Product-Line Verification with Feature-Oriented Contracts

Thomas Thüm
University of Magdeburg
Germany

ABSTRACT

Software product lines allow programmers to reuse code across similar software products. Software products are decomposed into separate modules representing user-visible features. Based on a selection of desired features, a customized software product can be generated automatically. However, these reuse mechanisms challenge existing techniques for specification and verification of software. Specifying and verifying each product involves redundant steps, and is often infeasible. We discuss how method contracts (i.e., preconditions and postconditions) can be used to efficiently specify and verify product lines.

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, feature-oriented programming, design by contract, verification, Java Modeling Language

1. INTRODUCTION

In software engineering, a major challenge is to reduce the effort for software development. On the one hand, the effort can be reduced using high-level reuse techniques for source code and specification. On the other hand, efficient techniques for testing and verification are needed. We focus on both, reuse for specifications and efficient verification.

In object-oriented programming, reuse *within* one software system is achieved by reuse techniques such as class inheritance. Feature-oriented programming facilitates reuse *across* several software systems based on object orientation [5]. Software systems are decomposed into *feature modules* according to user-visible features. Given a selection of desired features, a software product can be generated automatically by composing the respective feature modules. The set of software products that can be generated from a set of feature modules is called software product line [1].

We discuss how to specify feature modules such that the specification for each software product can be generated based on a feature selection. For specification, we rely on design by contract; a methodology to formally specify object-oriented systems in terms of method contracts [16, 11]. A *method contract* consists of a *precondition* stating properties the caller of a method needs to ensure, and a *postcondition* stating properties the caller can rely on.

In addition, we discuss approaches to verify software product lines based on feature-oriented contracts. An advantage of contracts compared to other specification techniques is that they can be used for runtime assertion checking, static analysis, and deductive verification [8, 4]. We discuss how all these verification approaches can be applied to feature modules. In our experience, each approach has unique strengths and weaknesses regarding soundness, completeness, and efficiency. Consequently, we need these approaches in concert when verifying software product lines.

2. STATE OF THE ART

In a recent survey [19], we give an overview and classification of product-line specification and verification. For brevity, we select representative approaches to illustrate the state-of-the-art and the research gap we are aiming at.

Product-Line Specification. A product line can be specified using a *global specification* that all products need to establish [19]. For instance, in a product line of pacemakers, all products must generate a heart beat whenever the heart stops beating. Global specifications were used for model checking [9] and static analyses [6]. We found that a global specification is too restrictive for product lines, because only behavior that is common to all products can be specified [24].

Another strategy is to specify each product separately in a *product-based specification* [19]. There is no proposal pursuing a product-based specification in the literature. However, in principle, every specification approach for a single software system can be applied to products individually. A product-based specification does scale well, because it does not facilitate any reuse opportunities across products, and thus is not an option for us.

In contrast, *feature-based specifications* aim at reuse: We can specify each feature individually, and compose these specifications based on a feature selection [19]. Feature-based specifications have been used only for model check-

ISSTA'13, July 15–20, 2013, Lugano, Switzerland
Copyright 2013 ACM 978-1-4503-2159-4/13/07...$15.00
http://dx.doi.org/10.1145/2483760.2492396

ing [15]. We rely on feature-based specifications, because we can define contracts directly at the source code, as proposed by design by contract [16].

Product-Line Verification. A strategy for product-line verification is to generate and verify all products separately in a *product-based verification* [19]. In principle, any standard verification technique applicable to the generated products can be used for product-based verification. Product-based verification is used for model checking [14] and theorem proving [7]. However, product-based verification inherently involves redundant effort.

In a *feature-based verification*, the implementation of each feature is verified in isolation, without considering other features [19]. As feature-based verification can detect only issues *within* features and not *across* features, it is often combined with product-based verification to achieve synergies [19]. In such a *feature-product-based verification*, features are verified as far as possible in isolation, and all remaining verification tasks are done for each product. A feature-product-based strategy has been proposed for type checking [2], model checking [9], and theorem proving [17].

