Potential Synergies of Theorem Proving and Model Checking for Software Product Lines

Conference Paper · September 2014

DOI: 10.1145/2648511.2648530

CITATIONS
11

READS
92

6 authors, including:

Thomas Thüm
Technische Universität Braunschweig
105 PUBLICATIONS **2,014** CITATIONS
SEE PROFILE

Fabian Benduhn
Otto-von-Guericke-Universität Magdeburg
31 PUBLICATIONS **398** CITATIONS
SEE PROFILE

Alexander von Rhein
Universität Passau
21 PUBLICATIONS **579** CITATIONS
SEE PROFILE

Gunter Saake
Otto-von-Guericke-Universität Magdeburg
632 PUBLICATIONS **5,963** CITATIONS
SEE PROFILE

Potential Synergies of Theorem Proving and Model Checking for Software Product Lines

Thomas Thüm
University of Magdeburg
Germany

Jens Meinicke
University of Magdeburg
Germany

Fabian Benduhn
University of Magdeburg
Germany

Martin Hentschel
University of Darmstadt
Germany

Alexander von Rhein
University of Passau
Germany

Gunter Saake
University of Magdeburg
Germany

ABSTRACT

The verification of software product lines is an active research area. A challenge is to efficiently verify similar products without the need to generate and verify them individually. As solution, researchers suggest family-based verification approaches, which either transform compile-time into runtime variability or make verification tools variability-aware. Existing approaches either focus on theorem proving, model checking, or other verification techniques. For the first time, we combine theorem proving and model checking to evaluate their synergies for product-line verification. We provide tool support by connecting five existing tools, namely FeatureIDE and FeatureHouse for product-line development, as well as KeY, JPF, and OpenJML for verification of Java programs. In an experiment, we found the synergy of improved effectiveness and efficiency, especially for product lines with few defects. Further, we experienced that model checking and theorem proving are more efficient and effective if the product line contains more defects.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.13 [**Software Engineering**]: Reusable Software

General Terms

Design, Languages, Verification

Keywords

Software product lines, theorem proving, model checking, design by contract, feature-based specification, family-based verification, variability encoding, feature-oriented contracts

1. INTRODUCTION

Many software systems are not developed from scratch, but rather by starting from existing systems [62, 47]. Consequently, these software systems share similarities with each other. Software product-line engineering is a means to take advantage of these similarities [42]. A *software product line* is a set of products that share substantial similarities and are created from a set of reusable parts [2]. These products are typically distinguished in terms of the features they provide, and each product can be generated automatically based on a selection of features.

As software product lines are increasingly used in mission-critical and safety-critical systems, such as medical, avionic, and automotive systems [61], there is a need for their verification. A simple strategy for product-line verification is to generate and verify all products individually, known as *product-based verification* [55]. However, product-based verification involves redundant computations, because products have similarities, and is often even infeasible due to a large number of products.

In the last decade, several verification approaches have been proposed that take commonality and variability in a product line into account [55]. A well-known strategy is *family-based verification*, in which all products are verified simultaneously – either by making tools variability-aware or by transforming compile-time into runtime variability [55]. We focus on the latter, which is known as *configuration lifting* [44] or *variability encoding* [6]. The family-based strategy has been applied to scale different verification techniques to software product lines, such as type checking [7, 54, 32, 21, 35, 34], dataflow analyses [14, 35, 13], model checking [23, 25, 44, 4, 18, 33, 6], and theorem proving [58, 26].

Each of these verification techniques has unique strengths and weaknesses. Type checking is relatively fast, but limited in the defects that can be detected [41]. Dataflow analyses are typically fast, but often unsound or incomplete (e.g., they may miss actual defects or may produce false positives) [39]. In contrast, if supported by the particular specification language, arbitrary reachability and safety properties can be verified with model checking and theorem proving. While model checking often requires experts for parametrization to avoid the state space explosion and may produce hard-to-understand traces [17], theorem proving often requires user interaction, such as providing invariants or applying proof tactics, and the understanding of a logical representation of the program [50]. Consequently, it seems useful to *combine several techniques* for product-line verification [55]. However, to the best of our knowledge, combinations have not yet been used to verify the same property for a particular product line.

SPLC '14, September 15–19 2014, Florence, Italy

A challenge in combining several verification techniques for the very same property is that this property needs to be translated into the input language of each verifier. Fortunately, *design by contract* is a methodology for specifications that has already been extensively applied to many verification techniques, such as static analysis, model checking, and theorem proving [16, 46, 11, 8, 28]. In design by contract, a contract is assigned to a method with a precondition stating a property that needs to be ensured by the caller, and a postcondition being a property ensured by the callee [38]. Researchers already proposed contracts for software product lines [60, 49, 5] and checked them by means of runtime assertion checking [56], static analysis [48], and theorem proving [15, 58, 26, 20]. We focus on the combination of theorem proving and model checking for product-line verification. To the best of our knowledge, this is the first time that product-line contracts are verified using model checking.

We provide tool support for contract-based verification of product lines by extending and reusing several existing tools. We extended FeatureHouse [3], a tool for the composition of feature-oriented source code, with support for variability encoding for contracts written in JML [16]. An advantage of variability encoding is that the result is a Java program with JML specifications that can be verified by a variety of JML tools. We use KeY [11] for automated theorem proving. For model checking, we utilize Java PathFinder (JPF) [29], which, however, has no built-in support for contracts. For that reason, we transform contracts into runtime assertions using OpenJML [19] before analyzing the product line with JPF. Finally, we extended FeatureIDE [57] to integrate these complex tool chains into Eclipse. Our extension provides the first tool support for (a) variability encoding of contracts, (b) family-based theorem proving, and (c) family-based model checking with contracts.

In our evaluation, we compare model checking and theorem proving for product-line verification. In particular, an interesting question is whether one of them is superior for earlier or later stages in the development process, in which the product line may contain more or less defects. Consequently, we consider the number of defects as independent variable. We introduce artificial defects by means of mutation techniques as known from mutation testing [31]. As dependent variables, we focus on effectiveness, performance, and efficiency to find the first defect of the product line. For that, we measure how often defects are found, the time for verification, and the ratio of both. In contrast, existing evaluations of product-line verification techniques have either verified a product line with [4, 18, 6] or without [44, 59, 58, 35, 14, 13] defects, or have not compared these verification times [54, 32, 33, 34]. Furthermore, the influence of defects on verification has not yet been explored.

