Verification of Software Product Lines with Delta-oriented Slicing

Daniel Bruns[1], Vladimir Klebanov[1], and Ina Schaefer[2]⋆

[1] Karlsruhe Institute of Technology, 76128 Karlsruhe, Germany
`{bruns, klebanov}@kit.edu`
[2] Chalmers University of Technology, 421 96 Gothenburg, Sweden
`schaefer@chalmers.se`

Abstract. Software product line (SPL) engineering is a well-known approach to develop industry-size adaptable software systems. SPL are often used in domains where high-quality software is desirable; the overwhelming product diversity, however, remains a challenge for assuring correctness. In this paper, we present delta-oriented slicing, an approach to reduce the deductive verification effort across an SPL where individual products are Java programs and their relations are described by deltas. On the specification side, we extend the delta language to deal with formal specifications. On the verification side, we combine proof slicing and similarity-guided proof reuse to ease the verification process.

1 Introduction

A software product line (SPL) [18] is a set of software systems (called *products*) with well-defined commonalities and variabilities. SPL are often used in domains (e.g., communications, medical, transportation) where high-quality software is desirable; the overwhelming product diversity, however, remains a challenge for assuring correctness by any method.

Even without formal verification, the dimensions and complexity of product lines make it essential to model the relationships between products explicitly. One of the authors has been working on the software engineering aspects of SPL [22, 23, 21]. This has resulted in a modeling approach called delta-oriented programming (Sect. 2). Our current effort aims to exploit the structural information available in an SPL model to reuse verification results obtained from verifying one product when considering another product. Where necessary, we enrich the model with semantical information (such as formal specifications, Sect. 3). Considering other possibilities to verify SPL that are more meta-level (like generic or partial proofs) and require more semantical information, we decided to go on with a more light-weight approach first.

The technology that we are using to illustrate our approach is Java for programming single products, JML [13] for formal specifications and the KeY sys-

⋆ This author has been supported by Deutsche Forschungsgemeinschaft (DFG) and by the EU project FP7-ICT-2007-3 HATS.

tem [4] for deductive verification. However, we only make the following assumptions about the verification system:

- We concentrate on systems that manipulate an explicit proof object in the proof assistant style, but do discuss systems operating in the verifying compiler style (a verification condition-generating tool chain with an SMT solver at its end).
- We support both ways in which verification systems can treat method calls: using the method contract or inlining the implementation. Using the contract is inherently modular while inlining is not, but it still has its advantages. It is simple, does not force the developer to write "trivial" contracts for helper methods, and reduces the number of commitments that need to be updated as the code evolves.
- Our method is also parametric on how a verification system treats invariants. In the worst case, all methods in the program have to be verified to preserve every invariant, as the invariant vocabulary is (in general) unrestricted. In practice, verification systems use criteria such as visibility, syntax and typing, assignable clauses or ownership to reduce the workload. We simply limit ourselves to requiring that all *relevant* invariants must be checked.

In our approach we analyze the SPL model to determine which parts of the original product are unchanged in the new product and also do not have to be verified again. This analysis constitutes proof slicing (Sect. 4).

For the modified or otherwise affected product parts, we apply a previously-developed proof reuse technique based on the assumption of similarity between the two implementation variants. (Sect. 5).

We present related work in Sect. 6 and draw conclusions in Sect. 7.

2 Delta-oriented Programming of Software Product Lines

Delta-oriented programming (DOP) [22, 23, 21] is a novel approach for implementing software product lines. Delta-oriented programming offers an expressive and flexible "programming meta-language" for specifying a set of products. Its aim is to relax the restrictions of currently established SPL description formalisms such as feature-oriented programming (FOP) [3] by adding the explicit possibility to remove parts of a program. For a more detailed comparison between delta-oriented and feature-oriented programming, the reader is referred to [22].

In delta-oriented programming, an SPL is implemented as a *core module* together with a set of *delta modules*. The core module contains a complete product implementation for some valid feature configuration, which can be developed by conventional single-application engineering techniques. Delta modules specify changes to be applied to the core module in order to implement other products.

