Automated Test Design for Boundaries of Product Line Variants

Stephan Weißleder[1], Florian Wartenberg[1], and Hartmut Lackner[2](✉)

[1] Thales Transportation Systems, Schützenstraße 25, 10117 Berlin, Germany
{stephan.weissleder,florian.wartenberg}@thalesgroup.com
[2] Humboldt-Universität zu Berlin, Rudower Chaussee 25, 12489 Berlin, Germany
lackner@informatik.hu-berlin.de

Abstract. Developing product lines is usually more efficient than developing single products because of the reuse of single components. Testing, however, has to consider complete, integrated systems. To prevent testing every product on system level, the whole product line should be analyzed with the aim of selecting distinguishing product behavior and a minimum of system products to test. In this paper, we present a model-based test design approach for testing the selected behavior of products, but also their deselected behavior. A major challenge of this approach is that the deselected behavior of a product is often not part of its behavioral model. Thus, we use the variability model to transform the behavioral model so that showing the exclusion of the deselected behavior is also covered by tests. We present the approach, a corresponding prototypical implementation, and our experiences using a set of examples.

1 Introduction

Configurability is a key selling point of many systems. For instance, every possible variant of a German car is sold only once or twice on average. Correspondingly, it is infeasible to design every variant from scratch. Instead, system components are reused to a maximum extent. The reuse and variability of system components can be described in a variability model like, e.g., a feature model with features linked to system components. Although this process leads to a significant gain in development efficiency, system test efficiency is not impacted, because the whole integrated system has to be tested.

Quality assurance is one of the most important aspects in systems engineering. Low quality usually results in high costs for defect rectification. Testing is an important quality assurance technique. In many cases, however, it is considered to be very expensive. Automated test execution helps in reducing test execution time and costs. It faces, however, the issue of high costs for test design adaptation. The automation of test design is a solution to such issues.

Existing approaches are focused on covering the selected behavior of a product, i.e., they check if everything that should be in the product is really implemented. In this paper, we focus on covering the deselected behavior by checking if everything that should not be in the product variant is really not implemented. Common approaches cannot be used for this, because they are focused

K. El-Fakih et al. (Eds.): ICTSS 2015, LNCS 9447, pp. 86–101, 2015.
DOI: 10.1007/978-3-319-25945-1_6

on cutting away all deselected behavior for a variant, and thus the model for the variant does not contain deselected behavior, anymore. To overcome this issue, we introduce model transformations that create new model elements describing the non-existence of deselected behavior.

The paper is structured as follows. In Sect. 2, we present the preliminaries. Section 3 describes our approach. The implementation and experiments are described in Sect. 4. Section 5 contains an analysis of the related work. In Sect. 6, we conclude and discuss our approach including threats to validity.

2 Preliminaries

This section contains the preliminaries of this paper. It describes automated test design, model-based product line (PL) engineering, and the combination of both.

2.1 Automated Test Design

Testing is a common approach to quality assurance. The idea is to systematically compare the observed system behavior with the expected one. There are various approaches and tools to automate the test execution. The biggest issue of this approach are changes. A change of requirements or customer wishes results in high effort for test design adaptation. In the worst case, test design adaptation costs outweigh the costs saved by automated test execution. In order to solve this issue, test design also needs to be automated. An often used approach for this is model-based testing [5,24]. We apply state machines of the Unified Modeling Language (UML) as the basis for automated test design [28]. UML state machines are used to express state-based system behavior. There are several corresponding test generators [9,20,27].

2.2 Model-Based Product-Line Engineering

A demand for high configurability at low costs drives engineering disciplines to increase the number of product features while keeping systems engineering costs at a reasonable level. Reusing system components helps in reducing engineering costs. A PL is a set of related products that share a common core of assets (commonalities), but can be distinguished (variabilities) [21]. Consequently, PL engineering is a technique to fulfill the wish for high configurability at low costs.

PL engineering can be supported by models like, e.g., feature models that enable facilitating the explicit design of global system variation points [15]. As a consequence, system variation points are not spread across one or multiple domain models or code fragments anymore, but instead linked to one core of variability description.

A feature model is a tree with root feature and linked feature children (see Fig. 1). A parent feature can have the following relations to its child features: (a) *Mandatory*: child feature is required, (b) *Optional*: child feature is optional, (c) *Or*: at least one of the child features must be selected, and (d) *Alternative*: exactly one of the child features must be selected. Furthermore, cross-tree

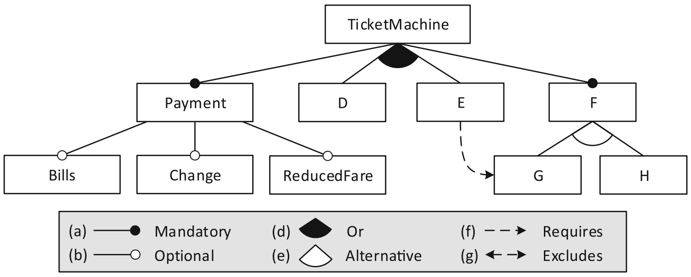

Fig. 1. A feature model for the ticket machine example.

constraints between two features A and B are possible: (f) A *requires* B: the selection of A implies the selection of B, and (g) A *excludes* B: both features A and B must not be selected for the same product.