A further strategy is to verify all products simultaneously using a *family-based verification* [19]. To simulate all products, verification tools are made variability-aware, or compile-time variability is translated into runtime variability [19]. A family-based strategy has been applied to type checking [13], model checking [10], and static analyses [6].

3. PROBLEM STATEMENT

Software product lines require reuse mechanisms for both, source code and specifications. While there are several mechanisms for code reuse [1], there is a lack of *sophisticated reuse mechanisms for specifications*. Some approaches to define feature-based specifications have been presented in the literature (e.g., [15], see further examples in our survey [19]). However, specification approaches are usually taken as given and not justified by theoretical or practical evaluations, as the focus is on the verification, rather than specification.

Besides reuse in source code and specifications, we need efficient techniques for product-line testing and verification. In our survey [19], we noticed that existing work usually focuses on one verification technique (e.g., model checking), and thus the specification approach is usually specific to a certain technique. For example, a system specified in the input language of a model checker can hardly be verified by means of theorem proving. However, we and others [8, 4] argue that once a system is specified formally, we need a *combination of several static and dynamic techniques for verification*. When formally proving the correctness, the program should already be tested forehand, because formal verification is too expensive for extensive bug finding [8]. 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 [4]. Even static techniques such as theorem proving, model checking, and static analysis may be combined to achieve synergies and efficiently prove product lines.

We tackle both problems with design by contract, because it is a well understood approach to formally specify and verify software [16, 8, 4, 11]. In particular, we can use contracts for both, dynamic and static verification techniques [8, 4]. In addition, contracts bring a new form of compositionality

feature module *BankAccount*

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

feature module *DailyLimit*

```
refines class Account {
  final static int DAILY_LIMIT = -1000;
  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;
} }
```

Figure 1: Feature modules specified using explicit contract refinement.

into static verification, because methods can be analyzed in isolation with only referring to the contracts of other methods. The benefit of such compositionality is twofold. First, the static verification of a product line can be split into manageable pieces. Second, once a product line is verified, we only need to consider parts of it when the product line evolves. Both advantages are especially helpful for product-line verification, as product lines inherently have a larger code base than single systems and evolve more frequently. In particular, we address the following research questions:

RQ1 How to make product-line specifications reusable and applicable to several verification techniques?

RQ2 Which combinations of verification techniques (e.g., theorem proving) and strategies (e.g., family-based) are feasible and what are their advantages?

4. CONTRACTS IN FEATURE MODULES

We answer RQ1 by extending an existing implementation technique for product lines with support for design by contracts, because contracts should be defined directly at the source code [16]. While there are several implementation techniques for software product lines, we focus on techniques that allow to automatically generate products for a given feature selection [1]. We choose feature-oriented programming for our work, because it enables the modularization of large-scale refinements [5] (e.g., in contrast to preprocessor directives). Furthermore, feature-oriented programming supports the modularization of most cross-cutting concerns with lightweight language extensions [12] (e.g., in contrast to aspect-oriented programming).

Feature-Oriented Programming. A *feature module* encapsulates a set of classes and class refinements. A *class refinement* is a set of methods and fields [5]. When composing class A with a class refinement A', all methods and fields of A' are added to A and existing methods are refined.

