A Transformational Proof System for Delta-Oriented Programming [*]

Ferruccio Damiani
Università di Torino, Italy
ferruccio.damiani@unito.it

Johan Dovland
University of Oslo, Norway
johand@ifi.uio.no

Einar Broch Johnsen
University of Oslo, Norway
einarj@ifi.uio.no

Olaf Owe
University of Oslo, Norway
olaf@ifi.uio.no

Ina Schaefer
TU Braunschweig, Germany
i.schaefer@tu-bs.de

Ingrid Chieh Yu
University of Oslo, Norway
ingridcy@ifi.uio.no

ABSTRACT

Delta-oriented programming is a modular, yet flexible technique to implement software product lines. To efficiently verify the specifications of all possible product variants of a product line, it is usually infeasible to generate all product variants and to verify them individually. To counter this problem, we propose a transformational proof system in which the specifications in a delta module describe changes to previous specifications. Our approach allows each delta module to be verified in isolation, based on symbolic assumptions for calls to methods which may be in other delta modules. When product variants are generated from delta modules, these assumptions are instantiated by the actual guarantees of the methods in the considered product variant and used to derive the specifications of this product variant.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Design, Languages, Theory

Keywords

Program Verification, Proof System, Software Product Line

1. INTRODUCTION

Diversity is prevalent in modern software systems in order to meet different customer requirements and application contexts [20]. A software product line (SPL) realizes this diversity by providing a range of product variants. Formal modeling and verification of SPLs have attracted considerable interest recently [6, 9, 13, 22]. Given the combinatorial explosion in the number of possible product variants, efficient verification techniques for SPLs are essential. For verification techniques to scale, they have to be modular in the artifacts reused to build the different variants of the SPL.

This paper focuses on the verification of behavioral specifications for *delta-oriented programming* (DOP) [18, 19]. A DOP SPL is realized by a set of *delta modules* (*deltas*, for short) encapsulating modifications to OO programs. Deltas may add or remove classes or modify classes by adding or removing methods and fields or by wrapping methods. A particular program variant is obtained by applying a selected set of deltas to the empty program in a given order. For verification to scale, each delta should be verified in isolation and the verification results reused for different product variants in the same way as the delta for their generation. The challenge to realize this approach is that deltas are not self-contained and may comprise calls to methods defined in or modified by other deltas. Previous work [15] imposed very strong restrictions on how methods could be modified by deltas to achieve a compositional verification principle.

This paper proposes a different approach by developing a novel *transformational* proof system for DOP, which relaxes these restrictions but still allows modular verification of deltas. The methods of each delta are verified in isolation using symbolic assumptions on methods that are not contained in the delta itself, e.g., calls to the original version of a method or methods introduced or modified by other deltas. When deriving product variants by delta application, these assumptions are instantiated with the guarantees of the actual methods contained in the generated product variant. Then, the actual products are verified based on the specifications already established for the used deltas. The goal of the verification process at the level of the product variant is to guarantee that the derived specifications are *mutually consistent*; i.e., the derived specifications are strong enough to guarantee the symbolic assumptions after instantiation.

The paper presents the proposed proof system by an example of the verification of an SPL. The example is written in DeltaJ [8, 18, 19], a DOP language based on Java which is currently being developed. Sect. 2 explains DOP and introduces the programming language we use in this paper. Sect. 3 introduces the running example of the paper. Sect. 4

[*]Partly funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models (`http://www.hats-project.eu`), by the German Science Foundation (DFG - SCHA1635/2-1) and by the Italian MIUR (PRIN 2008 DISCO).

$$
\begin{array}{lll@{\qquad}r}
\texttt{CD} & ::= & \textbf{class}\ \texttt{C}\ \{\overline{\texttt{FD}};\ \overline{\texttt{MD}}\} & \text{classes}\\
\texttt{FD} & ::= & \texttt{N f} = \texttt{c} & \text{fields}\\
\texttt{N} & ::= & \texttt{C} \mid \textbf{int} \mid \textbf{boolean} & \text{nominal types}\\
\texttt{MD} & ::= & \texttt{N m}(\overline{\texttt{N x}})\{\overline{\texttt{N y}};\ \overline{\texttt{s}};\textbf{return}\ \texttt{e};\} & \text{methods}\\
\texttt{s} & ::= & \texttt{w} = \texttt{rhs} \mid \textbf{if}\,(\texttt{e})\ \texttt{s}\ \textbf{else}\ \texttt{s} \mid \{\overline{\texttt{s}};\} & \text{statements}\\
\texttt{w} & ::= & \texttt{f} \mid \texttt{y} & \text{assignable variables}\\
\texttt{rhs} & ::= & \texttt{m}(\bar{\texttt{e}}) \mid \texttt{v.m}(\bar{\texttt{e}}) \mid \textbf{new}\ \texttt{C}() \mid \texttt{e} & \text{assignment rhs}\\
 & & \mid \textbf{original}(\bar{\texttt{e}}) & \\
\texttt{v} & ::= & \texttt{f} \mid \texttt{x} \mid \texttt{y} & \text{variables}
\end{array}
$$

Figure 1: Syntax for classes (`C` ∈ class names; `f` ∈ field names; `c` ∈ constants (including null); `m` ∈ method names; `x` ∈ method parameter names; `y` ∈ method local variable names; `e` ∈ side effect free expressions over variables `v`, this and constants).

presents the specification language for deltas and Sect. 5 explains the transformational proof system. Sect. 6 discusses related work and Sect. 7 concludes the paper.

2. A LANGUAGE FOR DOP

For the purposes of this paper, we consider a fragment of DeltaJ [8] which highlights the specific features of DOP from a reasoning perspective. To simplify the presentation the class/method/field removal operations of DeltaJ deltas are not considered. Also many standard language features are omitted; e.g., subclassing is not included to put focus on deltas as the only code reuse mechanism in this paper.

Implementing Single Products. Single products are implemented in a simple subset of Java (see Figure 1). A class definition **class** `C` $\{\overline{\texttt{FD}};\ \overline{\texttt{MD}}\}$ consists of its name `C`, a list of field definitions $\overline{\texttt{FD}}$ and a list of method definitions $\overline{\texttt{MD}}$. (We use the overline notation for possibly empty sequences.) All fields are private and all methods are public. Each class is assumed to have an implicit constructor that initializes each field to the value specified in the definition of the field. Sequences of named elements (field, method or parameter names, field, method or class definitions,...) are assumed to contain no duplicate names.

A program is essentially a class table `CT` which maps class names to class definitions. We assume that `CT` satisfies the following sanity conditions: (*i*) $\texttt{CT}(\texttt{C}) = $ **class** `C` ... for every $\texttt{C} \in dom(\texttt{CT})$; and (*ii*) for every class name `C` (except library classes, including `Object`) appearing anywhere in `CT`, we have $\texttt{C} \in dom(\texttt{CT})$.

A class definition `CD` maps field and method names to field and method definitions. We let `a` range over field and method names and `AD` over field and method definitions.

