Verification of Software Product Lines Using Contracts

Thomas Thüm
School of Computer Science
University of Magdeburg
Magdeburg, Germany

***Abstract*—Software product lines are widely used to achieve high reuse of code artifacts for similar software products. While there are many efficient techniques to implement product lines, such as feature-oriented programming, the analysis and verification of product lines got only little attention so far. But as product lines are increasingly used in safety critical scenarios, efficient verification techniques are indispensable. We give an overview on the state-of-the-art in product-line verification, in which we classify approaches according to their strategy to scale specification and verification approaches known from single-system engineering. We propose to use contracts (i.e., preconditions and postconditions) to specify the intended behavior of a product line implemented with feature-oriented programming. Based on these contracts, we discuss different approaches to verify that all products of a product line fulfill its specification.**

***Keywords*-Software product lines, feature-oriented programming, design by contract, specification, verification**

I. Introduction

A major challenge in software engineering is to reduce the effort required to implement a certain functionality. In the last century, software engineering focused on reuse *within* one software system. For example, imperative programming encapsulates functionality in procedures to enable software reuse, whereas object-oriented programming provides more high-level reuse techniques such as class inheritance [1].

Software reuse *across* multiple software systems has gained much attention in the last decades [2]–[6]. The idea is to develop similar software systems not from scratch, but rather define the commonalities and variabilities between them in a software product line. A *software product line* (or short product line) is a set of software systems sharing a common code base [3]–[5]. The software systems (a.k.a. software products) are distinguished in terms of features. A *feature* is a prominent or distinctive user-visible aspect, quality, or characteristic of a software system [2]. We focus on feature-oriented programming for the implementation of product lines, in which each feature is implemented in a separate module [7], [8]. Given a particular selection of features, a customized product can be generated automatically by composing the corresponding modules [3], [6].

Another major challenge in software engineering is to verify the correctness of software systems. Especially safety-critical and mission-critical software systems need to be verified in order to prove the absence of failures. *Design by contract* is a methodology to formally specify object-oriented systems in terms of method contracts [9]. A *method contract* (or short contract) is assigned to each or at least each safety-critical method consisting of a *precondition* stating what the caller of a method needs to ensure and a *postcondition* stating what the caller can rely on. Contracts can be used to formally specify the intended behavior of a software system, which in turn can be used to verify that the software system fulfills its specification.

When product lines are used to implement safety-critical software systems, we need to apply specification and verification techniques from single-system engineering to product lines. A simple strategy is to specify and verify each product separately. Unfortunately, this strategy involves redundant effort for specification as well as for verification, because products of a product line have commonalities [10]. Furthermore, the number of products is up-to exponential in the number of implemented features, and thus the strategy infeasible for large product lines [10].

Our goal is to develop efficient verification techniques for product lines implemented with feature-oriented programming. In order to verify that a product line is correct, the intended behavior need to be specified efficiently. We propose to use contracts to formally specify feature-oriented programs [11]. We assign contracts to each feature from which the specification of each product can be derived automatically [11]. Based on contracts for each feature, we discuss different approaches ranging from testing to static analysis [12] and theorem proving [13], [14], which can verify that the implementation of each feature fulfills its contracts. Each approach has strengths and weaknesses regarding soundness, completeness, and efficiency. We summarize preliminary results on the scalability of these approaches.

II. Background

We present basic concepts that are necessary to understand our remaining discussion. We give a short overview on software product lines and feature-oriented programming.

A. Software Product Lines

A software product line is a set of products defined on a set of features F. A software product P is as a subset of all features $P \subseteq F$. Theoretically, we can combine the features in F in all combinations defined by the power set 2^F.

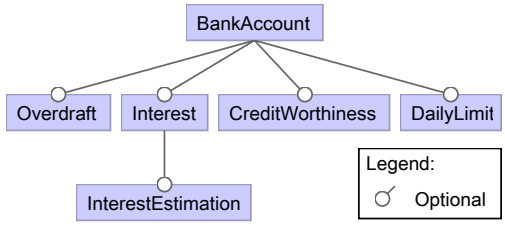

Figure 1. A feature model of a product line of bank accounts.

Practically, features may require other features or features may exclude each other [2], [3], [15]. For example, in a product line of database management systems, we may have support for different operating systems such as Windows and Unix, and a database product can either run on Windows or Unix. Hence, a software product line L is defined as a subset of all possible feature combinations: $L \subseteq 2^F$.

The formalization of a product line as a set of sets of features is often laborious. A common means to compactly describe valid feature combinations is a feature model [2], [3], [15]. A *feature model* is a hierarchy of features [2], in which each feature requires its parent feature [15]. A subfeature can be either mandatory, optional, or part of a group of alternative features [2].

