Model-Based Coverage-Driven Test Suite Generation for Software Product Lines

Harald Cichos[1], Sebastian Oster[1], Malte Lochau[2], and Andy Schürr[1]

[1] TU Darmstadt
Real-Time Systems Lab
{cichos,oster,schuerr}@es.tu-darmstadt.de
[2] TU Braunschweig
Institute for Programming and Reactive Systems
lochau@ips.cs.tu-bs.de

Abstract. Software Product Line (SPL) engineering is a popular approach for the systematic reuse of software artifacts across a large number of similar products. Unfortunately, testing each product of an SPL separately is often unfeasible. Consequently, SPL engineering is in conflict with standards like ISO 26262, which require each installed software configuration of safety-critical SPLs to be tested using a model-based approach with well-defined coverage criteria.

In this paper we address this dilemma and present a new SPL test suite generation algorithm that uses model-based testing techniques to derive a small test suite from one variable 150% test model of the SPL such that a given coverage criterion is satisfied for the test model of every product. Furthermore, our algorithm simplifies the subsequent selection of a small, representative set of products (w.r.t. the given coverage criterion) on which the generated test suite can be executed.

1 Introduction

Software Product Line (SPL) engineering is a popular approach for the systematic reuse of software artifacts across a large number of similar products [1]. Unfortunately, engineers of different domains are nowadays developing SPLs for embedded, safety-critical systems without knowing how to test the large number of their product configurations systematically and efficiently in strict accordance with new software development standards. For example the new standard ISO 26262 [2] for safety-critical automotive software recommends that each software configurations has been tested thoroughly using model-based techniques and guaranteeing degrees of coverage according to certain criteria. But, nowadays, in the automotive industry almost every car of a certain brand has its individual software configuration, so it is difficult to comply with this recommendation. So far, SPL testing approaches are not able to efficiently test large SPLs thoroughly for the following reasons: First of all, testing every single product configuration of an SPL individually by using common testing techniques is not acceptable for large SPLs [3]. Furthermore, testing all actually used products only following a

J. Whittle, T. Clark, and T. Kühne (Eds.): MODELS 2011, LNCS 6981, pp. 425–439, 2011.

demand-driven approach is inacceptable, too, due to the still large number of relevant products and the fact that the time available at the end of an assembly line for testing a just instantiated product is limited. Even exploiting *regression-based techniques* on SPLs to reduce the efforts for testing a single product based on the already spent efforts for testing other similar products previously is unfeasible as long as precise definitions of "similarity" w.r.t. a chosen coverage criterion are missing [4]. Finally, successfully used *subset selection heuristics* which generate small sets of products that are assumed to be representative for all SPL products are improper for testing safety-critical software systems as long as there is no proof that this small set is *really* representative.

We address this problem and present a new model-based coverage-criteria-driven approach for safety-critical SPL testing. We can prove that our new SPL test suite generation algorithm efficiently generates a set of test cases (*test suite*) that achieves a *complete* test model coverage for *every* product of an SPL w.r.t. the chosen coverage criterion. For this purpose, our approach makes use of the 150% test model [5] which contains all test models of an SPL as special cases. By using the 150% test model it is possible to determine if a created a test case is executable on more than one product. Additionally, our approach utilizes the Quine-McCluskey algorithm [6] (a method used for minimization of boolean functions) that helps to efficiently keep a record of all product configurations which are left to be processed. This makes it possible to create a test suite that achieves a *complete* test model coverage for *every* product of an SPL without processing each product individually. Furthermore, during test suite generation our algorithm gathers information that simplifies the subsequent selection of a small, representative set of products on which this test suite can be executed.To identify a small, representative set of products of an SPL, for every test-model-driven approach it is necessary to assume that products with similar behaviors specified in their test models have similar implementations, i.e. produce the same verdicts for a test case that has identical traces in their related test models.

The remainder of the paper is organized as follows: In the following section we introduce domain specific terms. After that, we present our approach in detail in Section 3 and discuss it in Section 4. In the subsequent Section 5, we show how our work stands out from related work. Finally, in Section 6 we conclude the paper and present our plans for future work.

2 Basic Terms of SPL Testing

In the following we explain basic terms from the domain of SPL testing and model-based testing. A *software product line* (SPL) defines a set of features $F = \{f_1, f_2, \ldots f_n\}$. *Features* are increments of functionality explicitly stating commonality and variability parameters for *product configurations* of the SPL [7]. Theses features are combined into one software product which is interacting with components of the environment, i.e. sensors and actuators.

In this paper we use an embedded Alarm System (AS) SPL as running example. This SPL provides nine features $F_{AS} = \{AS,\ C,\ O,\ P,\ W,\ S,\ V,\ M,\ U\}$

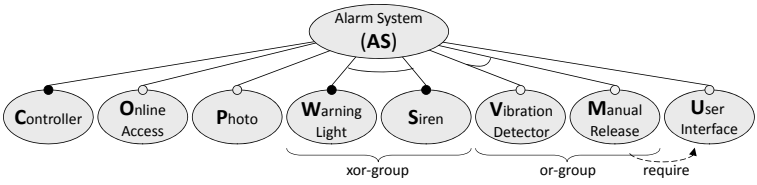


Fig. 1. Feature Model of the AS SPL

(cf. Figure 1). Depending on which features are integrated the functionality of a product in the AS SPL varies. The AS SPL contains products with up to two alarm levels. The alarm is set off if it is released manually (req. M) or the vibration detector detects a vibration over a certain time (req. V). By entering the first level, an alarm signal is sent out by a siren (req. S) or warning light (req. W). When the vibration did not stop after a certain time, the system enters level two. Entering this level the system may call the police (req. O) and/or send an evidence photo to the police (req. P). Additionally, the SPL offers the feature that a photo will be taken as security measure when a user interacts with the environment of the system (req. U).

