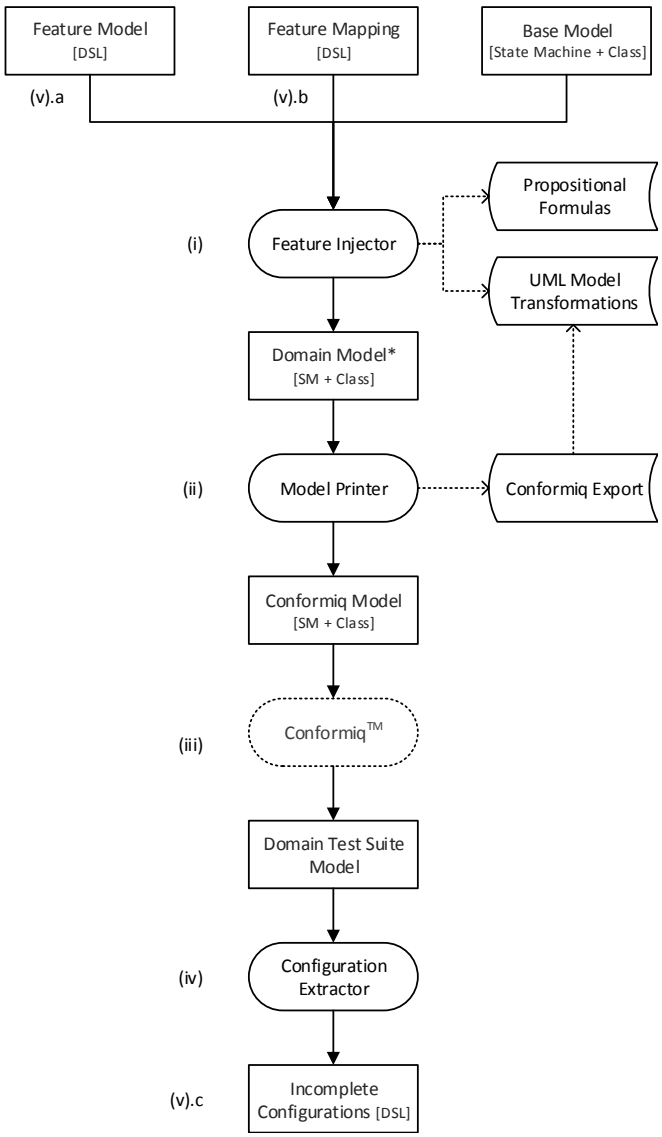

Figure 13: Workflow of the SPLTestbench.

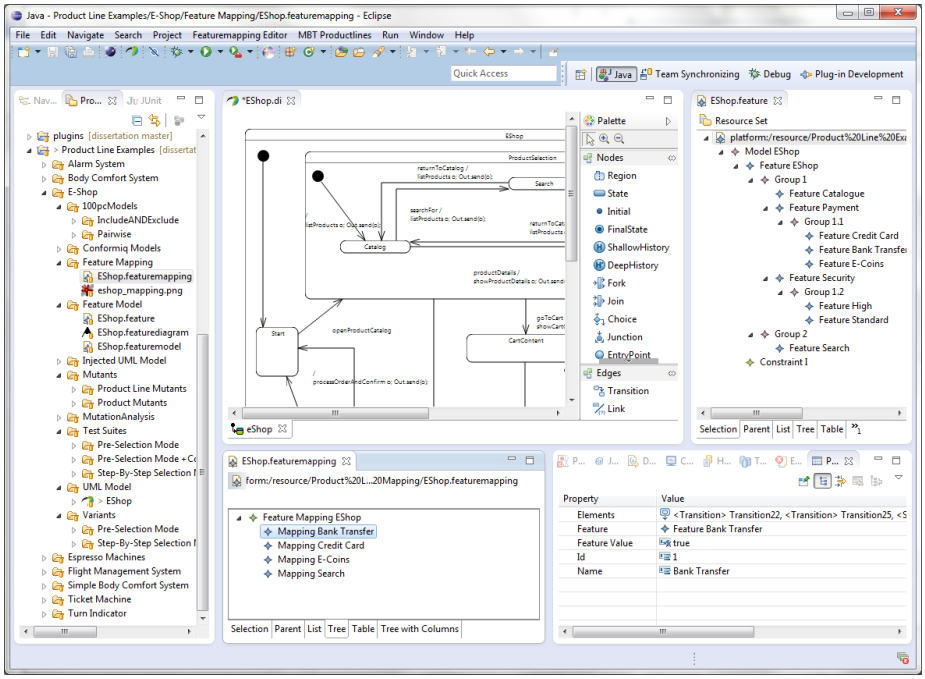

Figure 14: Perspective of the SPLTestbench.

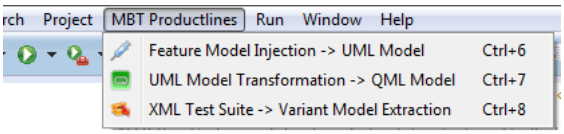

Figure 15: Menu options in the SPLTestbench.

academic sources [31]. In our implementation and experiments, we used Conformiq Designer [70] and Real-Time Tester [59] as external test case generators.

Figure 13 depicts the work flow a test engineer has to follow when using our SPLTestbench. The SPLTestbench is implemented as Eclipse plug-ins. The plug-ins are shown in the screenshot in Figure 14. On the left hand side the project layout is shown. In the center view, a currently opened UML state machine is depicted, which subsumes all possible behaviors of the SUT. On the right hand side, we see the feature model in a tree view editor. Below, we find the feature mapping editor with a property view on the currently selected feature mapping. Figure 15 shows how the components (i), (ii), and (iv) integrate into the Eclipse IDE. Each of the three menu items starts an individual wizard that guides the user through the details of the respective process.

3.2.2 Experimental Results

Now, we describe the experiments we performed to evaluate our approaches. We used several example product lines, including the eShop example from above. For our experiments we generated tests according to both presented approaches, AC and DC test design. For AC test design we sampled products according to two different feature model coverage criteria: *all-features-included-excluded* and *all-feature-pairs* [60]. As the name indicates, the criterion *all-features-included-excluded* is satisfied if in the set of sampled products, for each feature there is one product in which it is selected, and another one in which it is desected. The criterion *all-feature-pairs* or *pairwise* holds of a sampling, if for every pair of features (f_1, f_2) there are products where f_1 and f_2 are both selected, one where neither f_1 nor f_2 is selected, one where f_1 is selected and f_2 deselected, and one where f_1 is deselected and f_2 selected. Details of the model sampling procedure will be described in section 5.

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 model we required all-transition coverage from the test generator [73]. There would be other, more sophisticated metrics for state machines to consider [24, 18, 30]. However, with this criterion we maintain comparability to other studies, e.g., in [44].

We were able to generate test suites for both approaches with all the aforementioned parameters for all examples. In order to compare the results, we counted

- the number of test cases,

Table 1: Test cases, test steps, and configurations for each of the presented approaches with the eShop example.

	Approach			
	AC-IX	**AC-PW**	**DC-Pre**	**DC-Step**
Tests	20	71	13	13
Test Steps	135	486	48	39
Configurations	2	7	4	2

- the test steps that were generated by the test generator, and
- the number of configurations that are necessary to execute the test cases.

These are important criteria when estimating test efforts: The number of configurations is a major factor, since every configuration must be built, setup, and maintained for testing. The number of test steps is an indicator for the efforts for test design and maintenance. Finally, the number of tests of interest when put in relation to the test steps. This gives the average test case length, which affects the efforts for debugging. With increasing length of a test case it becomes more difficult to isolate a fault within the SUT.