In Figure 1, we give an example describing the features of a product line to manage bank accounts. The product line consists of six features. Feature *BankAccount* is part of all products and all other features are optional. Depending on the feature selection, a bank account may or may not support an overdraft limit (feature *Overdraft*), the calculation of interests (*Interest*), the estimation of credit worthiness (*CreditWorthiness*), or a maximum daily withdrawal (*DailyLimit*). Furthermore, feature *InterestEstimation* provides a calculation of the estimated interest until the end of the year. But, this feature requires feature *Interest* and is therefore modeled as a subfeature of it. Overall, the feature model defines a product line with 24 products.

B. Feature-Oriented Programming

So far, we discussed how to define valid combinations of features. Once these valid combinations have been defined, we need a product-line implementation technique that maps features to implementation units, in order to automatically generate software products for a given feature selection. While there are several implementation techniques [6], we focus on feature-oriented programming.

In feature-oriented programming, the code of each feature is encapsulated in a distinct feature module [7], [8]. A *feature module* is a set of classes and class refinements [8], in which a *class refinement* is a set of methods and fields. When composing class A with a class refinement A', all methods and fields of A' are added to A and existing methods are refined. Software products can be generated automatically by composing different combinations of feature modules.

feature module *BankAccount*

```
class Account {
  int balance = 0;
  void update(int x) {
    balance += x;
  }
}
class Application {
  Account account = new Account();
  void nextDay() {}
  void nextYear() {}
}
```

feature module *DailyLimit*

```
refines class Account {
  final static int DAILY_LIMIT = -1000;
  int withdrawal = 0;
  void update(int x) {
    original(x);
    if (x < 0) withdrawal += x;
  }
}
refines class Application {
  void nextDay() {
    original();
    account.withdrawal = 0;
  }
}
```

feature module *Interest*

```
refines class Account {
  final static int INTEREST_RATE = 2;
  int interest = 0;
  int calculateInterest() {
    return balance * INTEREST_RATE / 100 / 365;
  }
}
refines class Application {
  void nextDay() {
    original();
    account.interest +=
      account.calculateInterest();
  }
  void nextYear() {
    original();
    account.balance += account.interest;
    account.interest = 0;
  }
}
```

Figure 2. Three feature modules of the bank account product line.

In Figure 2, we illustrate three feature modules of the bank account product line. Feature module *BankAccount* provides a base implementation, which can store and update the current balance of an account. As defined in the feature model in Figure 1, we assume that this feature is part of every product, but its classes are refined by other features.

Feature module *DailyLimit* adds a field `withdrawal` to store the current withdrawal of the day. Method `update()` is refined to alter the withdrawal whenever the account balance is decreased. Keyword `original` refers to the method being subject of the refinement and is similar to `super` in object-oriented programming. Method `nextDay()` is assumed to be called every day at midnight and refined by feature module *DailyLimit* to reset the withdrawal of the day.

```
class Account {          product {BankAccount, DailyLimit}
  final static int DAILY_LIMIT = -1000;
  int balance = 0;
  int withdrawal = 0;
  void update(int x) {
    balance += x;
    if (x < 0) withdrawal += x;
  }
}
class Application {
  Account account = new Account();
  void nextDay() { account.withdrawal = 0; }
  void nextYear() {}
}
```

Figure 3. Composition of feature modules *BankAccount* and *DailyLimit*.

Similarly, feature module *Interest* adds a field `interest` to store the cumulated interests since beginning of the year, and adds a method to calculate the interests. The stored interests are updated daily using the refinement for method `nextDay()`, and credited to the account at the end of the year using the refinement for method `nextYear()`.

Four products can be created automatically by composing these three feature modules in different combinations: $\{B\}$, $\{B, I\}$, $\{B, D\}$, and $\{B, I, D\}$. In Figure 3, we illustrate the result of composing the feature modules *BankAccount* and *DailyLimit*. Classes are merged with identically named class refinements. The resulting classes contain all fields and methods defined in the composed feature modules. Already existing methods such as method `update()` are replaced, whereas the keyword `original` is substituted by the method body of the replaced method.

III. Product-Line Strategies

When software product lines are used in safety-critical contexts, we need to verify that all products behave as intended. Consequently, we need to specify the intended behavior of all products. We found several approaches for specification and verification of product lines in the literature. In recent work, we proposed a classification and survey on product-line analysis [10]. Using our classification, we identify strategies to scale specification and verification approaches to product lines. We briefly summarize our classification and discuss how different strategies deal with variability in specification and verification.

A. Specification Strategies for Product Lines

A simple strategy to specify a software product line is to define a specification that all products need to establish, called *global specification* [10]. For example, in a product line of pacemakers, all products have to admit to the same specification stating that a heart beat is generated whenever the heart stops beating [49]. Global specifications were used for different verification techniques such as model checking [17], [40], [42], [44] and static analyses [35], [36]. But in recent work, we found that global specifications are often too restrictive, because variability in implementation does usually require variability in specifications, too [11].