As Czarnecki et al. presented in [11], feature models can be transformed into propositional formulas defined over a set of Boolean variables, where each variable corresponds to a feature. This allows for checking every combination of features according to its validity, i.e., if it represents a valid variant of the feature model. For instance, the boolean formula for the Ticket Machine in Fig. 1 is:

$$\begin{aligned} FM = \; & TM \wedge (\neg Bills \vee Payment) \wedge (\neg Change \vee Payment) \\ & \wedge (\neg ReducedFare \vee Payment) \wedge (\neg Payment \vee TM) \\ & \wedge (\neg D \vee TM) \wedge (\neg E \vee TM) \wedge (\neg F \vee TM) \\ & \wedge (\neg G \vee F) \wedge (\neg H \vee F) \wedge (\neg E \vee G) \\ & \wedge (\neg TicketMachine \vee Payment) \wedge (\neg TicketMachine \vee F) \\ & \wedge (\neg D \vee E) \wedge ((G \wedge \neg H) \vee (\neg G \wedge H)) \end{aligned}$$

Any assignment that satisfies the formula is a valid configuration. The following formula is a valid configuration for the feature model presented in Fig. 1.

$$\begin{aligned} P = & \{TicketMachine, Payment, \neg Bills, \neg Change, \\ & \neg ReducedFare, D, E, F, G, \neg H\} \end{aligned}$$

2.3 Automated Test Design for Product Lines

A feature model contains the system's variation points. Its elements, however, are only symbols [10]. Semantics is provided by mapping features to artifacts with semantics such as system models or source code. Such a mapping can be defined using a mapping models that contain relations from features to artifacts with semantics.

In our case, the system model is designed as a so called 150 % model containing every element that is used in at least one potential product configuration and, thus, describing all possible variants [13]. Hence our PL model comprises a 150 % system model that is a UML state machine, a feature model explicitly

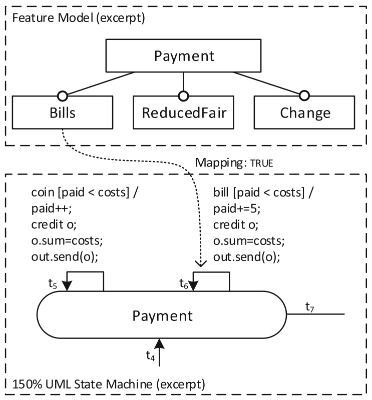
Feature Model (excerpt)
Payment
Bills
ReducedFair
Change
Mapping: TRUE
coin [paid < costs] / paid++; credit o; o.sum=costs; out.send(o);
bill [paid < costs] / paid+=5; credit o; o.sum=costs; out.send(o);
t5
t6
t7
t4
Payment
150% UML State Machine (excerpt)

Fig. 2. Excerpt of the product line model for the TicketMachine.

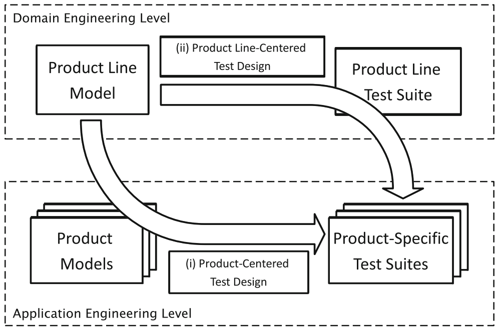
Domain Engineering Level
Product Line Model
(ii) Product Line-Centered Test Design
Product Line Test Suite
Product Models
(i) Product-Centered Test Design
Product-Specific Test Suites
Application Engineering Level

Fig. 3. Product-centered and product line-centered test design.

expressing the PL's variation points, and a feature mapping model that connects both. The current version of our work maps features to states and transitions. Each mapping has a Boolean flag that indicates whether the mapped model elements are part of the product when the feature is selected (*Mapping: TRUE*) or unselected (*Mapping: FALSE*).

In Fig. 2, we depict an excerpt of the PL model for our Ticket Machine example with feature model and UML state machine. In this excerpt, the system waits for coins or bills to be inserted until the costs for the selected tickets are covered. The dotted arrow maps the feature *Bills* to the transition t_6 in the state machine: If the Ticket Machine's configuration includes the feature *Bills*, then the mapped transition t_6 must be present in the corresponding product. If the feature is not selected, this transition is not part of the corresponding product and hence leaving the customer's only payment option to be coins as denoted in transition t_5.