The notation we use for Java programs constituting individual products is the following:

Definition 1. *A* program *is a set of class declarations (further called classes) and a binary inheritance relation on this set. We are primarily interested in the transitive closure of this relation* $\sqsubset$ *and the transitive reflexive closure* $\sqsubseteq$. $A \sqsubset B$ *means that the class* A *is below class* B *in the inheritance hierarchy. Abstract classes and interfaces are omitted in this paper for brevity.*

A class *is a set of field and method declarations (which are built up of names, types, parameters, bodies, etc., as appropriate in Java). If* C *is a class declaring a method with signature* m*, then we will refer to this particular implementation as* $C{::}m$.[3] *Vice versa, we identify the method signature* m *with a set of classes in a product that declare a method with that signature:* $C \in m$ *if* $C{::}m \in C$.

```
core Base {
    class Account extends Object {
        int balance;
        int bonus;
        void addBonus(int x){}
        void update(int x) {
            balance += x;
        }
    }
}
```

(a) Core module with `Account` class

```
delta DInvestment when Investment {
    modifies class Account {
        removes void addBonus(int x);
        adds void addBonus(int x) {
            bonus += x;
        }
        removes void update(int x);
        adds void update(int x){
            balance += x;
            if (x > 0) addBonus(x/2);
        }
    }
}
```

(b) Delta module for feature `Investment`

```
class Account extends Object {
    int balance;
    int bonus;
    void addBonus(int x){
        bonus += x;
    }
    void update(int x) {
        balance += x;
        if (x > 0) addBonus(x/2);
    }
}
```

(c) Result of delta module application

Fig. 1: Example of a delta-oriented product line.

Modification operations used in delta modules that we consider in this paper are the following:

– adding/removing a class declaration C: *adds*(C), *removes*(C)
– modifying class C by

[3] For simplicity, we assume the absence of method overloading. In Java, a class may contain several method implementations with the same identifier and compatible parameter types. This renders the lookup procedure far more complicated; c.f. [8, Sect. 15.12.2].

- adding/removing a field f: $\mathit{adds}(C::f), \mathit{removes}(C::f)$
- adding/removing a method declaration m: $\mathit{adds}(C::m), \mathit{removes}(C::m)$
- changing the direct superclass of C to C': $\mathit{reparents}(C, C')$

On an abstract level, the variability of an SPL is defined by the feature set F. Valid member products of an SPL are given by the feature model $\mathcal{F} \subseteq 2^F$. Each product uniquely corresponds to a combination of features, also called *feature configuration.* In the following, we identify products and feature configurations in $\mathcal{F}$. Each delta module d contains an *application condition* φ_d (the **when** clause in concrete syntax), which is a propositional formula over the feature set F. The application conditions specify which delta modules are necessary for which features. For every pair of valid products $P_1, P_2 \in \mathcal{F}$, $\Delta(P_1, P_2)$ is the set of delta modules that have to be applied to the product P_1 in order to obtain a product P_2 with a different feature configuration.[4] The original delta language proposal [22] demands a partial order on deltas to guarantee that the result of applying $\Delta(P_1, P_2)$ is unique, as well as certain other syntactical well-formedness conditions, which we are not concerned with in this paper.

Example 1. Our running example in this paper is a delta-oriented product line of bank accounts inspired by [7]. Figure 1a shows the core module of this SPL with the basic Account class. Figure 1b shows the delta module `DInvestment` for activating the `Investment` feature, which accumulates a bonus for each deposit made. Figure 1c contains the result of applying the delta module to the core, which is, again, a conventional Java class. Later on, in Example 3, we will also see the `Paycheck` feature adding the class `Employer` as a client of `Account`. $\Diamond$

3 Delta-oriented Formal Specification of Software Product Lines

We use the Java Modeling Language (JML) [13] for the formal specification of product properties. In this work, we concentrate on class invariants and method contracts with pre- and post-conditions. As JML specifications are written directly into Java source files as comments, it is possible to include them in the delta language introduced in Sect. 2. A core module is specified just as a conventional program. An example of a core module with JML specifications can be seen in the first listing of Example 3.

For delta modules, we extend the delta language with the following operations to manipulate specifications:

- adding an invariant to a class: $\mathit{adds}(C, I)$
- removing an invariant from a class: $\mathit{removes}(C, I)$
- adding a contract (pre-/post-condition pair) to a method: $\mathit{adds}(C::m, ct)$
- removing a contract from a method: $\mathit{removes}(C::m, ct)$

[4] This is a slight generalization of the original delta approach, where deltas could only be applied to the core product.

