Pairwise Feature-Interaction Testing for SPLs: Potentials and Limitations

Sebastian Oster TU Darmstadt
Darmstadt, Germany
oster@es.tu-darmstadt.de

Malte Lochau TU Braunschweig
Braunschweig, Germany
lochau@ips.cs.tu-bs.de

Marius Zink TU Darmstadt
Darmstadt, Germany
mzink@es.tu-darmstadt.de

Mark Grechanik Accenture Labs & UIC
Chicago, IL 60601
drmark@uic.edu

ABSTRACT

A fundamental problem of testing *Software Product Lines (SPLs)* is that variability enables the production of a large number of instances and it is difficult to construct and run test cases even for SPLs with a small number of variable features. Interacting features is a foundation of a fault model for SPLs, where faults are likely to be revealed at execution points where features exchange information with other features or influence one another. Therefore, a test adequacy criterion is to cover as many interactions among different features as possible, thus increasing the probability of finding bugs. Our approach combines a combinatorial designs algorithm for pairwise feature generation with model-based testing to reduce the size of the SPL required for comprehensive coverage of interacting features. We implemented our approach and applied it to an SPL from the automotive domain provided by one of our industrial partners. The results suggest that with our approach higher coverage of feature interactions is achieved at a fraction of cost when compared with a baseline approach of testing all feature interactions.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.8 [**Software Engineering**]: Metrics; D.2.13 [**Software Engineering**]: Reusable Software—Reuse models

General Terms

product lines, combinatorial testing, model-based testing

Keywords

reusable test model, feature model, feature interaction.

SPLC '11, August 21-26, 2011, Munich, Germany

1. INTRODUCTION

Software product-line (SPL) architectures are gaining a widespread acceptance, since they enable stakeholders to economically produce different applications from the same architecture family, where these applications are differentiated by features (i.e., units of functionality) [7]. **Variability** specifies at what point features are attached to other features [5, pages 93-95]. A fundamental problem of testing an SPL is that variability enables producing a large number of instances of this SPL, and it is often not feasible to test each instance individually. According to [24] testing each individual instance is only possible for SPLs with a small number of instances. A straw man solution is to generate an instance of an SPL under test containing all features and apply standard testing approaches to this instance. Unfortunately, this solution is not effective since it does not address situations when features are removed from an instance of the SPL, thereby changing the way how features interact and what faults may appear as a result of these interactions.

A feature interaction occurs when one or more features modify or influence other features [11]. Interacting features is a foundation of a fault model for SPL, where faults are likely to be revealed at execution points where features exchange information with other features or influence one another [1, page 557]. Therefore, a test adequacy criterion is to cover as many interactions among different feature as possible, thus increasing the probability of finding bugs.

We created a novel approach for effective testing of feature interactions in SPL by combining an *combinatorial designs (CD)* algorithm for pairwise feature generation with model-based testing to reduce the size of the product line required for comprehensive coverage of interacting features. That is, features from models are combined in products using a pairwise combinatorial strategy [3]. The intuition behind our approach is that we use statecharts as a test model for our SPL whose states are mapped to features in feature models. Pairing this combined feature-statechart model with the CD pairwise algorithm, we produce test models for the SPL that have a higher likelihood of diverse feature interactions. We are thus able to interrelate feature model coverage and test model coverage. This is, according to the best of our knowledge, the first contribution to do so. This paper makes the following contributions:

- We implemented our approach (MoSo-PoLiTe [18]) and we use it here on an SPL from the automotive domain, a simplified extract of a Body Comfort System [15]

provided by one of our industrial partners.

- We establish a coverage criteria for feature-oriented model-based testing and apply our MoSo-PoLiTe tool chain allowing to generate a subset of configurations realizing 100% pairwise feature interaction coverage and automatically generate corresponding test cases on the basis of model-based coverage criteria.

- Finally, we share the results of our industrial case study and discuss the potentials and limitations of the pairwise testing approach.

2. CASE STUDY: BCS-SPL

To illustrate the approach, we make use of a sample SPL from the automotive domain, a simplified extract of a *Body Comfort System* (BCS) [15] provided by one of our industrial partners serves as our running example. We adapted the original BCS at some points to obtain further interesting cases for our testing approach. Fig. 1 depicts the feature model of the BCS example.

Feature models (FMs) are frequently used to describe the variable and common parts within an SPL. Each feature represents a system property that is relevant to some stakeholder [4]. FMs are not able to capture all kinds of important properties of an SPL but are designated to be easily readable by the stakeholders. They are complemented by other development artifacts, e.g., natural language descriptions, executable models, or even code fragments. These artifacts are then mapped to the corresponding features by means of traceability relationships. We use a FODA-like FM [12] with mandatory, optional, or, and alternative features as well as binary exclude and require cross-tree dependencies.

In our running example the *HMI* provides control elements and displays for BCS functions and is optionally enhanced by further Status *LEDs* displaying current diagnostics values of BCS features at runtime. The Door System consists of a Power Window (PW) and an (optionally heatable) Electric Exterior Mirror (EM). The Power Window allows for moving the door window upwards and downwards by tipping corresponding control buttons. In case of a Manual PW ($ManPW$), the button is to be hold for the window to keep moving, against what the Automatic PW ($AutPW$) keeps the window moving until stopped, either by reaching the uppermost/lowermost position, or by (re-) pressing some button. Both PW variants are equipped with a finger protection (FP) mechanism that stops the upward movement if an obstacle is detected.