In Figure 1 the features of the AS SPL are arranged in a FODA *feature model* [7]. The root node of a feature model is a special mandatory feature denoting the name of the whole SPL. Subnodes introduce further variabilities to their parent feature nodes: singleton subfeatures can be either *mandatory* or *optional* variabilities for their parent features, and groups of subfeatures define either *or* or *xor* (i.e. *alternative*) subset constraints among features in that group. Consequently, feature models introduce dependencies and constraints on feature combinations, thus limiting the set $\mathcal{P}(F)$ of potential combinations to a subset of *valid product configurations* $PC = \{pc_1, pc_2, \ldots, pc_k\} \subseteq \mathcal{P}(F)$ where each $pc_i \in PC$ corresponds to exactly one subset of selected features of F. Due to the constraints in the feature model of the AS SPL 32 valid product configurations exist, e.g. $pc_i = \{AS, C, P, S, U, V\}$.

In model-based testing, *test models* are used to specify the abstract behavior of one corresponding system-under-test. A test model tm is used to derive a set of test cases (*test suite*) that satisfy certain *coverage criteria*. The derivation happens either manually or automatically by using a test case generator. In this paper, we use deterministic *state machines* as test models, simply consisting of sets of states and transitions.

In SPL testing, each valid product configuration has its own test model. This results in a large number of test models $TM = \{tm_1, \ldots, tm_k\}$. A function $map : PC \rightarrow TM$ maps any valid product configuration $pc_i \in PC$ onto its defined test model $tm_i = map(pc_i)$. We require map to be a bijection, thus every product configuration $pc_i \in PC$ owns a unique behavioral specification tm_i.

To achieve a better maintainability and a better overall view in SPL testing it makes sense to combine all test models of an SPL, which usually are rather similar, into one "super" test model stm, a so-called *150% test model*. For the

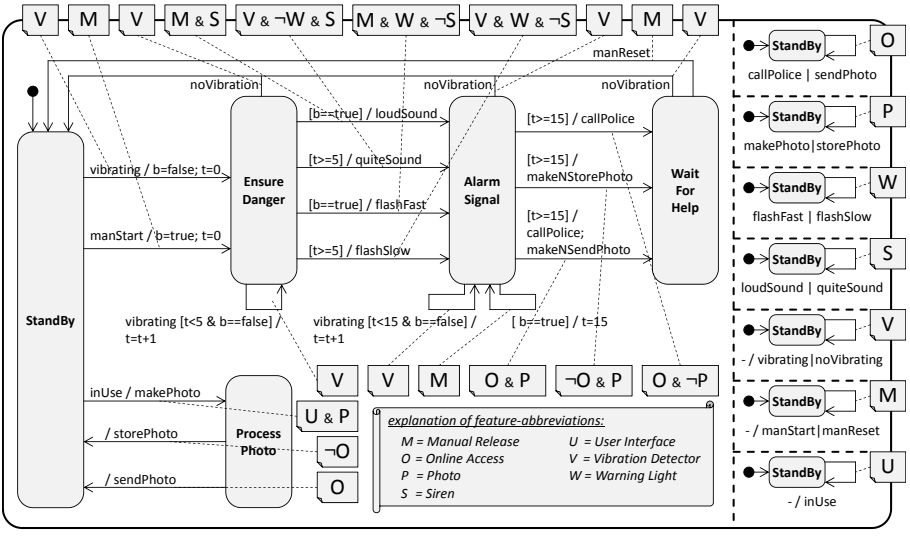
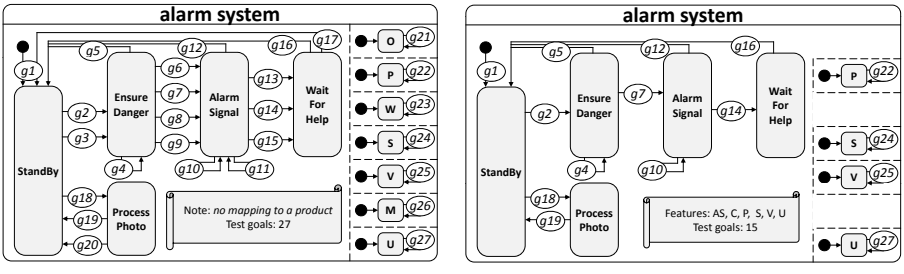


Fig. 2. 150% Test Model of the AS SPL with Annotated Selection Conditions



(a) 150% Test Model (Abstract) (b) 100% Test Model (Abstract)

Fig. 3. Abstract Test Models of the AS SPL with Annotated Test Goals (Transitions)

sake of a better discriminability, in the following, we call a test model for one specific product a *100% test model*. Each 100% test model $tm \in TM$ consists of a subset of states and transitions of the 150% test model *stm*, which is usually not an element of TM [5]. In our 150% test model, *map* is implemented by annotating states and transitions with logical formulas as *selection conditions* defined over features in F, which is exemplarily depicted in Figure 2. The 100% test model $tm_i = map(pc_i)$ for product configuration $pc_i \in PC$ can be derived by removing those states and transitions from the 150% model *stm* whose selection conditions are *not* satisfied for the feature combination in pc_i. For example, in Figure 3(b) the 100% test model of a product of the AS SPL is depicted.