In summary, we make the following contributions:

- We combine model checking and theorem proving for efficient and effective product-line verification.
- We provide tool support in FeatureIDE based on FeatureHouse, KeY, JPF, and OpenJML.
- We evaluate model checking and theorem proving for a small product line with no, some, or many defects.
- We evaluate synergies of model checking and theorem proving regarding their ability to identify defects.

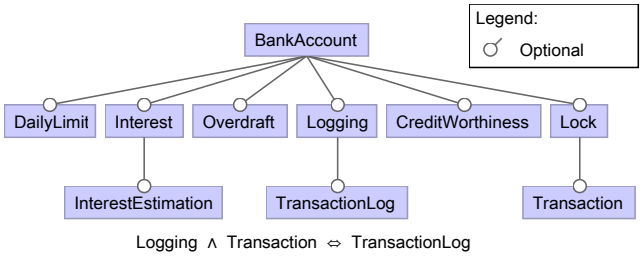

Figure 1: Feature model for bank account software.

2. FEATURE ORIENTATION

We base our work on feature-oriented software product lines [2], in which valid combinations are specified using feature models [9], the implementation is separated into feature modules [10, 45], and code-level specifications are given in feature modules by means of feature-oriented contracts [60]. We briefly exemplify these concepts in the following.

2.1 Feature Models

In product lines, not all combinations of features are meaningful. A *feature model* is often used to define the features of a product line and their valid combinations [9]. In Figure 1, we give an example feature model for a variable bank account software. Feature *BankAccount* represents a basic implementation of a bank account, which may be combined with several optional features to ensure a maximum daily withdrawal, to calculate and estimate interests, to support a negative balance to a certain overdraft limit, to log changes to accounts and transactions, to calculate credit worthiness, and to support locking of accounts and transactions. For our considerations, it is sufficient to know that feature models can be automatically translated into *propositional formulas* [9]. The example feature model is equivalent to the following propositional formula:

$$(\mathit{InterestEstimation} \Rightarrow \mathit{Interest}) \wedge (\mathit{Transaction} \Rightarrow \mathit{Lock}) \wedge$$
$$(\mathit{Logging} \wedge \mathit{Transaction} \Leftrightarrow \mathit{TransactionLog}) \wedge \mathit{BankAccount}$$

2.2 Feature Modules

A goal of feature-oriented software product lines is that products are generated automatically for a selection of features [2]. Consequently, a mapping from features to source code is needed. It is often not sufficient to map features to classes, because features are typically cross-cutting to classes [53]. In feature-oriented programming, a *feature module* is created for each feature consisting of a set of classes and class refinements [10, 45]. Class refinements can add new members, such as fields and methods, to existing classes and refine existing methods. Products are generated by automatically assembling selected feature modules [10].

In Figure 2, we show a simplified version of three feature modules for our bank account product line, namely feature *BankAccount*, *Transaction*, and *TransactionLog*. Feature *BankAccount* contains class Account, a simple Java class (Lines 1–9). It stores the balance for a bank account and enables changes to the balance using the method update.

Similarly, the feature modules *Transaction* and *TransactionLog* introduce the classes Transaction and Log, respectively. In addition, they also refine classes introduced in other feature modules. For example, class Account is refined by feature module *Transaction* with support for locking and

```
1  class Account {                                          BankAccount
2    //@ invariant balance >= 0;
3    int balance = 0;
4    /*@ requires balance + x >= 0;
5      @ ensures balance == \old(balance) + x; @*/
6    boolean update(int x) {
7      balance = balance + x;
8      return true;
9  } }
```

```
10 refines class Account {                                  Transaction
11   boolean lock = false;
12   /*@ requires !lock; ensures lock; @*/
13   void lock() { lock = true; }
14   /*@ ensures !lock; @*/
15   void unLock() { lock = false; }
16   /*@ ensures \result == lock; @*/
17   boolean isLocked() { return lock; }
18 }
19 class Transaction {
20   /*@ requires src != null && dest != null;
21     @ requires src != dest && x > 0;
22     @ ensures \result ==>
23     @   (\old(dest.balance) + x == dest.balance) &&
24     @   (\old(src.balance) - x == src.balance);
25     @ ensures !\result ==>
26     @   (\old(dest.balance) == dest.balance) &&
27     @   (\old(src.balance) == src.balance); @*/
28   boolean transfer(Account src, Account dest, int x) {
29     if (!lock(src, dest)) return false;
30     try {
31       if (!src.update(-x))
32         return false;
33       if (!dest.update(x))
34         { src.update(x); return false; }
35       return true;
36     } finally {
37       src.unLock(); dest.unLock();
38   } }
39   /*@ requires src != null && dest != null;
40     @ requires src != dest;
41     @ ensures \result ==> src.isLocked() &&
42     @   dest.isLocked(); @*/
43   static boolean lock(Account src,
44       Account dest) {
45     if (src.isLocked()) return false;
46     if (dest.isLocked()) return false;
47     src.lock(); dest.lock(); return true;
48 } }
```

```
49 refines class Transaction {                              TransactionLog
50   /*@ requires \original;
51     @ ensures \original && (\result <==>
52     @   Log.contains(src, dest, x)); @*/
53   boolean transfer(Account src, Account dest, int x) {
54     if (original(src, dest, x))
55       { Log.add(src, dest, x); return true; }
56     return false;
57 } }
58 class Log { /* ... */ }
```

Figure 2: Extract from three feature modules of a bank account product line.

unlocking the account (Lines 10–18), which is then used by class Transaction (e.g., Lines 45–47). Essentially, the class refinement adds a field and three new members to class Account, if the according feature is selected. Besides adding members, a class refinement can also refine existing methods, such as the method transfer in class Transaction. The keyword original may be used in a method refinement (e.g., Line 54) to refer to the method that is subject to refinement (similar to keyword proceed in aspect-oriented around advice). Based on a selection of features, the according feature modules are composed automatically to generate a product.

2.3 Feature-Oriented Contracts

Static verification techniques, such as theorem proving or model checking, can be used to verify general properties of programs (e.g., the absence of runtime exceptions). However, to verify that the program behaves as intended, we need to express our intention, for instance, by means of specifications. We specify the intended behavior by means of contracts defined in JML, because there is already tool support for several verification techniques [16, 46, 11].

For an example specification in JML, we refer to Figure 2 again. Lines 4–5 show a precondition stating that an update is only allowed if the balance does not become negative and a postcondition stating that the balance is updated correctly. The keyword old can be used in a postcondition to refer to the object state before method execution, and keyword result to refer to the return value of a method. Line 2 shows a class invariant stating that the balance is always non-negative.