The results for these measures are shown in Table 1 for each individual approach: AC with all-features-included-excluded (AC-IX) as well as with pairwise (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, while at the same time — as stated in section 3.1.2 — it is focused on covering every reachable transition. We take this as an indicator for DC test design to scale better than AC approaches. We performed similar experiments on several other examples, with comparable results.

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 section 5, this does not necessarily lead to a lower error detection capability.

4 Assessment of Product Line Test Suites

In the previous section, we evaluated test generation algorithms with respect to the size and complexity of test suites. However, testing usually is performed to *detect errors* in an implementation. Thus, the complexity may not be the only quality criterion which is important for a test suite. The “effectiveness” of a test

suite is often called its *error detection capability*. Given a test suite and an SUT, test execution will show up whether the implementation contains errors or not. Thus, by repeatedly executing a test suite with a faulty SUT, the error detection capability of the test suite can be measured. However, if the SUT doesn't exist yet, what is the error detection capability of the test suite? In a product line development, often it is impossible to construct all possible products in advance. As an example, the software for the embedded control units in a car is highly dependent on which features are present in the car. It is constructed only when the car is actually built, according to the orders of the customer. Given a test generation algorithm, how can we decide whether the generated test suite is adequate? Can we compare different product line test generation methods with respect to the quality of their result? Test assessment is an integral part for evaluation of the concepts presented in the previous section. This section builds the foundations for assessing the quality of a product line test suite in terms of its error detection capability. Though there are many methods proposed for testing a product line, until now, quality assessment of test suites mostly is limited to measuring code, model and/or requirements coverage [52, 17].

A generally acknowledged approach to measure the error detection capability of a test suite is *mutation analysis*. Artificial faults are inserted into an SUT to form a *mutant*. Then, the test suite is executed and it is observed how many of the errors in the mutant are detected. In product line engineering, mutation analysis so far has been applied to individual products of a product line only. This "product-level approach" has two major drawbacks:

- Firstly, developers can make errors in all kinds of artifacts, not only on product models. For example, there could be faults in the base model or variability model as well. This can lead to new kinds of faults in the products.

- Secondly, in the product-level approach the selection of products to mutate is biased by the selection of products for testing. Therefore, the mutation analysis assesses the quality of the tests for particular products. It is unclear how to combine the results to get an assessment for the whole product line.

To better understand which errors can occur in model-based product line engineering, we consider the different design paradigms discussed in section 2 (annotative, compositional and transformational variability modeling, see Figure 3). We analyze potential errors in the respective design processes. From this, we develop mutation operators for variability and base models to mimic possible faults in these models.

To tackle the second problem, we define a mutation system and specific mutation operators for domain-level artifacts. This enables us to assess the test quality independently from the tested products. Subsequently, the error detection capabilities of a test suite can be assessed for the complete product line.

4.1 Potential Errors in Model-Based Product Line Engineering

The feature mapping has a major impact on the resolved products in a product line. However, designing a variability model is a complex and error-prone task, which is hard to automate. We identify potential errors in this process in a systematic way. We check each modeling paradigm for possibilities to

- omit necessary elements,
- add superfluous element, or
- change the value of an element's attribute.

For each potential error we discuss its effects onto the resolved products.

Annotative Variability Modeling

In the annotative variability paradigm, the following model elements for potential errors in the variability model can be identified: 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:

1. *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 by this error, since the behavior is now part of the common core and, thus, enabled by default.

2. *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.

3. *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.

4. *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.

5. *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.

6. *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 in products with the associated feature and available to any product not including the said feature.

This enumeration covers all basic errors which can be made in the variability model. Other errors can be described by combinations of these basic errors. In the next subsection, we will use them to define fault injection operators.

Compositional Variability modeling

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

1. *Omitted mapping:* a necessary mapping is missing in its entirety.

2. *Superfluous mapping:* a superfluous mapping is added.

3. *Swapped modules:* the associated modules of two mappings are mutually exchanged.

4. *Swapped features:* the associated features of two mappings are mutually exchanged.

Transformational Variability Modeling

For other paradigms, like delta modeling [64], we make similar observations. In contrast to compositional variability models, transformational variability models start from an actual core product, instead of a base model. From this, only the differences from one product to another are defined by delta modules.

Similar to above, errors in delta modeling include omitted or superfluous deltas, omitted or superfluous base elements, and omitted or superfluous delta elements. Since there are no attributes in deltas other than add/remove, changes in deltas can be neglected.

To sum up, a systematic analysis according to the three categories ‘add’, ‘remove’, and ‘change’ yields a list of all potential errors for each modeling paradigm. Subsequently, we will use the identified error possibilities in a fault injection framework to assess product line test suites.

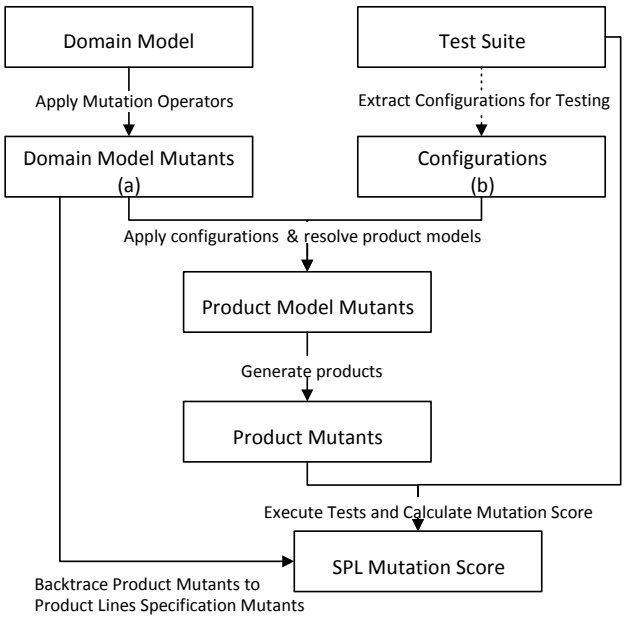

Figure 16: Mutation process for product line systems

4.2 Mutating Domain Models

As discussed in the previous subsection, model-based product line engineering bears the risk for new kinds of errors compared 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 assessing the error detection capability of a product line test suite. Note that in contrast to other authors, we assess the test quality for the whole product line rather than for single products.

To define our mutation system we need novel mutation operators. Mutation operators defined for non-variant systems cannot add or remove features. However, a high-quality test suite should also detect such faults. Hence, we propose new mutation operators based on the potential errors identified in Section 4.1.

4.2.1 A Mutation System for Product Lines

Performing mutation analysis on product line 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 product line test suite itself, since each test is applicable to just a subset of products.

In Figure 16, we depict a mutation process for assessing product line 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 resolved to mutant products, 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. If a product mutant is detected ("killed") by a test, we backtrack its domain model mutant and flag it accordingly. The mutation score is the quotient of the detected and overall number of mutants.

In the mutation or resolution process, models may be generated which are invalid (syntactically incorrect). Of course, such mutants must be discarded. For all mutation operators, there is a risk that mutants are materialized which behave equivalent to the original product (so-called "masked mutants"). For example, a masked mutant is generated when the mutated element is part of a disabled feature. Of course, masked mutants should be excluded from the scoring. However, the question whether a mutant is masked is in general undecidable. Our mutation systems filters mutants which are structurally equivalent to the original product model.

4.2.2 Product Line Mutation Operators

Here, we present a list of mutation operators for variability models with annotative variability. We start with standard state machine mutation operators, applied to the domain-level.

Mutation operators for the base model: In general, for product line test assessment it is not necessary to mutate classes in UML state machines [38]. The system's logic is designed in the state machine diagrams, while the classes are merely containers for variables and diagrams. Many mutation operators for state/transition systems have been defined in the literature, see, e.g., [23, 55, 7, 8]. For product line tests, it is sufficient to consider only operators based on transitions, as these have the strongest impact on the behavior of the SUT.

