Conflict Detection in Delta-Oriented Programming*

Michael Lienhardt[1] and Dave Clarke[2]

[1] University of Bologna, Italy
lienhard@cs.unibo.it
[2] IBBT-DistriNet Katholieke Universiteit Leuven, Belgium
Dave.Clarke@cs.kuleuven.be

Abstract. This paper study the notion of conflict for a variant of Delta-Oriented Programming (DOP) without features. Specifically, we define a language for this subset of DOP and give a precise, formal definition of conflict. We then define a type system based on row-polymorphism that ensures that the computation of a well-typed product will always succeed and have an unambiguous result.

1 Introduction

Delta-oriented programming (DOP) [21, 22] is a recent approach to developing Software Product Lines (SPLs) [6] that addresses several limitations of previous approaches: it completely dissociates feature models from feature modules (now called *deltas*), which allows features to be implemented using more than one delta and deltas to be used by several features, thus improving modularity, reuse and flexibility; moreover, DOP enables non-monotonic modifications of the core architecture, including the removal of fields, methods and even classes. DOP is flexible and enables the modular construction of SPLs. However, tool support for DOP is is not as mature as for other SPL approaches. In particular, the issue of validating delta-oriented programs has not fully been addressed. Schaefer et al. [20] propose to generate a collection of constraints for delta-oriented product lines, ensuring that the manipulations done on the core product are sound and the resulting products are type safe. However, this work have several limitations: i) as it is based on constraints, the types does not reflect the structure of the deltas; ii) it presupposes that the order in which the deltas are applied on a core is totally specified; and iii) it generates a set of constraints per product, which means that the complexity is exponential in the number of deltas. More recently, the present authors proposed an approach [16] that addresses the first of these limitations using *row polymorphism* [19] to capture the structure of products and the semantics of deltas in the types. This paper presents an extension of this approach to deal with the second limitation.

* This research is partly funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models (http://www.hats-project.eu)

```
core k {
 class Settings {
  int coffee;
 }

 class CMachine {
   Settings conf;

   void make() {...}
   void makeCoffee() {...}
 }
}

delta Choco {
 modifies class Settings {
  adds int chocolate;
 }
 modifies class CMachine {
  adds makeChoco() {...}
  modifies make() {...}
 }
}
```

```
delta Sugar {
 modifies class Settings {
  adds int sugar;
 }
 modifies class CMachine {
  modifies make {...}
 }
}

delta ColorPrint {
 modifies class CMachine {
  modifies make() {...}
  modifies makeCoffee() {...}
  modifies makeChoco() {...}
 }
}


product p_s {Choco Sugar} k
product p_h {Choco ColorPrint} k
```

Fig. 1: Soft and Hard Conflicts

In DOP if feature modules are applied to a core product in a different order it is not necessarily the case that all computations give the same result. This is illustrated with the code in Figure 1. This example models a coffee machine with core `k` comprised of a class `Setting` storing the type of coffee to brew, and a class `CMachine` with a generic `make` method and a method `makeCoffee`, called by `make` to prepare coffee. In addition to this core, there are three deltas: `Choco` adds the capability of brewing hot chocolate; `Sugar` adds the possibility of setting the quantity of sugar; and `ColorPrint` changes the `make*` methods so that messages are printed in color. Finally, there are two different products: $\mathtt{p}_s$ applies deltas `Choco` and `Sugar` on the core `k`, and product $\mathtt{p}_h$ is constructed by applying deltas `Choco` and `ColorPrint` to the core. The order in which the deltas are applied is free in this example, and thus $\mathtt{p}_s$ and $\mathtt{p}_h$ can either be computed by applying either delta `Choco` or the other one first.

Applying first `Choco` and then `Sugar` in $\mathtt{p}_s$ results in a product with the method `make` defined by the delta `Sugar`, whereas if `Sugar` is applied first, the method `make` is defined by `Choco`: the result of the computation of $\mathtt{p}_s$ is non-deterministic, caused by a *soft conflict* between the deltas `Choco` and `Sugar`. Such soft conflicts can be dealt with in two ways: by defining a partial order between delta: for instance, by stating that `Choco` is always applied after `Sugar` the computation of $\mathtt{p}_s$ is deterministic; and by defining another delta to *resolve*