Usually, in a testing process it is hard to know when to stop testing, thus a test end criterion must be selected. In model-based testing such test end criteria

are usually defined by means of *coverage criteria*, concerning fragments of a test model to be traversed in test case executions. Therefore, coverage criteria impose requirements for test suite generation from test models. Applied to test model tm, a criterion C selects sets of model fragments, so-called *test goals* $G = \{g_1, g_2, \ldots, g_l\}$, that refer to state machine artifacts, e.g., *all-states*, *all-transitions*, *all-transition-pairs*, etc. [8]. In this paper the set of test goals G is selected using the 150% test model stm as input. This set of test goals G is a superset of all test goals of all 100% test models of the SPL. For instance, considering *all-transitions-coverage* criterion applied to the 150% test model of the AS SPL selects all transitions as test goals, thus leading to 27 test goals $G = \{g_1, g_2, \ldots, g_{27}\}$ as shown in Fig. 3(a). Correspondingly, each 100% test model $tm_i \in TM$ contains a subset $G_i \subseteq G$ of these test goals depending on the transitions selected from stm via map. For instance, the 100% test model in Figure 3(b) owns 15 goals.

A *test case* consists of a sequence of inputs and expected outputs. A *test suite* $T = \{t_1, t_2, \ldots, t_m\}$ is a set of test cases $t_i \in T$. For a test suite T generated from a 150% test model stm, each test case $t_i \in T$ corresponds to a unique execution path of transitions in stm. We consider the following relations:

- $exec \subseteq T \times TM$, where $exec(t, tm) :\Leftrightarrow$ test case t is *executable* on the 100% test model tm, i.e., the execution path of t is contained in tm as it only consists of transitions of stm mapped into tm via map,
- $satisfy \subseteq T \times G$, where $satisfy(t, g) :\Leftrightarrow$ the execution of test case t *satisfies* test goal g selected for some coverage criterion C on stm, and
- $valid \subseteq T \times G \times PC$, where $valid(t, g, pc) :\Leftrightarrow satisfy(t, g) \wedge exec(t, map(pc))$, i.e., test case t is *valid* for test goal g on product pc if it is executable on the test model of product configuration pc and satisfies test goal g.

3 Complete SPL Test Suites

An *SPL test suite* $TS = (T', PC')$ contains test cases $t' \in T'$ for a set of products $PC' \subseteq PC$ of the SPL. We denote the set of all SPL test suites by $TS_{SPL} = \mathcal{P}(T) \times \mathcal{P}(PC)$, where T refers to the set of all test cases executable on stm. A *complete* SPL test suite TS_C achieves a *complete* test model coverage for every product of the SPL w.r.t. a certain coverage criterion C which defines a set of test goals G.

Definition 1. *(Complete SPL Test Suite)*
SPL test suite $TS_C = (T_C, PC') \in TS_{SPL}$ *with a set of test cases* $T_C \subseteq T$ *and valid product configurations* $PC' \subseteq PC$ *is* complete *for a set of test goals* G*, iff*

$$\forall g \in G, pc \in PC : \ (\exists t \in T : valid(t, g, pc) \Rightarrow (\exists t_g \in T_C : valid(t_g, g, pc))$$

The easiest way to obtain *complete* test model coverage for every product of the SPL is to compute for each 100% test model of an SPL a test suite that achieves complete coverage, and, afterwards, combine all test suites to TS_C.

This procedure follows a product-by-product approach and is inefficient for large SPLs. Instead, our algorithm avoids the iteration over every single PC. Our algorithm analyzes each created test case and if this test case is valid for more than one PC, all the PCs in this set are processed at once. For this purpose, our approach computes TS_C by deriving test cases from the 150% test model of an SPL and not from each 100% test model.

An SPL test suite derived from the 150% test model is *complete* if for all products of an SPL and for each test goal g in the 150% test model, this SPL test suite contains at least one test case $t_g \in T_C$ such that $valid(t_g, g, pc)$ is true for every product configuration $pc \in PC$ whose 100% test model contains the respective test goal g. It is important to recognize that a complete SPL test suite derived from a 150% test model is a superset of a test suite that achieves a complete 150% test model coverage under the assumption that both test suites use the same coverage criterion.

During test suite generation, our approach already associates each test case with a set of PCs for which this test case is valid. After the test suite generation is finished, these associated sets make it easier to select a small, representative set of products $PC_R \subseteq PC$ for the complete SPL test suite.

Definition 2. *(Representative Set of Products)*
A set of products $PC_R \subseteq PC$ of a complete SPL test suite $TS_C = (T_C, PC_R) \in TS_{SPL}$ is representative *for all product configurations PC, iff*

$$\forall g \in G, pc \in PC, t_g \in T_C : \ valid(t_g, g, pc) \Rightarrow (\exists pc_R \in PC_R : valid(t_g, g, pc_R))$$

3.1 Complete SPL Test Suite Generation – An Example

In this section, we explain our *complete SPL test suite generation* algorithm by applying it to the 150% test model of the AS SPL. The following explanation refers to the pseudo code of the algorithm in Figure 4. Additionally, we use Figure 5 to illustrate each step in the pseudo code.