Note that we only consider pairs of exactly one pre- and post-condition to be added or removed together. In case one of them is trivial (i.e., `true`), it is omitted.

Example 2. Figure 2 shows the delta module `DInvestmentSpec` changing the specifications in class `Account`. It is applied for the same configurations as the code delta `DInvestment`, since it has the same application condition.[5] ◊

In general, there is no concordance between code deltas and specification deltas for one product. It is perfectly conceivable to change the code without changing the specification or the other way round. However, there are (at least) the following exceptions where code changes influence the specification:

– Removing a class or a method induces the removal of attached specifications.
– JML enforces behavioral subtyping, i.e., subclasses inherit the specifications of the superclass. Changing the inheritance hierarchy, thus, also changes the specification.
– JML by default enforces non-nullness of fields, variables, etc. Adding a field of reference type to a class automatically creates an implicit invariant about this field.
– Changing a (pure) method changes the semantics of specifications using this method.

```
delta DInvestmentSpec when Investment {
    modifies class Account {
        removes //@ ensures bonus == \old(bonus);
          from void addBonus(int x);
        adds //@ requires x >= 0;
            //@ ensures bonus == \old(bonus) + x;
          to void addBonus(int x);
}
```

Fig. 2: A specification delta adds and removes pre- and post-conditions from a method.

4 Delta-oriented Slicing

When a new product is derived by delta application, in general, both the implementation as well as the specification change. However, from the structural information available in the used delta modules, we are able to conservatively infer which specifications of the new product remain valid (i.e., the proofs done for the old product are not affected by the change) and which parts have to

[5] It is possible to specify code and specification changes in the same delta module. The separation at this point is for presentation reasons.

be (re-)proven in order to establish the specified properties. We call the latter *delta-oriented slice*. Slicing originated as a program analysis technique answering the question of which program statements influence the value of a given variable. Our algorithm answers the question of which proofs are influenced by a delta module.

Of course, the simplest and safest way to achieve assurance for a changed product is to redo all proofs. However, at the current state of hardware and deduction technology, this approach is too slow for any product of non-trivial size. Our approach is much less computationally expensive as it only involves a deterministic static analysis of different artifacts. This way, proof slicing can quickly provide feedback to the engineer on what impact a certain change to the product will have.

Proof Modularity

The key to obtaining a sound slicing algorithm is identifying non-modular proof steps. The issue of proof non-modularity arises if the validity of certain proof steps in a verification proof is lost when the program that is to be verified is changed or extended.

The change may be explicit, i.e., concerning the source code of the very method being verified, or implicit, i.e., concerning program entities that are only referenced (e.g., other methods called). Explicit changes are easy to detect, and if they are benign, they can be treated by proof reuse (Sect. 5). Implicit changes are more involved, and their impact depends both on the semantics of the programming language, as well as on the particular verification calculus. Implicit change is the case that we concentrate on in the following.

Proof modularity has been recognized as an issue for quite some time, focusing, naturally, on adding/removing classes and overriding methods. Particularly relevant to our effort are a previous account for the KeY system [20, Sect. 6.2] as well as a comprehensive survey for the KIV system [24, Chap. 6]. As our change vocabulary is larger, we have to address this issue anew. In the KeY system, identifying rules resulting in non-modular proof steps is made easier by the fact that the class declarations and the class hierarchy are not part of the original proof obligation. This information is available in the background (i.e., in the prover implementation) and can be introduced into a logical sequent by rules containing *metaconstructs* (functions that are not logically specified, but programmed in the prover). These functions make non-modular rules easily identifiable syntactically, which we have done for the KeY rule base. In the KeY calculus, we discern rules giving rise to proof steps whose validity is:

(A) not affected by implicit program changes (rewriting, propositional, and the like, but also many symbolic execution rules, e.g., for conditionals, loops, etc.);
(B) affected by presence or absence of classes regardless of their content;[6]

[6] The rules of this type are rare and the KeY system has only two of them: TYPEABSTRACT and ARRAYSTORESTATICANALYSE. The former allows deducing the dynamic

(C) affected by methods declared in classes; these rules are the non-modular method invocation rules inlining the method implementations and simulating dynamic binding;
(D) affected by fields declared in classes regardless whether these fields are used in the program; these are the instance creation rules assigning default values to fields;
(E) affected by inheritance relationship between classes; these are the rules for tackling the inheritance predicate $\sqsubseteq$.