While the specification of feature module *BankAccount* is given in JML, for the other feature modules we use a feature-oriented extension of JML, named explicit contract refinement [60]. With this JML extension, we can specify method contracts and class invariants similar to JML. The difference is that contracts for method refinements may contain keyword original to refer to the contract of the method that is subject to refinement. The semantics is similar as for source code: original in a precondition or postcondition refers to the previous precondition or postcondition, respectively. For example, Lines 50–52 in feature module *TransactionLog* show a contract refinement. The precondition remains as specified in feature module *Transaction* and the postcondition is maintained, but concatenated with a further condition stating that the transaction is logged, if the method returns true. Given a selection of features, we can compose the feature modules including their feature-oriented contracts to a Java program with a JML specification.

3. VARIABILITY ENCODING

In variability encoding, compile-time variability of a product line is translated into run-time or load-time variability for verification purposes. The result is a *metaproduct*, which simulates the behavior of all products. Our approach and evaluation is based on variability encoding, because existing verification techniques and tools can be reused for software product lines. Variability encoding has been proposed for model checking [44, 4, 6, 33] and theorem proving [58] before. We give a brief overview on variability encoding.

Variability Encoding for Feature Models. The feature model is encoded into the metaproduct to simulate exactly those feature selections that are valid. A boolean class variable is created for each feature, whereas the variable as-

signment true indicates that the feature is selected and false that it is not. However, verification tools are configured to treat those *feature variables* as not initialized to consider all combinations. The dependencies between features are translated into a propositional formula into the host language (e.g., Java in our case). This formula is then used to prohibit any execution of non-valid feature selections.

Variability Encoding for Feature Modules. In contrast to the generation of products, where typically only a subset of feature modules is composed, all feature modules are encoded into the metaproduct. To simulate all products, a branching statement over feature variables is included at the beginning of each method that is refined. Depending on the feature selection, either the code of the current method refinement will be executed or the previous method refinement in the refinement chain. For a given feature selection, each method of the metaproduct behaves as the method in the according product (except for some further branching statements and method calls). Similarly, variability encoding can be applied to constructors and fields [36].

Variability Encoding for Contracts. Contracts and invariants are encoded into the metaproduct by introducing implications with feature variables. Each precondition and postcondition (for short condition) c defined in a feature f is rewritten as $f \Rightarrow c$ to ensure that the condition is checked only if the according feature is selected. Similar to the encoding of feature modules, we need to specify the behavior if f is not selected. Given the previous condition c' in the refinement chain, we also add the condition $\neg f \Rightarrow c'$. As invariants cannot be refined [60, 12], it is sufficient to rewrite each invariant i in feature f as $f \Rightarrow i$.

Theorem Proving. In theorem proving, programs are typically verified method-by-method. Given a method and its contract, a theorem prover transforms the precondition while symbolically executing the method. Then, it checks whether the transformed precondition is a model of the postcondition (i.e., it implies the postcondition). That each method is verified without its calling context requires two adaptions to variability encoding for contracts. First, to consider only valid feature selections, a precondition with the feature model needs to be added to every method (either directly, or as an invariant). Otherwise, we may not be able to prove certain contracts, because they are not fulfilled for invalid feature selections. Second, to only verify each method for those feature selections, in which the method is available, we add a precondition to forbid calls otherwise.

Software Model Checking. In software model checking, programs are usually verified by means of test scenarios [4]. A model checker takes the program and test scenarios as input and exhaustively searches for possible violations. The difference of test scenarios compared to test cases is that they can include arbitrary values (e.g., a boolean value or a positive integer), which are all considered during model checking. Similarly to theorem proving, we need to rule out all execution paths that are not available in the products. However, for model checking, it is sufficient to check this once at the beginning of each test scenario.

4. TOOL SUPPORT

We provide tool support for variability encoding of feature modules and their contracts as illustrated in Figure 3.

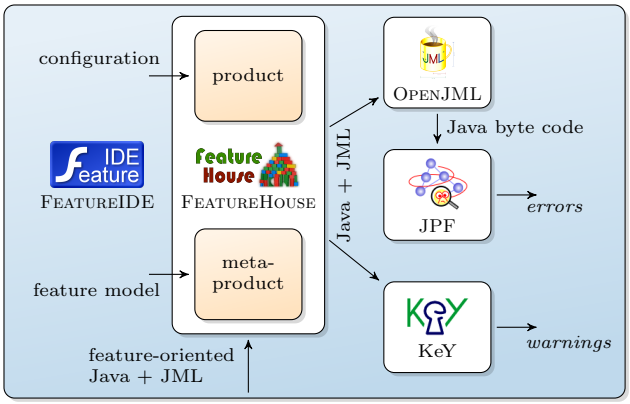

Figure 3: Product-line verification with FeatureIDE.

Previous work either required manual assembly of metaproducts [44, 58], which is laborious and error-prone, or did not support contracts [4, 18, 6, 33], which we rely on to support several verification techniques. We implemented variability encoding based on FeatureHouse [3], because this tool already supports the composition of feature-oriented source code in several languages, such as Java and C. For our extension, we had to provide a new grammar for JML based on the existing one for Java. Furthermore, we implemented the composition of products and variability encoding as described in Section 3.

As the generated metaproduct is a JML-annotated Java program, we can use any JML tool for verification. However, as described in Section 3, theorem provers and model checkers require smaller modifications of the metaproduct. Our extension of FeatureHouse currently supports theorem proving with KeY [11] and model checking with JPF [29]. As JPF has no built-in support for contracts, we use OpenJML [19] to translate contracts of the metaproduct into runtime assertions before model checking. Each of these verification tools is reused as-is.

Finally, FeatureIDE [57] is an Eclipse-based development environment that already integrates several generation tools, such as FeatureHouse. We extended FeatureIDE with support for contracts and variability encoding. First, the user can choose whether products or metaproducts are generated for each project. Second, we implemented error propagation for metaproducts, which propagates error markers in the metaproduct to the source feature module. Third, we extended several FeatureIDE views with support for contracts, such as the collaboration diagram and outline.

All our extensions are open-source and available in FeatureIDE v2.7.0. We plan to use the tool support for teaching product-line specification and verification. Furthermore, we envision that our tool support is a stepping stone to transfer research on product-line verification to industry.