Our algorithm iterates over all test goals and repeats each time the same steps (cf. line 7 of Figure 4). Consequently, it is sufficient to focus on one test goal. We chose test goal 14 (cf. Figure 3(a)). Out of all 32 valid PCs only 10 PCs have a corresponding 100% test model that contains test goal 14. For each of these 10 PCs at least one valid test case has to be created. Using a common product-by-product approach, it would be necessary to create 10 test cases - one for each PC. Applying our algorithm, only 4 instead of 10 test cases are created.

At the beginning (cf. first iteration in Figure 5), the test suite is empty and does not contain any test cases that satisfy test goal 14 for any PC, respectively. Before test case generation starts, the set of not yet processed PCs (*processPCset*) is reduced from 512 PCs to 32 valid PCs due to the constraints of the feature model (cf. line 9). The resulting formula is minimized to a DNF-formula by applying the well-known Quine-McCluskey algorithm [6]. This is necessary to efficiently keep a record of all PCs which are left to be processed. This minimized DNF-formula is depicted in *processPCset* of the first iteration

```
1  SuperTestModel stm;                    // 150% test model
2  dnfFormula processPCset := empty;      // DNF formula that describes set of not yet processed product configurations
3  cFormula inputPCset, outputPCset;      // conjunctions describing subsets of (un-)processed product configurations
4  List<TestCase> testsuite := empty;     // generated representative set of all product configurations plus test cases
5
6  // create for each test goal in G a representative set of test products plus test cases
7  for each g in G do {
8        // translate feature model with all constraints into DNF formula
9        processPCset := QMC.minimizeDNF( FeatureModel.getConstraintsAsPropositionalLogicFormula() );
10       do {
11             // select conjunction in DNF formula that references the smallest number of features
12             inputPCset := processPCset.getTermWithSmallestNumberOfLiterals();
13             // try to create a test case for selected subset inputPCset
14             testcase := TestCaseGenerator.create( g, inputPCset , stm );
15
16             if testcase was created then {
17                   // find set of PCs for which testcase is valid by analyzing its feature-flags
18                   outputPCset := FlagAnalyzer.findPCset( testcase );
19                   testsuite.add( testcase , outputPCset );
20                   // remove successfully processed subset of product configurations from DNF description
21                   processPCset : = QMC.minimizeDNF( processPCset ∧ ¬outputPCset );
22             } else {
23                   // removes subset for which no test case was found from the set of all not yet processed PCs
24                   processPCset := QMC.minimizeDNF( processPCset ∧ ¬inputPCset );
25             }
26       } while ( processPCset ≠ empty )
27 }
```