Deltas and Product Generation. Figure 2 shows the abstract syntax of deltas of the considered DOP language. A delta definition `DD` has a name δ and a list of class operations $\overline{\texttt{CO}}$. A *class operation* `CO` specifies the addition of a new class or the modification of a named class `C` as a sequence of *attribute operations* `AO`, defining modifications of methods and additions of fields and methods. A class-modify operation `CO` maps field and method names to attribute operations. A method-modify operation either replaces the method body or wraps the existing method using the **original** construct. In both cases, the modified method must have the same header as the unmodified method. The call **original**($\bar{\texttt{e}}$),

$$
\begin{array}{lll@{\qquad}r}
\texttt{DD} & ::= & \textbf{delta}\ \delta\ \{\overline{\texttt{CO}}\} & \text{deltas}\\
\texttt{CO} & ::= & \textbf{adds}\ \texttt{CD} \mid \textbf{modifies}\ \texttt{C}\ \{\ \overline{\texttt{AO}}\ \} & \text{class operations}\\
\texttt{AO} & ::= & \textbf{adds}\ \texttt{FD} & \text{attribute operations}\\
 & & \mid \textbf{adds}\ \texttt{MD} \mid \textbf{modifies}\ \texttt{MD} &
\end{array}
$$

Figure 2: Syntax for deltas.

which may only occur in the body of the method `MD` provided by a method-modify operation **modifies** `MD`, expresses a call to the method with the same name before the modifications and is bound when the product is generated.

A delta δ may be seen as a mapping from class names to class operations. A delta is *applicable* to a class table `CT` if each class to be *modified* exists and, for every class-modify operation, each method to be modified exists and has the same header as in the method-modify operation; and if each class, method, or field to be *added* does not exist.

Given a delta δ and a class table `CT` such that δ is applicable to `CT`, the application of δ to `CT`, denoted by $\textsc{apply}(\delta, \texttt{CT})$, is the class table $\texttt{CT}'$ defined as follows:

$$
\texttt{CT}'(\texttt{C}) = \begin{cases}
\texttt{CT}(\texttt{C}) & \text{if } \texttt{C} \notin dom(\delta)\\
\texttt{CD} & \text{if } \delta(\texttt{C}) = \textbf{adds}\ \texttt{CD}\\
\textsc{apply}_{\delta}(\delta(\texttt{C}), \texttt{CT}(\texttt{C})) & \text{if } \delta(\texttt{C}) = \textbf{modifies}\ \texttt{C}\cdots
\end{cases}
$$

where $\textsc{apply}_{\delta}(\delta(\texttt{C}), \texttt{CT}(\texttt{C}))$, the application of the class-modify operation $\delta(\texttt{C}) = \texttt{CO}$ to the class definition $\texttt{CT}(\texttt{C}) = \texttt{CD}$, is the class definition $\texttt{CD}'$ defined as follows:

$$
\texttt{CD}'(\texttt{a}) = \begin{cases}
\texttt{AD} & \text{if } \texttt{CO}(\texttt{a}) = \textbf{adds}\ \texttt{AD}\\
\texttt{MD}[\texttt{a}\$\delta/\textbf{original}] & \text{if } \texttt{CO}(\texttt{a}) = \textbf{modifies}\ \texttt{MD}\\
\texttt{A a}(\bar{\texttt{A}}\,\bar{\texttt{x}})\,\texttt{MB} & \text{if } \texttt{a} = \texttt{m}\$\delta \text{ for some } \texttt{m} \text{ such that:}\\
 & \quad \texttt{CO}(\texttt{m}) = \textbf{modifies}\ \texttt{MD},\\
 & \quad \textbf{original} \in \texttt{MD} \text{ and}\\
 & \quad \texttt{CD}(\texttt{m}) = \texttt{A m}(\bar{\texttt{A}}\,\bar{\texttt{x}})\,\texttt{MB}\\
\texttt{CD}(\texttt{a}) & \text{if } \texttt{a} \notin dom(\texttt{CO}) \text{ and } \texttt{a} \neq \texttt{m}\$\cdots\\
 & \quad \text{for some } \texttt{m} \text{ such that}\\
 & \quad (\texttt{CO}(\texttt{m}) = \textbf{modifies}\ \texttt{MD}\\
 & \quad \text{and } \textbf{original} \notin \texttt{MD})
\end{cases}
$$

The semantics of the **original** construct is modeled by the second, third, and fourth cases of the definition of $\texttt{CD}'$. In the second case the modified method is obtained by replacing each call to **original** in the old method definition by a call to $\texttt{a}\$\delta$, where `a` is the name of the modified method, \$ is a special character that is assumed to not occur in the names introduced by the programmer, and δ is the name of the delta containing the method-modify operation. In the third case the definition `MD` of a method `m` supplied by a method-modify operation contains a call to **original**, and a new method $\texttt{m}\$\delta$ is introduced with the same body as the original method `m`. The fourth case ensures that, for each method `m` that is modified without using the **original** construct, the auxiliary methods $\texttt{m}\$\cdots$ introduced by previously applied deltas are removed.

Implementing Product Lines. Let φ and ψ range over feature names. A *delta table* `DT` maps delta names to delta definitions. A DeltaJ SPL is a 5-tuple $\texttt{L} = (\{\overline{\varphi}\}, \Phi, \texttt{DT}, \Delta, \Pi)$ consisting of: (*i*) the set of the features $\{\overline{\varphi}\} = \{\varphi_1, \ldots, \varphi_n\}$ $(n \geq 1)$ of the SPL; (*ii*) the set of the valid feature configurations $\Phi \subseteq \mathcal{P}(\{\overline{\varphi}\})$; (*iii*) a delta table `DT` containing the deltas; (*iv*) a mapping $\Delta : \Phi \to \mathcal{P}(dom(\texttt{DT}))$ determining for which feature configurations a delta must be applied;

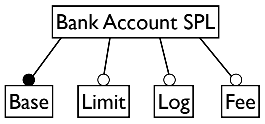

Figure 3: Feature Model of the Bank Account SPL

and (v) a total order Π of $dom(\texttt{DT})$, describing the application order of the deltas. The ordering captures semantic requires-relations that are necessary for the applicability of the deltas.

The delta table DT represents the *code base*, while the 4-tuple $(\{\overline{\varphi}\}, \Phi, \Delta, \Pi)$ represents the *product line declaration* which creates the connection to the product line variability specified in terms of product features.

The application order Π defines a *product generation mapping*, that is, a partial mapping from each feature configuration $\{\overline{\psi}\}$ in Φ to the class table of the product that is obtained by applying the deltas in $\Delta(\{\overline{\psi}\})$ to the empty class table according to the application order. The product generation mapping can be partial since a non-applicable delta may be encountered during product generation, such that the resulting product is undefined. The products generated by the product generation mapping are the products of the product line L. We write $\texttt{CT}_{\{\overline{\psi}\}}$ to denote the class table generated for the feature configuration $\{\overline{\psi}\}$.