5. EVALUATION

Given our tool support in FeatureIDE it is possible to implement feature modules, specify feature-oriented contracts, and verify them using theorem proving and model checking by means of variability encoding. This gives rise to a number of questions. What are the benefits of combining theorem proving and model checking? Which verification technique is a programmer supposed to use when? How do

Source/Target	Target/Source	In Java	In JML	Sum
false	true	27	1	27
*	/	12	0	12
-	+	7	8	15
+=	-=	4	0	4
<	<=	7	5	12
>	>=	1	12	13
&&	||	0	11	11
==>	<==>	0	27	27
==	!=	0	37	37
Sum		58	101	159

Table 1: Mutations applied to feature modules and feature-oriented contracts of the bank account.

the verification techniques scale depending on the number of features or defects? In the following, we describe our experiment to explore potential synergies of combining techniques, the results of the experiment, and threats to validity. We refer interested readers to our website containing screencasts, source code, and raw data for reproduction purposes.[1]

5.1 Experiment

In our experiment, we use a product line that is completely verified with KeY and JPF. In contrast to the tool support described above, we use MonKeY [58], an extension of KeY providing a batch mode for automatic verification. For model checking, we use JPF-BDD [6], an extension of JPF for product-line verification that symbolically encodes feature variables in a binary decision diagram for better performance. We deliberately introduced defects by means of mutations in feature modules and feature-oriented contracts, respectively. The goal of mutations is to simulate different phases of development (i.e., mature and less mature product lines). We measured the verification time and effectiveness of KeY and JPF for the product line containing no defects, some defects, and many defects.

Experiment Subjects. Our experiment is based on a bank account product line that has been previously used to evaluate techniques for product-line specification [60, 5] and verification [59, 58]. We extended the product line by four new features to ten features overall. The feature model of the product line has already been presented in Figure 1, and a simplified version of three feature modules and their contracts have been shown in Figure 2. Overall, the product line consists of four classes, ten class refinements, 17 unique methods with a contract each, six class invariants, eight method refinements, and six contract refinements. Quantifiers and model methods were not necessary for specifying the bank account product line. The test cases of the product line are composed along with the feature modules and achieve a method coverage of 100.0 %, an instruction coverage of 91.6 %, and a branch coverage of 72.2 % for the largest product. Based on this product line, we simulate different product-line sizes by successively removing existing features. The resulting product lines have between 2 and 10 features (2, 4, 6, 12, 24, 36, 36, 72, and 144 products).

Automatic Defect Generation. While the goal of product-line verification is a defect-free product line, verification tools are used to detect defects on the way towards a verified product line. Consequently, an interesting characteristic for evaluating verification techniques is how they perform for product lines containing many, some, or no defects. While extending the bank account product line, we introduced defects, but they are too few to make any general statements. Hence, we decided to automatically introduce defects as known from mutation testing [31]. We mutate feature modules before variability encoding to simulate realistic defects. Table 1 shows the considered mutation operators as well as the number of occurrences for our product line. These operators are typical for mutation testing [31]. As common in mutation testing, we use string replacements, as they are applicable to feature modules and contracts. With regular expressions, we identified possible positions for mutations and randomly selected mutations in our experiment.

[1] `http://fosd.de/spl-contracts`

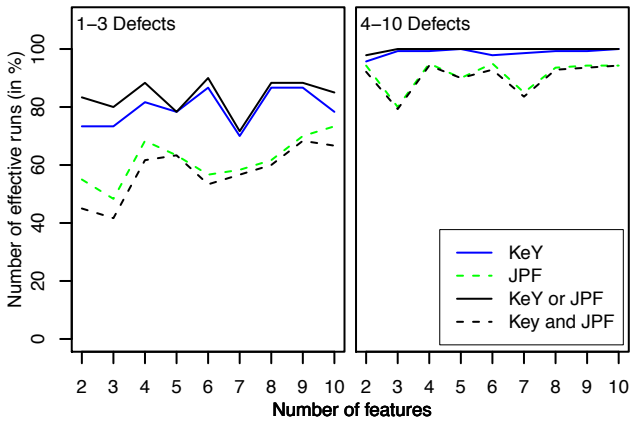

Figure 4: Effectiveness for finding one defect in product lines with some and many defects.

Experiment Set-Up. We computed all experiments on a lab computer with Intel Core i5 CPU with 3.33 GHz, 8 GB RAM, and Windows 7. In all runs, we measured the time for verification with KeY and JPF. We created separate runs for no defects, one defect, and so on, until reaching ten defects, whereas each run was repeated 20 times with different, randomized mutations each to avoid computation bias and bias due to mutations. We stopped both tools after the first defect had been identified, because this time is more critical for developers than the overall verification time; a developer can investigate the first defect already and need to start verification again after fixing the defect anyway. In particular, we stopped KeY if a proof obligation could not be proven automatically. In general, an open proof obligation does not necessarily indicate a defect, because it requires user interaction. However, we inspected all open proof obligations for each single mutation and they all indicate a defect.

5.2 Empirical Comparison

We evaluated effectiveness, performance, and efficiency of theorem proving and model checking for the bank account product line, and share our results in the following.

Effectiveness. We measured effectiveness as how often a verifier will find at least one defect, independent of whether the product line contains one, two, or more defects. In particular, we consider a verifier as *effective*, if it finds less defects than the product line contains. The rational behind this decision is that developers typically work on one defect at a time and then verify the product line again.