Based on this, we defined two approaches to automated test design for PLs [16] as depicted in Fig. 3: (i) *product-centered* (PC) and (ii) *product line-centered* (PLC). The product-centered approach consists of selecting a representative set of products (test models) and afterwards 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 applys the PL model for designing tests. The second approach is focused on the behavior defined at the PL 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.

3 Testing Boundaries of Products

A product is configured to include a subset of the specified behavior of a PL model, the rest is excluded. Model-based testing (MBT) is focused on creating test cases based on models. Typically, MBT designs tests for performig positive testing by means of checking the included behavior for conformance. The information about parts being explicitly excluded, however, is valuable too. A test designer can make use of this information by creating tests that actively try to invoke excluded behavior. We think of this as an attempt of breaching the boundaries of a product under test (PUT), where the boundary is predefined by the PUT's configuration. A boundary is overcome if an excluded behavior is invoked and executed as specified in the PL model.

3.1 Boundary Transitions

Inside the PUT's boundaries is the PL's core and all included features declared by the configuration. Outside its boundaries lie the excluded features. Figure 4 depicts an excerpt of a ticket machine product, in which the feature *Bills* is deactivated. Here, the state "Payment" and the transitions t_4, t_5, t_7 lie within the boundaries of the product. Transition t_6 as shown in the excerpt of Fig. 2 is not part of this product. We overcome this boundary, if we make the product process a bill in this state as defined in the PL model in Fig. 2. More formally speaking, we define a product's boundary by *boundary transitions* over UML State Machines. We define a boundary transition bt, where S be the set of states and T be the set of transitions in a PL model and $t(s, s')$ be a transition from state s to s' as:

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

Hence, a boundary transition is not part of the particular product. We call a product to have an *open boundary*, if behavior from an excluded feature can be invoked at some point of the PUT's execution.

In general, it is possible to detect open boundaries by stimulating the PUT with unexpected events in every state. This resembles sneak-path-analysis and is costly [14]. Here, we propose a method to reduce test effort by stimulating the PUT with unexpected events only if its active state has at least one boundary transition. In particular, we stimulate the PUT with only those events that could possibly trigger one of its boundary transitions.

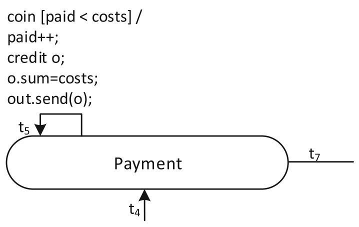

Fig. 4. Product of ticket machine excerpt without feature *Bills*.

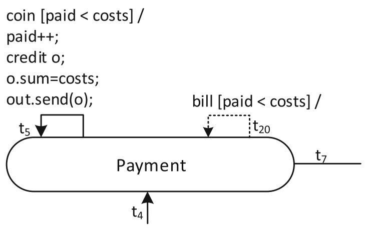

Fig. 5. Same product with additional complementary transition.

3.2 Turning Open Boundaries into Test Goals

We chose transition-based coverage criteria for selecting test goals. Our approach comprises introducing a transition for each boundary transition to which we refer to as *complementary transition.* The intention of this is to create transitions specifying that the PUT should stay in its current state and with events that are not expected to trigger product behavior. Hence, for every boundary transition, we add a complementary transition with its source state as target and source. For the presented ticket machine product without feature *Bills*, Fig. 5 shows the same excerpt of the product as in Fig. 4, but with the additional complementary transition t_{20}, which complements boundary transition t_6 of this product. The complementary transition must have no effect, since in the state "Payment" no reaction is expected for any product that does not include feature *Bills*. However, we should not add a complementary transition, if there is an explicitly specified behavior for processing the signal event when feature *Bills* is excluded as in state "Selection".

So far, we defined boundary transitions for a given product and outlined how to add complementary transitions. For PLC test design we must raise these concepts to the PL level, in order to set complementary transitions as test goals. Particularly, we define a transformation for adding complementary transitions to the PL model whenever there is a boundary transition of any product available. This enables PLC test design methods to consider complementary transition as test goals during test design. Also, PC test design methods can benefit from this approach, since the complementary transitions persist during the derivation process.