When global specifications are not applicable, we can specify each product of a product line separately, called *product-based specification* [10]. Clearly, specifying the behavior for every product scales only for product lines with few products. An optimization is to specify and analyze only a subset of all products, which is applicable if only this subset is used productively. We did not find any product-based specification approach in the literature, but every specification approach for a single software system can be applied to products individually. Product-based specifications may be useful if the product specifications are largely disjunct, and thus there is a low potential to reuse specifications over several products.

Another strategy is to specify each feature and to compose these specification in a similar manner as source code, called *feature-based specification* [10]. For example, in our bank account product line, we could add a specification to feature *DailyLimit* stating that the daily withdrawal never exceeds the limit. Then, this specification applies to all products containing feature *DailyLimit*. We identified that feature-based specifications were used for model checking only [18], [19], [31], [41], [43]. The main advantage of feature-based specifications is that specifications can be reused across several products. However, specifications applying to combinations of features cannot be defined.

A *family-based specification* is a specification annotated with a propositional formulas stating for which feature combinations the specification is assumed to hold [10]. For example, in a database management system, we might want to specify that statistics over transactions are gathered whenever both features are selected. Family-based specifications generalize product-based and feature-based specifications, because each product-based and feature-based specification is a family-based specifications per definition. Family-based specifications are used for model checking only [34].

B. Verification Strategies for Product Lines

A simple strategy to verify a software product line is to generate and verify all products separately, called *product-based verification* [10]. In principle, any standard verification technique applicable to the generated products can be used for product-based verification. But, product-based verification is feasible only for product lines with few products. We found no proposal in the literature explicitly suggesting an exhaustive product-based verification without any optimizations. But, we found some approaches that actually propose product-based analyses and do not discuss how to deal with many products; these approaches apply type checking [16], model checking [17]–[19], theorem proving [20], and runtime analyses [21] to product lines.

Verification Strategy	Type Checking	Model Checking	Theorem Proving	Other Techniques
Product-based	[16]	[17]–[19]	[20]	[21]
Family-based	[22]–[29]	[19], [25], [30]–[34]		[35], [36]
Feature-product-based	[37]–[39]	[40]–[44]	[43], [45], [46]	
Feature-family-based	[47]		[48]	

Table I
OVERVIEW ON VERIFICATION APPROACHES FOR SOFTWARE PRODUCT LINES.

A more efficient strategy is to verify all products simultaneously using a *family-based verification* [10]. The idea is either to make the verification tool variability-aware [30] or to generate a metaproduct simulating the behavior of all products [25]. For example, in our bank account product line, a metaproduct can be generated by composing *all* feature modules and transforming compile-time variability into runtime variability (i.e., creating a boolean variable for each feature and using dynamic branching to simulate the behavior of all feature combinations). A family-based strategy has been applied to type checking [22]–[29], model checking [19], [25], [30]–[34], and static analyses [35], [36].

Another strategy is to verify the implementation of each feature in isolation without considering other features, called *feature-based verification* [10]. The goal of feature-based verification is to reduce the potentially exponential number of verification tasks (i.e., for every product) to a linear number of verification tasks (i.e., for every feature). But, a feature-based verification can detect only issues *within* a certain feature and does not care about issues *across* features. However, a well-known problem are *feature interactions*: several features work as expected in isolation, but lead to unexpected behavior in combination [50]. Thus, a feature-based strategy can usually not be used for verification as-is. However, it can be combined with other strategies.

In the literature, we found that product-based, family-based, and feature-based strategies are also combined to achieve synergies [10]. The most commonly proposed combination is *feature-product-based verification*, in which features are verified as far as possible in isolation and all remaining verification tasks are done for each product. A feature-product-based strategy is applied to scale type checking [37]–[39], model checking [40]–[44], and theorem proving [43], [45], [46] to product lines. Similarly, *feature-family-based verification* has been proposed using type checking [47] and theorem proving [48].

In our recent survey, we give examples for each strategy and discuss advantages and disadvantages in detail [10]. In Table I, we give an overview on all classified approaches, from which we can make some observations regarding new and underrepresented research areas. First, feature-family-based verification is a young research area for which only two approaches exist so far. Second, while there is a large number of approaches for family-based type checking and family-based model checking, we found not a single approach applying a family-based strategy to theorem proving. But, we argue that several verification techniques such as type checking, model checking, and theorem proving should be applied to product lines, because each technique has strengths and weaknesses [10]. For example, type checking is limited in the errors that can be detected [51] and model checking might not terminate or run out of memory due to the state explosion [52]. We fill this gap by proposing *family-based theorem proving* using feature-based specifications.