The *Security* feature decompose into optional Central Locking System (CLS), Remote Control Key (RCK), and Alarm System (AS). In addition to locking the doors, CLS also hinders the PW to be opened, and, if $AutPW$ is selected, CLS automatically closes open windows before locking them. Automatic Locking (AL) further releases CLS if the car exceeds a specific velocity value. Besides locking/unlocking CLS, RCK has further variabilities for controlling the $AutPW$ (CAP), the Alarm System (CAS), and adjusting the Exterior Mirror (AEM). The Safety Function (SF) automatically locks the car after an idle period. Finally, the Alarm System is enhanceable by an Interior Monitoring (IM).

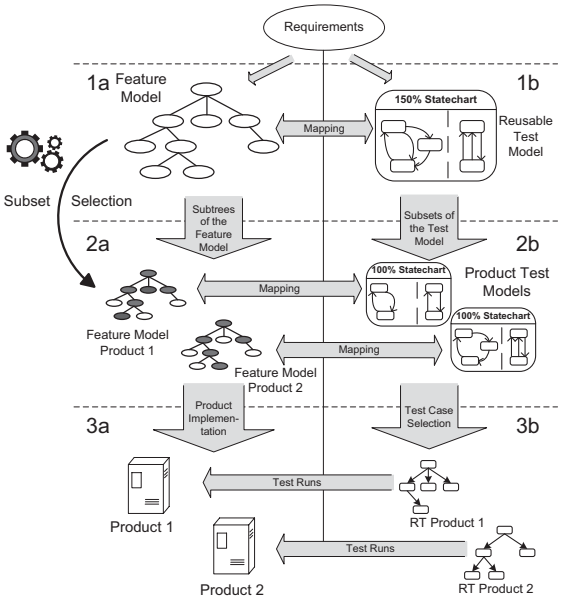

Figure 2: SPL Testing according to the MoSo-PoLiTe Concept [17]

3. COMBINATORIAL MODEL-BASED SPL TESTING MOSO-POLITE

In the following we describe the concept of our contribution which is based on our previous work [17, 13, 18]. MoSo-PoLiTe combines three basic concepts: **feature model-based testing**, **combinatorial testing**, and **model-based testing**. In [17], we focused on the combinatorial testing part of MoSo-PoLiTe, whereas, we here focus on the reusable test model and the results of applying MoSo-PoLiTe to our automotive case study.

Figure 2 depicts the combination of the three concepts in our testing approach. First, an FM comprising the functional properties of the SPL is created on the basis of the requirements (1a). The following assumptions led us to the idea of using an FM for our testing purposes: FMs are frequently used to parametrize products that are instances of some product line, thus the logical consequence is to use an FM for test configuration as well. The second reason is that FMs represent the variable and common requirements of an SPL, thus this structured schematic representation of SPL requirements should be taken into account to test against the SPL requirements.

A test model, representing the behavior of the entire SPL, is created on the basis of the requirements or the system specification (1b). Each valid subtree of the FM describes a variant of the SPL (2a) which is constructed via feature selection. In our case, the selection algorithm is our combinatorial subset selection. Using a suitable mapping, the selection of features results in the (semi)-automatic derivation of a corresponding configuration specific test model out of the SPL test model (2b). The selected configurations are completed to be real products (3a) and the test cases (3b) derived from the configuration specific test models cover the products to a certain extend which we are going to inspect in the remainder of this contribution.

Next, we describe the selection of the combinatorial subset

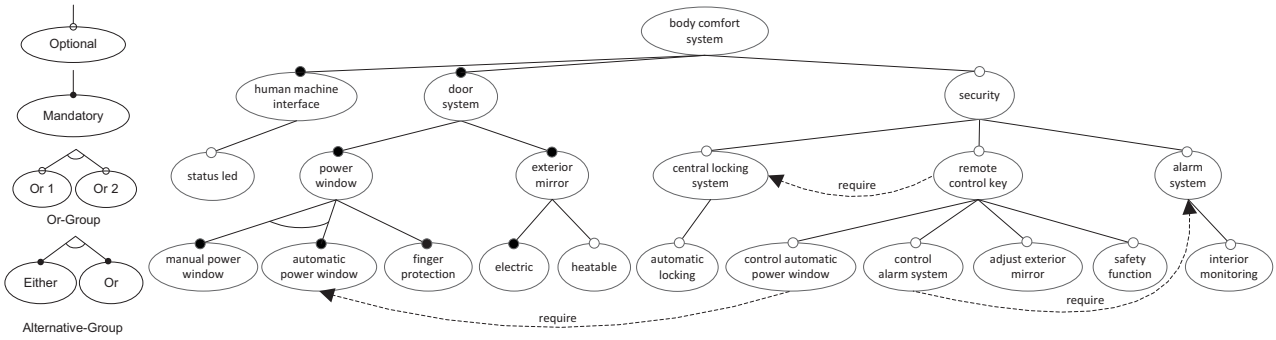

Figure 1: BCS-SPL – FODA Feature Model

followed by a description of our model-based test approach.

3.1 Subset Selection Heuristics

SPL testing based on subset approaches aim at selecting representative products-under-tests from the set of all valid product configurations on an SPL defined by the FM. An SPL organizes a set of features $F = \{f_1, f_2, \ldots, f_n\}$ in *feature models* $FM(F) \subseteq \mathcal{P}(F)$ that restrict feature combinations to *valid product configurations* $PC \in FM(F)$ For the set $\mathcal{FM}_F$ of all feature models $FM(F) \in \mathcal{FM}_F$ over F, a *subset selection function*:

$$S : \mathcal{FM}_F \rightarrow \mathcal{P}(\mathcal{P}(F))$$

selects feature combinations $PC_{UT} \subseteq FM(F)$ according to a combinatorial criterion on the product space $FM(F)$.