The slicing algorithm is based on these findings.

Other systems encode the class hierarchy as axioms that are part of the proof obligation from the start. Here, it is necessary to analyze the proofs constructed by the prover for occurrence of particular axioms. This may be difficult if there is no explicit proof object, but, for instance, the popular SMT prover Z3 often used in verifying compilers provides this information.

The Algorithm

In the following, we present the delta-oriented slicing algorithm. As the first step of the algorithm, we copy all finished proofs from product P_1 into product P_2 regardless of their validity for P_2. In the resulting set of proofs for the new product, our algorithm identifies the proofs that do not hold in the new context and marks them as invalid. These proofs have to be redone. The algorithm also identifies new proof obligations that have to be discharged in order to obtain a full set of proofs for the specifications of P_2.

Input: A set of proofs for a product P_1, and the delta $\Delta(P_1, P_2)$[7]
Output: A set of valid proofs for the product $P_2 = P_1 + \Delta(P_1, P_2)$

1. Copy all proofs from P_1 to P_2 (regardless of validity). Weed out all proofs where the vocabulary involved (code or specification) is no longer present.

The following steps refer to the content of the delta module $\Delta(P_1, P_2)$. The algorithm currently considers only the structural change information available in the delta and does not take the content of the modified methods or specifications into account.

2. For each $adds(C)$:
   (a) do $adds(C{::}f)$ for each $f \in C$
   (b) do $adds(C{::}m)$ for each $m \in C$
   (c) invalidate all proofs with proof steps by non-modular rules of type (B) where C or any of its superclasses appear in the rule conclusion

type of an object pointed to by an expression with an abstract static type (this rule produces a disjunction over all subclasses). The latter uses a simple static analysis to check whether an array assignment can throw an `ArrayStoreException`.

[7] For the sake of the algorithm, we assume that $\Delta(P_1, P_2)$ contains exactly one delta module (i.e., we assume delta module composition).

3. For each *removes*(C):
   (a) do *removes*(C::f) for each $f \in C$
   (b) do *removes*(C::m) for each $m \in C$
   (c) invalidate all proofs with proof steps by non-modular rules of type (B) where C or any of its superclasses appear in the rule conclusion

Adding and removing methods. When adding methods, we have to distinguish if their invocation is treated by inlining and contract application. If an altered implementation is inlined, the proof, of course, will be invalidated. For a contract, this is different since the altered implementation is expected to fulfill the old contract. Contracts are also not affected by method removal. Even though an implementation has been removed, the contract still applies to some overriding implementation in a subclass.

4. For each *adds*(C::m):
   (a) invalidate all pre-existing proofs where m was inlined and C::m would have been among potentially referenced implementations (see Fig. 3)
   (b) proofs using the contracts for m remain valid
   (c) prove that C::m satisfies all specifications of C (either stated directly or inherited), as well as all other invariants
5. For each *removes*(C::m):
   (a) invalidate all pre-existing proofs where m was inlined and C::m would have been among potentially referenced implementations (Fig. 3)
   (b) proofs using the contracts for m remain valid

Adding and removing fields. In steps 6–7, it might not be immediately clear why adding or removing a field can invalidate a proof. Consider the following code snippet:

```
class A { Object f; }
class B extends A {  /*@ invariant f == ((A)this).f; @*/ }
```

The invariant in class `B` holds if and only if no field `f` is added to class `B`. Otherwise, the left occurrence of `f` would refer to `B::f`, while the right one would continue referring to `A::f` as fields are bound statically in Java.

Adding or removing fields also invalidates proofs containing instance creation, as this process must assign all fields a default value, resulting in varying intermediate states.