Fig. 4. Algorithm to Generate a Complete SPL Test Suite

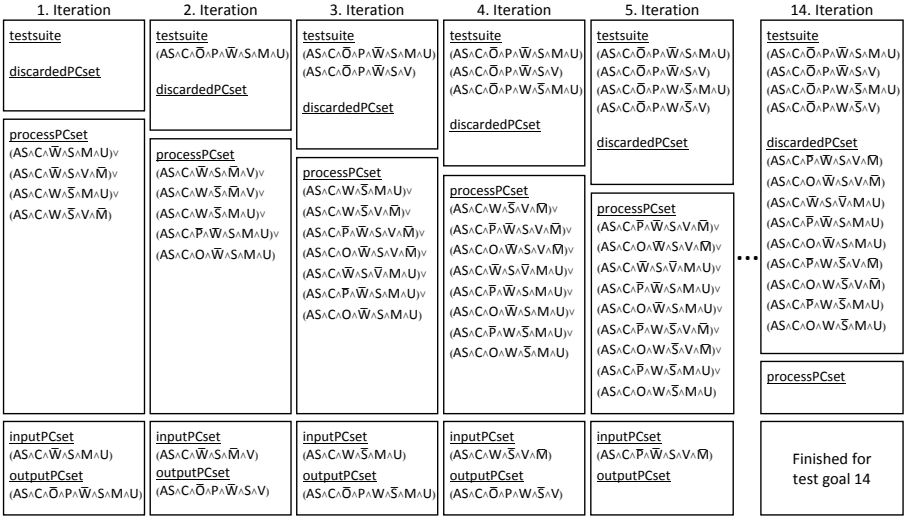


Fig. 5. Generating Test Cases for Test Goal 14 from the AS SPL 150% Test Model

in Figure 5. After that, for efficiency reasons a subformula *inputPCset* of the formula *processPCset* is selected which references the smallest number of features of the SPL (one of the conjunctions of *processPCset* which has a DNF representation). The subformula *inputPCset* represents the set of all PCs that contain features AS, C, S, M, and U, but not W. This subformula is passed to the test case generator combined with the 150% test model and test goal 14 (cf. line 14). The test case generator creates a test case for any appropriate PC in this passed set of PCs, represented by the subformula *inputPCset*. Due to some preparations in the 150% test model of the AS SPL (see Sections 3.2 and 3.3) it is possible to identify all PCs for which the generated test case is also valid (cf. line 18). These PCs are described by *outputPCset*. In the first iteration *outputPCset* describes two PCs ($AS \wedge C \wedge \neg O \wedge P \wedge \neg W \wedge S \wedge M \wedge U \wedge (V \vee \neg V)$). Next, the generated test case and its associated PCs (*outputPCset*), for which the test case is valid, is added to the test suite (cf. line 19). Finally, due to the fact that a valid test case was created that satisfies test goal 14 for the two PCs described by *outputPCset*, these two PCs are excluded from the unprocessed PCs in *processPCset* (cf. line 21). In the second iteration, the test suite contains the previously generated test case. From the 2nd to 4th iteration, three additional test cases are created. In the 5th iteration, four test cases that satisfies test goal 14 were generated. For each of these 10 PCs, one of these four test cases is valid owing to their different execution paths in the 150% test model. That means for any of these 10 PCs at least one test case is executable on the corresponding 100% test model and satisfies test goal 14. From the 5th to 14th iteration the test case generator cannot create any more test cases for the remaining 22 PCs in *processPCset* because there exists no test case satisfying test goal 14. The number of iterations needed from the 5th iteration to the end can be shortened by selecting more than one conjunction in line 12. Summarizing, our algorithm generated not more than 4 test cases satisfying test goal 14 for 10 PCs of the AS SPL compared to 10 test cases generated by an product-by-product approach.

3.2 150% Test Model Preparation

The derivation of a *complete SPL test suite* TS_C from a 150% test model *stm* requires an appropriate *test case generator*. So far, common test case generators support interfaces to pass the test goal g and the test model *tm*, i.e. `createTestCase(g,tm)`. But this is insufficient for the generation of test cases from a 150% test model *stm*. It is necessary to pass a valid product configuration $pc \in PC$ as well, such that the test case generator can instantiate the corresponding 100% test model from *stm*. Consequently, an appropriate test case generator must support at least the interface `createTestCase(g,stm,pc)`.

Prior to test case generation, an embedding of the mapping function *map* (see Section 2) into the 150% test model is to be provided. More precisely, each transition which is annotated by a selection condition now includes this condition as additional clause in its transition guard (cf. Figure 6). After that, only *valid* test cases for a valid product configuration *pc* are generated from 150% test model *stm* by internally instantiating it to the 100% test model $tm = map(pc)$.

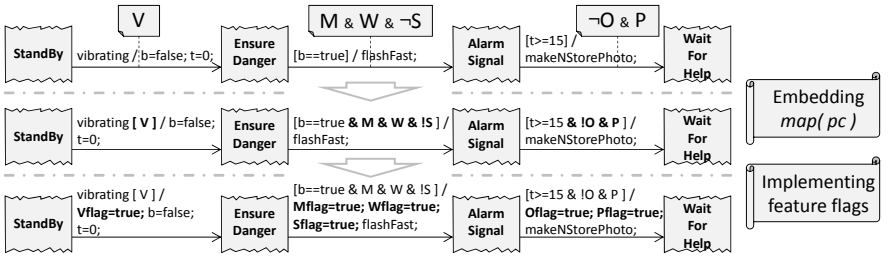


Fig. 6. 150% Test Model Preparation for SPL Test Suite Generation

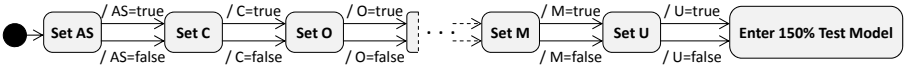


Fig. 7. Setup Section for the 150% Test Model of the AS SPL

We use the model-based testing framework Azmun [9] as test case generator, which is based on the model checker NuSMV. We extended Azmun by implementing a plug-in that supports the interface `createTestCase(g,stm,PC`$_I$`)`. As third input parameter we use a *set* of product configurations $PC_I \subseteq PC$ instead of a single product configuration $pc \in PC$. The advantage is that the test case generator has the possibility to search in this set PC_I of product configurations for any product configuration $pc \in PC_I$ for which a *valid* test case t for test goal g exists in the corresponding test model $tm = map(pc)$. If no test case for test goal g can be found, then there exists no test case for test goal g for any of these product configurations in PC_I. In this case, all $pc \in PC_I$ were processed in one go and discarded (as irrelevant) for this specific test goal (cf. *discardedPCset* in Figure 5). In our algorithm, we specify the input set of product configurations PC_I in an implicit way using a propositional formula *inputPCset* (cf. line 14 in Figure 4). Examples for such formulas *inputPCset* are depicted in Figure 5. *inputPCset* assigns conditional values to some features in F, denoting either presence (*true*) or absence (*false*) constraints on those features to hold for all product configurations in PC_I. The formula then conjuncts all predicates over features for which a constraint is given.

In the usual case, PC_I contains more than one product configuration. To ensure that a created test case is *valid* for at least one $pc \in PC_I$, it must be guaranteed that the variable 150% test model is instantiated to exactly one 100% test model before the test case generator starts searching for the test case. This is achieved by determining the product configuration beforehand. Therefore, we add a *setup section* to the 150% test model consisting of a chain of transitions for setting the *feature variable* of each feature, depending on whether this feature should be *present* (`true`) or *absent* (`false`). In Figure 7, the setup section of the AS SPL 150% test model is depicted. The test case generator arbitrarily decides the presence/absence for each feature provided that the resultant product configuration pc is in PC_I.

By initially running through some path of this setup section, the test case generator configures the subsequent 100% test model to be $tm = map(pc)$. When this setup is completed, the values of the feature variables cannot be changed afterwards in the 150% test model. For considering another $pc' \in PC$, the setup section has to be traversed again.