In Fig. 6, we depict the desired outcome of the transformation: we added a complementary transition t_{20} to state *Payment* for transition t_6, which is a boundary transition for any product not including the feature *Bills*. Hence, the complementary transition is mapped to feature *Bills* with the mapping's flag set to *false*, denoting the transition is only to be included when the feature *Bills* is deselected. We present the pseudo code to achieve the result shown in Fig. 6 in Algorithm 1. Let $SM(S,T)$ be a state machine, where S is the set of states and T the set of transitions. For each transition $t \in T$ we define:

- source(t) as the source state of t,
- target(t) as the target state of t,
- triggers(t) as the triggers of t,
- triggers $*(t)$ as the triggers from all transitions leaving target(t), if triggers(t) is empty, and triggers(t) otherwise. Since this is a recursive definition, triggers $*(t)$ must stop once all $t \in T$ are traversed.
- features(e) as the set of feature selections mapped to an UML element $e \in SM$. A feature selection states whether a feature must be selected or deselected to include e.
- *concurrentGuards*(t) as a conjunction of guard conditions. The conditions are collected from transitions that can be concurrently enabled with t.

First a set of transitions for storing complementary transitions during this procedure is initialized. Then for all transitions of the state machine the following

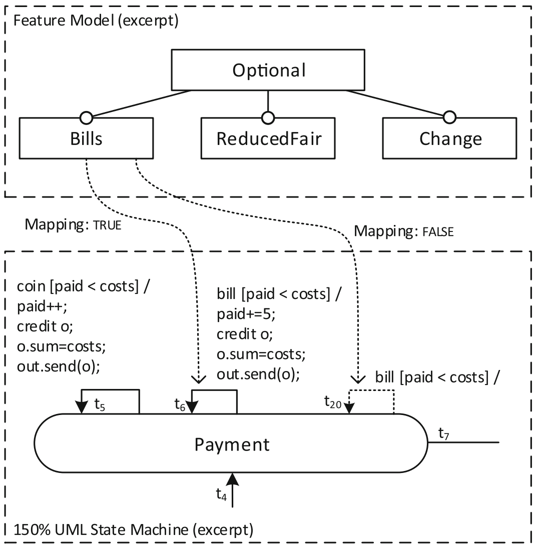
Feature Model (excerpt)
Optional
Bills
ReducedFair
Change
Mapping: TRUE
Mapping: FALSE
coin [paid < costs] / paid++; credit o; o.sum=costs; out.send(o);
bill [paid < costs] / paid+=5; credit o; o.sum=costs; out.send(o);
bill [paid < costs] /
t5
t6
t20
t7
t4
Payment
150% UML State Machine (excerpt)

Fig. 6. PL model example: ticket machine with complementary transition.

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

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

The outcome of this procedure when applied to the ticket machine's PL model is depicted in Fig. 7. We denote the mappings from the feature model by feature formulas in the transition's guards analog to Featured Transition System (FTS) introduced by Classen [8]. We use the following acronyms: B for *Bills*, C for *Change*, and R for *ReducedFare*. The complementary transitions added by our transformation procedure are denoted by dotted arcs (transitions t_{19}–t_{22}). Beginning from the initial state, we find the first state with at least one boundary transition to be "Selection". The boundary transition here is t_3, which is enabled when the feature *ReducedFare* is part of a product. Hence, t_{19} is added to the state machine for serving as an additional test goal to any product not including

Algorithm 1. Adds Complementary Transitions to a Region

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

ReducedFare. To achieve all-transition coverage, a test case must include sending the signal event “reducedTicket” when the feature *ReducedFare* is disabled while the state machine is supposed stay in state “Selection”. Analog to this, transition t_{20} is added for boundary transition t_6 in state “Payment”.

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

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

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

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

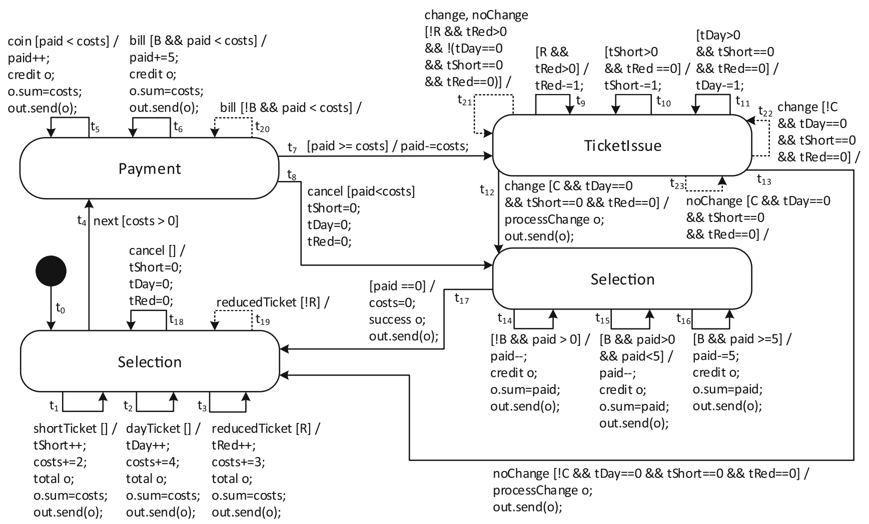
coin [paid < costs] / paid++; credit o; o.sum=costs; out.send(o);
bill [B && paid < costs] / paid+=5; credit o; o.sum=costs; out.send(o);
bill [!B && paid < costs] /
t5
t6
t20
Payment
t7 [paid >= costs] / paid-=costs;
t8 cancel [paid<costs] tShort=0; tDay=0; tRed=0;
change, noChange [!R && tRed>0 &&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

Fig. 7. PL model example: TicketMachine model with added feature formulas and complementary transitions.