Our mutation system provides the following mutation operators for the base model:

- Delete Transition (DTR)
- Change Transition Target (CTT)
- Delete Effect (DEF)
- Delete Trigger (DTI)
- Insert Trigger (ITG)

- Delete Guard (DGD)
- Change Guard (CGD)

These operators are selected and implemented such that they cover most of the state machine mutation operators which are found in the literature.

Mutation operators for the variability model: Now, we describe mutation operators according to the potential errors in the feature mapping identified in Section 4.1. We comment on hidden, invalid, and equivalent mutants where appropriate.

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 variability 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.

There are other possible domain model mutation operators, which, however, are of minor importance. For example, inserting superfluous mappings does not seem to be necessary: it remains unclear which and how many UML elements should be selected for the mapping. In most cases, such an operation will lead to invalid mutants.

In a basic mutation experiment, each operator is used for each applicable model element exactly once. To get an idea of the complexity, consider the following numbers: For our eShop example, which has about 10 features and 44 state chart elements (states and transitions), after filtering this yields 152 product line mutants, which are resolved into 574 mutated products. Of course, it is possible to generate an arbitrarily higher amount of mutants by repeated

application and combination of mutation operators. However, it is doubtful whether this brings about new insights.

4.3 Evaluation

To evaluate the above ideas, we performed a mutation analysis on three example product lines, including our eShop example. With our prototypical SPLTestbench, we developed and adapted the domain models for the examples. Then we derived a test suite for each example by applying the techniques described in section 3. The external test generator was set to transitions coverage. For the eShop example, this generated 13 test cases with a total of 103 test steps. From the generated test suite, the SPLTestbench selected 4 variants for testing and resolved mutated product models from the mutated domain model.

Since there were no implementations available for our examples, we decided to generate code from the mutated product models and run the tests against them. (Note: Generating code *and* tests from the same basis for testing the code is not advisable 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.) To derive our implementations, we implemented a small code generator transforming 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.

For the eShop example, on the 574 mutated products there were in total 1855 tests executed. All of the transformations above and the mutation system are integrated into the SPLTestbench. The test execution is fully automatic and is done within a few minutes.

For each mutation operator we measured the mutation score as the quotient of detected mutants by generated mutants. The results for all mutation operators from the previous subsection is shown in Table 2. In the last column of this table, we show the weighted average for each mutation operator. The weighted average takes the number of product line mutants for each example and operator into consideration.

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 [72, 44]. 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 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

Table 2: Mutation scores per operator (and number of products)

Op.	eShop	Example #2	Example #3	Average
DMP	0% (4)	0% (5)	0% (8)	0%
DME	0% (14)	0% (8)	0% (21)	0%
IME	75% (4)	40% (5)	50% (8)	53%
SWP	100% (4)	60% (5)	63% (8)	71%
CFV	100% (4)	100% (5)	87.50 (8)	94%
DTR	89% (28)	84% (19)	63% (19)	80%
CTT	64% (28)	63% (19)	37% (19)	56%
DEF	100% (16)	82% (17)	62% (13)	83%
DTI	83% (23)	100% (13)	94% (17)	91%
ITG	21% (24)	28% (18)	17% (18)	22%
DGD	0% (1)	43% (14)	50% (2)	41%
CGD	100% (2)	69% (48)	90% (10)	73%

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 [74]. 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.

5 Test-driven Product Sampling

In a large product line with one hundred or more features, there might be up to $2^{100} \simeq 10^{30}$ possible products. Clearly, it is impossible to build and test all of these products. In this section, we address the question which products of a large product line should be selected for testing. In practical applications, the choice is often done by some heuristics (e.g., "test one minimal and one maximal product with respect to the number of features", or "test those products which the customers are most likely to order").

However, it is not clear whether the error detection capabilities of these heuristic choices are acceptable. From a systematic point of view, there are different criteria on the selection. One possibility is to select a minimal number of different configurations such that each test case is executable in at least one product. This minimizes the amount of tested products, and thus reduces the testing effort. Other possibilities are to minimize or maximize the size of product configurations in terms of activated features, or the diversity of configurations. The question is which method offers an acceptable error detection rate in relation to the testing effort.

Subsequently, we address the question how the sampling of configurations

from test cases affects the effectivity of product line testing. Using our domain-centered test design methods described in section 3.1, the domain test cases can be used to sample product configurations for testing. We describe experiments to measure the effect of different sampling criteria on the error detection rate. We consider the following criteria:

- Sampling as many or as few configurations as possible,
- sampling large or small products in terms of activated features, and
- sampling diverse or random products.

Our expectation is that selecting many, large, and diverse products yields the highest fault detection capability and test effort. Firstly, testing many products should decrease the chance of missing faults which are specific to some particular combination of features. Secondly, selecting large products for testing should expose faults arising from feature interactions. Thirdly, testing diverse products, rather than testing similar products, should increase the error detection capability.

In order to confirm these expectations, we develop a method for product selection from domain test cases. This method allows us to perform experiments with different sampling criteria. The assessment of error detection capabilities is done by the mutation system presented in the previous section. We analyze the results for all three examples of the previous section, including the eShop, and give recommendations for practical use.

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 optimization criteria for sampling configurations from such generic test cases.

5.1 Sampling Configurations from Generic Test Cases

In this subsection we address the question how to sample product configurations from a test suite. In domain-centered test design, an incomplete product configuration is created and stored with each test case during test generation. Product configurations can be sampled from these incomplete configurations by assigning a concrete value (*true* of *false*) to each undecided feature. Since there is a choice to make, we can apply an optimization criterion. The sampling should be such that the likelihood of detecting faults in the product line should be maximal, while the test effort is kept reasonably low. Since test cases were selected according to a model coverage criterion, every test case should be executable at least once. Moreover, in order to keep test efforts low, test cases should not be executed more than once.

Due to the fact that feature models are representable as propositional formulas, the problem of sampling configurations can be viewed as a boolean satisfiability problem. In order to search for an solution to an optimization criterion, we represent it as a constraint satisfaction problem. First, we describe the problem of sampling a product configuration from an incomplete configuration as a

propositional formula. On this basis, we then define minimization and maximization criteria to sample large, small, few, many, and diverse variants from a set of incomplete configurations.

5.1.1 Random Sampling

The first challenge is to complete a given incomplete product configuration. This problem can be solved straight-forward by a SAT solver. We use the Java constraint programming solver JaCoP [42] which is based on the DPLL SAT algorithm. We declare for each feature f in F an new variable v_f with an appropriate domain. The domain varies depending on the feature's assignment:

- $f = true$ then the domain of v_f is $\{1\}$
- $f = undecided$ then the domain of v_f is $\{0, 1\}$
- $f = false$ then the domain of v_f is $\{0\}$

An additional constraint is the propositional formula representing the feature model (see Figure 2). JaCoP is now able to make a random assignment to each undecided feature and check the solution for consistency with the feature model. This method can be repeated individually for every incomplete configuration in the given test suite.

5.1.2 Targeted Sampling

The above procedure yields a set of product configurations which may not be optimal. From each test case, a separate product is sampled. There is no systematic check whether a product is appropriate for several test cases. In the following, we define and use optimization criteria for this purpose. We consider the following criteria:

- Few/many configurations,
- small/large configurations (in terms of selected features),
- diverse/random configurations, and
- combinations of these criteria.

The optimization should be such that for every test case there is a valid configuration, and for every sampled configuration there is an associated test case. Thus, for a test suite with m test cases, we will not sample more than m products. For an automated sampling of products, we now formulate these criteria as boolean optimization problems, which can be solved automatically by a constraint solver.