$$
\begin{array}{rll}
PL ::= & 0 \quad | \quad PLE\ PL & DL ::= \epsilon \quad | \quad \mathtt{d}\ DL \\
PLE ::= & \textbf{core}\ \mathtt{k}\ \{CL\} & CL ::= \epsilon \quad | \quad C\ CL \\
 & |\ \textbf{delta}\ \mathtt{d}\ \textbf{after}\ DL\ \{COL\} & FL ::= \epsilon \quad | \quad F;\ FL \\
 & |\ \textbf{product}\ \mathtt{p}\ \mathtt{=}\ \{DL\}\ \mathtt{k} & C ::= \textbf{class}\ \mathtt{c}\ \{FL\} \\
COL ::= & CO \quad | \quad CO;\ COL & F ::= T\ \mathtt{f}\ \mathit{def} \\
FOL ::= & FO \quad | \quad FO;\ FOL & \\
CO ::= & \textbf{adds}\ C \quad | \quad \textbf{removes}\ \mathtt{c} \quad | \quad \textbf{modifies}\ \mathtt{c}\ \{FOL\} & \\
FO ::= & \textbf{adds}\ F \quad | \quad \textbf{removes}\ \mathtt{f} \quad | \quad \textbf{modifies}\ \mathtt{f} &
\end{array}
$$

Fig. 2: Calculus Syntax

the conflict: adding to $\mathtt{p}_s$ delta `SweetChoco` that replaces the method `make` after applying `Choco` and `Sugar` will also make the computation deterministic.

The product $\mathtt{p}_h$ presents another kind of conflict, called *hard conflicts*. While first applying the delta `Choco` and then `ColorPrint` results in a valid computation, first applying `ColorPrint` results in an error because `ColorPrint` tries to modify method `makeChoco` before it exists. Such hard conflicts can only be resolved by imposing an ordering on the deltas specifying that `ColorPrint` must be applied after `Choco`.

Roadmap. The paper is structured as follow. Section 2 describes a DOP language focusing on deltas and conflicts. Section 3 presents a formal definition of soft and hard conflicts. Section 4 introduces our type system to capture runtime errors and conflicts. Section 5 compares our approach with related work and Section 6 concludes the paper.

2 Delta-Oriented Programming

In the rest of the paper, we will use the term *field* for either a method or a field of a class. The syntax of our delta-oriented programming language is presented in Figure 2. A product line PL is a sequence of element declaration PLE. An element can either be a delta **delta** `d` **after** DL `{` COL `}`, where DL defines the partial order between the deltas and COL is the body of `d`; a core product **core** `k {` CL `}`, where CL is the set of classes defining the core `k`; or a product **product** `p = {` DL `} k`, where DL are the deltas to be applied to the core `k` to produce `p`. CO and FO are the operations on classes and fields, respectively. It is possible to add, remove and modify both classes and fields. The modification of a class is done with a sequence of operations on fields FOL, while the modification of field is not specified in our language that only focuses on the manipulation on the structure of the cores, not their behavior.

Free names. The declaration of a new delta `d`, of a new core `k` or of a new product `p`, binds that name in the rest of the program. This notion of binders and free names implicitly creates a form of α-conversion on our PL terms. We note

$PL =_\alpha PL'$ when PL is α-equivalent to PL'. Using this notion of α-conversion, we can assume that all declared deltas, cores and products of a product line PL have different names.

Definition 1. *A product line PL with no free names (i.e. $fn(PL) = \emptyset$) is said to be* closed. *We note $\mathcal{P}^{cl}$ the set of all closed product lines.*

Semantics. The full semantics of the language is presented elsewhere [5, 15]. The principle of the computation of a product in our language is quite simple. The delta names in DL are sorted to match the order given by the keyword **after**. When the order is not total, several sequences of deltas are possible, creating the possibility of conflicts. Then, the code of the deltas are applied in order to the core, thus computing the product. The following definition presents the formal construction of the order between deltas, which is necessary to define the notion of conflicts.

Definition 2. *A* general context $\mathbb{G}$ *is a product line with a hole $\bullet$. Given a product line PL and a term* `t` *given by one of the productions in Figure 2. Say that* `t` is in PL *(denoted* $\mathtt{t} \in PL$*) if there exists a general context $\mathbb{G}$ such that* $PL = \mathbb{G}[\mathtt{t}]$. *The relation $<_{PL}$ between delta names is defined as the smallest transitive relation satisfying the following property*

$$\forall \mathtt{d}, \overline{\mathtt{d}_i}, COL\, (\textbf{delta}\ \mathtt{d}\ \textbf{after}\ \mathtt{d}_1 \ldots \mathtt{d}_n\ \{COL\} \in PL \Rightarrow \mathtt{d} <_{PL} \mathtt{d}_i).$$

Finally, the next definition presents the equivalence relation used to sort delta names, which is used in our type system.