For instance, our pairwise testing approach generates a subset of all possible products of the SPL that covers all valid pairs of features. Valid pairs are pairs of features which can be included within a product considering the hierarchy, dependencies, and the feature notations. Furthermore, a valid pair can also consist of unselected features e.g. if two features f_i and f_j exclude each other then f_i and *not* f_j and vice versa are valid pairs (see Sect. 4).

Selecting a minimal subset of configurations covering all pairs of features can be described as follows.
Given: All possible product configurations *products*(FM) of the SPL, a set of *valid* feature pairs f_i, f_j.
Problem: Find a minimal subset $S(FM)$ covering all pairs of features.

Such a representative set of products that covers all pairs is called a *hitting set* [23]. Actually, the optimization problem of finding the *minimal* subset of products is the minimum cardinality hitting set problem which is, in the general case, NP-hard [23].

We apply pairwise testing to SPLs by translating the FM into a binary constraint satisfaction problem (CSP) [17]. We then generate a set of valid configurations containing all valid pairs of features. Testing this set of products is equivalent to pairwise testing the whole SPL. The developed pairwise algorithm can handle *dependencies* between features and guarantees the generation of valid configurations containing all valid pairs of features [17].

3.2 Reusable Statechart Test Model

Model-based testing (MBT) [26] aims at the automation of design and application steps of testing activities for detecting failures in (software) system implementations. We focus on the generation of (abstract) test cases with oracle from a *behavioral* model: for systems with potentially infinitely many executions, MBT techniques generate a finite set of behavioral test cases. Therefore, the model provides a behavioral specification that relates system inputs to expected outputs. Representative executions are successively selected as test cases from the model until some test end criterion, usually a coverage criterion, is met. Concerning embedded systems in particular, the emulation of environmental sensor/user stimuli sequences serve as test inputs, and the oracles define corresponding output signals expected for the actuator components.

For our upcoming discussions we simply regard a test model $TM = \{E_1, E_2, \ldots, E_n\}$ to consist of a (finite) collection of modeling artifacts $E_i \in TM$, $1 \leq i \leq n$. These artifacts are of arbitrary internal structure and their interrelations and compositions depend on the modeling formalism under consideration. A coverage criterion applied to test model TM then selects (assemblies) of artifacts to be covered, i.e., traversed by test case executions.

Here, we use statecharts (SC) as test models [8] . Statechart modeling is nowadays widely used and adopted, e.g., as UML state machines. For the development and implementation of reactive/embedded control systems, statecharts constitute an industrial de-facto standard, e.g., underlying the MATLAB/Simulink/Stateflow tool set.

A statechart is an Extended Finite State Machine (EFSM) enriched with several additional concepts for intuitively specifying complex system behavior: (1) composite states for nested and concurrent sub machines, (2) complex transition labels for explicit computational effects on shared variables, and (3) a communication mechanism between sub systems via event broadcasts. Based on well-defined operational semantics [9], statechart specifications can be used for simulation and (automated) implementation derivation, as well as for static analysis of properties such as deadlocks, reachability, and validity of execution sequences [14].

We will use statechart specifications for both, as a basis for capturing commonalities and variations of product implementations in an SPL, and as test models for reasoning about test case selection and coverage criteria. Fig. 3 shows a sample statechart, the *Manual Power Window* sub machine from the BCS-SPL.

The basic states encode vertical window positions `up`, `down`, and `pending`, and transitions are labeled with `trigger-`

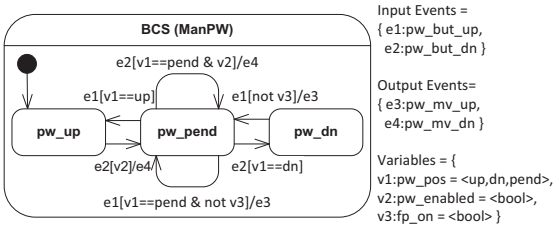

Figure 3: Statechart Test Model of *ManPW*

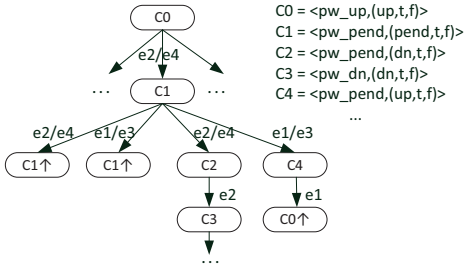

Figure 4: Reachability Tree of *ManPW*

`[guard]/actions`. The user triggers vertical window movements via buttons that release according input events, and, depending on the internal state, release corresponding output events for controlling the movement actuators. Additional transition guards possibly hinder window movements, e.g., via status flags for *CLS* and *FP* features.

Statecharts are also established to serve as test models, where test case generation is supported by various tools like Rhapsody/ATG and ParTeG [28]. Considering statecharts as test models TM_{SC}, modeling artifacts $E \in TM_{SC}$ refer to (composite and basic) states, transitions, labels, etc. Corresponding, statechart coverage criteria are mainly structural control/data flow oriented, e.g., *all-states*, *all-transitions*, *all-transition-pairs* [26], etc.

Approaches like *reachability trees* (RT) [14] provide a basis for reasoning and constructing valid test sequences from statechart test models. Transforming a statechart into an RT makes explicit all its (potentially infinite) executions within the reachable state space. Test cases are obtained from RTs by means of consecutive branches (partial paths), e.g. covering all reachable *configurations*, *branches*, distinguishing sequences for configuration pairs, etc. Fig. 4 shows a sample RT extract of the *ManPW* statechart. Artifacts $E \in TM_{RT}$ are configuration nodes C (sets of concurrently active states, variable values, etc.), and branching edges (sets of concurrent transitions in a step). Nodes marked with $\uparrow$ refer to a previous node labeled with the same configuration, thus ensuring the RT to be finite.

