Model-Based Test Design of Product Lines: Raising Test Design to the Product Line Level

Hartmut Lackner*, Martin Thomas†, Florian Wartenberg ‡, and Stephan Weiβleder†

*Humboldt-Universität zu Berlin
Rudower Chaussee 25, Berlin, Germany
Email: lackner@informatik.hu-berlin.de
†Fraunhofer FOKUS
Kaiserin-Augusta-Allee 31, Berlin, Germany
Email: [martin.thomas|florian.wartenberg|stephan.weissleder]@fokus.fraunhofer.de

Abstract—System quality assurance techniques like testing are important for high-quality products and processes. The effort for applying them is usually high, but can be reduced using automation. Automated test design is possible by using models to specify test-relevant aspects and by generating tests on this basis. Testing multiple variants of a system like, e.g., a product line of a German car manufacturer, results in a significant, additional effort. In this paper, we deal with model-based testing of product lines. We combine feature models that are used to describe product lines and models that are used for automated model-based test design. Our main contribution is the definition of a test generation approach on the product line level, i.e., that does not depend on resolving single product variants. Furthermore, we compare our approach to other test generation approaches and evaluate it using our tool chain SPLTestbench for some product line examples.

I. INTRODUCTION

Quality assurance is of vital importance for achieving and maintaining a high quality of all kinds of artifacts such as products and processes. There are many well-known quality assurance techniques such as reviews or testing. In industry, testing is the most wide-spread quality assurance technique. For safety-critical systems, testing and satisfying coverage criteria on the source code of the system under test (SUT) is recommended by corresponding standards such as the ISO 61508. The costs of testing range between 30 and 50 percent [1], [2] and for safety-critical systems up to 80 percent. As a result, increasing the testing efficiency is a major concern of all system engineering activities. A common means of increasing efficiency is automation of the test execution and the test design. In this paper, we are focusing on the automation of test design. Model-based testing [3], [4] (MBT) is considered one of the most mature automated test design techniques. Here, models are used to describe only the test-relevant aspects of the SUT, its environment, or both. There are several reports about the successful application of MBT in industrial practice [5]–[7].

System engineering faces several challenges. One of the most important ones is the management of reuse of system artifacts for similar products. This challenge is especially important for product line engineering [8], in which two aspects have to be combined: the end user perspective and the engineering perspective. End users wish to select their favorite product features and to also pay only for the selected ones. System engineers have to make sure that all offered combinations of product features in one product are correctly implemented. For keeping engineering effort at a reasonable level, reuse of system artifacts is of high importance. Managing both sides is a crucial and complicated task. Like for testing, models can be used to focus on the relevant aspects of product lines: feature models are one way of describing the product line features and their relations.

For testing single products of a product line (or even the whole product line), the information about the product line features has to be integrated with the test-relevant information. There are several possible ways to do so. Since the quality assurance effort for several elements of a product line corresponds more or less to the product of the number of products and the effort for a single product, the need for efficiency improvement is several times higher compared to the quality assurance of single products. In this paper, we deal with test design automation for product lines. We present the intuitive approach of resolving a representative set of products from the product line, introduce an improved approach that works on the product line level without resolving single products, and compare both. Both approaches are implemented in our prototype implementation SPLTestBench and evaluated using several examples and case studies.

The paper is structured as follows. Section II contains a short introduction to model-based testing and product line engineering. In Section III, we define both mentioned automated test design approaches for product lines. We present the case studies and the comparison of both approaches in Section IV. Subsequently, we show the related work in Section V and conclude in Section VI.

II. PRELIMINARIES

Our approach is founded on the two disciplines of model-based test design and model-based product line engineering. In this section, we provide the fundamentals of these disciplines and the establishment of a relationship between them.

A. Model-Based Test Design

Testing is managing the risk of undetected defects. Test design plays a crucial part in influencing the likelihood of
detecting defects by tests. The quality of test design is usually measured in terms of requirements coverage and code coverage. Requirements, however, are often described as continuous text. Formalizing them, e.g., by using formal models, helps in measuring requirements coverage and in deriving corresponding tests. A test model describes test-relevant aspects of the system under test (SUT).