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 exactly once. From a product configuration with features $F_n : f_1, \ldots, f_k$, in JaCoP we define the binary number

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

where v_{f_i} is the variable associated with feature f_i (see above). For a test suite with m test cases, we define the set Z as the collection of all b_i, where $1 \leq n \leq m$:

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

With such an encoding, we can ask the constraint solver to find a variable assignment for the v_{f_i} respecting additional criteria. To optimize the amount of distinct configurations, we define a cost function as the cardinality of Z:

$$cost_a = |Z|$$

Now we can ask JaCoP to find a solution which minimizes or maximizes this cost function. The resulting set of configurations has between 1 and m elements.

b) Optimizing the Size of all Configurations. For minimizing the size of a set of configurations, we define its size as the sum of all selected features in all configurations. For constraint solving, we interpret 'selected' as numerical value '1' and 'deselected' as '0' respectively. In JaCoP, we define the size of a single product configuration as follows:

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

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

$$cost_b = \textstyle\sum_{n=1}^{m} s_n.$$

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

c) Optimizing the Diversity of Configurations Similar to a) and b), 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 to obtain the 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 $1 \leq i \leq k$:

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

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:

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

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.

It should be noted that this approach does not maximize the amount of sampled product configurations, but their diversity. Inherently, this leads to a solution with few unique product configurations, although there might be solutions with more product configurations but less diversity. A possibility to increase the amount of product configurations is to combine the two criteria diversity and maximization.

d) Combinations Now we look at combinations of the previously defined constraints, e.g., many with large and diverse configurations. All the above criteria can be combined, with the exception of

- few with many product configurations, and
- small with large product configurations.

Of course, costs cannot be summed up directly if the optimization targets are conflictive, 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.

5.2 Evaluation

For evaluation of the proposed methods, we extended our SPLTestbench to support optimization-driven product sampling from incomplete product configurations. The sampling process is supported by Eclipse plug-ins to configure the sampling and start the sampling process as depicted in Figure 17.

As an example to evaluate the optimization criteria, we use our eShop product line. Test case generation from the variability model was discussed in section 3.1. Here, we use the test cases generated with the step-by-step method to measure the effects of sampling for different optimization criteria on the test case's error detection capability. We compare the results to include/exclude-all-features and pair-wise combinatorial testing from application-centered test design.

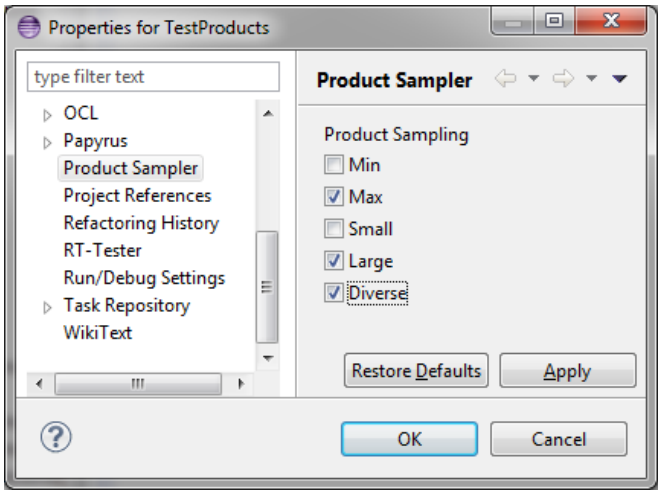

Figure 17: Project options in SPLTestbench for sampling products.

For assessing the error detection capability, we use the product line mutation framework presented in section 4.2.1. For our 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 variability model. However, we did not apply the operators DMP, DME and ITG. As discussed before, mostly these operators add superfluous behavior, which cannot be detected by model-generated test suites.

We sampled configurations for all sampling criteria in isolation and for the four combinations Few+Small+Div, Few+Large+Div, Many+Small+Div, and Many+Large+Div. As a comparison value, we also consider the numbers for AC test design. The results can be seen in Table 3. In this table, the number of automatically sampled configurations is shown in the first column. This number is responsible for much of the testing effort. The second column shows the number of test executions. For a given DC sampling, every test case is executed exactly once. In contrast, for AC test design, test cases are generated individually for each variant. Hence, here we gain more test cases and subsequently more test executions. The third and fourth column give the achieved mutation score on the base model and on the variability model, respectively.

The test suite contains 13 test cases with 48 test steps. Since this test suites is the same for all samplings, the procedure indeed assesses the impact of the different sampling criteria on the test suite's error detection capability. As can be seen from the data, the error detection capability of the test suite varies with the applied sampling criterion. The mutation scores for the base model exposes only marginal variance (76.5%-77.6%). This is due to our test design approach where all-transitions as test selection criterion is applied to base model and product models.

More variance is found in the error detection capability for errors in the

Table 3: Results of product sampling

Isolated DC samplings	configurations	executed test steps	Mutation score base model	Mutation score variability model
Few	1	48	78%	33%
Many	13	48	77%	92%
Small	4	48	77%	92%
Large	1	48	78%	33%
Div	7	48	78%	67%
Combined DC samplings				
Few+Small+Div	4	48	77%	92%
Few+Large+Div	4	48	78%	67%
Many+Small+Div	10	48	77%	92%
Many+Large+Div	10	48	78%	67%
AC samplings				
Include/exclude-all	2	135	77%	67%
Two-pair	7	486	78%	100%

variability model: the highest error detection capability of isolated criteria are achieved by maximized amount of products (Many) and small products (Small). In contradiction to our expectations, sampling large products reduces the error detection capability. This may be due to the fact that only few products are being sampled and tested. Though the criteria 'Many' and 'Small' have a comparable error detection capability (91.7%), 'Small' is more efficient in terms of testing effort, since more products have to be provided.

For combined sampling criteria the variance of the mutation scores declines (66.7%-91.7%). The highest scores in the group of combined criteria are achieved by the following two combinations: maximized amount with small and diverse products (Many+Small+Div) and minimized amount with small and diverse products (Few+Small+Div). For these, efficiency is higher for Few+Small+Div than for Many+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.

The DC scores lie in between the two presented AC criteria. However, AC testing needs much more resources: For a comparable score, two-pair testing needs ten times more test steps than any DC test.

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. We performed a similar analysis for the two other examples mentioned in the previous section, with comparable results.

6 Assignment of Product Line Test Cases

In section 3, we considered different ways of generating test suites from domain models. Subsequently, we used these tests for product sampling and mutation analysis. Thus, the domain model was primarily used for test generation. In model-based engineering, either products or tests can be derived from a given model. In this section, we consider the setting that domain models are used for deriving implementations, whereas test cases are developed separately and manually. This setting describes the actual situation in most medium and large companies, where testing and implementation is done in different departments. In fact, in this section we do not care about the test case development process; we just assume that there is a significant body of test cases available, and that these test cases were developed independently from the products. This situation is found in all companies transitioning to a model-based software engineering process, which already have a large number of legacy test cases.

For such a setting, the question is, which test cases should be executed with which products? Certainly, there are syntactic criteria excluding a test case to be applicable for a certain product. For example, a test case might address an interface which is not present in the product. In such a case, the test cannot be executed with this product.

However, even if all syntactic criteria are being met, a test case might not be relevant for a certain product. For example, the specification may not determine whether the behavior tested by t_1 is required or not for a product p. That is, p may or may not show the behavior which is tested by t_1. In such a case, no new information about the conformance of the product would be obtained by executing the test case. Thus, it does not make sense to execute t_1 with the SUT p.

As another example, consider a test case t_2 for an advanced feature which is not present in a basic product p_b. Here, we would expect the test execution of t_2 with SUT p_b to fail. It may also be advantageous to execute such a *negative test case* in order to check that the basic product has no superfluous behavior.