4 Examples and Evaluation

In this section, we present the evaluation of the product line's test suites, with and without the presented model transformations. We assess all tests by means of fault detection capability. First, we introduce the used approach of measuring the fault detection capability of the test suite. Afterwards, we describe the used examples, the test setup, and the results.

4.1 Mutation System for PLs

Mutation analysis (also mutation testing) [12] is a fault-based testing technique with the intended purpose to assess the quality of tests by introducing faults into a system and measuring the success rate of fault detection.

The process of mutation analysis inserts defects into software by creating multiple versions of the original software, where each created version contains one deviation. Afterwards, existing test cases are used to execute the faulty versions (*mutants*) with the goal to distinguish the faulty ones (*to kill a mutant*) from the original software. The ratio of killed mutants to generated mutants is called *mutation score*. The main goal of the test designer is to maximize the mutation score. A mutation score of 100 % is seldom possible, because some deviations may lead to an unchanged system behavior, i.e. semantically equivalent mutants.

We think that mutation systems for PLs need novel mutation operators and mutation processes. The reason for this is the separation of concerns in model-based PL engineering, where variability and domain engineering are split into

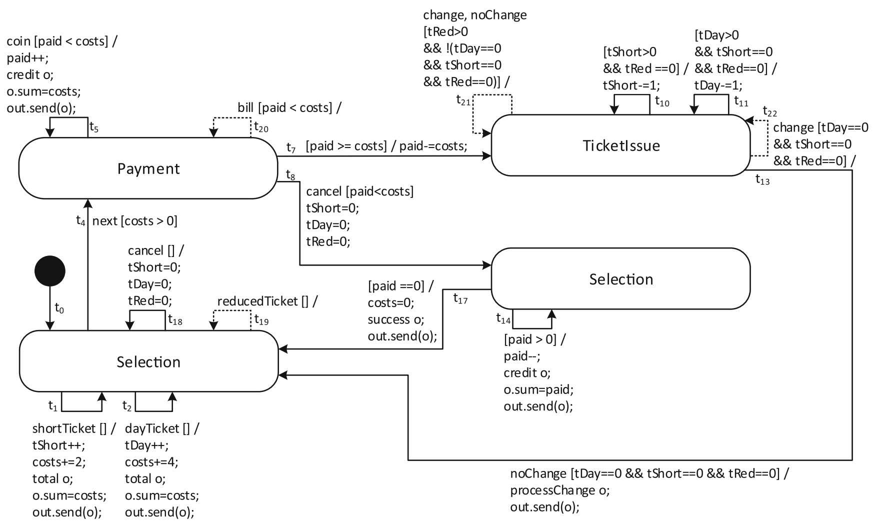
coin [paid < costs] / paid++; credit o; o.sum=costs; out.send(o);
t5
bill [paid < costs] /
t20
change, noChange [tRed>0 && !(tDay==0 && tShort==0 && tRed==0)] /
t21
[tShort>0 && tRed ==0] / tShort-=1;
t10
[tDay>0 && tShort==0 && tRed==0] / tDay-=1;
t11
t22
change [tDay==0 && tShort==0 && tRed==0] /
t13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

Fig. 8. Product example: state machine model of a ticket machine without Bills, Change, and ReducedFare.

different phases and models. Hence of new modeling languages used in PL engineering, more *kinds* of errors can be made on the model-level than in non-variable systems engineering. In our case, new errors occur in feature mapping models. Of course, the here defined operators are only useful, if system engineering was facilitated by feature models and feature mappings with negative variability. Otherwise, the here described errors are unlikely and hence not applicable.

Mutation processes for PLs differ from conventional mutation processes, since a mutated PL model is not executable per se. Thus, testing cannot be performed until a decision is made towards a set of products for testing. This decision depends on the PL test suite itself, since each test is applicable to just a subset of products. In Fig. 9, we depict a mutation process for assessing PL test suites, which addresses this issue. Independently from each other, we gain (a) a set of PL model mutants by applying mutation operators to the PL model and identify (b) a set of configurations describing the applicable products for testing. We apply every configuration from (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 removed and does not participate in the scoring. The model mutants are then derived to product mutants and finally, tests are executed. Our mutation scores are based on the PL model mutants, hence we established bidirectional traceability from any PL model mutant to all its associated product mutants and back again. If a product mutant is killed by a test, we backtrack its original PL model mutant and flag it as killed. The final mutation score is then calculated from the set of killed and the overall number of PL model mutants.