For applying MBT to test (and cover) whole SPLs, i.e., families of similar systems, we use a *reusable* test model, which we call a 150% test model $TM_{150\%}$. The 150% integrates all common and variable behaviors of every valid product configuration. It contains the behavioral artifacts of every feature, no matter if those exclude each other or if those are within an alternative group. A variant including two features that exclude each other is inconsistent.

Features $F = \{f_1, f_2, \ldots, f_n\}$ of an SPL organized in a feature model $FM(F) \subseteq \mathcal{P}(F)$ are abstract entities that describe characteristics of product variants from the user's point of view. To give features a meaning, e.g., behavioral semantics as a basis for implementation and testing, a *mapping* to appropriate models like statecharts is required. We restrict our discussions to a simple mapping function:

$$\sigma : \mathcal{P}(F) \rightharpoonup \mathcal{P}(TM_{150\%})$$

i.e., a partial bijection from (sets of) features $F_i \in dom(\sigma)$ to sets of test model artifacts $TM_i = \sigma(F_i) \subseteq TM_{150\%}$. We further constitute σ to partition $TM_{150\%}$, i.e., for all $F_i \in dom(\sigma)$ and $F_j \in dom(\sigma)$, $F_i \neq F_j$, we require $\sigma(F_i) \cap \sigma(F_j) = \emptyset$. Mappings to model artifacts associated with particular features $f \in dom(\sigma)$ represent behavioral commonalities and variabilities of product configurations in an SPL. Mappings for feature sets $F_i \in dom$, $|F_i| > 1$, introduce further fragments for coordinating the interplay of certain feature *combinations*.

We refer to a test model specification of a full variant, i.e., a valid product configuration $PC \in FM(F)$ assembled from all fragments of the variant's features as its 100% test model in the following. For the remainder, we assume the 100% test model $TM_{PC100\%} \subseteq TM_{150\%}$ for a product configuration $PC \in FM(F)$ to be given as:

$$TM_{PC100\%} = \bigcup_{F_i \subseteq PC, F_i \in dom(\sigma)} \sigma(F_i)$$

i.e., all model artifacts associated to (combinations of) features present in PC are conservatively added to the test model. In general, mapping tools like pure::variants usually allow for model artifact annotations using logical propositions. Therefore, artifact assemblies depend on presence/ absence/ combination conditions over features. For such non-conservative constructions, fragments $F_i \in dom(\sigma)$ may also replace/remove artifacts $\sigma(F_j) \in dom(\sigma)$, where $F_j \subset F_i$. By $TM_{SC150\%}$ we refer to a 150% statechart test model of an SPL and assume the construction to be conservative.

4. POTENTIALS AND LIMITATIONS

In this section, we discuss the adequacy of FM-based SPL testing in general, and, in particular, T-wise (especially pairwise) selection criteria as proposed in our framework.

4.1 Feature Dependencies and Interactions

Features of an SPL are interrelated in two different ways: (1) via dependencies (including constraints) defined in an FM, and (2) via interactions.

Considering (1), a feature model $FM(F) \subseteq \mathcal{P}(F)$ defines dependencies by means of sets of variability relations D and constraining relations C on the set of features F. For instance, in the FODA FM used in our case study we have:

- *mandatory* variabilities $D_{man} \subseteq F \times F$
- *optional* variabilities $D_{opt} \subseteq F \times F$
- *alternative* group variabilities $D_{alt} \subseteq F \times \mathcal{P}(F)$
- *or* group variabilities $D_{or} \subseteq F \times \mathcal{P}(F)$

where the dependencies D define the sub feature hierarchy (feature tree) on F as shown in Fig. 1, and:

- require constraints $C_{req} \subseteq F \times F$

- exclude constraints $C_{exc} \subseteq F \times F$

introduce further conditions C on the validity of product configurations $PC \in FM(F)$. As a consequence, we distinguish three kinds of *binary* dependencies, one of each holding for every feature pair $\{f_i, f_j\} \subseteq F$ organized in a feature model $FM(F)$:

1. f_i implies f_j, written $f_i \Rightarrow f_j$,
2. f_i excludes f_j, written $f_i \lightning f_j$,
3. f_i and f_j are independent, written $f_i \perp f_j$

In case 1), the presence of feature f_i in a configuration implies the presence of feature f_j in that configuration, i.e.:

- features depend on their parent features,
- mandatory features: if $(f_i, f_j) \in D_{man}$ then $f_i \Rightarrow f_j$, and
- required features: if $(f_i, f_j) \in C_{req}$ then $f_i \Rightarrow f_j$.

The binary relation $\Rightarrow \subseteq F \times F$ is reflexive, non-symmetrical, and transitive. We write $f_i \Leftrightarrow f_j$ for short if f_i implies f_j *and* f_j implies f_i.

In case 2), the presence of feature f_i in a configuration implies the absence of feature f_j in the same configuration:

- alternative feature groupings: if $(f, F_i) \in D_{alt}$, then $f_j \lightning f_k$ for each pair $\{f_j, f_k\} \subseteq F_i$, and
- excluded features: if $(f_i, f_j) \in C_{exc}$, then $f_i \lightning f_j$, and $f_j \lightning f_i$.