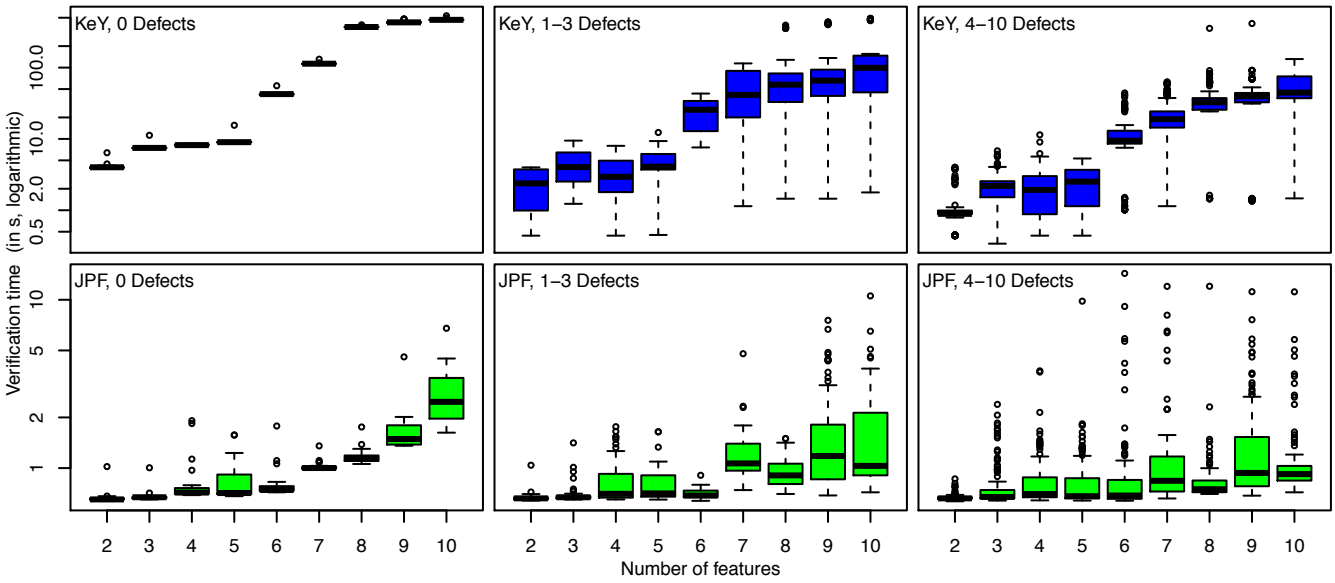

Figure 5: Performance for finding the first defect in product lines with no, some, and many defects.

In Figure 4, we show the effectiveness of theorem proving and model checking for some defects (left diagram) and many defects (right diagram). With regard to the size of our subject product line, we already consider more than three defects as many. Each run is either effective or not, and we show the percentage of runs in which KeY (solid, blue line) and JPF (dashed, green line) were effective. The x-axis represents the number of features of the product line in each run. Furthermore, we show the percentage of runs in which both, KeY and JPF, were effective (dashed, black line, named "KeY and JPF") and the percentage in which at least one of them was effective (solid, black line, named "KeY or JPF"). We computed the percentage for each number of defects separately and show mean values in the diagram (e.g., for one, two, and three defects in the left diagram).

These diagrams lead to the following observations. First, both verification techniques are more effective, when more defects are introduced, because it is more likely that they detect one defect. Second, JPF is, in general, less effective than KeY. A code inspection in these cases revealed that our test scenarios were not sufficient to detect several defects. Third, JPF detects some defects that KeY does not. One reason is that the test scenarios can expose defects in addition to the contracts. Another reason is that KeY does not check precondition violations of called methods when a method body is inlined instead of applying its contract. Fourth, the effectiveness varies for different sizes of product lines, which is caused by our rather small product line with only few mutations (58 mutations in Java and 101 mutations in JML, cf. Table 1). Finally, using both verifiers leads to better effectiveness compared to each of them. This is especially the case if there are only few defects in the product line. That the combined effectiveness is not 100 % is due to the fact that some mutations can simply not be detected with the given contracts and test scenarios. Overall, we found that the combination of theorem proving and model checking improves the effectiveness, especially if there are only a few defects (i.e., in later development stages).

Performance. We assess the performance of theorem proving and model checking as the time needed to detect the first defect. If no defects were found, we consider the time to completely verify the product line. For JPF, we measured the time until the first runtime assertion was violated. For KeY, we measured the time until the first proof obligation could not be proven automatically. In Figure 5, we present the performance of KeY and JPF in box plots (created with default parameters of R). Note that the y-axes are logarithmic. Compared to effectiveness, we also consider the product line without defects.

Again, we make several observations. First, the most obvious result is that JPF is significantly faster than KeY (39.5 times in average). However, this result heavily depends on the test scenarios and should not be overestimated. Second, the verification time grows with the size of the product line, which is due to the larger code base. Third, the deviation of verification is larger for some and many defects than for no defects. The reason is that the computation bias is much smaller than the bias introduced by mutations; the left diagrams only show computation bias as we repeatedly verified the source code without mutations. Finally, the average verification time of both verifiers reduces when more defects are added; for KeY from 163.4 s to 50.7 s and 19.3 s, and for JPF from 1.19 s to 1.06 s and 1.01 s, for some and many defects, respectively. Thus, the speed-up from no to many defects is 8.5 for KeY and 1.2 for JPF. The better average performance is caused by the fact that both verifiers are aborted once they detect a defect. In contrast, larger verification times in some cases indicate that verifiers have, depending on the mutation, some extra effort. In those cases, JPF executes more statements and KeY needs to consider further proof rules than for the defect-free product line.

Efficiency. Based on the effectiveness and performance, we measured efficiency as the ratio of both. However, with our insight that JPF was faster than KeY, we propose to combine both verifier as follows: the product line is checked with JPF first and, depending on the result, KeY is only

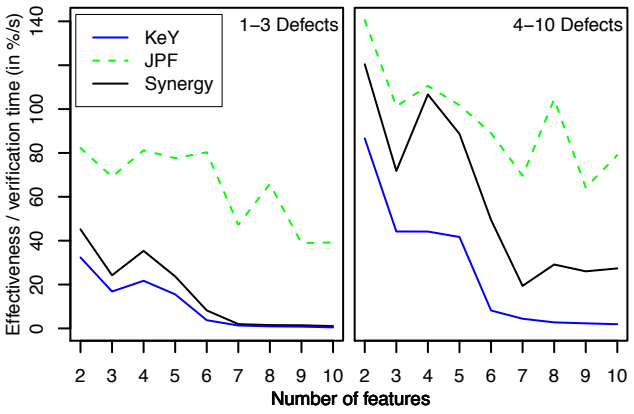

Figure 6: Efficiency as the ratio of effectiveness and performance (larger values are better).

used if no defects were identified by JPF, resulting in the same effectiveness as always running both verifiers (i.e., "KeY or JPF" in Figure 4). Another reason for using JPF first is that it always indicates actual defects, compared to an open proof obligation in KeY. We show our results in Figure 6 (the new strategy is named Synergy), whereas larger values indicate higher efficiency. In the combination, KeY was utilized in 17.6 % of all runs and consumed 90.2 % of the verification time. However, KeY indicated additional defects in 71.3 % of those runs, in which JPF did not find a defect.

Based on the diagrams in Figure 6, we make the following observations. First, each verifier and the combination of both is more efficient for many defects than for some defects. The reason is that they are more effective and require less time. Second, JPF is more efficient for all sizes of product lines as well as some and many defects, because of the better performance. Third, the combination of both verifiers is less efficient than JPF but more efficient than KeY. This is an interesting result, as the combination achieves a better effectiveness than each verifier individually, but the efficiency is still better than KeY. The reason is that we do only verify a product line with KeY, if no defects were found with JPF. In summary, combining KeY and JPF seems beneficial, as it increases the effectiveness compared to both in isolation and the efficiency compared to KeY.