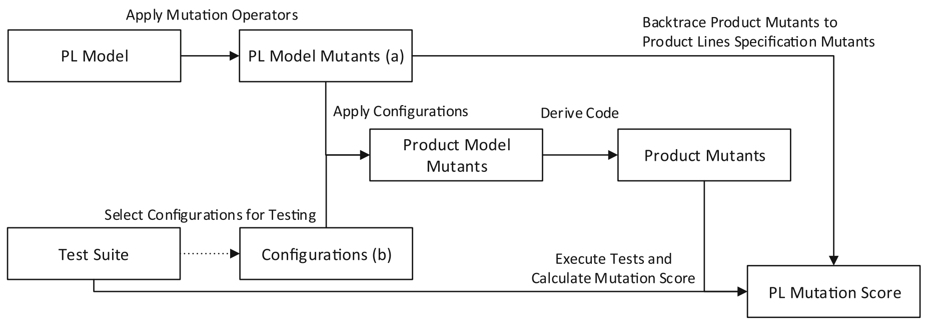

Fig. 9. Mutation process for PLs

We provide the following mutation operators for the mapping model:

- *Delete Mapping (DMP)*: Deletes a mapping from the mapping model. This enables all referenced elements that have no other mappings.
- *Delete Mapped Element (DME)*: Deletes an UML element from a mapping. This enables all referenced elements that have no other mappings.
- *Insert Mapped Element (IME)*: Adds a new UML element to the mapping. This element will only be available in products including the mapped feature.
- *Change Feature Value (CFV)*: Flips the feature value of a mapping so that the UML element is included when it has been excluded before and vice versa.
- *Swap Feature (SWF)*: Substitutes a feature from a mapping by another feature from the mapping model.

For our experiment, we perform mutation analysis with all of these operators.

4.2 Examples

We assessed the quality for three test suites, where each test suite belongs to a different case study. These case studies represent three kinds of systems: an e-commerce shop (eShop), which makes contains many signals, but only few guards, the Ticket Machine (TM) that uses less signals and in contrast more guards, and lastly, an alarm system (AS), which is offers most product variations.

In the eShop example, 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 constraint ensures that credit card payment is only offered if the eShop also implements a high security server.

The TM example is adopted from Cichos et al. [7]. 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 optional subfeatures attached to it. Depending on the selected features, the machine offers reduced tickets, accepts not only coins but also bills, and/or will dispense change.

The AS example is adopted from Cichos et al. [6]. 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 PL of the AS provides a feature for taking a photo of any operator that configures the system for security measures. We adopted the AS model by removing manual timers that were implemented as guard conditions.

4.3 Setup

We design two test suites for each example. For the first test suite we use the original models, for the second we apply our transformations first and then run the test design process. The design of each test suite is facilitated by model-based testing techniques. In particular, we used a product line-centered test design process as defined in [16], where tests are designed based on the PL model.

We apply transition coverage for test selection. A test generator then automatically designed the tests. From the tests, SPLTestbench selected products for testing and derived them from the mutated PL models into product model mutants. Since our examples lack implementations, we decided to generate code from the product model mutants and run the tests on them.

4.4 Results

In Table 1, we show the test assessment results of test suites, that were designed with the original models. In each row, we show the mutation results for all examples in the form of killed mutants/all mutants. As supposed, mutations with behavior that is not described by the test model (DME, DMP) are not detected. For the other two mutation types which alter specified behavior (IME, CFV), we receive mixed results in the range of 40 % to 100 %. In contrast, Table 2 depicts the assessment results for the test suites that were created from our transformed

Table 1. Mutation scores for regular tests

Op	TM	eShop	AS	p.Op
DMP	1/5	0/4	0/8	1/17
DME	1/8	0/14	0/21	1/43
IME	2/5	1/4	2/8	5/17
CFV	5/5	4/4	6/8	15/17
SWV	3/5	2/4	3/8	8/17
per Ex.	12/28	7/30	11/53	

Table 2. Mutation scores for tests with transformations

Op	TM	eShop	AS	p.Op
DMP	3/5	4/4	5/8	12/17
DME	3/8	4/14	8/21	15/43
IME	3/5	4/4	2/8	9/17
CFV	5/5	4/4	7/8	16/17
SWV	4/5	4/4	4/8	12/17
per Ex.	18/28	20/30	26/53	

models. Again in each row, we show the mutation results for all examples in the form of killed mutants/all mutants. We observe increased scores for every mutation operator on any of our examples.