Owing to these arrangements it is guaranteed that a valid test case will be derived from the 150% test model if a valid set of product configurations will be passed to the test case generator.

3.3 Valid Test Case for a Set of Product Configurations

As described previously, the test case generator creates (if possible) a test case t for test goal g from the 150% test model for *some* appropriate product configuration $pc \in PC_I$. Usually, the created test case t is valid for many PCs. By $PC_O \subseteq PC$, we refer to the set of *output* product configurations for which the generated test case t is valid. There exists at least one $pc \in PC_O \cap PC_I$. To derive the whole set PC_O, it is necessary to know which features must be present and which features must be absent in the product configuration of each $pc' \in PC_O$ to traverse the transitions of the execution path of the test case. For that reason, we keep a record which features' presence or absence is necessary for the execution of the generated test case.

This is done using a flag variable for each feature. These flags are implemented in the action part of transitions in *stm*. As a result, a flag is set to `true` if exactly this value of the corresponding feature variable is necessary to traverse at least one transition in the execution path of the test case. If this flag remains `false` (default value) then the presence or absence of the corresponding feature has no impact on whether the generated test case is executable on $pc' \in PC_O$.

For example, consider Figure 6: to traverse the second transition the feature `M` and `W` must be present, but feature `S` must be absent. For that reason, the corresponding flags `Mflag`, `Wflag`, and `Sflag` are set to *true* in the action part. By analyzing the values of the feature variable and the feature flag of each corresponding feature $f \in F$, it is possible to determine for which PCs the test case t is valid. In our algorithm, PC_O is specified by a formula *outputPCset* (cf. line 18 in Figure 4) which is constructed by conjunction of values of feature variables whose corresponding flags are set to true.

3.4 Complete SPL Test Suite Generation

The following descriptions relate to the algorithm presented in Figure 4 as well as to Definition 1 and 2. A full execution of the presented algorithm generates an SPL test suite $TS_C = (T_C, PC_R) \in TS_{SPL}$ from a given 150% test model of the SPL for coverage criterion C.

Theorem 1. *SPL test suite* $TS_C = (T_C, PC_R)$, $T_C \subseteq T$, *is* complete *and the set* $PC_R \subseteq PC$ *of products is* representative *w.r.t. to coverage criterion* C.

Sketch of Proof: For each test goal g in the 150% test model the outer loop (cf. line 7-27) generates a formula *processPCset* that describes all the valid PCs that have not yet been processed and for which a valid test case has to be generated. Afterwards, the inner loop (cf. line 10-26) generates a subformula *inputPCset* of the formula *processPCset* (cf. line 12) that describes a subset of those PCs for which a test case that satisfies g is still missing. Then, the algorithm generates (if possible) for one product configuration in the set of PCs described by *inputPCset* a new test case t for test goal g (cf. line 14). If such a test case t does not exist then the formula *inputPCset* describes a set of PCs for which no test case exists such that test goal g is satisfied. Otherwise, the test case t is added to the test suite combined with the associated set of PCs described by *outputPCset* (cf. line 19). The formula *outputPCset* characterizes a nonempty subset of *processPCset* for which the created test case t is valid (cf. line 18 or see Section 3.3). The inner loop ends by computing a new formula that describes the *new* set of not yet processed PCs by concatenating the old formula stored in *processPCset* with the negation of either *outputPCset* (cf. line 21) or *inputPCset* (cf. line 24).

In each iteration, either ($inputPCset \cap outputPCset$) or at least *inputPCset* describe nonempty subsets of the set of not yet processed PCs described by *processPCset*. Therefore, the inner loop (cf. line 10-26) reduces the number of unprocessed PCs with each iteration. As a consequence the inner loop of the algorithm always terminates and generates for the just regarded test goal g a set of test cases such that for each PC, that contains test goal g, at least one valid test case is in this set. Furthermore, the outer loop (cf. line 7-27) repeats the process for all test goals. This loop terminates due to the fact that G is a finite set. In the end, our algorithm creates a TS_C (cf. Definition 1) which contains for each test case t the associated set of PCs for which t is valid.

To derive an explicitly defined representative set of products PC_R for the complete SPL test suite TS_C it is necessary to select for each test case t at least one PC for which test case t is valid (cf. Definition 2). This can be easily ensured by selecting one PC from the set of products that was associated with test case t during test case generation (cf. line 19). In the end, the number of products in the representative set PC_R depends on the heuristics which is used to select the PCs. A small, representative set of products is achievable by selecting only those products that were already selected for other test cases. The development of a sophisticated algorithm, which searches for a minimal, representative set of products, is subject of our future research activities.

4 Discussion

In our running example, the AS SPL, there exist 32 valid products. If these 32 products are tested individually by using a brute-force "product-by-product" approach (which does not select a representative set of products) it would be necessary to create 432 test cases in total to achieve a full test model coverage w.r.t. the all-transitions coverage criterion for all 32 products. For comparison, by applying our *complete SPL test suite generation* approach to the 150% test

model of the AS SPL it is possible to achieve the same full test model coverage by only creating 43 test cases. In addition to this, it was possible to select a representative set of 6 products from 32 possible products by selecting suitable products from those sets that are associated with the test cases in the complete SPL test suite. As a first step we only used a brute-force approach for the selection, although we are planning to do research for suitable heuristics. If these 6 products are tested individually then 120 test cases in total have to be created. Using our new approach it is sufficient to create only 43 test cases to achieve the same complete coverage. Detailed data about our evaluation experiments are published in [10].