In this paper, we apply state machines of the Unified Modeling Language (UML) [9] as models for automated test design. UML state machines are a common modeling language to express behavioral test models and they are supported by commercial and academic tools [10]–[12]. Typical test targets for state machines are criteria for control flow, data flow, conditional branching, use cases, and requirements [13]. Test designers create a test suite that fulfills a prescribed set of criteria to a certain degree. The resulting tests can be then be adapted to the interface of the SUT and executed.

B. Model-Based Product-Line Engineering

Individual customer expectations and the reuse of existing assets in a product’s design are two driving factors for the emergence of product line engineering: increasing the number of product features while keeping system engineering costs at a reasonable level. A higher number of possible product configurations usually attracts more customers than a smaller one. Engineering costs are often reduced by reusing system components. In terms of software engineering, a software product line (SPL) is a set of related software products that share a common core of software assets (commonalities), but can be distinguished (variabilities) [8].

Like many methodologies, SPL engineering can be supported by model-based abstractions such as feature models. Feature models offer a way to overcome the aforementioned challenges by facilitating the explicit design of global system variation points [14]. In consequence, variation points are not spread across one or multiple domain models anymore, but instead linked to one core of variability description.

A feature model has a tree structure in which a feature can be decomposed into sub-features (fig. 1). A parent feature can have the following relations to its sub-features: (a) Mandatory: child feature is required, (b) Optional: child feature is optional, (c) Or: at least one of the children features must be selected, and (d) Alternative: exactly one of the children features must be selected. Furthermore, one may specify additional cross-tree constraints between two features A and B: (i) A requires B: the selection of A implies the selection of B, and (ii) A excludes B: both features A and B must not be selected for the same product.

A key finding for our work in this paper is that feature models can be transformed into propositional formulas [15]. We employ this finding to automatically validate product configurations against a feature model. The outcome of such a validation can then be used to guide the human test designer or the automated test generator through a domain model.

Such a propositional formula can be constructed from a feature model as a conjunction of (a) implications from all child features to their parent features, (b) additional implications from parents to all their mandatory features, (c) implications from parents to groups, and (d) any additional constraints represented as propositional formulas [16]. Each literal in the resulting formula represents a feature. Since the empty configuration is considered correct under the above premises, we include the root feature as an additional conjoined literal to the formula. The assignment of a value to a literal indicates whether the corresponding feature is selected (true) or deselected (false).

For instance, the boolean formula for the eShop example in Figure 1 is:

\[
FM = \text{eShop} \land (\neg\text{eShop} \lor \text{Catalog}) \\
\land (\neg\text{eShop} \lor \text{Payment}) \land (\neg\text{eShop} \lor \text{Security}) \\
\land (\neg\text{Payment} \lor \text{CreditCard} \\
\lor \text{BankTransfer} \lor \text{eCoins}) \\
\land (\neg\text{Security} \lor (\text{High} \land \neg\text{Standard}) \\
\lor (\text{Standard} \land \neg\text{High})) \\
\land (\neg\text{Catalog} \lor \text{eShop}) \\
\land (\neg\text{Payment} \lor \text{eShop}) \\
\land (\neg\text{CreditCard} \lor \text{Payment}) \\
\land (\neg\text{BankTransfer} \lor \text{eShop}) \\
\land (\neg\text{eCoins} \lor \text{Payment}) \\
\land (\neg\text{Security} \lor \text{eShop}) \\
\land (\neg\text{High} \lor \text{Security}) \land (\neg\text{Standard} \lor \text{Security}) \\
\land (\neg\text{Search} \lor \text{eShop}) \land (\neg\text{CreditCard} \lor \text{High})
\]

Any variable assignment that satisfies the formula is a valid configuration for the product line. For instance, the following formula is a valid configuration for the feature model presented in Figure 1.

\[
P = \text{Catalog, Payment, BankTransfer,} \\
\neg\text{CreditCard, \neg\text{eCoins, \neg\text{Search, Security,} \text{High, \neg\text{Standard}}}
\]

Furthermore, a partial formula can be derived for individual features of the feature model. 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:

\[
F = \text{CreditCard} \land \text{High} \land \neg \text{Standard}
\]

We apply these findings to the product line-centered test design process presented in section III-B. There, we employ these formulas as guidance for the test design process.

C. Combining Model-Based Testing and Model-Based Product Line Engineering

A feature model captures the system’s variation points in a concise form. Its elements, however, are only symbols [17]. Their semantics has to be provided by mapping them to models with semantics. Such a mapping can be defined using an explicit mapping model. A mapping model consists of relations from feature model elements to domain model elements.