In the last row of each table, we show the overall results for each example. Furthermore, in the last column we present the accumulated scores of every mutation operator over all examples.

5 Related Work

In recent years model-based testing (MBT) emerged as an efficient test design paradim that yields a number of improvements compared to conventional test design such as higher test coverage or earlier defect detection. There are several surveys on the effectiveness of MBT in general [5,25,29] and MBT of software product lines [18]. In contrast to this, we combine the application of model-based software product line testing with a product line-specific sneak path analysis. To our knowledge, this combination has not been covered before.

In earlier work [16], we present two approaches for product line test design automation. However, the current paper is focused on testing whether unselected features are actually excluded from the product variant. Our approach reuses the concept of *Simulated Satisfaction* of coverage criteria by transforming the test model instead of improving the applied test generation tools [26]. Hence, the herein presented approach is independent of the test design method, as long as it relies on models.

There are many studies on fault detection effectiveness of model-based test generation using mutation analysis [1,2,19,22,23]. In order to further assess our approach we extended our SPLTestBench by a mutation framework and defined mutation operators for feature models, feature mappings, and the test model.

An early evaluation of the mutation scores suggests that our generated test suites satisfying all-transitions coverage are capable of detecting many seeded faults except unspecified behavior, so-called sneak paths [3]. In safety-critical systems, an unintentional sneak path may have catastrophic consequences. Sneak path testing aims at verifying the absence of sneak paths and at showing that the software under test handles them in a correct way. Several studies showed that sneak path testing improves the fault detection capabilities [4,14,17]. However, the effort spent for sneak path testing is considerably high. Here, we present a novel, more efficient approach for detecting unspecified behavior in product line engineering: We define boundary transitions that stimulate the product under test with only those events that could possibly trigger a transition that would invoke excluded behavior. To our knowledge, this approach has not been applied in the context of software product line engineering before.

6 Conclusions, Discussion and Future Work

Conclusions: In this paper, we combined model-based test design for software product lines with boundary transition analysis. We extended our previous work on product-line centered model-based test design with model transformations that increase the fault detection capabilities of the generated test suites.

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

Discussions: Our results support the recommendation of Binder [3] and the conclusions drawn by Mouchawrab et al. [17] and Holt et al. [14]: Testing sneak paths (in our case as boundaries of product line variants) is an essential component of state-based testing and drastically increases fault detection capabilities. Furthermore the results indicate that sneak path testing is a necessary step in state-based testing due to the same observations made by Holt et al. [14]: (1) The proportion of sneak paths in the collected fault data was high (61,5 %), and (2) the presence of sneak paths is undetectable by conformance testing.

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

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

Future Work: In our future, we plan to apply our approach to a real case study. We also want to review the defined mutation operators and compare the effects when applying well-known mutation operators.

Acknowledgments. This work is partially supported by grants from Deutsche Forschungsgemeinschaft, Graduiertenkolleg METRIK (GRK 1324).