A DeltaJ product line is *well-formed* if its product generation mapping is total and all its products are well-typed Java programs. The DeltaJ type system, formalized for the minimal core calculus IFΔJ (Imperative Featherweight Delta Java) [19], guarantees that every well-typed DeltaJ product line is well-formed. In the sequel, we only consider well-formed DeltaJ product lines. For the purpose of this paper we consider one application order for each product-line declaration. We refer to [19] for a more general understanding of application orders.

3. A BANK ACCOUNT PRODUCT LINE

The proposed approach to verifying DOP product lines is illustrated by an example of a simple Bank Account product line. Figure 3 depicts the feature model of the product line as a feature diagram. Here, only the feature Base is mandatory, while all other features are optional. Listing 1 shows a delta-oriented product line declaration for the Bank Account product line. The valid feature configurations Φ are specified by means of a propositional constraint over the set of features (see, e.g., [5]). In the example the propositional formula consists of the propositional variable Base, which represents the valid feature configurations described by the feature diagram in Figure 3, i.e., the 8 ($= 2^3$) feature configurations in $\mathcal{P}(\{\text{Base}, \text{Limit}, \text{Log}, \text{Fee}\})$ that contain the feature Base. The total order Π is defined by writing the deltas in a sequence, where each delta in the sequence is enclosed by `[ ... ]`. The mapping Δ is described by attaching to each delta (by means of a **when** clause) a propositional constraint over the set of features specifying for which feature configurations the delta has to be applied. Since only feature configurations that are valid according to the feature model are used for product generation, the application conditions are understood as a conjunction with the formula describing the set of valid feature configurations. Listing 2 shows the code base for the Bank Account product line (sequence append is denoted by ++ and "$\texttt{N y = rhs;}$" is short for "$\texttt{N y; y = rhs;}$"). Listing 3 shows the product generated when all the features are selected.

```
features Base, Limit, Log, Fee
configurations Base
deltas
  [ DBase ]
  [ DLimit when Limit ]
  [ DLog when Log ]
  [ DFee when Fee ]
```
Listing 1: Declaration of the Bank Account SPL

4. SPECIFICATION OF DELTAS

This section considers the specification of delta-oriented programs in a Hoare logic style [16], adapted to the object-oriented setting; in particular, de Boer's technique using sequences in assertions addresses the issue of object creation [11]. Let $\mathcal{A} \vdash \{p\}\overline{\mathtt{s}}\{q\}$ denote that the triple $\{p\}\overline{\mathtt{s}}\{q\}$ can be derived in Hoare logic from the axioms $\mathcal{A}$. Triples $\{p\}\overline{\mathtt{s}}\{q\}$ have a standard partial correctness semantics [2,3]; If $\overline{\mathtt{s}}$ is executed in a state where the *precondition* p holds and the execution terminates, then the *postcondition* q holds, after $\overline{\mathtt{s}}$ has terminated. Pre- and postconditions are assertions of type **boolean**, expressed in an assertion language.

4.1 The Assertion Language

We consider an assertion language defined by

$$\begin{aligned}
a &::= v \mid z \mid u \mid op(\overline{a}) \mid a\sigma \\
v &::= \texttt{this} \mid \texttt{result} \mid \ldots \\
u &::= P \mid Q \mid \ldots \\
\sigma &::= \varepsilon \mid [v_0, \ldots, v_i := a_0, \ldots, a_i] \text{ (where } 0 \leq i)
\end{aligned}$$

Assertions a are program variables v, logical variables z, uninterpreted assertions u, expressions $op(\overline{a})$ which apply an operator op to a list of assertions, and the application $a\sigma$ of a substitution σ to an assertion a. In the assertion language, the program variables v of Fig. 1 are extended with `this` and `result` (avoiding name capture), which are used for the identity of the current object and the return value of the current method. We let w range over the *assignable variables*, i.e., the program's fields and local variables. (Although `null`, booleans and integers are technically modelled by 0-ary operator application, $op()$, in the examples we will for convenience write `null`, `true`, `false`, 0, 1, 2, etc.)

Uninterpreted assertions u play the role of placeholders in symbolic assumptions, and represent the pre- and postconditions of methods where the exact specifications are unknown (e.g., to the specifications of **original** calls in method redefinition). Uninterpreted assertions are conventionally capitalized. Remark that we use the assertion $v \notin Q$ to express that the uninterpreted assertion Q may not refer to a variable v. A *concrete* assertion contains no uninterpreted assertions.

Substitutions σ bind program variables v to assertions a. Denote by ε the empty substitution, by $[v := a]$ the substitution which binds v to a, by $[v_0, \ldots, v_i := a_0, \ldots, a_i]$ the substitution $[v_0 := a_0] \ldots [v_i := a_i]$ (for $0 \leq i$), and by $dom(\sigma)$ the variables bound in σ (so $dom([v_0, \ldots, v_i := a_0, \ldots, a_i]) = \{v_0, \ldots, v_i\}$). We define the application of a substitution to a variable in its domain in the standard way: $[v_0, \ldots, v_i := a_0, \ldots, a_i](v_j) = a_j$ for $0 \leq j \leq i$. The application of a substitution to an assertion is defined over the

```
delta DBase {
  adds class Account {
    int bal = 0;  // the balance
    boolean update(int x) {
      bal = bal + x; return true;
    }
    boolean deposit(int x){ // for increasing the balance
      boolean b = false; if (x >=0) { b = update(x); } return b;
    }
    boolean withdraw(int x){ // for decreasing the balance
      boolean b = false; if (x >=0) { b = update(−x); } return b;
    }
  }
}
delta DLimit {
  modifies Account {
    adds int limit = 0;
    modifies boolean update(int x) {
      boolean b = false; if (bal+x > limit) { b = original(x); }
      return b;
    }
  }
}
delta DLog {
  modifies Account {
    adds Seq[int]= empty;
    modifies boolean update(int x) {
      boolean b=original(x); if (b) { log = log++x; }
      return b;
    }
  }
}
delta DFee {
  modifies Account {
    adds int fee = 1;
    modifies boolean update(int x) {
      boolean b;
      if (x<0) { b = original(x−fee); }
      else { b = original(x); }
      return b;
    }
  }
}
```