Definition 3. *A relation R is* closed under general context *iff*

$$\forall \mathbb{G}, \quad x\, R\, y \quad \Rightarrow \quad \mathbb{G}[x]\, R\, \mathbb{G}[y]$$

Define the relation $\equiv_{dl}$ as the smallest equivalence relation closed under general context validating the following rule:

$$\frac{DL \equiv_{dl} DL'}{\mathtt{d}_1\ \mathtt{d}_2\ DL \equiv_{dl} \mathtt{d}_2\ \mathtt{d}_1\ DL'}.$$

3 Conflicts

Clarke et al. [4] define the notion of conflict for an abstract notion of delta, but they do not capture hard conflicts. This section proposes a more precise definition based the notion of *action.*

Actions. The following syntax gives the elements E manipulated by a delta and how they can be manipulated:

$$\begin{array}{l} E ::= \mathtt{c} \quad | \quad \mathtt{c.f} \\ A ::= \bot \quad | \quad \textbf{add} \quad | \quad \textbf{rem} \quad | \quad \textbf{mod} \end{array}$$

Given a class `c` or a field `c.f` (annotated with its class name `c`), a delta can either do nothing with it ($\bot$); add it (**add**); remove it (**rem**); or modify it (**mod**).

$$
\begin{gathered}
\mathit{act}_c(\textbf{adds class } \texttt{c}\ \{FL\}) \triangleq \overline{\{\texttt{c}\} \cup \mathit{fields}(\texttt{c}, FL) \rightarrow \textbf{add}} \\
\mathit{act}_c(\textbf{removes class } \texttt{c}) \triangleq \overline{\{\texttt{c}\} \cup \{\texttt{c.f} \mid \texttt{f} \in \mathcal{F}\} \rightarrow \textbf{rem}} \\
\mathit{act}_c(\textbf{modifies class } \texttt{c}\ \{FOL\}) \triangleq \overline{\{\texttt{c}\} \rightarrow \textbf{mod}} \rhd \mathit{act}_f(\texttt{c}, FOL) \\
\mathit{act}_c(CO; COL) \triangleq \mathit{act}_c(CO) \rhd \mathit{act}_c(COL)
\end{gathered}
$$

$$
\begin{gathered}
\mathit{act}_f(\texttt{c}, \textbf{adds } T\ \texttt{f}\ \mathit{def}) \triangleq \overline{\{\texttt{c.f}\} \rightarrow \textbf{add}} \qquad \mathit{act}_f(\texttt{c}, \textbf{removes } \texttt{f}) \triangleq \overline{\{\texttt{c.f}\} \rightarrow \textbf{rem}} \\
\mathit{act}_f(\texttt{c}, \textbf{modifies } \texttt{f}) \triangleq \overline{\{\texttt{c.f}\} \rightarrow \textbf{mod}} \\
\mathit{act}_f(\texttt{c}, FO; FOL) \triangleq \mathit{act}_f(\texttt{c}, FO) \rhd \mathit{act}_f(\texttt{c}, FOL)
\end{gathered}
$$

$$
\begin{gathered}
A \rhd \bot = A \qquad \bot \rhd A = A \\
\textbf{add} \rhd \textbf{add} \triangleq \textbf{add} \qquad \textbf{add} \rhd \textbf{rem} \triangleq \bot \qquad \textbf{add} \rhd \textbf{mod} \triangleq \textbf{add} \\
\textbf{rem} \rhd \textbf{add} \triangleq \textbf{mod} \qquad \textbf{rem} \rhd \textbf{rem} \triangleq \textbf{rem} \qquad \textbf{rem} \rhd \textbf{mod} \triangleq \textbf{rem} \\
\textbf{mod} \rhd \textbf{add} \triangleq \textbf{mod} \qquad \textbf{mod} \rhd \textbf{rem} \triangleq \textbf{rem} \qquad \textbf{mod} \rhd \textbf{mod} \triangleq \textbf{mod}
\end{gathered}
$$

Fig. 3: Actions of Operations

Denoting the set of all elements E (resp. all actions A) by $\mathcal{E}$ (resp. $\mathcal{A}$), we define the action of a delta as a function from $\mathcal{E}$ to $\mathcal{A}$ which maps all elements to the actions performed by the delta on them. The action of the code of a delta is defined inductively in Fig. 3, based on the following notions.