In our case, the domain model is designed in terms of a so-called 150% model. A 150% model contains every element that is used in at least one product configuration and, thus, subsumes every possible product [18]. This approach is also known as declarative mapping with negative variability [19]. In some cases, the 150% model can also be a product of the product line.

In the current version of our work, we map features only to transitions. Each mapping has a Boolean flag that indicates whether the mapped model elements are part of the product when the feature is selected (true) or unselected (false).

In Figure 2 we depict an excerpt of the product line model for our eShop example. The upper compartment shows a small part of the feature model introduced above. In the lower compartment, an excerpt of the UML state machine model is shown, where the system waits for the user to choose his preferred payment method. The dotted arrow line maps the feature CreditCard to the credit card payment process in the state machine. The notion of the mapping is as follows: If the eShop’s configuration includes the feature CreditCard, then the mapped transition must be present in the corresponding product. However, if the feature is not present, this transition is not part of the corresponding product and thus rendering the state for processing this payment method unreachable.

III. AUTOMATED TEST DESIGN FOR PRODUCT LINES

Tests for product lines face two challenges: covering a significant subset of products and covering a significant subset of the test focus on the product line level (in our case behavior of the 150% state machine). As the products share commonalities, some test cases may be applicable to more than one product.

Our product line model (PLM) consists of the previously introduced 150% domain model by means of UML state machines, a feature model explicitly expressing the product line’s variation points, and a feature mapping model that connects the two. This is essentially the view of domain engineering, where commonalities and variabilities are specified. Of course, this view is also applicable from the point of testing. From the domain artifacts, application artifacts can be derived, i.e. products and tests for them. Based on this, we define two approaches to automated test design for product lines as depicted in Figure 3: (i) product-centered (PC) and (ii) product line-centered (PLC).

The PC approach consists of first selecting a representative set of products (test models) and second generating test cases from each of these models. This approach is focused on satisfying a defined coverage on each test model, which also leads to an overlap of the resulting test cases. In contrast, the PLC approach directly applies the product line model for designing tests. This second approach is focused on the behavior defined at the product line level and does not focus on covering single products. Instead, there is still variability in the choice of the concrete products for which the test cases will be executed. Both approaches are investigated in more detail in the following paragraphs.
A. Product-Centered Test Design

We call any test design method that binds the variability by selecting products before the test design phase product-centered (see Figure 4). In this approach, product models are selected from the PLM according to a predefined feature coverage criterion first. Since testing all products individually is usually not feasible, these criteria are applied to gain a representative set of configurations from the product line for testing. Criteria like n-wise are presented in [20], [21].

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

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

B. Product Line-Centered Test Design

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

A major advantage of this approach is the focus on the test aspects of the product line model without deriving single products first. This approach maximizes the coverage of the test targets and thus should lead to high-quality test cases, while at the same time it classifies the products into sets that are most test-worthy due to their diverse behavior.

Though, using the 150% model as the only input for test design is not sufficient as it lacks information about the model element’s associations to the features and the features itself, since they impose additional constraints on the system’s behavior. Thus, the main challenge of this approach is to merge the PLMs into a single model artifact that a standard test generator will accept as valid input (fig. 5). We identified two solutions to this problem: 1) the step-by-step approach: sequentially excluding non-conforming configurations during test design time, and 2) the pre-configuration approach: choosing a valid configuration before designing test cases.

1) The Step-by-Step Approach: The key idea of the step-by-step approach is to sustain the variability until it becomes necessary to bind it. Therefore, at the beginning of each test case design the assumption is made that the test case is applicable to any valid product of the product line. Ideally, this holds true until test case design is finished. Since not necessarily all valid paths in the 150% model are applicable to all products, the test designer must take account of test steps that bind variability. A test step must bind variability if not all products do conform. Subsequently, the set of valid products for this particular test case must be reduced by the set of non-conforming products. Hence, each test case is valid for any of the remaining products that do conform.