Hence, $\lightning \subseteq F \times F$ is irreflexive, symmetrical, and non-transitive. Instead, we define an alternative kind of (non-reflexive) transitive closure $\lightning^{(+)}$ as follows:

$$\lightning^{(+)} = \{(f_i, f_j) \in F \times F \mid f_i \Rightarrow f_k \lightning f_l \Leftarrow f_j\}$$

where $\Leftarrow$ denotes $\Rightarrow^{-1}$. By intuition, two features f_i and f_j exclude each other, if they imply some features f_k and f_l, respectively, that exclude each other. In any other case, two features f_i and f_j are *independent*, written $f_i \perp f_j$, where:

$$\perp = F \times F \setminus \{ \Rightarrow \cup \Leftarrow \cup \lightning^{(+)} \}$$

Thus, if two features f_i and f_j neither have implication, nor exclusion dependencies among each other, their presence/absence within the same product configuration is mutually independent. The relation $\perp$ is irreflexive, symmetrical, and non transitive.

The notion (2) of (pairwise) *feature interactions* is far less straight forward than the FM-based *dependency* relations. Various descriptions of feature interaction phenomena are given in the literature (e.g. in [2]). We refer to the characterization of [6] in the following: *A feature f_i interacts with a feature f_j if the behavior of f_i depends on whether f_j is present or absent.*

Interactions introduce runtime relationships among features. Therefore, correctness of SPL implementations is not verifiable for every feature in isolation, e.g., by means of feature unit testing. Instead, behavioral influences arising from varying product configurations contexts are to be taken into account. Concerning the corresponding problem of feature integration testing in particular, we consider the categories listed in Table 1.

	positive	**negative**
intended	feature cooperation	feature vetoing
unintended	undesired interference	required but missing

Table 1: Categories of Feature Interactions

Feature interactions can be either *intended*, i.e., being an integral part of the product line requirements, or they are *unintended*, therefore potentially leading to failures. We will mainly focus on those interactions arising at the functional/logical level, therefore being detectable on the basis of test model specifications. Nevertheless, interactions might also arise at some other level of abstraction, e.g., by shared resources accesses and even environmental influences [6].

We further distinguish *positive* interactions, e.g., features *cooperate* by complementing one another to provide some additional functionality, and *negative* interactions, where a feature *vetoes*, i.e., hinders another feature to be executed, e.g., for safety critical reasons. Accordingly, unintended interactions are, again, either *positive*, now leading to some additional *undesired influences* among features, or they are *negative*, i.e., an *intended* interaction is required but missing/erroneous.

For example, in the BCS-SPL, feature *CLS* cooperates with *AutPW* to close windows when locking, and, at the same time, *CLS* vetoes *AutPW* to (re-)open the windows while being locked. Unintended interferences arise, e.g., if *FP* fails to veto *AutPW* when interfering with *CLS* and/or *CAP*.

Feature dependencies imply valid feature combinations in product configurations. Therefore, dependencies restrict potential feature interactions: interactions between features only potentially arise in some product configuration, if those features are not excluding each other. Further dependency and interaction interrelations among features are, in general, incomparable, i.e., dependencies do not imply interactions, and interactions do not imply dependencies. Therefore, we further distinguish *mandatory* and *optional* feature interactions of non-excluding, interacting features f_i and f_j: if $f_i \Rightarrow f_j$ then f_i *mandatorily interacts* with f_j, else f_i *optionally interacts* with f_j. Accordingly, if $f_i \Leftrightarrow f_j$, then f_i and f_j mandatorily interact, and if $f_i \perp f_j$, then f_i and f_j optionally interact. According to this definition, every feature mandatorily interacts with all its parent features in the feature tree by *varying* them.

4.2 Potentials of Pairwise Feature Interaction Coverage

For reliably detecting, testing, and exhaustively covering potentially erroneous feature interactions, first of all, an appropriate test model is required.

We assume a test model $TM_{150\%}$ to (explicitely) specify (intended) feature interactions at a given level of abstraction. Features $F_i \subseteq F$, $|F_i| > 1$, with $F_i \in dom(\sigma)$ of mapping function $\sigma : \mathcal{P}(F) \rightharpoonup \mathcal{P}(TM_{150\%})$ *share* test model fragments $\sigma(F_i) \subseteq TM_{150\%}$. For example, if $TM_{150\%}$ is a statechart model, $\sigma(\{f_i, f_j\})$ maps to fragments that define the intended cooperation and/or vetoing behavior between features f_i and f_j in a product configuration. If $TM_{150\%}$ is a reachability tree, $\sigma(\{f_i, f_j\})$ maps to paths that correspond to behaviors exclusively resulting from intended interactions between features f_i and f_j. For example, in the BCS-SPL, $\{CLS, AutPW\} \in dom(\sigma)$ introduces additional artifacts to

the *AutPW* statechart allowing *CLS* to control it when activated. In contrast, interactions among *LEDs* and features whose statuses are shown is realized via shared variables accesses, thus only detectable on the basis of the reachability tree test model.

For our pairwise test approach, we make the following assumption: Implementations of features $f \in F$ and feature combinations $F_i \subseteq F$ are not testable as isolated units, but rather are to be assembled into some *valid* product configuration $PC \in FM(F)$ *under test*. A subset heuristics S then selects representative product configurations under test $S(FM(F)) \subseteq FM(F)$ that fulfill a given (combinatorial) coverage criterion. As a consequence, interactions among features $F_i \subseteq F$ are only testable, if the criterion matches F_i to be selected.

The inevitable criterion *one-wise* ensures every feature $f \in F$ to be assembled to at least one product under test, and the exhaustive criterion *N-wise* enforces every valid combination to be covered.