Listing 2: Code base of the Bank Account SPL

structure of assertions as follows:

$$v\sigma = \begin{cases} \sigma(v) & \text{if } v \in dom(\sigma) \\ v & \text{if } v \notin dom(\sigma) \end{cases}$$

$$op(a_1,\ldots,a_n)\sigma = op(a_1\sigma,\ldots,a_n\sigma).$$

Observe that substitutions applied to uninterpreted assertions $P\sigma$ and $Q\sigma$ cannot be reduced before P and Q have been instantiated in the inference system. For assertions a and p and an uninterpreted assertion P, the assertion *instantiation* $\langle\!\langle a \rangle\!\rangle^P_p$, which replaces each P in a by p, is defined inductively by:

$$\begin{array}{ll} \langle\!\langle P \rangle\!\rangle^P_p = p & \langle\!\langle a \rangle\!\rangle^P_p = a \text{ if } a \in v \mid z \\ \langle\!\langle u \rangle\!\rangle^P_p = u \text{ if } u \neq P & \langle\!\langle op(a_1,\ldots,a_n) \rangle\!\rangle^P_p \\ \langle\!\langle a\sigma \rangle\!\rangle^P_p = \langle\!\langle a \rangle\!\rangle^P_p \, \sigma & \quad = op(\langle\!\langle a_1 \rangle\!\rangle^P_p,\ldots,\langle\!\langle a_n \rangle\!\rangle^P_p) \end{array}$$

4.2 Annotated Deltas

The *contract* of a method $\mathtt{N\ m}(\overline{\mathtt{N\ x}})\{\overline{\mathtt{N\ y}};\ \overline{\mathtt{s}};\textbf{return}\ \mathtt{e};\}$ is a set S of assertion pairs (p,q) such that: (*i*) the assertion p does not contain `result` and local variables; (*ii*) the assertion q does not contain local variables; and (*iii*) $\mathcal{R} \vdash \{p\}\ \overline{\mathtt{N\ y}};\ \overline{\mathtt{s}};\textbf{return}\ \mathtt{e}\{q\}$ holds for all $(p,q) \in S$, where the set $\mathcal{R}$ of axioms annotates the method calls in $\overline{\mathtt{s}}$ with sets of *required* contracts. If $\mathtt{n} : \{(p,q)\} \cup R \in \mathcal{R}$ for some method n, then we can infer

$$\mathcal{R} \vdash \{p[\overline{x} := \overline{e}]\}\ \mathtt{w}{=}\mathtt{n}(\overline{e})\ \{q[\overline{x},\mathtt{result} := \overline{e}, w]\}.$$

```
class Account {
  int bal = 0;  // the balance
  int limit = 0;
  Seq[int]= empty;
  int fee = 1;
  boolean update$DLimit(int x) {
    bal = bal + x; return true;
  }
  boolean update$DLog(int x) {
    boolean b = false;
    if (bal+x > limit) { b = update$DLimit(x); }
    return b;
  }
  boolean update$DFee(int x) {
    boolean b=update$DLog(x); if (b) { log = log++x; }
    return b;
  }
  boolean update(int x) {
    boolean b;
    if (x<0) { b= update$DFee(x−fee); }
    else { b = update$DFee(x); }
    return b;
  }
  boolean deposit(int x){
    boolean b = false; if (x >=0) { b = update(x); } return b;
  }
  boolean withdraw(int x){
    boolean b = false; if (x >=0) { b = update(−x); } return b;
  }
}
```

Listing 3: Product with features Base, Limit, Log and Fee

We call the pair $\langle S, \mathcal{R} \rangle$, written **guar** S **req** $\mathcal{R}$, a *specification* of m. A specification $\langle S, \mathcal{R} \rangle$ of m reflects proof outlines [17] for the method body of m for all $(p,q) \in S$ with the same annotations $\mathcal{R}$ for auxiliary calls in all the proof outlines. Assertion pairs ap, requirements req, specifications sp, and annotated method definitions AMD have the syntax

$$\begin{array}{rl} \mathtt{ap} & ::= (a,a) \mid \textbf{readonly}\ \overline{f} \\ \mathtt{req} & ::= \mathtt{m} : \{\overline{\mathtt{ap}}\} \\ \mathtt{sp} & ::= \textbf{guar}\ \{\overline{\mathtt{ap}}\}\ \textbf{req}\ \{\overline{\mathtt{req}}\} \\ \mathtt{AMD} & ::= \mathtt{MD}\ \overline{\mathtt{sp}} \end{array}$$

In ap, **readonly** $\overline{f}$ expresses that the fields $\overline{f}$ are not in the write-set of a given method; i.e., if a method with body $\{\overline{\mathtt{N\ y}};\ \overline{\mathtt{s}};\textbf{return}\ \mathtt{e};\}$ has the property **readonly** $\overline{f}$, this implies $\{\overline{f} == \overline{z}\}\ \overline{\mathtt{s}}\ \{\overline{f} == \overline{z}\}$ for some logical variables $\overline{z}$.

In an *annotated delta*, all method declarations contained in a method add or modify operation are annotated. For annotated methods $\mathtt{MD}\ \overline{\mathtt{sp}}$, we extend APPLY (cf. Sect. 2) such that each requirement **original** : $\{\overline{\mathtt{ap}}\}$ in $\overline{\mathtt{sp}}$ is replaced by $\mathtt{m}\$\delta : \{\overline{\mathtt{ap}}\}$ when **original** calls in MD are replaced by $\mathtt{m}\$\delta$. Guarantees, requirements, and specifications are concrete if their assertions are concrete.

Specifying Deltas of Bank Account SPL. Listing 4 gives specifications for the methods defined in Listing 2. Primed variables are logical, and used to fix the initial values of variables; e.g., for `update` in `DBase`, the guarantee $(bal{=}{=}bal', \mathtt{result} \wedge bal == bal' + x)$ expresses that the method returns true and that the final value of *bal* equals the initial value plus the argument x. We provide two specifications for `deposit`. If $x < 0$, the guarantee expresses that `deposit` returns false and the balance is not modified. Otherwise, the guarantee is expressed using the uninterpreted requirement (P, Q) on the call to `update`. This requirement reflects that the actual implementation of `update` is not known in `DBase`. Especially, the `update` method found in `DBase`

Specifications of **delta** DBase:
update: **guar** $\{(bal{=}{=}bal', \texttt{result} \wedge bal{=}{=}bal'+x)\}$
deposit: **guar** $\{(x<0 \wedge bal{=}{=}bal', \neg\texttt{result} \wedge bal{=}{=}bal')\}$
    guar $\{(P \wedge x \geq 0, Q)\}$ **req** $\{$update: $(P, Q)\}$
withdraw: **guar** $\{(x<0 \wedge bal{=}{=}bal', \neg\texttt{result} \wedge bal{=}{=}bal')\}$
    guar $\{(x \geq 0 \wedge P[x:=-x], Q[x:=-x])\}$
      req $\{$update: $\{(P, Q)\}\}$

Specifications of **delta** DLimit:
update: **guar** $\{$**readonly** *limit*$\}$ **req** $\{$**original**: $\{$**readonly** *limit*$\}\}$
    guar $\{(bal{=}{=}bal' \wedge bal+x \leq limit, \neg\texttt{result} \wedge bal{=}{=}bal')\}$
    guar $\{(bal+x > limit \wedge P, Q)\}$ **req** $\{$**original**: $\{(P, Q)\}\}$