6. For each *adds*(C::f):
   (a) find the set of method implementations M referring to C::f in P_2
   (b) invalidate all pre-existing proofs about any C'::$m \in M$
   (c) invalidate all pre-existing proofs inlining any C'::$m \in M$
   (d) invalidate all pre-existing proofs of specifications referring to C::f in P_2
   (e) invalidate all pre-existing proofs with proof steps assigning default values (during instance creation) to fields of an object with type $A \sqsubseteq C$
7. For each *removes*(C::f): same as step 6, but look for C::f in P_1

Class reparenting. Reparenting is an invasive operation, which is illustrated in Fig. 4. *reparents*(C, C') moves C from under its old direct supertype $\widetilde{C}$ and beneath C', and with it *movedPart* $= \{K \mid K \sqsubseteq C\}$. As $\widehat{C}$ we then denote the least common supertype of $\widetilde{C}$ and C'.

Reparenting class C makes C and its subclasses lose features (implementations and specifications) inherited from *oldBranch* $= \{K \mid \widetilde{C} \sqsubseteq K \sqsubset \widehat{C}\}$ and inherit new features from *newBranch* $= \{K \mid C' \sqsubseteq K \sqsubset \widehat{C}\}$.

8. For each *reparents*(C, C'):
   (a) invalidate all pre-existing proofs inlining method bodies for any virtual method call e.m`()` with S as the static type of e and
      i. $S \in$ *newBranch*
      ii. $\widehat{C} \sqsubseteq S$
   or, if at least one method body K::m was inlined such that
      iii. $S \in$ *movedPart* and $K \in$ *oldBranch*
      iv. $S \in$ *oldBranch* and $K \in$ *movedPart*
   This step reacts to changes in the big case distinction simulating dynamic binding.
   (b) invalidate all pre-existing proofs about/inlining any method implementation C::m containing a method call of the form `super.`m'`()` (as the superclass will change)
   (c) invalidate all pre-existing proofs about/inlining any method implementation K::m, $K \in$ *movedPart* that references a field K'::f declared in *oldPart* (as this reference would change its meaning after the move)
   (d) contracts for methods in reparented classes remain valid unless the contract no longer exists (i.e., it was inherited from *oldBranch*)
   (e) invalidate proofs for specifications inherited from any class in *oldBranch*
   (f) prove that all classes $K \in$ *movedPart* satisfy the specifications inherited from new superclasses in *newBranch*
   (g) invalidate all proofs containing a proof step deciding the predicate $A \sqsubseteq B$ if $A \sqsubseteq C$ and $B \in$ *oldBranch*

Adding and removing specifications.

9. For each *adds*$(C$::m, *ct*$)$
   (a) prove that the contract *ct* is fulfilled by all C'::m with $C' \sqsubseteq C$
10. For each *removes*$(C$::m, *ct*$)$
   (a) invalidate all pre-existing proofs that use the contract *ct*
11. For each *adds*(C, I)
   (a) prove that the invariant I is fulfilled by all relevant implementations
12. For each *removes*(C, I)
   (a) invalidate all pre-existing proofs that assume the invariant I

For some of the algorithm steps, we need to determine whether an implementation $C::m$ is potentially referenced by the method invocation expression `e.m()`.
We consider the three different method invocation modes available in Java, defining for each mode a starting point class S of method lookup. The relation of S and C determines the answer:

Instance or "virtual" mode. This is the most common mode. The target expression `e` (of type S) references an object (it may be an implicit `this` reference), and the method is not declared static or private. This invocation mode requires dynamic binding.

- The implementation is in S or one of its subclasses: If $C \sqsubseteq S$, then "yes"
- The implementation is in a superclass of S, but it is inherited by S or one of its subclasses (i.e., it is not overridden between C and S): If $S \sqsubseteq C$ such that for all K with $S \sqsubseteq K \sqsubset C$ holds $K \notin m$, then "yes" (cf. Fig. 5).
- Otherwise, "no".

Static mode (m is declared static or private). In this case, no dynamic binding is performed. The implementation to invoke is determined in accordance with the declared static type S of `e`. If $C = S$ then "yes", otherwise "no".

Super mode (`e` is the keyword `super`). This mode is used to access the methods of the immediate superclass S (of the class containing the invocation expression `super.m()`).

- If $S \sqsubseteq C$ and for all K with $S \sqsubseteq K \sqsubset C$ holds $K \notin m$, then "yes".
- Otherwise, "no".

Fig. 3: Subroutine: When is a method implementation potentially referenced?