T-wise criteria, $1 < T < N$ propose a reasonable trade off, especially $T = 2$, i.e., *pairwise* feature combination coverage. Concerning potential interactions between feature pairs as characterized above, our pairwise subset heuristics is designed such that:

1. all *valid* combinations of feature pairs $\{f_i, f_j\} \subseteq F$, are covered. Thus, we do not only cover *intended* interactions $\{f_i, f_j\} \in dom(\sigma)$, but also (potentially) *unintended* pairwise feature interactions, and
2. all valid pairwise presence/absence combinations are covered, thus covering all ways of *optional* pairwise interactions.

For 2), considering features to be combinatorial parameters of product configurations, the coverage requirements for a pair $\{f_i, f_j\} \subseteq F$ are as follows: $(\neg f_i, \neg f_j)$

- if $f_i \Rightarrow f_j$, then (f_i, f_j), $(\neg f_i, \neg f_j)$, and $(\neg f_i, f_j)$ are valid pairs, i.e., f_j is to be tested in the presence as well as in the absence of f_i,
- if $f_i \Leftrightarrow f_j$, then (f_i, f_j) and $(\neg f_i, \neg f_j)$ are valid pairs, i.e., f_i is only testable in the presence of f_j,
- if $f_i \perp f_j$ then (f_i, f_j), $(\neg f_i, \neg f_j)$, $(\neg f_i, f_j)$, and $(f_i, \neg f_j)$ are valid pairs, i.e., f_i is to be tested in the presence and in the absence of f_j and vice versa, and
- if $f_i \not\downarrow f_j$, then $(\neg f_i, f_j)$, $(\neg f_i, \neg f_j)$, and $(f_i, \neg f_j)$ are valid pairs, i.e., no interaction is to be tested.

Therefore, pairwise SPL testing suffices to support test cases for all (pairwise) interactions according to Table 1. If there is an *intended* interaction modeled by fragments $\sigma(\{f_i, f_j\})$, then 1) either *intended positive/negative* interactions are tested to correctly cooperate/veto, or 2) *unintended negative* interactions arise, if the interaction is missing/faulty. Otherwise, if $\{f_i, f_j\} \notin dom(\sigma)$, the absence of *unintended negative* interactions by means of behavioral influences is tested. Furthermore, for *optional* interactions, 1) the correctness of *intended* interactions in the presence of both features is tested, and 2) the conceptional independence of both features is tested by isolating them from each other. Please note, that interaction errors are only reliably detectable provided that the actual test models, the model-based testing tools, and coverage criteria applied are adequate.

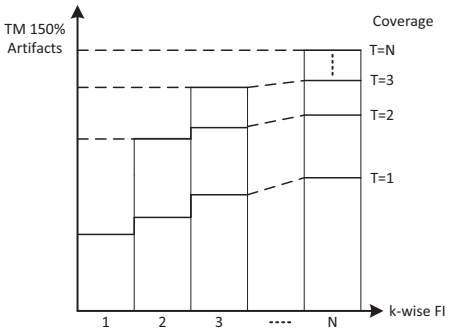

Figure 5: T-wise Coverage of k-wise Interactions

4.3 Limitations of Pairwise Testing

Pairwise combinatorial subset selection ensures every valid combination of feature pairs $\{f_i, f_j\} \subseteq F$ to be covered by at least one product-under-test. In contrast, e.g., for an interaction $\{f_i, f_j, f_k\} \in dom(\sigma)$, no general coverage statement can be given.

Generalizing the problem, for $F_i \in dom(\sigma)$, $|F_i| = k$, i.e., a *k-wise* interaction specified in a test model $TM_{150\%}$ and a *T-wise*, $T < k$, i.e., a combinatorial T-tuple feature selection, the coverage degree depends on 1) the (in-) dependencies between features in F_i, and 2) the subset selection heuristics algorithm used. Considering 1), it follows that the more independent the features are, the more optional interactions are present, and the more potentially unintended/erroneous feature interactions are to be covered. Thus, for *fully dependent* features, i.e., *k-wise mandatory* interactions, $T = 1$ suffices to cover them, against what for *fully independent* features, i.e., *k-wise optional* interactions, $T = k$ is required. Otherwise, in case of *partial* dependencies, some $1 < T < k$ is satisfactory. This relationship between k and T is illustrated in Fig. 5: T-wise suffices to cover all k-wise interactions, $k \leq T$, and some further interactions with $k > T$, that are "accidentally" assembled to some product-under-test. But, even if we are able to deduce *k-wise* interactions from the test model and conduct an appropriate subset selection, several uncertainties remain: (1) is every feature interaction captured in the test model? — especially positive unintended interactions potentially arise everywhere, (2) is every feature interaction reliably detectable in the test model? — for instance, the complexity of statechart configuration spaces, in general, degrades efficient feature interaction detection at RT level, and, accordingly, (3) how to keep the testing effort for increasing k maintainable?

5. BCS-SPL RESULTS

We evaluated our SPL testing approach considering the feature interaction coverage under pairwise feature selection in the sample BCS-SPL introduced in Sect. 2.

The BCS-SPL FM in Fig. 1, consisting of a set F of 21 features, constraints the product space to $|FM(F)| = 560$ valid product configurations.