Specification of **delta** Dlog:
update: **guar** $\{(P \wedge log{=}{=}log', Q \wedge (\texttt{result} \Rightarrow log{=}{=}log'{+}{+}x)$
      $\wedge (\neg\texttt{result} \Rightarrow log{=}{=}log'))\}$
    req $\{$**original**: $\{(P, Q \wedge log \notin Q)$, **readonly** *log*$\}\}$

Specifications of **delta** DFee:
update: **guar** $\{$**readonly** *fee*$\}$ **req** $\{$**original**: $\{$**readonly** *fee*$\}\}$
    guar $\{(x<0 \wedge P[x:=x-fee], Q[x:=x-fee]), (x \geq 0 \wedge P, Q)\}$
    req $\{$**original**: $\{(P, Q)$, **readonly** *fee*$\}$

Listing 4: Specifications for the deltas given in Listing 2. For each added or modified method in a delta, we list the associated specifications.

may be modified by delta applications in a final product. Still, this specification technique facilitates local and compositional reasoning. The specification may be locally verified, e.g., by supplying a proof outline for the method body. Different product specifications can be constructed by instantiating P and Q: The guarantee for `deposit` follows by transformation when P and Q are known. The remaining methods in Listing 4 are similarly specified in terms of uninterpreted requirements. Requirements for **original** calls are uninterpreted, since these calls are bound depending on the selected features in the different products. A specification **guar** $\{$**readonly** `f`$\}$ **req** $\{$`m` : $\{$**readonly** `f`$\}\}$ expresses a read-only guarantee for the field `f`, assuming that method `m` only reads `f`.

5. VERIFICATION

From the perspective of deductive verification, the challenge of delta-oriented programming is to find a compositional way to verify deltas, independent of how they are assembled into products. The problem is due to the fact that the exact definition of auxiliary methods is not known before the product is assembled. Even methods defined in the *same* delta may be redefined in the final product. In particular, calls to **original** may be used to chain redefinitions of a method through the modify operations of a number of deltas in the final product (see Listing 3).

To avoid introducing restrictive behavioral constraints on method operations, we distinguish the *local reasoning* about deltas from the *global reasoning* about assembled products. In our approach a delta is a *transformer* of method specifications, which takes assumptions about the underlying methods and produces specifications for the methods defined in that delta. For this purpose, uninterpreted assertions are used in the delta specifications as *placeholders* for unknown assumptions related to auxiliary method calls and **original**. At the level of a delta, the verification process consists of checking the correctness of a method operation as a modifier of the specifications of the **original** method, relative to *assumed* specifications of the auxiliary calls. When a product has been assembled so the total order of deltas is known, the product verification consists of applying the specification transformations and checking assumed specifications of auxiliary methods.

5.1 Local Reasoning in Deltas

This section considers the local reasoning in deltas. For a specification **guar** $\{\overline{\texttt{ap}}\}$ **req** $\{\overline{\texttt{req}}\}$ of some method `m`, the goal of the local verification process is to provide a proof for each guarantee in $\overline{\texttt{ap}}$, assuming that requirements in $\overline{\texttt{req}}$ are satisfied. We assume a standard Hoare Logic, adjusted to uninterpreted assertions as explained below. Rules for object creation, conditionals, and sequential composition are standard. The statement **return** e is treated as the assignment `result` $= e$, and we reason about assignments by $\{a[w:=e]\}$ `w=e` $\{a\}$.

Method calls cannot generally be resolved by the given local specifications of these methods, since methods may be modified (cf. Sect. 4). Uninterpreted assertions (P, Q) are used as placeholders when the exact pre- and postcondition of the called method are unknown. Under the assumption that `m` satisfies (P, Q), we may reason about calls to `m` by

$$\{P[\overline{x}:=\overline{e}]\}\ \texttt{w=m}(\overline{e})\ \{Q[\overline{x}, \texttt{result}:=\overline{e}, w]\} \qquad (1)$$

where $\overline{x}$ are the formal parameters of `m`.

If `m` additionally has the guarantee **readonly** `f`, we may strengthen any guarantees (p, q) of `m` to $(p \wedge \texttt{f} == z, q \wedge \texttt{f} == z)$ for some fresh logical variable z. Remark that the call may be of the form **original**$(\overline{e})$, where the signature for **original** is the same as the signature of the currently analyzed method. Assuming that **original** satisfies (P, Q), we may then use triple (1) to reason about calls to **original**.

Reasoning about remote method calls is handled by uninterpreted assertions in a corresponding manner:

$$\{P[\overline{x}, \texttt{this}:=\overline{e}, v]\}\ \texttt{w=v.m}(\overline{e}) \\ \{Q[\overline{x}, \texttt{this}, \texttt{result}:=\overline{e}, v, w]\} \qquad (2)$$

assuming that the remote method `m` satisfies (P, Q), and $\overline{x}$ are the formal parameters of `m`.

Adaptation. Since we deal with sets of assertion pairs, we use an adaptation rule adjusted to this setting: From a set of assertion pairs (p_i, q_i), we may for any assertion a derive an assertion pair with exactly a as postcondition:

$$(\forall \overline{u}\ .\ \bigwedge_i (\forall \overline{z}_i\ .\ p_i \Rightarrow q_i[\overline{w} := \overline{u}]) \Rightarrow a[\overline{w} := \overline{u}], a)$$

where $\overline{u}$ are fresh, $\overline{z}_i$ the logical variables in (p_i, q_i), and $\overline{w}$ variables which may be updated by the method. We assume that these variables can be statically determined, which is ensured when adaptation is applied at the product level.

Verifying the Deltas of Bank Account SPL.

We consider the specifications in Listing 4 of Sect. 4.2. The methods can be verified locally by standard reasoning, assuming the **readonly** requirements and that **original** calls are decorated with assumptions according to Equation 1. The verification of `update` in `DBase` is straightforward since there are no calls in the body. Verification details for `update` in `DLimit`, `DLog`, and `DFee` are shown in Fig. 4.

The verification of `deposit` and `withdraw` follows the same pattern as outlined in Fig. 4. For `withdraw`, the requirement (P, Q) on `update` leads to the decorated call $\{P[x:=-x]\}$ `b=update(-x)` $\{Q[x, \texttt{result}:=-x, b]\}$, and the method guarantee then follows. The call in `deposit`

DLimit:

```
{$bal+x \leq limit \wedge bal{=}{=}bal'$}
Bool b = false; {$bal+x \leq limit \wedge bal{=}{=}bal' \wedge \neg b$}
if (bal+x>limit) {b=original(x)};
{$bal+x \leq limit \wedge bal{=}{=}bal' \wedge \neg b$}
return b; {$\neg$result $\wedge bal{=}{=}bal'$}
```

```
{$bal+x > limit \wedge P$} Bool b = false;
{$bal+x > limit \wedge P \wedge \neg b$} if (bal+x > limit) {
   {$P$} b=original(x) {$Q[$result$:=b]$} };
{$Q[$result$:=b]$} return b; {$Q$}
```