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

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

$$G_t = \bigwedge_{i=1}^{n} p_{f, i, t}.$$  

We still have to incorporate the protection against repeated writings by substituting each feature literal in $G_t$ with the following expression: $(-f_e \lor (f_v == m_{f, t}))$, where $f_e$ is the control variable of feature $f$, $f_v$ is the feature variable of feature $f$, 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 $f_e$ 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 $t$:

```plaintext
if not f_e then
    f_e ← true
    f_v ← m_{f, t}
end if
```

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 6 shows the result of merging the product line model into a single UML state machine for the excerpt of the eShop introduced in Figure 2.

After test generation has finished, the valid configurations for a particular test case can be read from the product instance in each test case. Since the test cases may contain variability we obtain an abstract configuration from each test case. An abstract 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 (present/absent), the third stands for free. A free feature is not yet decided if it will be part of the configuration or not and thus expresses variability. Hence, each of the resulting test cases is applicable to 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 present/absent in the product. Features for which the respective control variable evaluates to false are free and thus not evaluated.

On the resulting product line test suite, new selection criteria for product selection can be applied. We think of coverage criteria of the following kinds:

- **fewest products** for exercising the full set of test cases,
- **most products** when each test case is executed once,
- **two-/n-products** for each test case in the test suite.

This list should not be considered complete and is open to further investigations. Furthermore, already established criteria, like $n$-wise, can be applied to this approach as well.

2) The Pre-Configuration Approach: In the pre-configuration approach, test targets are selected from the product line 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 and must not change before a new test case is created. Consequently, within a test case the test designer is limited to test targets that are specific to the selected product. Thus, satisfying all product line model test targets is a matter of finding the right configurations.

We implemented the pre-configuration approach by adding a signal to the very beginning of the 150% model for configuring the model. Therefore, we introduce a new state to the state machine, redirect all transitions leaving the initial state to leaving this new state, and add a transition between the initial state and the new state. Due to the UML specification the redirected transitions must not have 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, like in the step-by-step approach to limit the 150% model behavior to a behavior an actual product can conform to. However, control variables must not 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 \( f \) that are mapped to a transition \( t \) by a mapping \( m_{f,t} \) and construct a conjunction from these formulas:

\[
G_t = \bigwedge_{i=1}^{n} pf_{f_i,j}.
\]

For conjoining \( G_t \) with \( t \)'s guard, of course, the feature literals must be exchanged by the corresponding feature variables from the class. Figure 7 depicts the resulting merged product line model for this approach. As a result, no product can conform to any test case’s first step, since it was used to set the configuration and presents not the real system’s behavior. In a simple post-processing action this configuration step must be removed before testing can be performed.

With these transformations made to the 150% model, a test designer can already create test cases for the product line. However, each test case will be specific to one configuration. As pointed out, a generalization is possible, though we have not yet implemented it. The additional transformation steps consist of adding Boolean control variables for each non-core feature to the system class and initializing them with \( false \) and adding effects that set these variables to \( true \) when associated variability is affected. For every transition \( t \) that is mapped to a feature \( f \) by a mapping \( m_{f,t} \), the following code needs to be appended to \( t \)'s effect for every mapped feature \( f \).

```java
if not \( f_c \) then
  \( f_c \leftarrow true \)
end if
```

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.

IV. EVALUATION OF BOTH APPROACHES

In this section, we present the implementation of both PLC test design approaches. Furthermore, we apply them to three examples to compare them to PC test design.
and provides basic mappings between a Feature from a FeatureModel and a set of arbitrary UML Elements from an EMF UML model. Although for the case studies presented in this contribution, we mapped features solely to UML Transitions. Additionally, the Mapping class provides the flag featureValue that indicates whether the associated UML Elements must be present or absent when this feature is selected. The domain models are designed as 150% UML models in EMF with Papyrus [24], [25].

As defined in III-B, the first step towards a product line test suite is to merge the individual product line models into one. This task is performed by the feature injector by limiting the behavior in the 150% model according to feature and mapping model. We created two libraries for this purpose: the first library generates propositional formulas for a given feature model and for individual features as discussed in the preliminaries II. The latter library consists of typical transformations on UML models for facilitating the creation and manipulation of states, transitions, guards, triggers, effects, and signals. Eventually, the newly gained model must be exported into a format for a particular test generator. Therefore, we implemented a model printer that reuses our UML model transformations library to prepare the model to be exported and then prints it into the target file format.