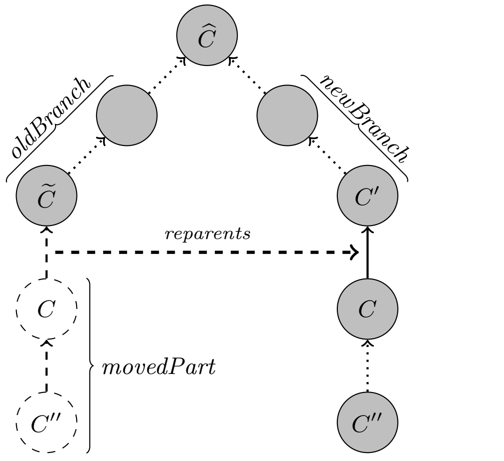

Fig. 4: Illustration of *reparents*(C, C'). Solid lines represent the direct subtype relation, dotted lines its transitive closure, and dashed lines show relations of the previous product.

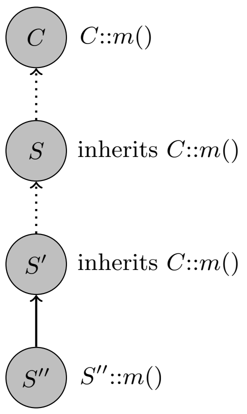

Fig. 5: Virtual method invocation mode and method overriding.

An Example

Example 3. (i) We return to the bank account example introduced in Sect. 2. The core product with the basic `Account` class now contains specifications (see below). It can easily be proven that both methods satisfy their contracts and the class invariant.

```
core Base {
    class Account extends Object {
        //@ invariant bonus >= 0;
        int balance;
        int bonus;

        //@ ensures bonus == \old(bonus);
        void addBonus(int x){}

        /*@ ensures balance == \old(balance) + x;
          @         && bonus >= \old(bonus); @*/
        void update(int x) {
            balance += x;
        }
    }
}
```

(ii) Next, we apply the delta module shown below in order to generate a new product with the additional feature `Paycheck`. This module adds an `Employer` class with a reference to the account and a `payday()` method with a corresponding specification. In order to determine which proofs for the basic bank account are still valid, we use the delta-oriented slicing algorithm. We perform step 2 for the added class, leading to step 4 for the added method, step 6 for the added field and step 9 for the added contract. Only step 4c is non-trivial, since the method `payday()` did not exist before. The method can be verified easily – either by inlining the implementation of `addBonus()` and `update()` or by applying their contracts. There is no existing proof to reuse. Step 6 is trivial (the set M is empty) as the field `a` did not exist previously. Step 9 is subsumed by step 4 as `Employer` has no subclasses. No proofs are invalidated.

```
delta DPaycheck when Paycheck {
    adds class Employer extends Object {
        Account a;

        /*@ requires x >= 0 && bonus >= 0;
          @ ensures a.balance == \old(a.balance) + x
          @             && a.bonus >= \old(a.bonus);
          @*/
        void payday(int x, int bonus) {
            a.addBonus(bonus);
            a.update(x);
        }
    }
}
```

(iii) If we now want to incorporate the `Investment` feature as well, we apply the deltas `DInvestment` (Fig. 1b) and `DInvestmentSpec` (Fig. 2) to the latest product. These two deltas modify the implementation and specification of the method `addBonus()` and the implementation of the method `update()` in the class `Account`. The slicing steps to take to determine which proofs from the previous product are still valid are: step 4 for the added methods, step 5 for the removed methods, step 9 for the added contract and step 10 for the removed contract.

Steps 4c and 9 dictate that both `update()` and `addBonus()` have to be reproven for conformance with the class invariant and their respective (modified) contracts. Proof reuse is feasible here (see Sect. 5). In contrast, `payday()` has not changed (neither code nor specification), but the proof that it satisfies its contract is now invalid. The proof has been invalidated by step 4a or 10, since it (the proof) depends on either the implementation or the contract of `addBonus()`. The proof reuse mechanism may be applied here to find a new proof efficiently. The contract of `update()` has not changed, and all proofs using it remain valid (step 4b). $\Diamond$

5 Proof Reuse for Changed Methods

In this section, we point to the existing technique of proof reuse [11] as a natural complement to delta-oriented proof slicing. This part of our approach is tailored to interactive verification systems like KeY, where the user provides hints to the prover by manipulating an explicit proof object. In practice (although not in our illustrating example), proofs contain proof steps which cannot be (efficiently) found automatically. Users have to instantiate quantifiers, provide lemmas, loop invariants, and guide proof search in other ways. These efforts can be recycled through proof reuse.