DLog:

```
{$P \wedge log{=}{=}log'$}
b=original(x); {$Q[$result$:=b] \wedge log{=}{=}log'$}
if b { {$Q[$result$:=b] \wedge log{=}{=}log' \wedge b$}
   log = log++x; {$Q[$result$:=b] \wedge log{=}{=}log'{+}{+}x \wedge b$}
}; {$Q[$result$:=b] \wedge (b \Rightarrow log{=}{=}log'{+}{+}x)$
   $\wedge (\neg b \Rightarrow log{=}{=}log')$} return b;
{$Q \wedge$ (result $\Rightarrow log{=}{=}log'{+}{+}x)$
  $\wedge$ ($\neg$result $\Rightarrow log{=}{=}log')$}
```

DFee:

```
{$x<0 \wedge P[x:=x-fee]$}
if (x<0) { {$P[x:=x-fee]$}
   b=original(x-fee) {$Q[x,$ result$:=x-fee, b]$}
} else { b=original(x)}
{$Q[x,$ result$:=x-fee, b]$} return b; {$Q[x:=x-fee]$}
```

```
{$x \geq 0 \wedge P$} if (x<0) { b = original(x-fee)}
else { {$x \geq 0 \wedge P$} b=original(x) {$Q[$result$:=b]$} };
{$Q[$result$:=b]$} return b; {$Q$}
```

Figure 4: Verification details for the update method in the different deltas.

is decorated by $\{P\}$ `b=update(x)` $\{Q[\texttt{result} := b]\}$, and the verification of the guarantee is then straightforward.

Remark that all specifications in Listing 4 satisfy the guarantee $(bal{=}{=}bal', \neg\texttt{result} \Rightarrow bal{=}{=}bal')$, expressing that the balance is not modified if a method returns false. Since this guarantee is maintained by all methods, it holds in any product of the product line.

5.2 Verifying Mutual Consistency in Products

This section considers the analysis of products $\texttt{CT}_{\{\overline{\psi}\}}$, constructed by applying a sequence of deltas to an initially empty class table. Assuming that all method specifications have been verified in their deltas, the overall goal of the product verification phase is to arrive at *concrete* method specifications that are *mutually consistent*. A product has mutually consistent method specifications if each assumption introduced in a method call holds for the implementation to which the call binds in the product variant.

In a product, all **original** calls have been replaced by calls of the form $\texttt{m}\$\delta$, where $\texttt{m}\$\delta$ is defined in the current class, each call $\texttt{m}(\overline{e})$ or $\texttt{v.m}(\overline{e})$ has a unique binding, and all available fields can be determined. Thus, all method names in a product variant are unique, and may be of the form $\texttt{m}\$\delta$.

Assume that p_i, q_i, r and s (for $1 \leq i \leq j$) are all concrete such that each (p_i, q_i) is a guarantee for `m`, and (r, s) is a requirement imposed by some call to `m`. Formally, consistency can be checked by adapting (p_i, q_i) to the postcondition s, and ensure that the resulting precondition follows from r:

$$r \Rightarrow (\forall \overline{u}\ .\ \bigwedge_i (\forall \overline{z}_i\ .\ p_i \Rightarrow q_i[\overline{w}{:=}\overline{u}]) \Rightarrow s[\overline{w}{:=}\overline{u}]) \qquad (3)$$

where $\overline{w}$ are variables which may be modified by `m`. We *discharge* the requirement $\texttt{m} : (r, s)$ if this implication holds. A requirement $\texttt{n} : \{\textbf{readonly}\ \texttt{f}\}$ is discharged by static analysis of the product code, ensuring that `f` is not written to in `n` or in methods called by `n`.

To obtain mutually consistent specifications, all uninterpreted assertions in the delta specifications must be instantiated. Consider a method `n` with the specification

$$\textbf{guar}\ \{(p, q)\}\ \textbf{req}\ \{\texttt{m} : (P, Q)\}$$

where p may contain P and q may contain Q, and assume that `m` has the guarantees (p_i, q_i), for $1 \leq i \leq j$. We may derive j specifications for `n` by instantiating (P, Q) to each (p_i, q_i), arriving at the transformed guarantees $(\langle\!\langle p \rangle\!\rangle^P_{p_i}, \langle\!\langle q \rangle\!\rangle^Q_{q_i})$ for `n`. Even though such transformations can be done automatically, user guidance may be needed in order to find concrete assertions for P and Q such that consistency is ensured; e.g., `m` and `n` may be mutually recursive with uninterpreted requirements on the recursive calls. If (p_i, q_i) are concrete, the above transformation ensures Equation 3 by construction. This is the case for the example in this paper: Uninterpreted assertions are always instantiated to concrete assertions. If all requirements for (p_i, q_i) have been discharged, there are no requirements left to discharge for $(\langle\!\langle p \rangle\!\rangle^P_{p_i}, \langle\!\langle q \rangle\!\rangle^Q_{q_i})$. If the instantiation reduces the precondition to false, the specification may be discarded since (false, q) is trivial. This situation is illustrated in the example below.

Verifying Products of the Bank Account SPL. We consider products which combine deltas in Listing 2, and focus on specifications with uninterpreted requirements.

Product $\texttt{CT}_{\{\mathsf{Base}\}}$. We first consider the product consisting of only the mandatory feature Base. The local calls to `update` are bound to the implementation found in delta `DBase`, which satisfies $(bal{=}{=}bal', \texttt{result} \wedge bal{=}{=}bal'{+}x)$. Guarantees for `deposit` and `withdraw` are derived by instantiating the uninterpreted requirements (P, Q) with this guarantee. Thus, we instantiate P to $bal{=}{=}bal'$ and Q to $\texttt{result} \wedge bal{=}{=}bal'{+}x$. Remark that x is substituted by $-x$ in the guarantee for `withdraw`, and we then arrive at the following guarantees for this product:

$$\begin{array}{ll}\texttt{deposit:} & (bal{=}{=}bal' \wedge x{\geq}0, \texttt{result} \wedge bal{=}{=}bal'{+}x) \\ \texttt{withdraw:} & (bal{=}{=}bal' \wedge x{\geq}0, \texttt{result} \wedge bal{=}{=}bal'{-}x)\end{array}$$

Product $\texttt{CT}_{\{\mathsf{Base},\mathsf{Fee}\}}$. In this product, the call to `update` binds to the implementation found in delta `DFee`. The original call, renamed to `update$DFee`, binds to `update` in delta `DBase`. Guarantees for the `update` method in this product are generated by instantiating the two specifications in `DFee` with the guarantee from `DBase`:

$$\begin{array}{l}(x{<}0 \wedge bal{=}{=}bal', \texttt{result} \wedge bal{=}{=}bal'{+}x{-}fee) \\ (x{\geq}0 \wedge bal{=}{=}bal', \texttt{result} \wedge bal{=}{=}bal'{+}x)\end{array}$$

Guarantees for `deposit` and `withdraw` can be derived by plugging these guarantees into the specifications in Listing 4.