Closely following ideas presented in [39], in this section we show how the product model can be used to determine the expected outcome of a test case to a product. The exposition is based on our previous work [40], where we give formal definitions, an extended example, and a detailed description of the algorithm.

6.1 Colored Test Cases

We use a three-valued coloring scheme to capture what design decisions have been made in the domain model and configuration with regards to a specific product: A test case is colored green if it reflects a behavior that is expected from this particular variant of a software product line. It is colored red if the variant should not allow the described behavior. Finally, a test case is colored yellow if the respective behavior is neither required nor disallowed by the specification of the variant. This can happen, e.g., if the specification is

non-deterministic or incomplete.

Intuitively, green test cases reflect *required* and red test cases *forbidden* behavioral properties of the specification. Yellow tests mirror *open design decisions*, i.e., properties which are not (yet) decided in the specification. Since the color of a test case depends on the base model as well as the variability model and its resolution for a particular variant, the same test case can be green for one product, but red or yellow for another one.

The three-valued scheme is similar to the definition of an incomplete configuration in domain-centered test design (subsection 3.1.2). The difference is that an incomplete configuration determines for each test case the set of products for which it is applicable, whereas the coloring determines for each test case and product whether it is a required, allowed or forbidden behavior.

In order to make these notions more precise, we briefly recall the UML stipulations on the execution of a model: In UML state machines, a transition $e[g]/a$ may have a *trigger* e, can be restricted by a *constraint* g, and can invoke a *behavior* a. The UML superstructure explains: "A trigger specifies an event that may cause the execution of an associated behavior. An event is often ultimately caused by the execution of an action, but need not be. [...] Upon their occurrence, events are placed into the input pool of the object where they occurred [...]. An event is dispatched when it is taken from the input pool and is processed by the classifier. At this point, the event is considered consumed and referred to as the current event." [54, p. 471sq.]. The constraint language is not specified in UML; "a constraint is a condition or restriction expressed in natural language text or in a machine readable language for the purpose of declaring some of the semantics of an element" [54, p. 57]. A behavior is a consequence of the execution of an action by some related object. The behavior invoked as the effect of a transition may contain several actions, e.g., calling an operation, changing variable values, or causing the occurrence of some event.

To define the notion of a test case, we fix a *test signature* Σ. In our approach, we assume that Σ is a subset of the occurrences and dispatches of events which are contained in the product model. In this case, we say that the test case is applicable to the product model.

Additionally, we require that stimuli can be sent to the SUT, for example pressing a button, from the outside. We represent this as the artificial entity *tester*. Events sent from the tester are called *input* events for the SUT, other events are called *output* events for the SUT. Intuitively, elements of the signature are the only events which can be "noticed" by the test case; events not in the signature are "invisible". A *test case* formally is a finite sequence of elements from the test signature Σ.

In order to fix the color of a test case, we assume that there exists a function *enabled* assigning to each configuration of a UML model the set of elements from Σ which may occur next. That is, an event $e \in \Sigma$ is in $enabled(c)$, if upon its occurrence there is a sequence $c_0 \xrightarrow{e_1} c_1 \xrightarrow{e_2} \cdots \xrightarrow{e_n} c_n$ of transitions such that $c_0 = c$ and $e_n = e$, and for all $i < n$ it holds that $e_i \notin \Sigma$. In this case, we say that c_n is *reached* from c by e. A useful assumption, which is, however,

not required for our theory, is that input events from the tester are enabled at any time. For an event $e \in \mathit{enabled}(c)$, we say that it is *testable* at c, if it is either an input event, or it is an output event and it is not the case that some output event $e' \in \Sigma$ different from e is enabled in c. Intuitively, if an output e is testable at c, it is the output from Σ which must occur next, if any.

Since UML contains semantic variation points, the function *enabled* is tool-dependent. In particular, UML does not impose an ordering on events in the event pool; furthermore, the mechanism for determining the behavior to be invoked as a result of a call operation is unspecified, and it is a semantic variation point whether one or more behaviors are triggered when an event satisfies multiple triggers. The UML allows an event to be dispatched in a configuration even if there is no transition taking this event as a trigger; in such a situation, this event is discarded.

The *color* of a test case $T = \langle e_1, \ldots, e_n \rangle$ in the signature Σ with respect to a product model is a value from $\{\mathit{green}, \mathit{red}, \mathit{yellow}\}$, such that

- $\mathit{color}(T) = \mathit{green}$ iff for all $k < n$ and every sequence $\langle c_0, c_1, \ldots, c_k \rangle$ of configurations such that c_0 is an initial configuration, and c_i is reached from c_{i-1} by e_i for all $1 \leq i \leq k$ it holds that e_{k+1} is testable at c_k;
- $\mathit{color}(T) = \mathit{red}$ if there is no sequence $\langle c_0, c_1, \ldots, c_n \rangle$ of configurations such that c_0 is an initial configuration, and c_i is reached from c_{i-1} by e_i for all $1 \leq i \leq n$; and
- $\mathit{color}(T) = \mathit{yellow}$, otherwise.

In other words, a test case is green if it can be observed in all possible executions of the model triggered by this test case. It is red if there is no possible execution where it can be observed. It is yellow if some executions show the behavior and others do not.

Note that our definition enforces that for each test case $T = \langle e_1, \ldots, e_n \rangle$ for which $\mathit{color}(T) = \mathit{green}$ there is at least one sequence $\langle c_0, c_1, \ldots, c_n \rangle$ such that c_0 is an initial configuration, and for all $1 \leq i \leq n$, configuration c_i is reached from c_{i-1} by e_i. That is, green test cases must indeed be observable in the system's executions.

For example, consider a minimal eShop where no search function is available. For such an eShop, the following test case would be green:

> *openProductCatalog* → *listProducts* → *productDetails* → *showProductDetails* → *addProductToCarts* → *proceedToCheckout* → *showSummary* → *selectPaymentMethod* → *startPayment* → *selectBankAccount* → *proceedPayment* → *validatePayment* → *paymentComplete* → *valid* → *processOrderAndConfirm*.

The following test case, which tests for the search feature to be absent, would be red:

> *openProductCatalog* → *listProducts* → *searchFor* → *listProducts*.

Here are some simple properties of our coloring.

- An empty test case (consisting of no events at all) is always green.
- A one-element test case is green if its event is enabled and testable in all initial configurations; it is red, if the event is initially not enabled; and yellow, if it is enabled in some initial configuration but not testable.
- Any initial fragment of a green test case is green; any extension of a red test case is red.
- If a state is non-deterministic, e.g., from state s there are transitions $/a$ and $/b$, then the test cases $\langle a \rangle$ and $\langle b \rangle$ are yellow, since $enabled(s) = \{a, b\}$, but a is not testable at s. Assuming that the test signature is $\{a, b, c\}$, the test case $\langle c \rangle$ is red, since neither $/a$ nor $/b$ produce c and thus c is not enabled in s.
- Consider a situation where the effect of a transition invokes a behavior expression including an operation for which only its signature is known (e.g., a transition $/obj.op(arg)$, where the operation op is declared in the class diagram, but the return value of op for a given argument arg is not specified). Then test cases using such a transition will be yellow, as all possible return values are enabled in the state machine; however, the test case contains only a specific one.

The test verdict (pass or fail) for a test is assigned by executing a green or red test case with a concrete product. A product passes a test suite, if it behaves as expected, i.e., if it exhibits the behavior described in all green test cases and deviates from the behavior described in all red test cases. Yellow test cases do not contribute to the detection of faults, thus we do not execute them.