The corresponding reusable statechart-based test model $TM_{SC150\%}$ consists of 93 states (20 composite states, 22 concurrent sub machines, and 51 basic states), and 107 transitions. The mapping function σ partitions artifacts in $TM_{SC150\%}$ into 30 separate fragments $F_i \in dom(\sigma)$, where

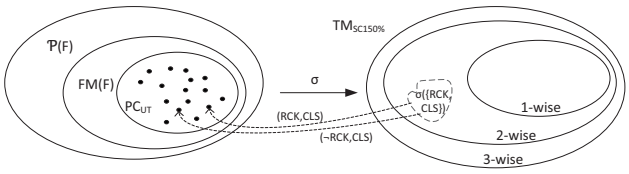

Figure 6: BCS Mapping and Pairwise Subset

	k=1	k=2	k=3	k>3	sum
# FI in SC	12	37	17	-	66
# covered	12	37	14	-	63
% covered	100	100	82,4	-	95,5

Table 2: BCS Statechart Coverage under Pairwise

for 12 of them $k = 1$, for 15 of them $k = 2$, and for 3 of them $k = 3$, $k = |F_i|$, holds. Thus, 9 features have no singleton mapping, but rather we assume a mandatory 0-wise mapping onto a *core* statechart model.

In particular, 3-wise interaction fragments arise for (1) $\{CLS, AutPW, RCK\}$, (2) $\{CLS, ManPW, RCK\}$, and (3) $\{CLS, AutPW, LED\}$. In case (1) and (2), we have $CLS \perp AutPW/ManPW$, $RCK \Rightarrow CLS$, and $RCK \perp AutPW/ManPW$, therefore 5 valid feature triples. In case (3), all features are mutually independent, thus 7 valid feature triples are to be tested to (optionally) interact in the intended way.

As illustrated in Fig. 6, MoSo-PoLiTe generates 16 representative products-under-test $PC_{UT} = S(FM(F))$ for our pairwise subset selection heuristics S. Table 2 summarizes the coverage of $TM_{SC150\%}$ achieved under pairwise. For the 15 pairwise interactions, all 37 valid pairs are covered in PC_{UT} as, e.g., illustrated for *RCK* and *CLS* in Fig. 6, against what for the three 3-wise interactions, just 14 of the 17 valid triples are covered in PC_{UT}.

For the test case generation and conduction on PC_{UT}, Rhapsody/ATG generates 62 test cases on average per product using *Model Element Coverage* and *Model Code Coverage* (MC/DC [26]) criteria. For evaluating the feasibility of the testing approach and the quality of the tests generated, we applied several *mutation operators* to the products-under-test implementation code. In the majority of cases in which mutation detection fails, the test cases generated by ATG were not sufficient to cover them, whereas only a small number of mutations were missed because they arise in particular, uncovered feature combinations. For instance, a faulty *LED* activation remained undetected because the unintended interaction only appears in the triple *(LED, AutPW, ¬CLS)* which is not covered by the subset under pairwise.

6. RELATED WORK

According to the best of our knowledge, our approach is the first attempt to provide a systematic integration of combinatorial testing and model-based testing. The CADeT approach [16] is the most similar methodology utilizing reusable test models by means of activity diagrams. The generation of a set of products using pairwise is mentioned but not described. Especially, the author does not sketch an approach of how to apply combinatorial testing to SPLs.

Other related work can be categorized into model-based testing of SPLs and combinatorial testing for SPLs. Frequently, statecharts, activity diagrams, and sequence diagrams are used for model-based testing. Similar to the CADeT approach, ScenTED [21] and Hartmann et al. [10] utilize activity diagram-based test models including variability. The test cases are then adapted for each individual product during application engineering for system testing. Weissleder et al. [28] use a single state machine as test model that describes the functionality of a whole SPL and the tool ParTeG for automatic test case generation.

With regard to our mapping approach, various methodologies for mapping features of a FM to elements of the SPL architecture exist. The approach in [22] identifies traceability between FMs and architecture models using Formal Concept Analysis. Wasowski [27] describes an approach to automatically generate variants of behavioral models by means of statecharts using a form of partial evaluation and slicing based on specified restrictions. Sochos et al. [25] propose Feature Architecture Mapping (FArM) for mappings between features and architectural components utilizing progressive transformations. Czarnecki [4] annotates a template design model with logical, feature-based presence conditions based on features. In [16] Olimpiew uses decision tables to map features to use cases to generate test cases for products derived from an SPL.

With regard to combinatorial testing the approach described in [19] is the only systematic approach to apply combinatorial testing to SPLs apart from our approach. Perrouin et al. translate the FM into an Alloy and use SAT-solvers to calculate a minimal set of products covering all T-wise feature interactions. A detailed comparison of our approach and the Alloy approach is currently under development.

7. CONCLUSIONS AND FUTURE WORK

In this paper we examine the potentials and limitations of pairwise testing for SPLs. Given the enormous number of product instances that can be created from even a modest SPL, it is important to reduce the testing space while preserving bug-finding power of testing. We created a novel approach for effective testing of feature interactions in SPLs by combining a combinatorial designs algorithm for pairwise feature generation with model-based testing to reduce the size of the SPL required for comprehensive coverage of interacting features. We implemented our approach and applied it to an SPL from the automotive domain provided by one of our industrial partners. The results suggest that with our approach higher coverage of feature interactions is achieved at a fraction of cost when compared with a baseline approach of testing all feature interactions.

In our future work, we will adapt our algorithm to combine specified sets of features in an arbitrary k-wise manner, where k is the number of features involved in interactions. In [20] the authors have shown that better coverage can be achieved by selecting input data objects that contain diverse values for configuration variables. A notion that plays a significant role is interaction strength, where interaction is defined as a partial setting of values of parameters that guarantees certain coverage, while this coverage is not guaranteed for any subset of this setting, and the strength of interaction is measured by the number of parameters to which unique combinations of values are assigned. This procedure requires a feature interaction analysis of the SPL under test

which is the scope of our future work and initially discussed in [13]. Therefore, we will extend Rhapsody/ATG to transform statecharts into corresponding reachability trees. Thus we are then able to apply more coverage criteria then currently available for ATG. In addition, we currently work on a more fine grained mapping between the FM and the test model to improve test case generation. We have already extended our algorithm from pairwise to N-wise and we will examine the potentials and limitations for N-wise approaches compared to pairwise and k-wise. With regard to feature modeling, we aim at support for cardinality based feature models.