Definition 4. *Given a set of elements $S \subseteq \mathcal{E}$ and an action A. Use $\overline{S \rightarrow A}$ to denote the function $f : \mathcal{E} \rightarrow \mathcal{A}$ such that $f(E) = A$ when $E \in S$ and $f(E) = \bot$ for all $E \notin S$. Given two functions $f, g : \mathcal{E} \rightarrow \mathcal{A}$. Use $f \rhd g$ to denote the function $h : \mathcal{E} \rightarrow \mathcal{A}$ such that $h(E) = f(E) \rhd g(E)$, where $\rhd$ is defined in Fig. 3.*

Adding a class corresponds to the action **add** on the class and on all of its fields. Removal corresponds to the action **rem** on the class and on all of its possible fields (noted $\mathcal{F}$). Modification corresponds to **mod**, plus all the actions done on the field level. Sequential composition of simple operators is handle by the operator $\rhd$ which gives the semantics of the sequential composition of actions.

Finally, the action of a product line that maps all delta names to their actions for a closed product line is defined as follows.

Definition 5. *Given a closed product line PL. The* action *of PL, denoted $\mathit{act}(PL)$, is a function that maps all the deltas declared in PL to their code's action:*

$$
\mathit{act}(PL)(\texttt{d}) \triangleq \begin{cases} \mathit{act}_c(COL) & \text{if } \textbf{delta } \texttt{d} \textbf{ after } DL\ \{COL\} \in PL \\ \overline{\mathcal{E} \rightarrow \bot} & \text{otherwise} \end{cases}
$$

Conflicts. The following definition captures the two kinds of conflict:

Definition 6. *Given a closed product line PL, an element E, a product name* $\texttt{p}$ *and two delta names $\texttt{d}_1$ and $\texttt{d}_2$ such that*

$$
\texttt{d}_1 \not<_{PL} \texttt{d}_2 \wedge \texttt{d}_2 \not<_{PL} \texttt{d}_1 \wedge (\textbf{product } \texttt{p} = \{DL\}\ \texttt{k} \in PL) \wedge (\texttt{d}_1, \texttt{d}_2 \in DL).
$$

$$\begin{array}{rl}
TP ::= & \{TCL^{\emptyset}\} \\
TCL^{c} ::= & \mathbf{Abs}^{c} \quad | \quad \rho^{c} \quad | \quad \mathtt{c} : CP;\ CL^{\{\mathtt{c}\}\uplus c} \\
CP ::= & \mathbf{Pre}_{J}(TC) \quad | \quad \mathbf{Abs}_{J}(TC) \\
TC ::= & \{FL^{\emptyset}\} \\
TFL^{f} ::= & \mathbf{Abs}^{f}_{J} \quad | \quad \rho^{f}_{\gamma} \quad | \quad \mathtt{f} : FP;\ FL^{\{\mathtt{f}\}\uplus f} \\
TFP ::= & \mathbf{Abs}_{J} \quad | \quad \mathbf{Pre}_{J} \\
TD ::= & TP \to TP \quad | \quad \forall \alpha.TD \\
TO ::= & TC \to TC \quad | \quad \forall \alpha.TO \\
J ::= & \gamma \quad | \quad \bot \quad | \quad J; (\mathtt{d}, A) \quad | \quad J; (\mathtt{d}_1 \ldots \mathtt{d}_n)
\end{array}$$

Fig. 4: Type Syntax

Product line PL has a soft conflict, *denoted $PL \vDash^{E}_{\mathtt{p}} \mathtt{d}_1 \lightning \mathtt{d}_2$, iff E is a field* $\mathtt{c.f}$ *and $act(PL)(\mathtt{d}_1)(E) = act(PL)(\mathtt{d}_2)(E) = \mathbf{mod}$. There is a* hard conflict, *denoted $PL \vDash^{E}_{\mathtt{p}} \mathtt{d}_1 \lightning\lightning \mathtt{d}_2$, iff both deltas are acting on E and one of them is not doing a simple modification. That is,*

$$(act(PL)(\mathtt{d}_1)(E), act(PL)(\mathtt{d}_2)(E)) \notin \{\bot\} \times \mathcal{A} \cup \mathcal{A} \times \{\bot\} \cup \{(\mathbf{mod}, \mathbf{mod})\}.$$