We also applied our approach successfully to a *body comfort system* (BCS) SPL, a real-world SPL from the automotive domain. The BCS SPL consists of 12 features and, due to the constraints in its feature model, 312 valid PCs exist. Its corresponding 150% test model contains 152 transitions and 55 states. We applied our algorithm on the BCS SPL and created a complete SPL test suite w.r.t the all-transitions coverage criterion. The generated complete SPL test suite contains not more than 283 test cases and the corresponding representative set contains not more than two products. This very small number of products in the representative set is caused by the small number of exclusion-constraints between the features. These remarkable results for both SPLs, AS SPL and BCS SPL, show how efficiently our new approach generates complete SPL test suites, leading to a considerable reduction of costs for SPL testing.

It is important to note that in this paper we ignored redundant test cases, i.e. it may happen that generated test cases satisfy more than one test goal (accidently). Hence, for our running example, the AS SPL, the number of test cases is rather large and contains quite a number of redundant test cases. In such a case the generated set of test cases may be reduced as, e.g., shown in [11].

4.1 Threats to Validity

A *complete SPL test suite* created by our algorithm allows the subsequent selection of a small, representative set of products. To ensure that this representative set is really representative, it is necessary to require a strong correlation between the similarity of product implementations and the similarity of their related test models. In other words, we assume that two products with similar test models have similar implementations and behavior. Consequently, when the execution path of a test case that is derived from the test model of product *p1* is also valid for the test model of product *p2*, then our approach implies that the test case will always produce identical verdicts (pass, fail, ...) when executed on both products, *p1* and *p2*, in practice. However, if the assumption is dropped then any test-model-driven attempt is doomed to fail that tries to identify a small, representative set of products of a large SPL.

In real-world automotive SPLs the size of used models is usually rather large. Our approach scales very well with large SPLs, because our algorithm avoids the iteration over every single PC to create a new test case. Instead, our algorithm analyzes each created test case and if this test case is valid for more than one PC,

all the PCs in this set are processed at once. For test case generation purposes we use the model-based testing framework Azmun [9], which integrates the model checker NuSMV. Testing with model checkers is still a field of research and the testing community has different opinions concerning its feasibility and the state space explosion problem [12]. For our research work a model checker is suitable due to its flexibility and great capabilities for model queries. For real-world SPLs with large models, we recommend more efficient model-based testing tools like Conformiq ATD or Rhapsody ATG. However, currently these tools do not support an appropriate interface that is needed for our approach (see Section 3.2).

5 Related Work

Studying related research we have identified three categories for SPL testing approaches. Approaches in these categories are more or less effective and efficient to achieve complete test model coverage for all products of an SPL.

Due to the fact that we pay particular attention to the automotive domain with large SPLs, we skip the first category "Contra-SPL-philosophy". Approaches in this category ignore the SPL-philosophy of reuse and, thus, are only appropriate for small SPLs [3].

The second category "Reuse-Techniques" includes techniques that are applied to reuse test artifacts (e.g. test cases and data) to reduce the test effort for SPLs. Typically, these approaches either make use of *regression testing techniques* to incrementally test products or *reuse and adapt domain tests* during application testing. The former ones are used in [4] to incrementally test products of an SPL treating the different variants of products as changes that have to be retested. This approach struggles with the challenging tasks to (1) identify a suitable product to start with and (2) to find out what needs to be retested. The latter ones, reusing and adapting domain tests, are created during domain engineering for product tests. Especially, model-based test approaches are used for that purpose. Model-based testing approaches provide the basis for SPL testing, due to their reusability and suitability to describe variability. A summary of model-based testing approaches for SPLs can be found in [1]. Frequently, statecharts, activity diagrams, and sequence diagrams are used to specify the behavior of software systems for model-based testing. CADeT [13], ScenTED [14] and Hartmann et al. [15] utilize reusable test models by means of activity diagrams. Instead of activity diagrams, we make use of state machines to derive test cases. In [16] a single state machine is used as test model that describes the functionality of an entire SPL. We also make use of one single test model, called a 150% test model, to derive test cases, according to the idea of [5]. The commercial variant management tool pure::variants [17] in interaction with the modeling tool IBM Rhapsody and ATG supports the modeling and trimming of a 150% model. One major drawback of this whole category is that all approaches still aim at deriving test cases for individual products. The test effort may be reduced because of reuse-techniques, but being confronted with millions of derivable products, these

approaches might still not be sufficient. Strategies for the selection of representative sets of products are also out-of-scope.

In the third category "Subset-Heuristics" a subset of products of the SPL for testing is created, instead of testing every possible product. The subsets are generated on the basis of a certain coverage criterion. Scheidemann introduced a heuristics to generate a representative subset of products covering all SPL requirements [3]. Unfortunately, her approach does not scale for large SPLs and does not give any guarantees concerning model/code-based coverage criteria. Kim et al. [18] use static analysis to determine for an existing test case, which features have to be mandatory present or absent for it to be executed. Thus, they are able to determine a set of products to execute all test cases for the entire SPL. In our approach we use a similar concept to generate a complete SPL test suite very efficiently. In [19] and [20] combinatorial feature combination is used to generate a set of products covering all t-wise feature combinations. The corresponding algorithms take all constraints and hierarchies of the feature model into account and generate small (representative) sets of products efficiently. Unfortunately, no guarantees are given concerning required model/requirements-based coverage criteria. Furthermore, generating test cases for the computed sets of products is usually done on a product-by-product basis even in the case of [19], where a 150% test model is used to generate test cases for selected SPL products.