The proof reuse technique has been originally developed for KeY by one of the authors to save verification effort during incremental development (i.e., after fixing a bug). Since then, the method has been applied to a number of different change management scenarios. It uses a similarity measure that determines which proof steps from proofs for the original product can be used to establish the proof obligations for the new product. It is a light-weight technique based on proof replay rather than on proof generation. For a full account of proof reuse in KeY we refer the reader to [11].

In the delta-oriented slicing step, we have identified which proofs have to be redone for the newly generated product. However, some of the changed method bodies may still have considerable similarities to the ones in the already verified product. The correctness proofs of such modified methods are likely to resemble the old proofs. Here proof reuse can help. Reuse can also be used in case of changed specifications but much less effectively. Specifications are less structured than programs, and proof shapes adhere to implementations rather than specifications, which makes finding reusable subproofs much harder.

6 Related Work

Formal methods are used in the context of software product lines for a variety of applications. A large body of work is concerned with the formal analysis of feature models [1] or product models [14]. Further approaches (e.g., [6]) verify that the variability specified by a feature model is correctly implemented in code. Efficient verification of product behavior, however, is not well established. In testing [15, 17] or model checking [12, 5] there is work to make validation of product lines more efficient, though.

In [2], a case study for the product line development of a compiler is considered. The compiler is developed by stepwise refinement or extension of the compiler functionality. The correctness proof of the compiler is extended and refined in line with the functional extensions by introduction or adaptation of invariants and the addition of case distinctions. This approach relies on a fixed structure of the induction proof for compiler correctness that allows determining in advance which modifications of the proof are required by functional changes.

Reuse of verification artifacts is also related to a whole plethora of work which is impossible to survey here, such as slicing for debugging [25, 27] or model checking [9], reuse of refined specifications [26], change management in theory development [16, 10], incremental compilation, refactoring, and software change impact analysis.

An interesting and closely related result from change impact analysis is the tool Chianti [19], which determines whether the results of a test are affected by changes to the source code. Changes to the program are decomposed into "atomic operations", which are similar to our delta operations. These are then analyzed for their impact on the program's call graph.

Of course, deriving a new product in a product line is also closely related to evolving a single product. Most verification systems implement some kind of proof management for this case. Alas, system developers apparently–and unjustifiedly, we think–tend to consider this important component an implementation detail, as published accounts on this subject are rare.

7 Conclusions

Working on verification of SPL, we have identified several interesting lines of future research. Most of them regard the transition from a syntactic modeling of SPL as in the current delta-oriented programming approach [22] to a more semantic-based modeling of SPL.

In order to define delta operations on specifications in a meaningful way, it is necessary to uniquely identify class invariants and method contracts (e.g., for removal or modification). This could be handled by introducing labels (as most tools probably already do internally).

So far the operations we have defined for specification deltas are rather basic. One reason for this is simplicity. Another reason is that at least with the current calculi, the shape of a proof follows rather closely the shape of the program, but it is much less related to the shape of a specification. It remains to be seen whether adding more fine-grained change information in the specification deltas helps obtaining new proofs more efficiently. Additional operators that appear promising to us are case distinctions and redundant specifications (lemmas).

Until now, the delta module operations (for code) and their applicability conditions are mostly syntactical. Greater power and precision can be achieved by adding more semantical information. For instance, such a description might dictate that a certain feature is only compatible with another if the base product preserves certain data invariants. New tools could be devised to assist in deriving consistent products with desired behavior based on semantical information.

Finally, getting the formal specification of a product right is difficult, but deriving a correct product from another also has its pitfalls. Even if two products P_1 and P_2 fulfill the specification I (as ensured by our approach), it is still only *syntactically* the same specification I. The product derivation process may seduce one to believe that I is still an adequate specification for the new product, which might not be the case. In the simplest case I might contain pure methods, which have changed between products. The issue is aggravated by the complicated and sometimes unclear semantics of modern specification languages and requires further investigation.