When coupled with Conformiq, SPLTestbench currently supports UML models that have a single class and one or more state machines that specify the class’ behavior. The class has to provide at least two ports, one for receiving signals from and the other for sending signals to the environment. One state machine must be selected as the class’ classifierBehavior and must own at most one region, while each region must own an initial state. A Transition may own a Trigger, a Guard, and/or an Effect. This far only SignalEvents are supported. Signals, SignalEvents, and PrimitiveTypes are stored in the same Package as the class. History, Fork/Join, and ConnectionPoints are currently not supported, but will be in future releases.

SPLTestbench is available as Eclipse-plugin. Figure 11 shows how the components (i), (ii), and (iv) integrate into the IDE. Each of the three menu items starts an individual wizard that guides the user through the details of the respective process.

B. Examples

To evaluate the PLC test design approaches with SPLTestbench, we created three model examples: An embedded alarm system, an e-commerce webshop, and a ticket machine. All of the model examples conform to the previously presented requirements for applying SPLTestbench.

The Ticket Machine is a simple case study and is adopted from Cichos et al. [26]. The functionality is as follows: a customer may select tickets, pay for them, receive the tickets, and collect change. The feature model has a root feature with three sub-features attached to it; all of them are optional. Depending on the selected features, the machine offers reduced tickets, accepts not only coins but also bills, and/or will dispense change.

The Alarm System example is also adopted from Cichos et al. [27] and more complex. The alarm may be set off manually or automatically by a vibration detector. Both features are part of an or-group and, thus, at least one of the two features must be present in every product. In the event of an alarm, a siren or a warning light will indicate the security breach. When the vibration does not stop after a predefined period of time, the system optionally escalates the alarm by calling police authorities and/or sending photos of evidence. Additionally to its alarming functionality, the Alarm System SPL provides a feature for taking a photo of any operator that configures the system for security measures.

We adopted the Alarm System models by removing manual timers that were implemented as guard conditions. Also a transition that eventually led to a life-lock condition was removed. Furthermore, we added cross-tree-constraints (CTC) and more features to the feature model for exercising the SPLTestbench’s functionalities more thoroughly.

The last example SPL is a fictional e-commerce Webshop (eShop) designed by us. A customer can browse the catalog...
of items, or if provided, use the search function. Once the customer puts items into the cart, he can checkout and may choose from up to three different payment options, depending on the eShop’s configuration. The transactions are secured by either a standard or high security server. A CTC ensures that credit card payment is only offered if the eShop also implements a high security server.

Table I and II summarize the individual case study models on structural level. The Alarm System case study is the most variable SPL in this comparison by means of possible configurations (CNF), offering 42, followed by the eShop with 20, and the Ticket Machine with only 8 configurations. Although the Alarm System has two features more than the eShop, it offers as twice as much configurations. This is a typical effect observable in variable systems, where adding only a few features can drastically increase the amount of configurations. There are further metrics for feature models available to measure analyzability, changeability, and understandability, which we did not apply so far [28].

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

C. Experiment Settings

For our case study experiment we generated tests according to both presented approaches, PLC and PC test design. We chose realistic coverage criteria for feature models, 100% state machine models, and 150% state machine models. For PC test design we selected two different feature model coverage criteria for individual comparison: all-features-included-excluded and all-feature-pairs [29].

For the actual test case design we employed Conformiq Designer for both approaches. Conformiq Designer supports control-flow, branching, requirements, and path coverage criteria. For the individual 100% state machine models as well as for the 150% models we applied all-transition coverage [30]. Though, there are many other more sophisticated metrics for state machines to choose from [31]–[33].

D. Results

We were able to generate test suites for both approaches with all the aforementioned parameters for all examples. Here, we present our first results. We counted the number of test cases and test steps that were generated by the test generator. Of course, the amount of configurations that is necessary to execute the test cases is of equal interest.

The results for these measures are shown in Table III for each individual approach: PC with all-features-included-excluded (PC-IX) as well as with pair-wise (PC-PW) coverage and PLC with pre-configuration (PLC-Pre) as well as with step-by-step (PLC-Step). The PC-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 PLC-Step yields the lowest scores for any measure except for tests steps on the Ticket Machine, while at the same time — as stated in section III-B — it is focused on covering every reachable transition. We take this as an indicator for PLC test design to scale better than PC approaches.