6 Conclusion and Future Work

The SPL test suite generation approach presented in this paper is - as far as we know - the first published approach that uses a 150% test model of the whole SPL as a starting point and generates a *complete* SPL test suite in such a way that (1) the created test cases satisfy required model-based coverage criteria for *every* product of the SPL and (2) the selection of a representative subset of all products is supported that allows for the execution of *all* test cases. To ensure that the selected representative set is really representative, a strong correlation between the similarity of product implementations and the similarity of their related 100% test models is required. Nevertheless, various publications with case studies from the automotive domain show that our SPL testing approach would be very useful in practice despite of the just mentioned restriction.

Our new approach was exemplary applied to a small SPL and additionally to a larger SPL from the automotive industry. It could be shown that our approach is efficient in complete SPL test suite generation for the test models of all products and still achieves full test model coverage. Additionally, the subsequent process for selecting a representative set of products is simplified by associating each generated test case with a set of products on which this test case is executable. Based on the promising results, in future research activities we will develop a more sophisticated algorithm for the minimization of the representative set of products and adapt our test suite reduction approach to a complete SPL test suite.

References

1. Oster, S., Wübbeke, A., Engels, G., Schürr, A.: Model-Based Software Product Lines Testing Survey. In: Zander, J., Schieferdecker, I., Mosterman, P. (eds.) Model-based Testing for Embedded Systems. CRC Press/Taylor&Francis (2011)
2. ISO: ISO - International Organization for Standardization. Website (2011), http://www.iso.org/iso/ (visited on May 2, 2011)
3. Scheidemann, K.: Verifying Families of System Configurations. PhD thesis, TU Munich (2007)
4. Engström, E., Skoglund, M., Runeson, P.: Empirical evaluations of regression test selection techniques. In: Rombach, H.D., Elbaum, S.G., Münch, J. (eds.) Proc. of ESEM 2008, pp. 22–31 (2008)
5. Grönniger, H., Krahn, H., Pinkernell, C., Rumpe, B.: Modeling Variants of Automotive Systems using Views. In: Modellierung (2008)
6. Jain, T.K., Kushwaha, D.S., Misra, A.K.: Optimization of the Quine-McCluskey Method for the Minimization of the Boolean Expressions. In: Proc. of the ICAS 2008, pp. 165–168. IEEE, Los Alamitos (2008)
7. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Carnegie-Mellon University Software Engineering Institute (1990)
8. Souza, S., Maldonado, J., Fabbri, S., Masiero, P.: Statecharts Specifications: A Family of Coverage Testing Criteria. In: CLEI 2000 (2000)
9. Haschemi, S.: Azmun - The Model-Based Testing Framework. Website (2011), http://www.azmun.de (visited on May 2, 2011)
10. Cichos, H., Oster, S., Lochau, M., Schürr, A.: Extended Version of Model-based Coverage-Driven Test Suite Generation for Software Product Lines. Technical Report 07, TU Braunschweig (2011)
11. Cichos, H., Heinze, T.S.: Efficient Test Suite Reduction by Merging Pairs of Suitable Test Cases. In: Dingel, J., Solberg, A. (eds.) MODELS 2010. LNCS, vol. 6627, pp. 244–258. Springer, Heidelberg (2011)
12. Fraser, G., Wotawa, F., Ammann, P.: Testing with Model Checkers: A Survey. Software Testing, Verification and Reliability 19, 215–261 (2009)
13. Olimpiew, E.M.: Model-Based Testing for Software Product Lines. PhD thesis, George Mason University (2008)
14. Reuys, A., Kamsties, E., Pohl, K., Reis, S.: Model-Based System Testing of Software Product Families. In: Pastor, Ó., Falcão e Cunha, J. (eds.) CAiSE 2005. LNCS, vol. 3520, pp. 519–534. Springer, Heidelberg (2005)
15. Hartmann, J., Vieira, M., Ruder, A.: A UML-based Approach for Validating Product Lines. In: Geppert, B., Krueger, C. (eds.) Proc. of the SPLiT 2004, pp. 58–65 (2004)
16. Weißleder, S., Sokenou, D., Schlingloff, H.: Reusing State Machines for Automatic Test Generation in ProductLines. In: Proc. of the MoTiP 2008 (2008)
17. Pure-Systems: pure-systems GmBH. Website (2011), http://www.pure-systems.com (visited on May 2, 2011)
18. Kim, C.H.P., Batory, D.S., Khurshid, S.: Reducing Combinatorics in Testing Product Lines. In: Proc. of the AOSD 2011, pp. 57–68. ACM, New York (2011)
19. Oster, S., Markert, F., Ritter, P.: Automated Incremental Pairwise Testing of Software Product Lines. In: Bosch, J., Lee, J. (eds.) SPLC 2010. LNCS, vol. 6287, pp. 196–210. Springer, Heidelberg (2010)
20. Perrouin, G., Sen, S., Klein, J., Traon, B.B.Y.L.: Automated and Scalable T-wise Test Case Generation Strategies forSoftware Product Lines. In: ICST 2010, pp. 459–468 (2010)