8. REFERENCES

[1] R. V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[2] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec. Feature Interaction: A Critical Review and onsidered Forecast. *Computer Networks*, 41(1):115–141, 2003.

[3] M. B. Cohen, P. B. Gibbons, W. B. Mugridge, and C. J. Colbourn. Constructing test suites for interaction testing. In *ICSE*, pages 38–48, 2003.

[4] K. Czarnecki. Mapping Features to Models: A Template Approach based on superimposed Variants. In *GPCE'05, vol. 3676 of LNCS*, pages 422–437, 2005.

[5] K. Czarnecki and U. W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[6] S. Ferber, J. Haag, and J. Savolainen. Feature Interaction and Dependencies: Modeling Features for Reengineering a Legacy Product Line. In *Proc. of the 2nd Int. Conf. on Software Product Lines*, pages 235–256, London, UK, 2002. Springer-Verlag.

[7] M. L. Griss. Implementing product-line features by composing aspects. In *SPLC*, pages 271–288, Norwell, MA, USA, 2000. Kluwer Academic Publishers.

[8] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[9] D. Harel and H. Kugler. The Rhapsody Semantics of Statecharts (or, On the Executable Core of the UML). In *In LNCS 3147*, pages 325–354. Springer, 2001.

[10] J. Hartmann, M. Vieira, and A. Ruder. A UML-based Approach for Validating Product Lines. In B. Geppert, C. Krueger, and J. Li, editors, *Proc. of the International Workshop on Software Product Line Testing (SPLiT 2004)*, pages 58–65, 2004.

[11] M. Jackson and P. Zave. Distributed feature composition: A virtual architecture for telecommunications services. *IEEE Transactions on Software Engineering*, 24:831–847, 1998.

[12] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, CMU Software Engineering Institute, November 1990.

[13] M. Lochau and U. Goltz. Feature Interaction Aware Test Case Generation for Embedded Control Systems. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 264:37–52, 2010.

[14] P. Masiero, J. Maldonado, and I. Boaventura. A Reachability Tree for Statecharts and Analysis of some Properties. *Information and Software Technology*, 36(10):615 – 624, 1994.

[15] T. Müller, M. Lochau, S. Detering, F. Saust, H. Garbers, L. Märtin, T. Form, and U. Goltz. A comprehensive Description of a Model-based, continuous Development Process for AUTOSAR Systems with integrated Quality Assurance. Technical Report 2009-06, TU Braunschweig, 2009.

[16] E. M. Olimpiew. *Model-Based Testing for Software Product Lines*. PhD thesis, George Mason University, 2008.

[17] S. Oster, F. Markert, and P. Ritter. Automated Incremental Pairwise Testing of Software Product Lines. In *Proc. of the 14th International Software Product Line Conference*, 2010.

[18] S. Oster, I. Zorcic, F. Markert, and M. Lochau. MoSo-PoLiTe - Tool Support for Pairwise and Model-Based Software Product Line Testing. In *Proc. of the Fifth Int. Workshop on Variability Modelling of Software-intensive System*, 2011.

[19] G. Perrouin, J. Klein, N. Guelfi, and J.-M. Jézéquel. Reconciling automation and flexibility in product derivation. In *SPLC*, pages 339–348, Limerick, Ireland, 2008. IEEE Computer Society.

[20] E. Reisner, C. Song, K.-K. Ma, J. S. Foster, and A. Porter. Using symbolic evaluation to understand behavior in configurable software systems. In *ICSE' 10)*, pages 445–454, 2010.

[21] A. Reuys, E. Kamsties, K. Pohl, and S. Reis. Model-based System Testing of Software Product Families. In *CAiSE*, pages 519–534, 2005.

[22] T. K. Satyananda, D. Lee, S. Kang, and S. I. Hashmi. Identifying traceability between feature model and software architecture in software product line using formal concept analysis. In *Proc. of Int. Conf. Computational Science and its Applications*, pages 380–388, Washington, DC, USA, 2007. IEEE CS.

[23] K. Scheidemann. Verifying Families of System Configurations. *Doctoral Thesis*, TU Munich, 2007.

[24] K. D. Scheidemann. Optimizing the Selection of Representative Configurations in Verification of Evolving Product Lines of Distributed Embedded Systems. In *Proc. of the Int. Software Product Line Conf*, pages 75–84, 2006.

[25] P. Sochos, M. Riebisch, and I. Philippow. The Feature-Architecture Mapping (FArM) Method for Feature-Oriented Development of Software Product Lines. In *Proc. of the 13th Annual IEEE Int. Symp. and Ws on Eng. of Computer Based Systems*, pages 308–318, Washington, DC, USA, 2006. IEEE CS.

[26] M. Utting and B. Legeard. *Practical Model-Based Testing. A Tools Approach*. Morgan Kaufmann, 2007.

[27] A. Wasowski. Automatic Generation of Program Families by Model Restrictions. In *SPLC*, pages 73–89, 2004.

[28] S. Weißleder, D. Sokenou, and B. Schlinglo. Reusing State Machines for Automatic Test Generation in Product Lines. In *Proc. of the 1st Workshop on Model-based Testing in Practice (MoTiP2008)*, 2008.