IV. SPECIFICATION OF FEATURE MODULES

Our goal is to verify feature modules using contracts, which naturally raises the question how to define contracts for feature modules and how to specify the intended behavior of all products. We give a short overview, how contracts can be defined for object-oriented classes. Then, we present our approach to define contracts for feature modules.

A. Contracts for Object-Oriented Classes

In 1949, Alan Turing formulated that the correctness of large methods should be verified using assertions to simplify the verification [53]. In 1969, Tony Hoare formalized assertions in terms of preconditions and postconditions using the well-known Hoare triple [54]. Two decades later, Bertrand Meyer made assertions popular in object-oriented programming as contracts and invariants [9]. *Contracts* are assigned to methods consisting of a precondition and a postcondition. The precondition is an assertion that callers of the method need to fulfill and the method can rely on. Vice versa, the postcondition must be fulfilled by the method and can be assumed by the caller. *Invariants* are assertions that must hold after each constructor call as well as before and after the execution of public methods.

We use the Java Modeling Language (JML) to define contracts, a behavioral specification language for Java with support for contracts and invariants [55]. In Figure 4, we give a JML specification of feature module *BankAccount*, which is a standard Java program. In JML, a contract is defined using keywords `requires` and `ensures`, denoting the precondition and postcondition, respectively. In our example, the precondition of method `update()` is always fulfilled and the postcondition is stating that the account balance is updated correctly. Additionally, an invariant is defined in class `Application` stating that field `account` is not null.

feature module *BankAccount*

```
class Account {
  int balance = 0;
  /*@
   @ requires true;
   @ ensures balance == \old(balance) + x;
   @*/
  void update(int x) {
    balance += x;
  }
}
class Application {
  //@ invariant account != null;
  Account account = new Account();
}
```

Figure 4. JML contracts and invariants in Java classes.

B. Contracts for Feature Modules

In recent work, we proposed and discussed five approaches to define contracts for feature modules [11]. All approaches enable feature-based specifications from which the specification of each product can be derived automatically. Contracts are composed in a similar manner as feature modules. Thus, specifications do not need to be defined for each product, because we can reuse specifications across several products. In the following, we exemplify one of these approaches, namely *explicit contract refinement*.

In Figure 5, we present contracts for class `Account` in *DailyLimit* and *Interest*. Contracts and invariants may be defined as known from object-oriented programming, except the specification of method contracts for refined methods. For example, method `calculateInterest()` is specified using a JML contract, which only holds when feature *Interest* is selected. Feature *DailyLimit* introduces a new invariant stating that the withdrawal is within the limit. Similarly, this invariant only needs to be established in all products containing feature *DailyLimit*.

The interesting case is the contract of method `update()`. In explicit contract refinement [11], a contract defined for a method refinement replaces the contract of the refined method. In our example, the contract defined in feature *DailyLimit* replaces the contract defined in feature *BankAccount*. But, the replaced contract can be reused with the keyword `original` in a similar manner as in the implementation: `original` in a precondition refers to the replaced precondition and `original` in a postcondition refers to the replaced postcondition. The refined contract extends the precondition to ensure that the daily withdrawal is within the limit and the postcondition to specify that the withdrawal is updated correctly whenever method `update()` is called.

feature module *DailyLimit*

```
refines class Account {
  final static int DAILY_LIMIT = -1000;
  //@ invariant withdrawal >= DAILY_LIMIT;
  int withdrawal = 0;
  /*@
   @ requires \original &&
   @   (withdrawal + x >= DAILY_LIMIT)
   @ ensures \original &&
   @   (x>=0 ==> withdrawal==\old(withdrawal)) &&
   @   (x<0 ==> withdrawal==\old(withdrawal)+x);
   @*/
  void update(int x) {
    original(x);
    if (x < 0) withdrawal += x;
  }
}
```

feature module *Interest*

```
refines class Account {
  final static int INTEREST_RATE = 2;
  int interest = 0;
  /*@
   @ requires true;
   @ ensures (balance >= 0 ==> \result >= 0) &&
   @   (balance <= 0 ==> \result <= 0);
   @*/
  int calculateInterest() {
    return balance * INTEREST_RATE / 100 / 365;
  }
}
```

Figure 5. Feature module specification using explicit contract refinement.

V. Verification of Feature Modules

Specifying Java programs with JML is not only beneficial for verification. Formal specification using JML can be used for documentation generation, runtime assertion checking, automatic test generation, extended static checking, and formal verification using theorem proving [56]. We and others argue that a multitude of techniques is needed to verify the correctness of programs [56], [57]. When formally proving the correctness, the program should already be tested forehand, because formal verification is too expensive for extensive bug finding [56]. Furthermore, certain properties are hard to be proved statically and should be checked at runtime, whereas not all properties should be checked at runtime to minimize the runtime overhead [57]. Hence, when verifying feature modules, a multitude of techniques is needed to efficiently detect errors as well as to prove the absence of errors. In the following, we summarize our approaches for the verification of feature modules.