In Figure 1, we illustrate two feature modules of a bank-account product line (ignore comments for now). Feature module *BankAccount* provides a base implementation, which can store and update the current balance of an account. Feature module *DailyLimit* adds a field `withdrawal` to store

```
class Account {                product {BankAccount, DailyLimit}
  final static int DAILY_LIMIT = -1000;
  int balance = 0;
  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 2: Composition of feature modules *BankAccount* and *DailyLimit*.

the withdrawal of the day. Method `update` is refined to alter the withdrawal whenever the balance is decreased. Keyword `original` refers to the method being subject of the refinement and is similar to `super` for inheritance.

A product can be generated by composing feature modules according to a selection desired features [5]. In Figure 2, 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. In our example, only feature *DailyLimit* is optional giving rise to two products. In general, with n optional, independent feature modules, we can generate 2^n products.

Explicit Contract Refinement. We extend feature modules with support for design by contract. Methods and method refinements can be annotated with method contracts to formally specify their behavior. In explicit contract refinement, contracts of method refinements can refer to the contract of the method that is subject to refinement [24].

We use a feature-oriented extension of the Java Modeling Language (JML) [8] for illustration. To realize the concept of explicit contract refinement, we extend JML by the keyword `\original` to support feature-oriented composition similar to method composition [24]. In Figure 1, we illustrate contracts for the feature modules *BankAccount* and *DailyLimit*. 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` in feature *BankAccount* is always fulfilled, and the postcondition is stating that the account balance is updated correctly. In feature *DailyLimit*, a new contract is defined for method `update` stating that the daily withdrawal is within the limit. Previously defined preconditions and postconditions are referenced using keyword `\original`. We illustrate the result of composition in Figure 2.

Explicit contract refinement is only one approach to define and compose feature-oriented contracts. In previous work, we proposed and discussed five approaches to define contracts for feature modules and evaluated them by means of case studies [24]. All approaches enable feature-based specifications from which the specification of each product can be derived automatically. However, they differ in their applicability and expressiveness. In future work, we want to formalize these approaches and evaluate them using further case studies to assess to which extent variability is required in contracts (RQ1). For evaluation, we specify existing feature modules, decompose existing programs with contracts, and develop feature modules with contracts from scratch.

5. FEATURE-MODULE VERIFICATION

Once we annotated feature modules with feature-oriented contracts, we can analyze whether these feature modules are correct with respect to their contracts. In principle, we can use any verification technique available for contracts such as runtime assertion checking, test-case generation, static analysis, model checking, and theorem proving [8, 4]. 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 discuss which verification strategies are applicable to which verification technique (RQ2).

Product-Based Runtime Assertion Checking. With special compilers (e.g., JMLc), contracts can be translated into runtime assertions [16, 8, 4]. We extended FeatureHouse [3] and FeatureIDE [21] to check the syntax of feature-oriented contracts (as shown in Figure 1), and automatically compose these contracts when generating products (as shown in Figure 2). As the result of composition is a Java program with JML annotations, we can use any JML compiler for product-based runtime assertion checking [20]. In contrast, feature-based runtime assertion checking for feature modules is hardly possible, because feature modules are fragments of a program, which cannot be compiled and tested in isolation. However, a possible optimization for product-based runtime assertion checking is to choose a subset of all products that is likely to detect many errors [19]. The advantage of runtime assertion checking is that contracts may be checked only when debugging the program, but do not cause runtime overhead in a release version compiled with a standard Java compiler.

Product-Based Static Analysis. The generated Java programs with JML specifications can also be used as input for static analysis tools. We pursued product-based static analysis using ESC/Java2 for the detection of feature interactions [18]. We were able to detect all feature interactions in a small product line of list implementations. However, 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. Hence, the tool can neither be used to prove the absence nor presence of errors, but it is still valuable for bug finding.