6.2 Automated Test Coloring via Model Checking

For automating the above defined test coloring procedure for a given materialization of a product line and a test case, we use the tool HUGO, which is a UML model translator for model checking [41]. In particular, HUGO resolves the UML's semantic variation points mentioned above in a particular way thus also fixing the *enabled* function: The event pool is implemented as bounded event queues; since inheritance is not supported, the dispatching algorithm becomes straightforward, as no overloading has to be considered; only a single, non-deterministically chosen behavior can be triggered by a given event; and events which trigger no outgoing transition in a state configuration are silently consumed.

HUGO translates both the materialization and a test case over a test signature into Promela, which is the input language of the model checker SPIN [35]; syntactically, it is first ensured that the test signature indeed is a subset of the possible event occurrences and dispatches of the materialization. The resulting

encoded product model shows instrumentation for observing all events: Whenever an event occurs or is dispatched in the product model, a notification is sent out which can be used by an observer. The test case results in an automaton process sending those events to the system which occur at the *tester* and also reacting to those produced by the system: If an event of the test case is observed, the test case automaton proceeds; if any other event which is part of the test signature happens, the automaton goes to a dedicated *failure state*; events not present in the test signature are ignored. After successful observation of the last entry of the test case sequence the automaton enters a dedicated *final state*.

We have implemented the above translations and colored a number of test cases. The results often are surprising, the calculated color is different from the intuition. Fortunately, due to the counterexample mechanism of the model checker, we were able to analyze this discrepancy. It turned out that in all cases the intuition was mistaken; test cases which we assumed to be green (required) or red (forbidden) really were yellow (allowed but neither required nor disallowed). This shows the viability of our approach.

In this section, we have presented a theory and prototypical implementation for test case assessment in the model-based development of multi-variant systems. To our knowledge, this is the first treatment of the subject in the context of UML-based software development.

We deal with both positive (green) and negative (red) test cases, and introduce a third color (yellow) for test cases whose outcome is not determined with a given product model. This means that it is needless to execute them with products based on this model. Our approach thus allows to assess and select those test cases from a universal test suite which are relevant for a given product. It would be a straightforward extension to extend our coloring to sets of products defined by incomplete configurations as defined above. Additionally, lifting our approach to logical, abstract test specifications in the universal test suite would be of interest. This would have to include different colorings for the different concretizations. For conciseness, we do not pursue these extensions further here.

Our test case coloring theory is well-suited for testing deterministic reactive systems under test, where the response functionally depends on the provided stimuli. In the UML specification, it can deal with indeterminacy caused by semantic variation points and nondeterminism by under-specification and open design decisions, by assigning the respective test cases the color yellow. The theory excludes to formulate test cases for systems which are inherently nondeterministic. This can be the case, e.g., for a network of cooperating devices with unpredictable message delays. To deal with such a situation, we are investigating trees and UML interactions as test cases and the relation to the testing theory of de Nicola and Hennessy [21].

7 Related Work

In this section, we give some references to related work. Due to the large volume of available literature on model-based testing and software product lines, this is necessarily only a small selection.

Specification of software product lines Software product families have already been proposed by Parnas in 1976 [58]. There are annual conferences dealing with product line engineering, e.g., the International Systems and Software Product Line Conference SPLC. An introductory textbook on this topic is by Pohl et al. [62].

A standardization attempt which incorporates many of the modeling concepts presented in this article has been the OMG initiative for CVL, the common variability language [19]. Unfortunately, due to patent disputes, this standardization could not be completed. Nevertheless, most of the concepts are now generally acknowledged and implemented by current product line engineering tools.

Model-based test generation 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 [3, 12, 53] and model-based testing [13, 70, 75]. modeling languages like the UML have been often used to create test models. For instance, Abdurazik and Offutt [55] automatically generate test cases from state machines.

Testing is an important topic also in the software product line literature. Systematic reviews can be found in [22], [51], and [45]. These surveys show that much of the work on SPL testing is concerned with the question of selecting products for testing (see below).

Feature modeling Feature models are commonly used to describe the variation points in product line systems. Our feature modeling language is similar to other standard feature languages, e.g. [14]. There are several approaches to apply feature models in quality assurance. For instance, Olimpiew and Gomaa [56] deal with test generation from product line systems and sequence diagrams. 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 [48] shows the importance of a well defined software product line testing process. Pohl and Metzger [63] emphasize the preservation of variability in test artifacts of software product line testing. Lochau et al. [47] 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.

Mutation analysis Mutation analysis for behavioral system models, e.g. finite state machines, is a well-established field. Fabbri et al. introduced mutation operators for finite state machines in [23]. Belli and Hollmann provide mutation operators for multiple formalisms: directed graphs, event sequence graphs [7], finite state machines [55], and basic state charts [8]. 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 [1].

Mutation analysis for software product lines has not received as much attention yet. In [67] Stephenson et al. propose the use of mutation testing for prioritizing test cases from a test suite in a product line environment. Henard et al. proposed two mutation operators for feature models based on propositional formulas in [34]. 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.

Sampling of products 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 [61, 57]. Still, the test effort is high, since feature interactions are selected for testing where no behavioral interaction is present. Baller et al. focus on heuristics for minimization of the test suite for the base model [5].

Similar to the notion of incremental test design, Beohar et al. propose spinal test suites [9]. 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.

Lochau et al. present incremental test design methods to subsequently test every specified behavior [47]. Here, configurations are sampled as needed to achieve the next test goal. The result is a set of test cases where each one is limited to a single product configuration. Our optimization criteria for sampling configurations allow for more flexibility since they use generic test cases.

Test case assignment Using three-valued logics for test assessment is an old idea. In 1983, Butler [15] uses the third value to denote either a missing or an incorrect test result. Zhao et al. [76] propose a symbolic three-valued logic analysis to improve the precision of static defect detection. The standardized testing language TTCN-3 [25] allows several verdicts for a test case such as *pass*, *fail*, *inconc*, *none*, and *error*.

To our knowledge, we were the first to propose a theory for a three-valued evaluation of test cases with respect to formal specifications [36]. In [39], we extended this theory to software product lines based on UML models.

Bertillon et al. [10] use a notation based on natural language descriptions of requirements to define test cases for product lines. The resulting test specification is generic in the product, and a set of relevant test scenarios for a customer specific application can be derived from it. This work complements our colouring method, since we assume that the test suite is designed separately.

8 Future Developments

Predicting the future of some field always is a risky enterprise, since trends in computer science tend to be short-lived. There are, however, a few clear development tendencies in industry and science which are sketched in this section.

Firstly, we expect product line development methods to become more "main stream" in the engineering of industrial software. Indications of this are, e.g., the availability of tools like pure::variants [11], which interacts with popular development environments such as Mathworks Simulink, IBM Rational DOORS and Rhapsody, and Sparx Systems Enterprise Architect. As another example, FeatureIDE [69] is an open-source tool which tries to "bring all phases of the development process together, consistently and in a user-friendly manner". It supports multiple paradigms for modeling variability such as annotation-based, delta-oriented, and aspect-oriented programming.

Secondly, there is a trend to standardization. A fist initiative was CVL, the common variability language mentioned above [33]. Since this endeavor has been discontinued, other ways of standardization are being explored. The Variability Exchange Language [29] provides a generic data exchange format for variant management. It is intended to be standardized within OASIS in the near future. The Variability Exchange Language defines a generic API (VariabilityAPI) that allows variant management tools to communicate with artifacts such as model based specifications, program code, or requirements documents. Since the communication is via a standardized interface, the number of tool adapters is significantly reduced.

Thirdly, the availability of concepts, standards, and unified APIs enables tool providers to focus on the development of features to facilitate the use of their tools. To minimize switching times between different tools and accelerate the overall workflow, a general trend is to integrate tools and support a growing number of other development environments. In particular, for product line development the link to software testing is being strengthened. In this paper, we discussed the selection of products for testing, and the selection of test cases for products. Integrated tool suites would support finding relevant artifacts for re-use and identifying linked artifacts from other projects. A related research question is how to automatically extract re-usable information from development artifacts.