A. Product-Based Runtime Assertion Checking

A popular application of contracts are runtime assertions [9], [56], [57]. The idea is to check contracts at runtime using a special compiler (e.g., JMLC). The advantage of runtime assertion checking is that contracts may be checked when testing the program, but do not cause runtime overhead in a release version compiled with a standard Java compiler.

We are currently implementing tool support for product-based runtime assertion checking in the integrated development environment FeatureIDE [58] based on an extension of the composer FeatureHouse [59]. When composing feature modules to generate a certain product, contracts are composed resulting in a standard Java program with

```
class Account {          product {BankAccount, DailyLimit}
  final static int DAILY_LIMIT = -1000;
  int balance = 0;
  //@ invariant withdrawal >= DAILY_LIMIT;
  int withdrawal = 0;
  /*@
   @ requires true &&
   @   (withdrawal + x >= DAILY_LIMIT)
   @ ensures balance == \old(balance) + x &&
   @   (x>=0 ==> withdrawal==\old(withdrawal)) &&
   @   (x<0 ==> withdrawal==\old(withdrawal)+x);
   @*/
  void update(int x) {
    balance += x;
    if (x < 0)
      withdrawal += x;
  }
}
```

Figure 6. Composition of specifications for *BankAccount* and *DailyLimit*.

JML specifications. We illustrate the result of composing class `Account` from features *BankAccount* and *DailyLimit* in Figure 6. Feature modules are composed as shown in Figure 3 and the keyword `original` in contracts is replaced similarly. The approach is product-based, because contract violations are detected at runtime for every product individually. A possible optimization is to choose a subset of all products that is likely to detect all errors [10].

B. Product-Based Extended Static Checking

The generated Java programs with JML specifications can also be used as input for static analysis tools such as extended static checkers. We pursued product-based extended static checking using ESC/JAVA2 for the detection of feature interactions [12]. We were able to detect all feature interactions in a small product line of list implementations, but the detection was not straightforward. The reason is that ESC/JAVA2 is unsound and incomplete (e.g., false positives and false negatives may occur), because loops are only unrolled a fixed number of times [60]. Hence, the tool cannot be used to prove the absence of errors, but as runtime assertion checking it is valuable for bug finding.

C. Feature-Product-Based Theorem Proving

Formal specifications in JML can be used to prove that a program behaves as intended. A verification tool translates a JML-annotated Java program into the input language of a theorem prover. One such tool is `Why/Krakatoa` [61], which supports multiple theorem provers such as COQ [62]. COQ is an interactive theorem prover meaning that user interaction is necessary to prove theorems. More precisely, the user needs to write textual commands (i.e., a proof script) that apply certain proof steps until the proof is finished.

We proposed feature-product-based theorem proving using the above mentioned tool chain and proof composition [13]. The idea is to write proof scripts for each feature and compose them together with specification and source code. Then, the composed proof scripts are checked for every product, but the user only needs to write proof scripts once per feature. Hence, this is a feature-product-based approach, in which only the feature-based part requires user interaction and the product-based part can be checked fully automatically using COQ. Using this approach, we were able to reduce the effort to write proof scripts by 88 % [13].

D. Family-Based Theorem Proving

All approaches discussed above rely on the generation of products, which can be problematically for large product lines, because the number of products is up to exponential in the number of features. When the goal is to find errors (e.g., with testing), it is usually sufficient to analyze only a subset of all products. But when the goal is to prove the absence of errors (e.g., with theorem proving), all products need to be generated to achieve completeness.

We proposed the first approach for family-based theorem proving avoiding the generation of all products [14]. The idea is to generate a metaproduct simulating all products and a metaspecification equivalent to all product specifications. We showed how to use the theorem prover KeY as-is for the verification of product lines. We evaluated our approach by means of a case study and measured that the automatic verification of the metaproduct saves 85 % calculation time compared to the individual verification of all products [14].

VI. Conclusion and Future Work

Efficient strategies for specification and verifications are indispensable for safety-critical software product lines. We classified existing approaches according to product-based, feature-based, and family-based strategies. For specification, we identified global specifications as a further strategy. For verification, also combined strategies such as feature-product-based or feature-family-based verification have been proposed in the literature.

We propose to use contracts to formally specify the intended behavior of product lines implemented with feature-oriented programming. Each feature module is specified using contracts and product specifications can be generated along with source code. Based on contract composition, we discussed product-based runtime assertion checking and extended static checking for bug finding. Furthermore, we discussed feature-product-based theorem proving and family-based theorem proving for proving the absence of errors.