5.3 Discussion

Besides our empirical evaluation, we discuss fundamental differences of theorem proving and model checking in the following, such as inherent guarantees, support for incomplete product lines, defect identification, and tool support.

Software model checking does not provide the same *guarantees* as theorem proving. With theorem proving, we verify methods for all given inputs, whereas software model checking is usually based on test scenarios, which are then executed symbolically. A test scenario is more than a simple test case, as it may consider a set of values at the same time. For example, method `update` in Figure 2 may be called with any positive integer value, which can be handled symbolically in JPF (with extensions dedicated to symbolic execution). While, in principle, for each possible defect we can write a test scenario exposing the defect, defects are typically not known in advance. Test scenarios are often incomplete and it is up to the developer to write meaningful tests.

In principle, theorem proving enables verification with *incomplete implementations* and model checking with *incomplete specifications*. In theorem proving, we can verify methods separately by only relying on contracts of called methods. By doing this, we can even verify product lines that are not completely implemented. In contrast, model checking requires that for each test scenario all called methods are implemented, but they do not need to have contracts. Modular verification with theorem proving may be impractical, because contracts of all called methods must be strong enough, which comes with the price of high specification effort. In particular, some contracts in the bank account product line are too weak and we configured KeY to use the implementation of called methods (a.k.a. method inlining).

A further characteristic difference is how the programmer can *identify defects*. Using JPF, the result is a trace that is essentially a counter example. With KeY, we can inspect the proof, in general, and the unclosed goal, in particular. Both, the trace and the proof, can be large and hard-to-understand. However, an advantage of combining theorem proving and model checking is that we can use both to locate the defect. Nevertheless, a benefit of contracts in both cases is that the violated method contract is already identified, and we only need to identify whether the defect is in the specification or implementation of that method.

A rather technical detail is that each JML *tool* implements a different set of keywords and a slightly different semantics, which requires some effort to use them in concert. For example, OpenJML reports invalid use of access modifiers (e.g., private, public) in contracts that KeY does not report. A further example is that OpenJML creates runtime assertions reporting every violation of a precondition, whereas KeY does only check precondition violations if a method call is treated by applying its contract (e.g., not for method inlining). In addition, each tool usually only supports a certain subset of Java (e.g., OpenJML does not support threads and KeY does not support floating numbers).

5.4 Threats to Validity

We discuss issues that may influence the internal or external validity of our experiments in the following.

Internal Validity. We measured the verification time for KeY and JPF until the first contract cannot be proven and the first assertion is thrown, respectively. While this time strongly depends on the order of test scenarios and proof order, we randomly generated a large number of mutations for each case to increase our confidence in the results. In general, an unproven contract in KeY does not necessarily mean that there is a defect; the theorem prover may need additional support, such as providing loop invariants [16, 11, 8]. However, we inspected all unproven contracts for single mutations and they all indicated a defect.

The mutations that we applied to the product line may not be considered as defects in all cases. To avoid this problem, first, our implementation makes sure that each position in the code is only mutated once, to avoid that two mutations compensate each other. Second, some of the mutations cannot be detected with the given method contracts, as contracts do typically not encode the behavior completely. However, this is independent of the verification technique and should not influence our comparison.

The verification with JPF is influenced by the test scenarios, which are not required for KeY. Whether our test

scenarios are meaningful for a comparison with KeY is questionable. As JPF with our test scenarios has a similar effectiveness as KeY indicates that our test scenarios are reasonable. Nevertheless, other test cases may lead to changes in effectiveness, performance, and efficiency of JPF.

The performance and efficiency of KeY could be better than measured in our experiment. The reason is that KeY can store proofs and check them after changes rather than finding a new proof. A common experience is that proof checking is magnitudes faster than proof finding [11]. In our experiment, we have not used the ability to store proofs, because it is questionable how to simulate two subsequent versions of a product line with mutations. Simply taking the product line with and without mutations does not seem realistic and it is future work to incorporate proof checking into the comparison. For model checking, there are similar strategies that save effort during evolution [52].

External Validity. It is questionable to which extent our results can be generalized to larger product lines (i.e., more features and larger feature implementations). While experiments with larger product lines would be more valuable, already verifying the bank account product line with several tools was a considerable effort. Furthermore, each verification tool does only support a certain subset of Java and JML, which rules out large product lines. Nevertheless, according to experience with previous studies [60], our subject product line including its contracts has typical characteristics (e.g., with respect to the mapping between features and classes). In addition, we simulate different sizes of product lines each bringing us to the same conclusions.

Our mutations may not represent real defects in product lines. As it was necessary for our evaluation to automatically generate defects into feature-oriented Java code and JML specifications, we decided to use mutation techniques. We used typical string replacement operators from mutation testing [31] to mutate feature modules and their contracts directly. More sophisticated operators operating on abstract syntax trees may provide more realistic defects, because a real defect may consist of not only slightly wrong parts, but also missing parts or wrong orders of statements. Furthermore, the generated mutations may represent typical feature-interaction bugs in product lines. The automatic generation of representative feature-interaction bugs is nontrivial and should be investigated in future research.

Other verification tools for theorem proving and model checking may lead to different results, which should be evaluated in further studies. We have chosen KeY and JPF as both tools have been used by many other researchers before and in particular also for product-line verification [58, 6].

Our comparison is based on fully automated verification, but in practice, both, theorem proving and model checking may depend on user input. Theorem proving may require to provide loop invariants or to guide the proof interactively. Besides creating test scenarios, model checking is automatic itself, but may require tuning of parameters to avoid the state explosion. Hence, further experiments are needed to assess the effort when evolving a product line.

6. RELATED WORK

Recent surveys on product-line analyses give a detailed overview on related work [55] and related tools [37]. In particular, several analysis techniques are discussed, such as type checking, dataflow analyses, model checking, and theorem proving. Besides the family-based strategy that we focus on, the surveys describe also product-based and feature-based strategies, as well as combinations thereof. So far, researchers evaluated product-line analyses only for product lines with defects or without [55]. In contrast, we measure the influence of defects on effectiveness and efficiency.

Other researchers proposed a combination of verification techniques. Liebig et al. combine type checking and dataflow analysis to find defects in the Linux kernel [35]. However, both techniques are used for different kinds of errors, while we focus on synergies for the same kind of error. Others used Event-B for product-line verification, which has support for theorem proving and model checking [43, 51, 24]. However, they do neither discuss nor evaluate the benefit of that combination. Besides product lines, several combinations of theorem proving and model checking have been proposed [27, 40, 22, 1, 30], but they all inherently require to create new or change existing verifiers, whereas we used each verifier as-is. Nevertheless, more sophisticated combinations of theorem proving and model checking should be considered for product lines in future work.