Continuing on the scientific side, of current scientific interest is the refactoring and evolution of software product lines. A question here is how changes in a product line affect subsequent artifacts like implementation models, test cases, and validation and verification documents [2].

Another trend in the research community is to consider models at runtime. With variability models, this would allow for runtime variants and dynamic reconfiguration of systems [46]. For example, one could imagine that additional features, e.g. advanced driver assistants in a car, are loaded according to current demands. Even though these techniques have a high potential, there are numerous safety and security problems to be solved.

Finally, a frequent reconfiguration leads to adaptive and self-learning systems. In fact, dynamic software product lines can be seen as a way to realize adaptive behavior in open contexts. Scientific challenges include the modeling and handling of uncertainties, the integration of feedback in the evolution, and the learning of adaptation rules [66].

9 Summary and Conclusion

In this article, we have collected and summarized our recent research on model-based testing for software product lines. With the example application of an eShop product line, we have described how to model a software product line: The feature model describing variation points, the base model describing the functionality, and the variability model which determines which features are realized by which base model elements.

Then, we showed how to automatically derive test suites for product lines from variability models. We distinguished between application-centered and product-centered test generation, and analyzed in detail two approaches of constructing a standard UML model from variability information. A comparison of these approaches showed that both approaches lead to very different test suites. Although they also cover all transitions of the model, domain-centered approaches lead to test suites with significantly fewer test steps. Thus, in terms of test steps and the amount of products to test, domain-centered test design scales better with respect to the size of the system.

We then analyzed the error detection capabilities of generated test suites. For software product lines, there are more sources where errors can occur than in ordinary software models. We defined mutation operators for domain models, and measured the detection rate on several examples. The results confirm that operators which add superfluous behavior to a model are hard to detect by model-based testing techniques.

Whereas usually in software engineering test suites are built for existing or envisioned software artifacts, in product lines it may be more adequate to build a software artifact for a given test suite. We considered the problem which products should be selected as representatives for a product line, to maximize the benefits from testing within given resource bounds. We found that domain-centered test generation can be very efficient for detecting errors in variability model, and that sampling small and divergent products is better than sampling large ones.

Finally, we looked at a setting where a test suite is developed independently from the product development in a model-based process. This is especially

relevant for legacy systems, where a huge set of test cases is already existing, whereas products are restructured along the line. Here, the question is which test cases are applicable for which products. We reduced this problem to a classical model-checking problem and showed how to solve it with existing tools.

While all these contributions can be considered as significant steps, the overall goal of a unified theory for quality assurance of software product lines remains unreached. The main challenge of software product line validation, as compared to "normal" software validation, is the significantly higher complexity. Due to the fact that a product line incorporates a large number of potential products, quality assurance methods must strive to handle many or all of them at once. With our prototypical SPLTestbench, we provide an integrated, open tool environment for this task. In the future, we plan to extend it with capabilities for formal verification and transformational development of software product lines.

References

[1] Bernhard K. Aichernig, Florian Lorber, and Dejan Ničković. Time for Mutants — Model-Based Mutation Testing with Timed Automata. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Margus Veanes, and Luca Viganò, editors, *Tests and Proofs*, volume 7942 of *Lecture Notes in Computer Science*, pages 20–38. Springer Berlin Heidelberg, Berlin and Heidelberg, 2013.

[2] Vander Alves, Rohit Gheyi, Tiago Massoni, Uirá Kulesza, Paulo Borba, and Carlos Lucena. Refactoring product lines. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, GPCE '06, pages 201–210, New York, NY, USA, 2006. ACM.

[3] Paul E. Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, New York and NY and USA, 2008.

[4] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer, 2013.

[5] Hauke Baller, Sascha Lity, Malte Lochau, and Ina Schaefer. Multi-objective Test Suite Optimization for Incremental Product Family Testing. In *Proc. ICST 2014*, pages 303–312, 2014.

[6] Don Batory. Feature models, grammars, and propositional formulas. In *Proceedings of the 9th international conference on Software Product Lines*, SPLC'05, pages 7–20, Berlin and Heidelberg, 2005. Springer-Verlag.

[7] Fevzi Belli, Christof J. Budnik, and Lee White. Event-based modelling, analysis and testing of user interactions: approach and case study. *Software Testing, Verification and Reliability*, 16(1):3–32, 2006.

[8] Fevzi Belli and Axel Hollmann. Test generation and minimization with basic statecharts. In Edward J. Delp and Ping Wah Wong, editors, *the 2008 ACM symposium*, volume vol. 5681, page 718, Bellingham and Wash and Springfield and Va, 2008. SPIE and IS&T.

[9] Harsh Beohar and Mohammad Reza Mousavi. Spinal Test Suites for Software Product Lines France, 6 April 2014. In Bernd-Holger Schlingloff and Alexander K. Petrenko, editors, *Proceedings Ninth Workshop on Model-Based Testing, MBT 2014, Grenoble, France, 6 April 2014*, volume 141 of *EPTCS*, pages 44–55, 2014.

[10] Antonia Bertolino and Stefania Gnesi. Use Case-based Testing of Product Lines. In *Proc. ESEC/FSE 2003*, pages 355–358. ACM, 2003.

[11] Danilo Beuche. pure::variants. In Rafael Capilla, Jan Bosch, and Kyo-Chul Kang, editors, *Systems and Software Variability Management*, pages 173–182. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[12] Robert V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley Longman Publishing Co., Inc, Boston and MA and USA, 1999.

[13] Manfred Broy, Bengt Jonsson, Joost-Pieter Katoen, Martin Leucker, and Alexander Pretschner, editors. *Model-based testing of reactive systems: Advanced lectures*, volume 3472 of *Lecture Notes in Computer Science*. Springer, Berlin, 2005.

[14] Michael G. Burke, Michal Antkiewicz, and Krzysztof Czarnecki. Feature-Plugin. In John M. Vlissides and Douglas C. Schmidt, editors, *OOPSLA 2004*, volume v. 39, no. 10 (Oct. 2004) of *ACM SIGPLAN notices*, pages 67–72, New York and NY, 2004. Association for Computing Machinery.

[15] J.T. Butler. *Relations among system diagnosis models with three-valued test outcomes*. IEEE,New York, NY, Jan 1983.

[16] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.

[17] Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. Coverage and adequacy in software product line testing. In *ROSATEA 2006*, pages 53–63. Association for Computing Machinery, Inc., New York and NY, 2006.

[18] José A. Cruz-Lemus, Ann Maes, Marcela Genero, Geert Poels, and Mario Piattini. The impact of structural complexity on the understandability of UML statechart diagrams. *Information Sciences*, 180(11):2209–2220, 2010.

[19] CVL Revised Submission.

[20] Krzysztof Czarnecki and Michal Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In Robert Glück, editor, *Generative programming and component engineering*, volume 3676 of *Lecture Notes in Computer Science*, pages 422–437. Springer, Berlin [u.a.], 2005.

[21] Rocco de Nicola and Matthew Hennessy. Testing Equivalences for Processes. *Theo. Comp. Sci.*, 34:83–133, 1984.

[22] Emelie Engstrm and Per Runeson. Software Product Line Testing — A Systematic Mapping Study. *Inf. Softw. Techn.*, 53(1):2–13, 2011.

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

[24] Marcela Genero, David Miranda, and Mario Piattini. Defining Metrics for UML Statechart Diagrams in a Methodological Way. In Gerhard Goos, Juris Hartmanis, Jan Leeuwen, Manfred A. Jeusfeld, and Óscar Pastor, editors, *Conceptual Modeling for Novel Application Domains*, volume 2814 of *Lecture Notes in Computer Science*, pages 118–128. Springer Berlin Heidelberg, Berlin and Heidelberg, 2003.