A conflict occurs when two operations on the same element may not produce the same result. An example *soft conflict* results from two modifications of a field: the two possible sequences can product a different result, thus causing *ambiguity*. Hard conflicts produce an error during the computation of a product. For instance, first modifying an element and then removing it is correct, whereas trying to modify an element that was removed is erroneous. Finally, it is possible to resolve soft conflicts with another delta that acts on the element after the conflict:

Definition 7. *Given a soft conflict $PL \vDash^{E}_{\mathtt{p}} \mathtt{d}_1 \lightning \mathtt{d}_2$. This conflict is* resolved *iff there exists $\mathtt{d} \in dep_{PL}(\mathtt{p})$ such that $\mathtt{d}_1 <_{PL} \mathtt{d}$, $\mathtt{d}_2 <_{PL} \mathtt{d}$ and $act(PL)(\mathtt{d})(E) \neq \bot$.*

Theorem 1. *A closed product line PL with no unresolved conflicts is unambiguous.*

4 Type System

The type system extends our previous work [16] to capture conflicts. Its syntax is presented in Figure 4. *Row types* [19] capture the structure of products and classes, *row polymorphism* is used to type deltas and *annotations*, J, capture the action of deltas and conflicts. The type of a product TP consists of a mapping between class names $\mathtt{c}$ and presence information that can either be $\mathbf{Pre}_{J}(TC)$, meaning that the class is present, where TC specifies which fields are present and which are absent from the class, and J specifies the previous actions done on the class; or $\mathbf{Abs}_{J}(TC)$, meaning that the class is not part of the product, where TC specifies all fields absent and stores the past actions done to them—normally this component would not be present, but the type system needs to track the previously applied actions, even when a field has been removed. Row polymorphism is enabled with variables ρ which stands for an unknown mapping. The structure of the type TC is similar to TP with two differences: as fields do not have an inner structure, presence information for them do not contain an inner type; and empty rows $\mathbf{Abs}^f_J$ (resp. row variables ρ^f_γ) are annotated with a general annotation J (resp. a *conflict* variable γ) to solve the technical difficulty of storing the past actions done on the fields of a deleted class. More details can be found in the companion report [15]. Mappings TCL^c (resp. TFL^f) are annotated with sets of class names c (resp. field names f) to ensure that they are just defined once in a type. Deltas are typed with functional types TD where α can either be a row variable ρ or a *conflict* variable γ.

Annotations, J, are used for conflict detection, which is structured into two steps. We first define the action of deltas inductively on their structure: each simple operator acting on an element E is typed with an annotation on E of the form $\gamma; (\mathtt{d}, A)$, where $\mathtt{d}$ is the name of the delta performing the operation, A is the performed action, and γ represents past actions done on E. Using type unification, sequential composition of operators on the same element E result in annotations of the form $\gamma; (\mathtt{d}, A_1); \ldots; (\mathtt{d}, A_n)$, which are transformed using a *rewriting relation* into $\gamma; (\mathtt{d}, A_1 \triangleright \ldots \triangleright A_n)$, corresponding to the action of $\mathtt{d}$ on the element E. Once the action of deltas are computed, we combine them while typing lists of delta names DL to detect conflicts. This detection is done inductively on the structure of DL: given a list $\mathtt{d}_1\ \ldots\ \mathtt{d}_n$ that result in annotation J on an element E and a delta $\mathtt{d}$ that performs the action $A \neq \perp$ on E, using type unification, the list $\mathtt{d}_1\ \ldots\ \mathtt{d}_n\ \mathtt{d}$ results in annotation $J; (\mathtt{d}, A)$ on E, which is then checked and possibly transformed to record soft conflicts using helper function called `detect`. Terms of the form $J; (\mathtt{d}_1 \ldots \mathtt{d}_n)$ generally represent soft conflicts involving the n deltas $\mathtt{d}_i$ (but can also be used to store delta names in some specific cases). Hard conflicts, corresponding to possible errors, are ill-typed and thus have no syntactic representation in J.