All approaches for product-line theorem proving are based on contracts. Thüm et al. [59] use interactive theorem proving with Coq to write proof scripts for each feature that are composed together with feature modules and feature-oriented contracts. Bruns et al. [15] combine KeY with slicing techniques to reuse verification effort from one product for other products. Damiani et al. [20] verify features as far as possible separately by means of uninterpreted assertions, and remaining proofs are done for products. However, all these approaches require to generate and verify each product. In contrast, two family-based approaches have been presented that avoid the generation of all products [58, 26]. Our comparison is based on one of them [58], but there was no tool support for variability encoding. The other approach by Hähnle and Schaefer [26] verifies features in isolation and than the whole product line similarly to variability encoding. However, they assume that features have to adhere to behavioral subtyping, which reduces the applicability [60] and is not assumed in our approach. All these approaches for theorem proving have neither been evaluated in the context of defects nor been compared to model checking.

Product-line model checking has been proposed for source code [44, 4, 33, 6], similar to our approach, and for abstract models [23, 25, 18]. Some of these approaches take advantage of variability encoding to reuse existing tools, such as NuSMV [18], CBMC [44], CPAchecker [4, 6], and JPF [33, 6] as we do. Others specify the expected behavior with an extension of μ-calculus [25], computation tree logic [18], and aspect-oriented programming [4, 6] (for further techniques see [55]), whereas we are the first to check feature-oriented contracts with model checking.

A common assumption of variability encoding is that all products of the product line are type safe (i.e., free of compiler errors) [58]. However, type safety can be efficiently checked for product lines [7, 54, 32, 21, 35, 34], and should be even combined with theorem proving and model checking.

7. CONCLUSION AND FUTURE WORK

Rather separate communities considered product-line verification by means of theorem proving and model checking, respectively. We propose to integrate both to verify feature-

oriented contracts. We use variability encoding to verify product lines without the need to generate all products. We provide tool support in Eclipse by extending FeatureIDE.

While theorem proving with KeY and model checking with JPF have been evaluated before, we are the first to evaluate the influence of defects and synergies of using both in concert, leading us to the following main results. First, theorem proving and model checking are both more effective and more efficient to find defects if the product line contains many defects rather than only some or no defects. Second, in our experiments, model checking was more efficient, but less effective than theorem proving. However, this heavily depends on the test scenarios used for model checking. Third, combining theorem proving and model checking improves effectiveness, especially if the product line contains only few defects, and may at the same time even improve efficiency compared to using only theorem proving.

In future work, our tool support should be used to to verify evolving product lines, in which non-artificial defects will occur. Our evaluation may be extended by considering the time to find at least one defect for each configuration, if existent [18, 6]. Besides theorem proving and model checking, further verification and testing techniques could be compared. A further dimension are other analysis strategies, such as sample-based or feature-family-based analyses, which could be considered for the integration of multiple verification techniques. Finally, it should be investigated how to combine verification techniques on a per-method basis (e.g., choosing a verification technique based on code metrics).

8. ACKNOWLEDGMENTS

This work is based on bachelor's theses by Jens Meinicke and Fabian Benduhn [12, 36]. We thank Sven Apel, Reiner Hähnle, Ina Schaefer, and Matthias Praast for discussions and the anonymous SPLC reviewers for their constructive reviews. We gratefully acknowledge Jörg Liebig and Christian Kästner for their help with FeatureHouse. This work is partially supported by the German Research Foundation (DFG – AP 206/4, SA 465/34-2).

9. REFERENCES

[1] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 1st edition, 2010.

[2] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer, 2013.

[3] S. Apel, C. Kästner, and C. Lengauer. Language-Independent and Automated Software Composition: The FeatureHouse Experience. *TSE*, 39(1):63–79, 2013.

[4] S. Apel, H. Speidel, P. Wendler, A. von Rhein, and D. Beyer. Detection of Feature Interactions using Feature-Aware Verification. In *ASE*, pages 372–375. IEEE, 2011.

[5] S. Apel, A. von Rhein, T. Thüm, and C. Kästner. Feature-Interaction Detection Based on Feature-Based Specifications. *ComNet*, 57(12):2399–2409, 2013.

[6] S. Apel, A. von Rhein, P. Wendler, A. Größlinger, and D. Beyer. Strategies for Product-Line Verification: Case Studies and Experiments. In *ICSE*, pages 482–491. IEEE, 2013.

[7] L. Aversano, M. D. Penta, and I. D. Baxter. Handling Preprocessor-Conditioned Declarations. In *SCAM*, pages 83–92. IEEE, 2002.

[8] M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and Verification: The Spec# Experience. *Comm. ACM*, 54:81–91, 2011.

[9] D. Batory. Feature Models, Grammars, and Propositional Formulas. In *SPLC*, pages 7–20. Springer, 2005.

[10] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *TSE*, 30(6):355–371, 2004.

[11] B. Beckert, R. Hähnle, and P. Schmitt. *Verification of Object-Oriented Software: The KeY Approach*. Springer, 2007.

[12] F. Benduhn. Contract-Aware Feature Composition. Bachelor's thesis, University of Magdeburg, Germany, 2012.

[13] E. Bodden, T. Tolêdo, M. Ribeiro, C. Brabrand, P. Borba, and M. Mezini. SPLLIFT: Statically Analyzing Software Product Lines in Minutes Instead of Years. In *PLDI*, pages 355–364. ACM, 2013.

[14] C. Brabrand, M. Ribeiro, T. Tolêdo, J. Winther, and P. Borba. Intraprocedural Dataflow Analysis for Software Product Lines. *TAOSD*, 10:73–108, 2013.

[15] D. Bruns, V. Klebanov, and I. Schaefer. Verification of Software Product Lines with Delta-Oriented Slicing. In *FoVeOOS*, pages 61–75. Springer, 2011.

[16] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An Overview of JML Tools and Applications. *STTT*, 7(3):212–232, 2005.

[17] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[18] A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. Symbolic Model Checking of Software Product Lines. In *ICSE*, pages 321–330. ACM, 2011.

[19] D. R. Cok. OpenJML: JML for Java 7 by Extending OpenJDK. In *NFM*, pages 472–479. Springer, 2011.

[20] F. Damiani, O. Owe, J. Dovland, I. Schaefer, E. B. Johnsen, and I. C. Yu. A Transformational Proof System for Delta-Oriented Programming. In *FMSPLE*, pages 53–60. ACM, 2012.