For deposit we then obtain the following two guarantees:

$$(x<0 \wedge bal{=}{=}bal' \wedge x\geq 0, \texttt{result} \wedge bal{=}{=}bal'+x-fee)$$
$$(x\geq 0 \wedge bal{=}{=}bal', \texttt{result} \wedge bal{=}{=}bal'+x)$$

The first guarantee is ignored since the precondition is always false and the second ensures that no fee is charged. For withdraw, the instantiation produces the guarantees:

$$(x>0 \wedge bal{=}{=}bal', \texttt{result} \wedge bal{=}{=}bal'-x-fee)$$
$$(x{=}{=}0 \wedge bal{=}{=}bal', \texttt{result} \wedge bal{=}{=}bal')$$

Thus, withdraw returns true if x is 0, and no fee is charged.

Product $\texttt{CT}_{\{Base,Log\}}$. Similar to product $\texttt{CT}_{\{\mathsf{Base},\mathsf{Fee}\}}$, the following guarantees for the product $\texttt{CT}_{\{\mathsf{Base},\mathsf{Log}\}}$ are derived:

$$\begin{array}{ll} \texttt{update:} & (log{=}{=}log' \wedge bal{=}{=}bal', \\ & \quad \texttt{result} \wedge bal{=}{=}bal'+x \wedge log{=}{=}log'{+}{+}x) \\ \texttt{deposit:} & (log{=}{=}log' \wedge bal{=}{=}bal' \wedge x\geq 0, \\ & \quad \texttt{result} \wedge bal{=}{=}bal'+x \wedge log{=}{=}log'{+}{+}x) \\ \texttt{withdraw:} & (log{=}{=}log' \wedge bal{=}{=}bal' \wedge x\geq 0, \\ & \quad \texttt{result} \wedge bal{=}{=}bal'-x \wedge log{=}{=}log'{+}{+}(-x)) \end{array}$$

The assertion $bal{=}{=}sum(log)$ expresses a relation between bal and log, where $sum(\epsilon) \triangleq 0$ and $sum(a{+}{+}x) \triangleq sum(a){+}x$. By adapting the guarantee for deposit to the postcondition $bal{=}{=}sum(log)$, we arrive at the precondition

$$\forall u_b, u_l \ . \ (\forall bal', log' \ . \ (bal{=}{=}bal' \wedge log{=}{=}log' \wedge x\geq 0) \Rightarrow$$
$$\texttt{result} \wedge u_b{=}{=}bal'+x \wedge u_l{=}{=}log'{+}{+}x) \Rightarrow u_b{=}{=}sum(u_l)$$

which follows from $x\geq 0 \wedge bal{=}{=}sum(log)$. We observe that the guarantee $(x<0 \wedge bal{=}{=}sum(log), bal{=}{=}sum(log))$ can be trivially verified for the implementation of deposit. For this product, deposit thereby satisfies the guarantee

$$(bal{=}{=}sum(log), bal{=}{=}sum(log)). \qquad (4)$$

Method withdraw satisfies (4) by a similar argument.

Product $\texttt{CT}_{\{Base,Limit,Log\}}$. If the deltas DLimit and DBase are applied, we have the following guarantees for update in the product $\texttt{CT}_{\{\mathsf{Base},\mathsf{Limit}\}}$:

$$(bal{=}{=}bal' \wedge bal{+}x\leq limit, \neg\texttt{result} \wedge bal{=}{=}bal')$$
$$(bal{=}{=}bal' \wedge bal{+}x>limit, \texttt{result} \wedge bal{=}{=}bal'+x)$$

Instantiating the specification in DLog with these guarantees yields guarantees for update in product $\texttt{CT}_{\{\mathsf{Base},\mathsf{Limit},\mathsf{Log}\}}$:

$$(log{=}{=}log' \wedge bal{=}{=}bal' \wedge bal{+}x\leq limit,$$
$$\neg\texttt{result} \wedge bal{=}{=}bal' \wedge log{=}{=}log')$$
$$(log{=}{=}log' \wedge bal{=}{=}bal' \wedge bal{+}x>limit,$$
$$\texttt{result} \wedge bal{=}{=}bal'+x \wedge log{=}{=}log'{+}{+}x)$$

Observe that if we modify the product line declaration in Listing 1 by switching the application order of deltas DLimit and DLog, generating the product $\texttt{CT}'_{\{\mathsf{Base},\mathsf{Limit},\mathsf{Log}\}}$, we obtain the same two guarantees when adding the trivial guarantee $(bal{-}x\leq limit \wedge log{=}{=}log', log{=}{=}log')$ for update in delta DLimit. Thus, the deltas DLimit and DLog can be applied in any order (after DBase) without affecting the guarantees of the resulting product. Especially, the resulting guarantees for deposit and withdraw are the same. As above, we can prove that guarantee (4) is satisfied in this product.

Product $\texttt{CT}_{\{Base,Log,Fee\}}$. For $\texttt{CT}_{\{\mathsf{Base},\mathsf{Log},\mathsf{Fee}\}}$ and the variant with the opposite application order for DLog and DFee $\texttt{CT}'_{\{\mathsf{Base},\mathsf{Log},\mathsf{Fee}\}}$, we get the following guarantees for update:

$\texttt{CT}_{\{\mathsf{Base},\mathsf{Log},\mathsf{Fee}\}}$:

$$(bal{=}{=}bal' \wedge log{=}{=}log' \wedge x<0,$$
$$\texttt{result} \wedge bal{=}{=}bal'+x-fee \wedge log{=}{=}log'{+}{+}(x-fee))$$
$$(bal{=}{=}bal' \wedge log{=}{=}log' \wedge x\geq 0,$$
$$\texttt{result} \wedge bal{=}{=}bal'+x \wedge log{=}{=}log'{+}{+}x)$$

$\texttt{CT}'_{\{\mathsf{Base},\mathsf{Log},\mathsf{Fee}\}}$:

$$(bal{=}{=}bal' \wedge log{=}{=}log' \wedge x<0,$$
$$\texttt{result} \wedge bal{=}{=}bal'+x-fee \wedge log{=}{=}log'{+}{+}x)$$
$$(bal{=}{=}bal' \wedge log{=}{=}log' \wedge x\geq 0,$$
$$\texttt{result} \wedge bal{=}{=}bal'+x \wedge log{=}{=}log'{+}{+}x)$$

Thus, $\texttt{CT}_{\{\mathsf{Base},\mathsf{Log},\mathsf{Fee}\}}$ satisfies guarantee (4), but $\texttt{CT}'_{\{\mathsf{Base},\mathsf{Log},\mathsf{Fee}\}}$ does not, since the fee is not recorded in the log if $x < 0$. The application order of deltas DLog and DFee is not arbitrary.