References

1. Andrews, J.H., Briand, L.C., Labiche, Y.: Is mutation an appropriate tool for testing experiments? In: Proceedings of the 27th International Conference on Software Engineering, ICSE 2005, pp. 402–411 (2005)
2. Andrews, J.H., Briand, L.C., Labiche, Y., Namin, A.S.: Using mutation analysis for assessing and comparing testing coverage criteria. IEEE Trans. Softw. Eng. **32**(8), 608–624 (2006)
3. Binder, R.V.: Testing Object-Oriented Systems: Models, Patterns, and Tools. Addison-Wesley Longman Publishing Co., Inc., Boston (1999)
4. Briand, L.C., Penta, M.D., Labiche, Y.: Assessing and improving state-based class testing: a series of experiments. IEEE Trans. Softw. Eng. **30**(11), 770–783 (2004)
5. Broy, M., Jonsson, B., Katoen, J.P.: Model-Based Testing of Reactive Systems: Advanced Lectures. Lecture Notes in Computer Science, vol. 3472. Springer, Heidelberg (2005)
6. Cichos, H., Heinze, T.S.: Efficient reduction of model-based generated test suites through test case pair prioritization. In: Proceedings of the 7th International Workshop on Model-Driven Engineering. Verification and Validation (MoDeVVa 10), pp. 37–42. IEEE Computer Society Press, Los Alamitos (2011)
7. Cichos, H., Lochau, M., Oster, S., Schürr, A.: Reduktion von testsuiten für software-produktlinien. In: Jähnichen, S., Küpper, A., Albayrak, S. (eds.) Software Engineering 2012: Fachtagung des GI-Fachbereichs Softwaretechnik, 27. Februar - 2. März 2012 in Berlin. LNI, vol. 198, pp. 143–154. GI (2012)
8. Classen, A., Heymans, P., Schobbens, P.Y., Legay, A.: Symbolic model checking of software product lines. In: 33rd International Conference on Software Engineering, ICSE 2011, May 21–28, 2011, Waikiki, Honolulu, Hawaii, Proceedings, pp. 321–330. ACM (2011)
9. Conformiq Qtronic: Semantics and Algorithms for Test Generation: A Conformiq Software Whitepaper (2008)
10. Czarnecki, K., Antkiewicz, M.: Mapping features to models: a template approach based on superimposed variants. In: Glück, R., Lowry, M. (eds.) GPCE 2005. LNCS, vol. 3676, pp. 422–437. Springer, Heidelberg (2005)
11. Czarnecki, K., Wasowski, A.: Feature diagrams and logics: there andback again. In: Software Product Line Conference, 2007. SPLC 2007. 11th International, pp. 23–34 (2007)
12. DeMillo, R.A.: Mutation Analysis as a Tool for Software Quality Assurance. In: COMPSAC 1980 (1980)
13. Grönniger, H., Krahn, H., Pinkernell, C., Rumpe, B.: Modeling variants of automotive systems using views. In: Kühne, T., Reisig, W., Steimann, F. (eds.) Tagungsband zur Modellierung 2008 (Berlin-Adlershof, Deutschland, 12–14. März 2008). LNI, Gesellschaft für Informatik, Bonn (2008)
14. Holt, N.E., Torkar, R., Briand, L.C., Hansen, K.: State-based testing: Industrial evaluation of the cost-effectiveness of round-trip path and sneak-path strategies. In: 23rd IEEE International Symposium on Software Reliability Engineering, ISSRE 2012, Dallas, TX, USA, November 27–30, pp. 321–330. IEEE Computer Society (2012)
15. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study (1990)

16. Lackner, H., Thomas, M., Wartenberg, F., Weißleder, S.: Model-based test design of product lines: raising test design to the product line level. In: ICST 2014: International Conference on Software Testing, Verification, and Validation, pp. 51–60. IEEE Computer Society (2014)
17. Mouchawrab, S., Briand, L.C., Labiche, Y., Di Penta, M.: Assessing, comparing, and combining state machine-based testing and structural testing: a series of experiments. IEEE Trans. Softw. Eng. **37**(2), 161–187 (2011)
18. Oster, S., Wubbeke, A., Engels, G., Schürr, A.: A survey of model-based software product lines testing. In: Zander, J., Schieferdecker, I., Mosterman, P.J. (eds.) Model-Based Testing for Embedded Systems. Computational Analysis, Synthesis, and Design of Dynamic Systems, pp. 339–384. CRC Press, Boca Raton (2011)
19. Paradkar, A.: Case studies on fault detection effectiveness of model based test generation techniques. In: Proceedings of the 1st International Workshop on Advances in Model-based Testing, A-MOST 2005, pp. 1–7 (2005)
20. Peleska, J.: RT-Tester Model-Based Test Case and Test Data Generator: User Manual: Version 9.0–1.0.0 (2013)
21. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer, Heidelberg (2005)
22. Siami Namin, A., Andrews, J.H., Murdoch, D.J.: Sufficient mutation operators for measuring test effectiveness. In: Proceedings of the 30th International Conference on Software Engineering, ICSE 2008, pp. 351–360 (2008)
23. Smith, B.H., Williams, L.: Should software testers use mutation analysis to augment a test set? J. Syst. Softw. **82**(11), 1819–1832 (2009)
24. Weißleder, S., Schlingloff, H.: An evaluation of model-based testing in embedded applications. In: ICST 2014: International Conference on Software Testing, Verification, and Validation. IEEE Computer Society (2014)
25. Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach, 1st edn. Morgan Kaufmann Publishers Inc., San Francisco (2006)
26. Weißleder, S.: Simulated satisfaction of coverage criteria on UML state machines. In: ICST - 3rd International Conference on Software Testing, Verification and Validation (2010)
27. Weißleder, S.: ParTeG (Partition Test Generator) (2009)
28. Weißleder, S., Schlingloff, H.: Automatic model-based test generation from UML state machines. In: Zander, J., Schieferdecker, I., Mosterman, P.J. (eds.) Model-Based Testing for Embedded Systems. Computational Analysis, Synthesis, and Design of Dynamic Systems. CRC Press, Boca Raton (2011)
29. Zander, J., Schieferdecker, I., Mosterman, P.J.: A taxonomy of model-based testing for embedded systems from multiple industry domains. In: Zander, J., Schieferdecker, I., Mosterman, P.J. (eds.) Model-based testing for embedded systems. Computational Analysis, Synthesis, and Design of Dynamic Systems. CRC Press, Boca Raton (2011)