[21] F. Damiani and I. Schaefer. Family-Based Analysis of Type Safety for Delta-Oriented Software Product Lines. In *ISOLA*, pages 193–207. Springer, 2012.

[22] P. Dybjer, Q. Haiyan, and M. Takeyama. Verifying Haskell Programs by Combining Testing, Model Checking and Interactive Theorem Proving. *IST*, 46(15):1011–1025, 2004.

[23] D. Fischbein, S. Uchitel, and V. Braberman. A Foundation for Behavioural Conformance in Software Product Line Architectures. In *ROSATEA*, pages 39–48. ACM, 2006.

[24] A. Gondal, M. Poppleton, and M. Butler. Composing Event-B Specifications: Case-Study Experience. In *SC*, pages 100–115. Springer, 2011.

[25] A. Gruler, M. Leucker, and K. Scheidemann. Modeling and Model Checking Software Product Lines. In *FMOODS*, pages 113–131. Springer, 2008.

[26] R. Hähnle and I. Schaefer. A Liskov Principle for

Delta-Oriented Programming. In *ISOLA*, pages 32–46. Springer, 2012.

[27] J. Y. Halpern and M. Y. Vardi. Model Checking vs. Theorem Proving: A Manifesto. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation*, pages 151–176. Academic Press Professional, Inc., 1991.

[28] J. Hatcliff, G. T. Leavens, K. R. M. Leino, P. Müller, and M. Parkinson. Behavioral Interface Specification Languages. *CSUR*, 44(3):16:1–16:58, 2012.

[29] K. Havelund and T. Pressburger. Model Checking Java Programs using Java PathFinder. *STTT*, 2(4):366–381, 2000.

[30] M. Ismail, O. Hasan, T. Ebi, M. Shafique, and J. Henkel. Formal Verification of Distributed Dynamic Thermal Management. In *ICCAD*, pages 248–255. IEEE, 2013.

[31] Y. Jia and M. Harman. An Analysis and Survey of the Development of Mutation Testing. *TSE*, 37(5):649–678, 2011.

[32] C. Kästner, S. Apel, T. Thüm, and G. Saake. Type Checking Annotation-Based Product Lines. *TOSEM*, 21(3):14:1–14:39, 2012.

[33] C. Kästner, A. von Rhein, S. Erdweg, J. Pusch, S. Apel, T. Rendel, and K. Ostermann. Toward Variability-Aware Testing. In *FOSD*, pages 1–8. ACM, 2012.

[34] S. Kolesnikov, A. von Rhein, C. Hunsen, and S. Apel. A Comparison of Product-based, Feature-based, and Family-based Type Checking. In *GPCE*, pages 115–124. ACM, 2013.

[35] J. Liebig, A. von Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. Scalable Analysis of Variable Software. In *ESECFSE*, pages 81–91. ACM, 2013.

[36] J. Meinicke. JML-Based Verification for Feature-Oriented Programming. Bachelor's thesis, University of Magdeburg, Germany, 2013.

[37] J. Meinicke, T. Thüm, R. Schöter, F. Benduhn, and G. Saake. An Overview on Analysis Tools for Software Product Lines. ACM, 2014. To appear.

[38] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., 1st edition, 1988.

[39] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 2010.

[40] S. Owre, S. P. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. PVS: Combining Specification, Proof Checking, and Model Checking. In R. Alur and T. A. Henzinger, editors, *CAV*, pages 411–414. Springer, 1996.

[41] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[42] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering : Foundations, Principles and Techniques*. Springer, 2005.

[43] M. Poppleton. Towards Feature-Oriented Specification and Development with Event-B. In *REFSQ*, pages 367–381. Springer, 2007.

[44] H. Post and C. Sinz. Configuration Lifting: Software Verification meets Software Configuration. In *ASE*, pages 347–350. IEEE, 2008.

[45] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *ECOOP*, pages 419–443. Springer, 1997.

[46] Robby, E. Rodríguez, M. B. Dwyer, and J. Hatcliff. Checking JML Specifications using an Extensible Software Model Checking Framework. *STTT*, 8(3):280–299, 2006.

[47] J. Rubin and M. Chechik. A Framework for Managing Cloned Product Variants. In *ICSE*, pages 1233–1236. IEEE, 2013.

[48] W. Scholz, T. Thüm, S. Apel, and C. Lengauer. Automatic Detection of Feature Interactions using the Java Modeling Language: An Experience Report. In *FOSD*, pages 7:1–7:8. ACM, 2011.

[49] R. Schröter, N. Siegmund, and T. Thüm. Towards Modular Analysis of Multi Product Lines. In *MultiPLE*, pages 96–99. ACM, 2013.

[50] J. Schumann. *Automated Theorem Proving in Software Engineering*. Springer, 2001.

[51] J. Sorge, M. Poppleton, and M. Butler. A Basis for Feature-Oriented Modelling in Event-B. In *ABZ*, pages 409–409. Springer, 2010.

[52] O. Strichman and B. Godlin. Verified Software: Theories, Tools, Experiments. In B. Meyer and J. Woodcock, editors, *Regression Verification - A Practical Way to Verify Programs*, pages 496–501. Springer, 2008.

[53] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *ICSE*, pages 107–119. ACM, 1999.

[54] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe Composition of Product Lines. In *GPCE*, pages 95–104. ACM, 2007.

[55] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. *CSUR*, 47(1):6:1–6:45, 2014.

[56] T. Thüm, S. Apel, A. Zelend, R. Schröter, and B. Möller. Subclack: Feature-Oriented Programming with Behavioral Feature Interfaces. In *MASPEGHI*, pages 1–8. ACM, 2013.

[57] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *SCP*, 79(0):70–85, 2014.

[58] T. Thüm, I. Schaefer, S. Apel, and M. Hentschel. Family-Based Deductive Verification of Software Product Lines. In *GPCE*, pages 11–20. ACM, 2012.

[59] T. Thüm, I. Schaefer, M. Kuhlemann, and S. Apel. Proof Composition for Deductive Verification of Software Product Lines. In *VAST*, pages 270–277. IEEE, 2011.

[60] T. Thüm, I. Schaefer, M. Kuhlemann, S. Apel, and G. Saake. Applying Design by Contract to Feature-Oriented Programming. In *FASE*, pages 255–269. Springer, 2012.

[61] D. M. Weiss. The Product Line Hall of Fame. In *SPLC*, page 395. IEEE, 2008.

[62] Y. Xue, Z. Xing, and S. Jarzabek. Feature Location in a Collection of Product Variants. In *WCRE*, pages 145–154. IEEE, 2012.