In future work, we intend to evaluate further approaches such as feature-family-based theorem proving and family-based testing. Furthermore, we continue building tool support for multiple specification approaches applying contracts to feature modules [58]. We also intend to formalize already presented specification approaches [11]. Finally, we are going to investigate how evolving product lines can be verified efficiently based on our previous work [63], [64].

Acknowledgment

I thank my supervisor Gunter Saake and Norbert Siegmund for comments on an earlier draft. I gratefully acknowledge the co-authors of previous publications, especially Christian Kästner, Ina Schaefer, Sven Apel, Don Batory, Martin Kuhlemann, and Fabian Benduhn.

References

[1] B. Meyer, *Object-Oriented Software Construction*, 1st ed. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1988.

[2] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Software Engineering Institute, Tech. Rep. CMU/SEI-90-TR-21, 1990.

[3] K. Czarnecki and U. W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. New York, NY, USA: ACM/Addison-Wesley, 2000.

[4] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*. Boston, MA, USA: Addison-Wesley, 2001.

[5] K. Pohl, G. Böckle, and F. J. van der Linden, *Software Product Line Engineering : Foundations, Principles and Techniques*. Berlin, Heidelberg, New York, London: Springer, 2005.

[6] S. Apel and C. Kästner, "An Overview of Feature-Oriented Software Development," *J. Object Technology (JOT)*, vol. 8, no. 5, pp. 49–84, 2009.

[7] C. Prehofer, "Feature-Oriented Programming: A Fresh Look at Objects," in *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*. Berlin, Heidelberg, New York, London: Springer, 1997, pp. 419–443.

[8] D. Batory, J. N. Sarvela, and A. Rauschmayer, "Scaling Step-Wise Refinement," *IEEE Trans. Software Engineering (TSE)*, vol. 30, no. 6, pp. 355–371, 2004.

[9] B. Meyer, "Applying Design by Contract," *IEEE Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[10] T. Thüm, S. Apel, C. Kästner, M. Kuhlemann, I. Schaefer, and G. Saake, "Analysis Strategies for Software Product Lines," School of Computer Science, University of Magdeburg, Germany, Tech. Rep. FIN-004-2012, 2012.

[11] T. Thüm, I. Schaefer, M. Kuhlemann, S. Apel, and G. Saake, "Applying Design by Contract to Feature-Oriented Programming," in *Proc. Int'l Conf. Fundamental Approaches to Software Engineering (FASE)*. Berlin, Heidelberg, New York, London: Springer, 2012, pp. 255–269.

[12] W. Scholz, T. Thüm, S. Apel, and C. Lengauer, "Automatic Detection of Feature Interactions using the Java Modeling Language: An Experience Report," in *Proc. Int'l Workshop Feature-Oriented Software Development (FOSD)*. New York, NY, USA: ACM, 2011, pp. 7:1–7:8.

[13] T. Thüm, I. Schaefer, M. Kuhlemann, and S. Apel, "Proof Composition for Deductive Verification of Software Product Lines," in *Proc. Int'l Workshop Variability-intensive Systems Testing, Validation and Verification (VAST)*. Washington, DC, USA: IEEE, 2011, pp. 270–277.

[14] T. Thüm, I. Schaefer, S. Apel, and M. Hentschel, "Family-Based Deductive Verification of Software Product Lines," in *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, 2012, to appear.

[15] D. Batory, "Feature Models, Grammars, and Propositional Formulas," in *Proc. Int'l Software Product Line Conference (SPLC)*. Berlin, Heidelberg, New York, London: Springer, 2005, pp. 7–20.

[16] S. Apel, C. Kästner, and C. Lengauer, "Feature Featherweight Java: A Calculus for Feature-Oriented Programming and Stepwise Refinement," in *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*. New York, NY, USA: ACM, 2008, pp. 101–112.

[17] T. Kishi and N. Noda, "Formal Verification and Software Product Lines," *Comm. ACM*, vol. 49, pp. 73–77, 2006.

[18] S. Apel, W. Scholz, C. Lengauer, and C. Kästner, "Detecting Dependences and Interactions in Feature-Oriented Design," in *Proc. Int'l Symposium Software Reliability Engineering (ISSRE)*. Washington, DC, USA: IEEE, 2010, pp. 161–170.

[19] S. Apel, H. Speidel, P. Wendler, A. von Rhein, and D. Beyer, "Detection of Feature Interactions using Feature-Aware Verification," in *Proc. Int'l Conf. Automated Software Engineering (ASE)*. Washington, DC, USA: IEEE, 2011, pp. 372–375.

[20] D. Bruns, V. Klebanov, and I. Schaefer, "Verification of Software Product Lines with Delta-Oriented Slicing," in *Proc. Int'l Conf. Formal Verification of Object-Oriented Software (FoVeOOS)*. Berlin, Heidelberg, New York, London: Springer, 2011, pp. 61–75.