V. RELATED WORK

Testing is one of the most important quality assurance techniques in industry. Since testing often consumes a high percentage of project budget, there are approaches to automate repeating activities like, e.g., regression tests. Some of these approaches are data-driven testing, keyword-driven testing, and model-based testing. There are many books that provide surveys of conventional standard testing [34]–[36] and model-based testing [4], [37], [38]. In this paper, we use model-based testing techniques and apply them to product lines. Modeling languages like the Unified Modeling Language (UML) [9] have been often used to create test models. We apply UML state machines.

Feature models are commonly used to describe the variation points in product lines. There are several approaches to apply feature models in quality assurance. For instance, Olimpiew and Gomaa [39] deal with test generation from product lines and sequence diagrams. However, we focus on UML state machines and describe different approaches for combining both. In contrast to sequence diagrams, state machines are commonly used to describe a higher number of possible behaviors, which makes the combination with feature models more complex than combining feature models and
sequence diagrams. As another example, McGregor [40] shows the importance of a well defined testing software product line process. Just like McGregor, the focus of our paper is only investigating the process of actually creating tests rather than defining the structural possible relations of feature models and state machines. Pohl and Metzger [41] emphasize the preservation of variability in test artifacts of software product line testing. As we derive test case design from models automatically, this variability is preserved. Lochau et al. [20] also focus on test design with feature models. In contrast to our work, they focus on defining and evaluating coverage criteria that can be applied to feature models. In the presented PC approaches, we strive for using such coverage criteria on feature models for the automation of test design. Furthermore, Lochau et al. recently proposed an extension for PC testing to decrease its efforts by omitting common test goals that were already exercised for other products of the product line [42]. Cichos et al. [27] also worked on an approach similar to the presented PLC approach. Their approach, however, requires users to provide a set of product variants to the used test generator to derive 100% models from the 150% model for automatic test generation. As a consequence, the test generator requires an additional input parameter and (as the authors state) no standard test generator can be applied for their approach. In contrast, both of our approaches allow for integrating commercial off the shelf test generators like in our case, Conformiq [10]. One of the most important aspects in our work is the ability to integrate our approach into existing tool chains. In [43], we already addressed model-based test generation for product lines. However, back then we focused on reusing state machines in multi-variant environments instead of describing the different automatic test design approaches for product lines.

VI. CONCLUSION, DISCUSSION, AND FUTURE WORK

In this paper, we presented different approaches to the automatic test design for product lines. We described the state of the art, presented the general idea of mapping feature to other model artifacts, and presented two approaches to use this mapping for automatic test design. Our main contributions are the definition and evaluation of test design approaches using three examples.

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

Currently, our approaches are limited to mapping features to transitions. In larger systems, not only transition, but also variables, default values, classes, whole components, and other behavioral or structural elements are potential targets for mappings. This need further investigation as one cannot simply apply our approach to these kinds of elements. For example, to allow a variable to have multiple default values is a non-trivial problem, not only because most UML editors will not allow you to create a second default value for a single variable. But also test generators will not accept a model with two variables that have the same name. Hence, efficient solutions must be found to deal with such syntactically incorrect 150% models.

Furthermore, we showed that PLC test design opens the fields for new feature coverage criteria. Since this test design method preserves variability throughout the test design process, products can be selected with other intentions, e.g., the fewest configurations for executing all tests, or the most configurations while each test case is executed only once. We plan to investigate these criteria in the future, but for evaluating them we have to implement and employ more sophisticated evaluation processes.

Hence in the near future, we will extend SPLITestbench for qualitative test case evaluation. Since it is not possible to obtain an implementation for every case study example, we plan to use techniques from mutation testing and raise them to the product line model level [44]. This can be achieved by applying mutations operators not only to 150% models, but also to feature models [45] and/or feature mapping models. The SPLITestbench should manage the overall mutation analysis process by creating mutant product line models, executing any given set of test cases against the mutated product line models, and storing the results. Of course, a mapping of the test cases to their corresponding configurations must be established and maintained throughout this process. Along with running larger examples with distributed systems, we expect to gain further evidence of scalability and test quality on our approaches.

Also, the retrieval of abstract configurations for the pre-configuration method of PLC test design will be implemented. This will enable us to conduct a meaningful comparison to the step-by-step approach. Especially the aforementioned mutation framework for SPLs would be helpful in this comparison, too.

REFERENCES