Product $\texttt{CT}_{\{Base,Limit,Log,Fee\}}$. For this product, which can be found in Listing 3, we can prove that the methods deposit and withdraw satisfy the guarantee:

$$(bal{=}{=}sum(log) \wedge bal>limit, bal{=}{=}sum(log) \wedge bal>limit)$$

6. RELATED WORK

Product line analysis techniques can be classified in three main categories [21]. First, product-based analyses consider each product variant separately. Product-based analyses can use any standard analysis technique for single products. These analyses work well when a relatively small number of products are generated (as it happens in many practical cases), but are in general infeasible when an exponential (in the number of features) number of products are generated. Second, family-based analysis checks the complete code base of the product line in a single run to obtain a result about all possible variants. Family-based product line analyses, which rely on a monolithic model of the product line, are currently used for type checking [1, 12] and model checking [4, 10] of product lines. Third, feature-based analysis considers the building blocks of the different product variants (the deltas in DOP) in isolation to derive results on all variants. Feature-based analyses usually only work for feature-compositional properties, such as syntax checking, or require a final product-based or family-based analysis phase in addition to the feature-based analysis phase. For instance, in compositional type checking [19], the feature-oriented phase generates constraints for every delta in isolation and the product-based phase checks these constraints for every possible product. For deductive verification of behavioral product properties, there are product-based, feature-based and combinations of product- and feature-based analyses.

The first group of approaches constructs proofs for program properties from (partial) proofs for single program features. In [6], proofs for single language features are incrementally constructed by a feature-based proof technique for type soundness of language extensions. In [13], Coq proofs for the soundness of a small compiler are composed feature-wise using explicit variation points. Composition scripts are built by hand, and it is not clear whether the technique applies to functional verification of general programs and properties. In [22], a feature- and product-based verification approach for behavioral program properties is proposed where for each feature module a partial proof script for Coq is manually generated. These proof scripts are composed and checked for single products.

The second group of deductive verification approaches relies on behavioral program specifications to modularize proofs. A product-based analysis approach is proposed in [9]. Assuming one product variant has been fully verified, from the structure of a delta to generate another program variant, it is analyzed which proof obligations remain valid in the new product variant and need not be reestablished. In [15], a feature-based verification approach for DOP is presented which relies on a Liskov principle for DOP; i.e., specifications of methods introduced by deltas must be more specific than previous versions of these methods. Then, each delta can be verified by approximating called methods defined in other deltas by the specification of their first introduction.

The approach presented in the current paper can be classified as a feature-based analysis approach with a final product-based phase relying on behavioral program specifications. After verifying the deltas in isolation, the actual products are verified based on the specification already established for the used deltas. By proposing a different approach based on transformational reasoning, we are able to relax the restrictions of previous work [15]. This transformational approach uses symbolic assumptions on called methods and thus separates the specifications of method implementations from the requirements to method calls in a way which is similar to lazy behavioral subtyping [14]. However, previous work on lazy behavioral subtyping uses explicit, but concrete assumptions on called methods whereas our transformational approach uses symbolic assumptions.

7. CONCLUSION AND FUTURE WORK

Efficient analyses of software product lines is challenging due to the combinatorial explosion of possible product variants. Compositional approaches typically address such combinatorial explosions by introducing explicit assumptions between modules, which allows modules to be analyzed separately. However, the flexibility of the reuse mechanisms in DOP are problematic for standard compositional techniques. To alleviate this problem, we have developed a transformational proof system for delta-oriented programming in which delta modules can be analyzed separately using symbolic assumptions, and specifications of product variants can be derived from the specifications of the delta modules by discharging the proof obligations obtained for the specific product variant through the instantiation of these symbolic assumptions. In future work, we aim at a full formalization and soundness proof for the transformational proof system with an extension of the considered delta operations, e.g., by including code removal. We also plan to extend the language with interface to introduce hiding mechanisms into the proof system. Furthermore, the presented approach will be implemented using the KeY verification system [7].

Acknowledgements. We thank the anonymous referees for comments and suggestions for improving the presentation.

8. REFERENCES

[1] S. Apel, C. Kästner, A. Grösslinger, and C. Lengauer. Type safety for feature-oriented product lines. *Automated Software Engineering*, 17(3):251–300, 2010.

[2] K. R. Apt. Ten years of Hoare's logic: A survey — Part I. *ACM TOPLAS*, 3(4):431–483, Oct. 1981.

[3] K. R. Apt, F. S. de Boer, and E.-R. Olderog. *Verification of Sequential and Concurrent Systems.* Springer, 2009.

[4] P. Asirelli, M. H. ter Beek, S. Gnesi, and A. Fantechi. Deontic logics for modeling behavioural variability. In *VaMoS*, pp. 71–76, January 2009.

[5] D. Batory. Feature Models, Grammars, and Propositional Formulas. In *SPLC*, LNCS 3714, pp. 7–20. Springer, 2005.

[6] D. S. Batory and E. Börger. Modularizing theorems for software product lines: The Jbook case study. *J. UCS*, 14(12):2059–2082, 2008.

[7] B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, *LNCS* 4334. Springer, 2007.

[8] L. Bettini, F. Damiani, D. Meglio, I. Schaefer, and F. Strocco. *DeltaJ website (New Version)*, September 2011. http://deltaj.sourceforge.net/.

[9] D. Bruns, V. Klebanov, and I. Schaefer. Verification of software product lines with delta-oriented slicing. In *FoVeOOS 2010*, *LNCS* 6528. Springer, 2011.

[10] 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 *ICSE*. IEEE, 2010.

[11] F. S. de Boer. A WP-calculus for OO. In *FOSSACS*, *LNCS* 1578, pp. 135–149. Springer, 1999.

[12] B. Delaware, W. Cook, and D. Batory. A Machine-Checked Model of Safe Composition. In *FOAL*, pp. 31–35. ACM, 2009.

[13] B. Delaware, W. Cook, and D. Batory. Theorem Proving for Product Lines. In *OOPSLA'11*, 2011.

[14] J. Dovland, E. B. Johnsen, O. Owe, and M. Steffen. Lazy behavioral subtyping. *Journal of Logic and Algebraic Programming*, 79(7):578–607, 2010.

[15] R. Hähnle and I. Schafer. A Liskov Principle for Delta-oriented Programming. In *FoVeOOS 2011*, *LNCS* 7421. Springer, 2012.

[16] C. A. R. Hoare. An Axiomatic Basis of Computer Programming. *Comm. ACM*, 12:576–580, 1969.

[17] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6(4):319–340, 1976.

[18] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. Delta-oriented Programming of Software Product Lines. In *SPLC*, *LNCS* 6287, pp. 77–91. Springer, 2010.

[19] I. Schaefer, L. Bettini, and F. Damiani. Compositional Type-Checking for Delta-Oriented Programming. In *AOSD*. ACM, 2011.

[20] I. Schaefer and R. Hähnle. Formal methods in software product line engineering. *IEEE Computer*, 44(2):82–85, 2011.

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

[22] T. Thüm, I. Schaefer, M. Kuhlemann, and S. Apel. Proof composition for deductive verification of software product lines. In *VAST*, pp. 270–277. IEEE, 2011.