Feature-Product-Based Theorem Proving. The absence of errors can be proved with verification tools translating the program and its specifications into the input language of a theorem prover. We used the verification framework Why/Krakatoa for feature-product-based theorem proving [23]. The framework supports automated theorem provers such as Simplify and interactive theorem provers such as Coq. We used Coq, in which the user needs to write textual commands (i.e., a proof script) that apply certain proof steps until the proof is finished. We wrote feature-oriented proof scripts for each feature, and composed them together with feature modules and contracts. The composed proof scripts are checked for every product, but the user only needs to write proof scripts once per feature. This approach reduces the effort to write proof scripts by 88 % [23].

Family-Based Theorem Proving. All approaches discussed above rely on the generation of products, which is infeasible for the verification of large product lines. We proposed family-based theorem proving avoiding the generation of all products [22]. By translating compile-time into runtime variability, we generate a metaproduct simulating all products and a metaspecification equivalent to all product specifications. We can use the theorem prover KeY as-is, because the metaproduct is a standard Java program with JML specifications. We measured that automatic verification of the metaproduct saved 85 % of the calculation time compared to product-based theorem proving [22].

6. CONCLUSIONS AND FUTURE WORK

With feature-oriented programming, we can modularize large-scale refinements into feature modules. Feature modules can be composed automatically to generate products of a software product line. We argue that such a high-level reuse is also necessary for specifications, and apply design by contract to feature modules. We formally specify feature modules with feature-oriented contracts, and use them for testing by means of runtime assertion checking and verification by means of static analysis and theorem proving.

In ongoing and future work, we formalize different approaches to define and compose feature-oriented contracts, and evaluate these approaches with further case studies to asses their reuse capabilities. In addition, we plan to compare product-line approaches for runtime assertion checking, model checking, static analysis, and theorem proving regarding their efficiency and effectiveness. We want to verify product lines from scratch and apply techniques known from mutation testing to introduce bugs into already verified product lines. The comparison of approaches is crucial for product-line developers, as they need to choose an approach from a pool of available approaches.

7. ACKNOWLEDGMENTS

I thank my supervisor Gunter Saake for his support, and gratefully acknowledge the co-authors of previous publications, especially Sven Apel, Christian Kästner, Ina Schaefer, Fabian Benduhn, Jens Meinicke, and Martin Hentschel.

8. REFERENCES

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

[2] S. Apel and D. Hutchins. A Calculus for Uniform Feature Composition. *TOPLAS*, 32:19:1–19:33, 2010.

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

[4] 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.

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

[6] C. Brabrand, M. Ribeiro, T. Tolêdo, and P. Borba. Intraprocedural Dataflow Analysis for Software Product Lines. In *AOSD*, pages 13–24. ACM, 2012.

[7] D. Bruns, V. Klebanov, and I. Schaefer. Verification of Software Product Lines with Delta-Oriented Slicing. In *FoVeOOS*, volume 6528 of *LNCS*, pages 61–75. Springer, 2011.

[8] 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.

[9] K. Fisler and S. Krishnamurthi. Modular Verification of Collaboration-based Software Designs. In *ESECFSE*, pages 152–163. ACM, 2001.

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

[11] 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.

[12] C. Kästner, S. Apel, and D. Batory. A Case Study Implementing Features Using AspectJ. In *SPLC*, pages 223–232. IEEE, 2007.

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

[14] T. Kishi and N. Noda. Formal Verification and Software Product Lines. *Comm. ACM*, 49:73–77, 2006.

[15] H. Li, S. Krishnamurthi, and K. Fisler. Verifying Cross-Cutting Features as Open Systems. *Softw. Eng. Notes*, 27(6):89–98, 2002.

[16] B. Meyer. Applying Design by Contract. *Computer*, 25(10):40–51, 1992.

[17] M. Poppleton. Towards Feature-Oriented Specification and Development with Event-B. In *REFSQ*, volume 4542 of *LNCS*, pages 367–381. Springer, 2007.

[18] 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.

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

[20] T. Thüm, S. Apel, A. Zelend, R. Schröter, and B. Möller. Subclack: Feature-Oriented Programming with Behavioral Feature Interfaces. In *MASPEGHI*, 2013. To appear.

[21] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *SCP*, 2013. To appear.

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

[23] 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.

[24] T. Thüm, I. Schaefer, M. Kuhlemann, S. Apel, and G. Saake. Applying Design by Contract to Feature-Oriented Programming. In *FASE*, volume 7212 of *LNCS*, pages 255–269. Springer, 2012.