[21] V. V. Rubanov and E. A. Shatokhin, "Runtime Verification of Linux Kernel Modules Based on Call Interception," in *Proc. Int'l Conf. Software Testing, Verification and Validation (ICST)*. Washington, DC, USA: IEEE, 2011, pp. 180–189.

[22] L. Aversano, M. D. Penta, and I. D. Baxter, "Handling Preprocessor-Conditioned Declarations," in *Proc. Int'l Workshop Source Code Analysis and Manipulation (SCAM)*. Washington, DC, USA: IEEE, 2002, pp. 83–92.

[23] K. Czarnecki and K. Pietroszek, "Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints," in *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*. New York, NY, USA: ACM, 2006, pp. 211–220.

[24] S. Thaker, D. Batory, D. Kitchin, and W. Cook, "Safe Composition of Product Lines," in *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*. New York, NY, USA: ACM, 2007, pp. 95–104.

[25] H. Post and C. Sinz, "Configuration Lifting: Software Verification meets Software Configuration," in *Proc. Int'l Conf. Automated Software Engineering (ASE)*. Washington, DC, USA: IEEE, 2008, pp. 347–350.

[26] M. Kuhlemann, D. Batory, and C. Kästner, "Safe Composition of Non-Monotonic Features," in *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*. New York, NY, USA: ACM, 2009, pp. 177–186.

[27] F. Heidenreich, "Towards Systematic Ensuring Well-Formedness of Software Product Lines," in *Proc. Int'l Workshop Feature-Oriented Software Development (FOSD)*. New York, NY, USA: ACM, 2009, pp. 69–74.

[28] S. Apel, C. Kästner, A. Größlinger, and C. Lengauer, "Type Safety for Feature-Oriented Product Lines," *Automated Software Engineering*, vol. 17, no. 3, pp. 251–300, 2010.

[29] C. Kästner, S. Apel, T. Thüm, and G. Saake, "Type Checking Annotation-Based Product Lines," *Trans. Software Engineering and Methodology (TOSEM)*, vol. 21, no. 3, 2012, to appear.

[30] A. Gruler, M. Leucker, and K. Scheidemann, "Modeling and Model Checking Software Product Lines," in *Proc. IFIP Int'l Conf. Formal Methods for Open Object-based Distributed Systems (FMOODS)*. Berlin, Heidelberg, New York, London: Springer, 2008, pp. 113–131.

[31] K. Lauenroth, K. Pohl, and S. Toehning, "Model Checking of Domain Artifacts in Product Line Engineering," in *Proc. Int'l*

Conf. Automated Software Engineering (ASE). Washington, DC, USA: IEEE, 2009, pp. 269–280.

[32] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin, "Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines," in *Proc. Int'l Conf. Software Engineering (ICSE)*. New York, NY, USA: ACM, 2010, pp. 335–344.

[33] I. Schaefer, D. Gurov, and S. Soleimanifard, "Compositional Algorithmic Verification of Software Product Lines," in *Proc. Int'l Symposium on Formal Methods for Components and Objects (FMCO)*. Berlin, Heidelberg, New York, London: Springer, 2010, pp. 184–203.

[34] A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay, "Symbolic Model Checking of Software Product Lines," in *Proc. Int'l Conf. Software Engineering (ICSE)*. New York, NY, USA: ACM, 2011, pp. 321–330.

[35] C. Brabrand, M. Ribeiro, T. Tolêdo, and P. Borba, "Intraprocedural Dataflow Analysis for Software Product Lines," in *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*. New York, NY, USA: ACM, 2012, pp. 13–24.

[36] E. Bodden, "Position Paper: Static Flow-Sensitive & Context-Sensitive Information-Flow Analysis for Software Product Lines," in *Proc. Workshop Programming Languages and Analysis for Security (PLAS)*, 2012, to appear.

[37] S. Apel and D. Hutchins, "A Calculus for Uniform Feature Composition," *Trans. Programming Languages and Systems (TOPLAS)*, vol. 32, pp. 19:1–19:33, 2010.

[38] L. Bettini, F. Damiani, and I. Schaefer, "Implementing Software Product Lines Using Traits," in *Proc. ACM Symposium on Applied Computing (SAC)*. New York, NY, USA: ACM, 2010, pp. 2096–2102.

[39] I. Schaefer, L. Bettini, and F. Damiani, "Compositional Type-Checking for Delta-Oriented Programming," in *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*. New York, NY, USA: ACM, 2011, pp. 43–56.

[40] K. Fisler and S. Krishnamurthi, "Modular Verification of Collaboration-based Software Designs," in *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. New York, NY, USA: ACM, 2001, pp. 152–163.