[25] Jens Grabowski, Dieter Hogrefe, György Rethy, Ina Schieferdecker, Anthony Wiles, and Colin Willcock. An introduction to the testing and test control notation (TTCN-3). *Computer Networks*, 42:375403, 2003.

[26] Matthias Grochtmann and Klaus Grimm. Classification trees for partition testing. *Software Testing, Verification and Reliability*, 3(2):63–82, 1993.

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

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

[29] Martin Groe-Rhode, Michael Himsolt, and Michael Schulze. The Variability Exchange Language, version 1.0. *Fnfter Workshop zur Zukunft der Entwicklung softwareintensiver, eingebetteter Systeme (ENVISION2020) im Rahmen der SE2015, Dresden*, 2015.

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

[31] Helmut Gtz, Markus Nickolaus, Thomas Roner, and Knut Salomon. *Modellbasiertes Testen: Modellierung und Generierung von Tests - Grundlagen, Kriterien für Werkzeugeinsatz, Werkzeuge in der Übersicht*, volume 01/2009 of *iX Studie*. Heise Verlag, 2009.

[32] Bill Hasling, Helmut Goetz, and Klaus Beetz. Model Based Testing of System Requirements using UML Use Case Models. In *1st International Conference on Software Testing, Verification and Validation, 2008*, pages 367–376, Piscataway and NJ, 2008. IEEE.

[33] Øystein Haugen, Birger Møller-Pedersen, Jon Oldevik, Gøran K. Olsen, and Andreas Svendsen. Adding Standardized Variability to Domain Specific Languages. In *SPLC 2008*, pages 139–148, Los Alamitos and Calif, 2008. IEEE Computer Society.

[34] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. 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*, pages 188–197, 2013.

[35] Gerard J. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2003.

[36] Temeshgen Kahsai, Markus Roggenbach, and Bernd-Holger Schlingloff. Specification-based testing for refinement. In *SEFM 2007 - Proc. 5th IEEE International Conference on. Software Engineering and Formal Methods , London*, 2007.

[37] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study, 1990.

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

[39] Alexander Knapp, Markus Roggenbach, and Bernd-Holger Schlingloff. On the use of test cases in model-based software product line development. In *Proc. SPLC 2014 - 18th International Software Product Line Conference. Florence*, 2014.

[40] Alexander Knapp, Markus Roggenbach, and Bernd-Holger Schlingloff. Automating test case selection in model-based software product line development. *IJSI*, 9, No. 2:153–175, 2015.

[41] Alexander Knapp and Jochen Wuttke. Model Checking of UML 2.0 Interactions. In *Proc. MoDELS 2006 Wsh.s*, LNCS 4364, pages 42–51. Springer, 2007.

[42] Krzysztof Kuchcinski and Radoslaw Szymanek. JaCoP - Java Constraint Programming Solver. Technical report, Lund University, 2013.

[43] Hartmut Lackner. *Domain-Centered Product Line Testing*. PhD thesis, Humboldt-Universität zu Berlin, 2017.

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

[45] Beatriz Prez Lamancha, Macario Polo, and Mario Piattini. Systematic Review on Software Product Line Testing. *Comm. Comp. Inf. Sci.*, 170:58–71, 2013.

[46] Malte Lochau, Johannes Bürdek, Stefan Hölzle, and Andy Schürr. Specification and automated validation of staged reconfiguration processes for dynamic software product lines. *Software & Systems Modeling*, 16(1):125–152, Feb 2017.

[47] Malte Lochau, Ina Schaefer, Jochen Kamischke, and Sascha Lity. Incremental Model-Based Testing of Delta-Oriented Software Product Lines. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Achim D. Brucker, and Jacques Julliand, editors, *Tests and Proofs*, volume 7305 of *Lecture Notes in Computer Science*, pages 67–82. Springer Berlin Heidelberg, Berlin and Heidelberg, 2012.

[48] John D. McGregor. Testing a Software Product Line, 2001.

[49] John D. McGregor, Linda M. Northrop, Salah Jarrad, and Klaus Pohl. Initiating Software Product Lines. *IEEE Softw.*, 19(4):24–27, 2002.

[50] Atif Memon. *Advances in Computers*, volume 86. Academic Press, 2012.

[51] Paulo Anselmo da Mota Silveira Neto, Ivan do Carmo Machado, John D. McGregor, Eduardo Santana de Almeida, and Silvio Romero de Lemos Meira. A Systematic Mapping Study of Software Product Lines Testing. *Inf. Softw. Techn.*, 53(5):407–423, 2011.

[52] H. Muccini and A. van der Hoek. Towards Testing Product Line Architectures. *Electronic Notes in Theoretical Computer Science*, 82(6):99–109, 2003.

[53] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, Hoboken and N.J, 3rd ed edition, 2012.

[54] Object Management Group. Unified Modeling Language Superstructure. Version 2.4.1. Specification, OMG, 2011.

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

[56] Erika Mir Olimpiew and Hassan Gomaa. Model-Based Testing for Applications Derived from Software Product Lines. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–7, 2005.

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

[58] David Lorge Parnas. On the Design and Development of Program Families. *IEEE Trans. Softw. Eng.*, 2(1):1–9, 1976.

[59] Jan Peleska. Industrial-Strength Model-Based Testing - State of the Art and Current Challenges. *Electronic Proceedings in Theoretical Computer Science*, 111:3–28, 2013.

[60] Gilles Perrouin, Sebastian Oster, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Traon. Pairwise testing for software product lines: comparison of two approaches. *Software Quality Journal*, 20(3-4):605–643, 2012.

[61] Gilles Perrouin, Sagar Sen, J. Klein, B. Baudry, and Yves Le Traon. Automated and Scalable T-wise Test Case Generation Strategies for Software Product Lines. In *ICST '10: International Conference on Software Testing, Verification and Validation*, pages 459–468, Los Alamitos and Calif and Piscataway and N.J, 2010. IEEE Computer Society and IEEE.

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

[63] Klaus Pohl and Andreas Metzger. Software Product Line Testing. *Communications of the ACM*, 49(12):78–81, 2006.

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

[65] Klaus Schmid and Holger Eichelberger. EASy-Producer: From product lines to variability-rich software ecosystems. In *Proceedings of the 20th International Systems and Software Product Line Conference*, pages 309–309. ACM, 2016.

[66] Amir Molzam Sharifloo, Andreas Metzger, Clément Quinton, Luciano Baresi, and Klaus Pohl. Learning and evolution in dynamic software product lines. In *Proceedings of the 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '16, pages 158–164, New York, NY, USA, 2016. ACM.

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

[68] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. FeatureIDE: An Extensible Framework for Feature-oriented Software Development. *Sci. Comp. Prog.*, 79:70–85, 2014.

[69] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. Featureide: An extensible framework for feature-oriented software development. *Sci. Comput. Program.*, 79:70–85, January 2014.

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

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

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

[73] Stephan Weißleder and Dehla Sokenou. ParTeG - A Model-Based Testing Tool. *Softwaretechnik-Trends*, 30(2), 2010.

[74] Stephan Weißleder, Florian Wartenberg, and Hartmut Lackner. Automated Test Design for Boundaries of Product Line Variants. In Khaled El-Fakih, Gerassimos Barlas, and Nina Yevtushenko, editors, *Testing Software and Systems*, volume 9447, pages 86–101. Springer International Publishing, Cham, 2015.

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

[76] Yunshan Zhao, Yawen Wang, Yunzhan Gong, Honghe Chen, Qing Xiao, and Zhaohong Yang. STVL: Improve the precision of static defect detection with symbolic three-valued logic. *APSEC*, pages 179–186, 2011.