References

1. Don S. Batory, David Benavides, and Antonio Ruiz-Cortés. Automated analysis of feature models: Challenges ahead. *Communications of the ACM*, 49(12), 2006.
2. Don S. Batory and Egon Börger. Modularizing theorems for software product lines: The Jbook case study. *Journal of Universal Computer Science*, 14(12), 2008.
3. Don S. Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. *IEEE Trans. Software Eng.*, 30(6):355–371, 2004.
4. Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *Lecture Notes in Computer Science*. Springer-Verlag, 2007.
5. Andreas Classen, Patrick Heymans, Pierre-Yves Schobbens, Axel Legay, and Jean-Franois Raskin. Model checking lots of systems: Efficient verification of temporal properties in software product lines (to appear). In *32nd International Conference on Software Engineering, ICSE 2010, May 2-8, 2010, Cape Town, South Africa, Proceedings*. IEEE, 2010.
6. Krzysztof Czarnecki and Krzysztof Pietroszek. Verifying feature-based model templates against well-formedness OCL constraints. In *Conf. on Generative Programming and Component Engineering (GPCE)*, 2006.
7. Benjamin Delaware, William Cook, and Don Batory. A Machine-Checked Model of Safe Composition. In *Foundations of Aspect-Oriented Languages (FOAL)*, pages 31–35. ACM, 2009.
8. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley Longman, Amsterdam, 3rd edition, 2005.
9. John Hatcliff, Matthew B. Dwyer, and Hongjun Zheng. Slicing software for model construction. *Higher-Order and Symbolic Computation*, 13(4):315–353, 2000.
10. Dieter Hutter. Management of change in structured verification. In *Automated Software Engineering (ASE)*, page 23, 2000.

11. Vladimir Klebanov. Proof reuse. In Beckert et al. [4].
12. Kim Lauenroth, Klaus Pohl, and Simon Toehning. Model checking of domain artifacts in product line engineering. In *Automated Software Engineering (ASE)*, pages 269–280. IEEE Computer Society, 2009.
13. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.
14. Mike Mannion. Using first-order logic for product line model validation. In Garry Chastek, editor, *Software Product Lines: Proceedings of the Second Software Product Line Conference (SPLC2)*, LNCS 2379, pages 176–187, San Diego, CA, August 2002. Springer.
15. John D. McGregor. Testing a software product line. Technical Report CMU/SEI-2001-TR-022, Software Engineering Institute, Carnegie Mellon University, December 2001.
16. Till Mossakowski. Heterogeneous theories and the heterogeneous tool set. In Yannis Kalfoglou, W. Marco Schorlemmer, Amit P. Sheth, Steffen Staab, and Michael Uschold, editors, *Semantic Interoperability and Integration*, volume 04391 of *Dagstuhl Seminar Proceedings*. IBFI, Schloss Dagstuhl, Germany, 2005.
17. Henry Muccini and André van der Hoek. Towards testing product line architectures. *Electr. Notes Theor. Comput. Sci*, 82(6), 2003.
18. Klaus Pohl, Günther Böckle, and Frank van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, Heidelberg, 2005.
19. Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. Chianti: A tool for change impact analysis of Java programs. In John M. Vlissides and Douglas C. Schmidt, editors, *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28, 2004, Vancouver, BC, Canada*, pages 432–448. ACM, 2004.
20. Andreas Roth. *Specification and Verification of Object-oriented Software Components*. PhD thesis, Universität Karlsruhe, 2006.
21. Ina Schaefer. Variability modelling for model-driven development of software product lines. In *4th Int. Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, Linz, Austria, January 2010.
22. Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. In *Proceedings, 14th International Software Product Line Conference*, Lecture Notes in Computer Science, Jeju, South Korea, September 13–17 2010. Springer. To appear.
23. Ina Schaefer, Alexander Worret, and Arndt Poetzsch-Heffter. A model-based framework for automated product derivation. In *Model-driven Approaches in Software Product Line Engineering (MAPLE 2009)*, 2009.
24. Kurt Stenzel. *Verification of Java Card Programs*. PhD thesis, Fakultät für angewandte Informatik, University of Augsburg, 2005.
25. Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3), 1995.
26. Heike Wehrheim. Slicing techniques for verification re-use. *Theor. Comput. Sci*, 343(3):509–528, 2005.
27. Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, August 1984.