[41] H. C. Li, S. Krishnamurthi, and K. Fisler, "Interfaces for Modular Feature Verification," in *Proc. Int'l Conf. Automated Software Engineering (ASE)*. Washington, DC, USA: IEEE, 2002, pp. 195–204.

[42] ——, "Modular Verification of Open Features Using Three-Valued Model Checking," *Automated Software Engineering*, vol. 12, no. 3, pp. 349–382, 2005.

[43] M. Poppleton, "Towards Feature-Oriented Specification and Development with Event-B," in *Proc. Int'l Working Conf. Requirements Engineering: Foundation for Software Quality (REFSQ)*. Berlin, Heidelberg, New York, London: Springer, 2007, pp. 367–381.

[44] J. Liu, S. Basu, and R. Lutz, "Compositional Model Checking of Software Product Lines using Variation Point Obligations," *Automated Software Engineering*, vol. 18, no. 1, pp. 39–76, 2011.

[45] D. Batory and E. Börger, "Modularizing Theorems for Software Product Lines: The Jbook Case Study," *J. Universal Computer Science (J.UCS)*, vol. 14, no. 12, pp. 2059–2082, 2008.

[46] B. Delaware, W. Cook, and D. Batory, "Product Lines of Theorems," in *Proc. Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. New York, NY, USA: ACM, 2011, pp. 595–608.

[47] B. Delaware, W. R. Cook, and D. Batory, "Fitting the Pieces Together: A Machine-Checked Model of Safe Composition," in *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*. New York, NY, USA: ACM, 2009, pp. 243–252.

[48] R. Hähnle and I. Schaefer, "A Liskov Principle for Delta-oriented Programming," in *Proc. Int'l Conf. Formal Verification of Object-Oriented Software (FoVeOOS)*. Karlsruhe, Germany: Technical Report 2011-26, Department of Informatics, Karlsruhe Institute of Technology, 2011, pp. 190–207.

[49] J. Liu, J. Dehlinger, and R. Lutz, "Safety Analysis of Software Product Lines using State-based Modeling," *J. Systems and Software (JSS)*, vol. 80, no. 11, pp. 1879–1892, 2007.

[50] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec, "Feature Interaction: A Critical Review and Considered Forecast," *Computer Networks*, vol. 41, no. 1, pp. 115–141, 2003.

[51] B. C. Pierce, *Types and Programming Languages*. Cambridge, Massachusetts, USA: MIT Press, 2002.

[52] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, Massachussetts: The MIT Press, 1999.

[53] A. Turing, "Checking a Large Routine," in *Conference on High Speed Automatic Calculating Machines*, 1949, pp. 67–69.

[54] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," *Comm. ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[55] G. T. Leavens, A. L. Baker, and C. Ruby, "Preliminary Design of JML: A Behavioral Interface Specification Language for Java," *SIGSOFT Softw. Eng. Notes*, vol. 31, no. 3, pp. 1–38, 2006.

[56] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll, "An Overview of JML Tools and Applications," *Int'l J. Software Tools for Technology Transfer (STTT)*, vol. 7, no. 3, pp. 212–232, 2005.

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

[58] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich, "FeatureIDE: An Extensible Framework for Feature-Oriented Software Development," *Science of Computer Programming (SCP)*, 2012, to appear; accepted 2012-06-07.

[59] S. Apel, C. Kästner, and C. Lengauer, "FeatureHouse: Language-Independent, Automated Software Composition," in *Proc. Int'l Conf. Software Engineering (ICSE)*. Washington, DC, USA: IEEE, 2009, pp. 221–231.

[60] P. Chalin, J. R. Kiniry, G. T. Leavens, and E. Poll, "Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2," in *Proc. Int'l Symposium on Formal Methods for Components and Objects (FMCO)*. Berlin, Heidelberg, New York, London: Springer, 2005, pp. 342–363.

[61] J.-C. Filliâtre and C. Marché, "The Why/Krakatoa/Caduceus Platform for Deductive Program Verification," in *Computer Aided Verification*. Berlin, Heidelberg, New York, London: Springer, 2007, pp. 173–177.

[62] Coq Development Team, *The Coq Proof Assistant Reference Manual*, LogiCal Project, 2010, version 8.3.

[63] T. Thüm, D. Batory, and C. Kästner, "Reasoning about Edits to Feature Models," in *Proc. Int'l Conf. Software Engineering (ICSE)*. Washington, DC, USA: IEEE, 2009, pp. 254–264.

[64] T. Thüm, C. Kästner, S. Erdweg, and N. Siegmund, "Abstract Features in Feature Modeling," in *Proc. Int'l Software Product Line Conference (SPLC)*. Washington, DC, USA: IEEE, 2011, pp. 191–200.