Finally, to ensure the correctness of the conflict detection algorithm, past actions done on deleted fields need to be remembered, even when the class itself has been deleted. This means that the type of the deletion of the class $\mathtt{c}$ should be able to identify each field $\mathtt{f}_i$ in $\mathtt{c}$, take their annotation J_i, and specify that the output product is typed with $\mathtt{c} : \mathbf{Abs}(\{\overline{\mathtt{f}_i : \mathbf{Abs}_{J_i;(\mathtt{d},\mathbf{rem})}}\})$. As discussed in the companion report [15], it is not possible using standard unification to compute such a type. We solve this problem by using *local substitutions* [17, 15], which allow conflict variables γ to be substituted locally into an element E. For instance, it is possible to type the removal of class $\mathtt{c}$ with

$$\{\mathtt{c} : \mathbf{Pre}_\gamma(\{\rho_{\gamma'}\})\} \rightarrow \{\mathtt{c} : \mathbf{Abs}_{\gamma;(\mathtt{d},\mathbf{rem})}(\{\mathbf{Abs}_{\gamma';(\mathtt{d},\mathbf{rem})}\})\}.$$

It is possible to type the application of this operator to a product with class $\mathtt{c}$ by first unifying ρ with the structure of $\mathtt{c}$, producing an input type of the form $\{\mathtt{c} : \mathbf{Pre}_J(\{\mathtt{f}_1 : \mathbf{Pre}_{\gamma'}; \ldots; \mathtt{f}_n : \mathbf{Pre}_{\gamma'}; \mathbf{Abs}_{\gamma'}\})\}$, and then unifying each instance of variable γ' with the annotation local to each field.

$$\text{T:FL}\quad \frac{\Phi = \bot \Rightarrow (J = J' = \bot) \qquad \Phi = \mathtt{d}, \gamma \Rightarrow (J = \gamma; (\mathtt{d}, \mathbf{add}) \wedge J' = \gamma)}{\Phi \vdash T_1\ \mathtt{f}_1\ \mathit{def}_1; \ldots; T_n\ \mathtt{f}_n\ \mathit{def}_n : \{\mathtt{f}_1 : \mathbf{Pre}_J; \ldots; \mathtt{f}_n : \mathbf{Pre}_J; \mathbf{Abs}_{J'}\}}$$

$$\text{T:CL}\quad \frac{\Phi \vdash FL_i : TC_i \quad i \in 1..n \qquad \Phi = \bot \Rightarrow J = \bot \qquad \Phi = \mathtt{d}, \gamma \Rightarrow J = \gamma; (\mathtt{d}, \mathbf{add})}{\begin{array}{l}\Phi \vdash \mathbf{class}\ \mathtt{C}_1\ \{FL_1\}\ \ldots\ \mathbf{class}\ \mathtt{C}_n\ \{FL_n\} \\ \qquad : \{\mathtt{C}_1 : \mathbf{Pre}_J(TC_1); \ldots; \mathtt{C}_n : \mathbf{Pre}_J(TC_n); \mathbf{Abs}\}\end{array}}$$

Fig. 5: Typing Core Products

Relations. We define two relations on our type syntax: a structural equivalence that identifies types with the same semantics, and the rewriting relation used for the computation the action of a delta.

Definition 8. *A* type context $\mathbb{T}$ *is any type term with a hole* $\bullet$*. Moreover, we say that a relation R is* closed under type context *iff*

$$\forall \mathbb{T}, \quad x\, R\, y \quad \Rightarrow \quad \mathbb{T}[x]\ R\ \mathbb{T}[y]$$

*The structural equivalence $\equiv$ between types is the smallest equivalence closed under type context satisfying the following rules, where a denotes either a class name or a field name, K denotes either **Abs**, **Pre** or **Pre**(TC) and W^l denotes either a class list TCL^c or a field list TFL^f:*

$$a : K; b : K'; W^l \equiv b : K'; a : K; W^l$$
$$\boldsymbol{Abs}^{c \uplus \{\mathtt{c}\}} \equiv \mathtt{c} : \boldsymbol{Abs}_{\bot}; \boldsymbol{Abs}^{c} \qquad\qquad \boldsymbol{Abs}_J^{f \uplus \{\mathtt{f}\}} \equiv \mathtt{f} : \boldsymbol{Abs}_J; \boldsymbol{Abs}_J^{f}.$$

The rewriting relation $\blacktriangleright$ is the smallest reflexive and transitive relation that is closed under type context satisfying the following rules:

$$(\mathtt{d}, A); (\mathtt{d}, A') \blacktriangleright (\mathtt{d}, A \triangleright A')$$

This equivalence relation states that the order in which the classes and fields are typed is not important, and that the empty row $\mathbf{Abs}^l$ corresponds to classes and fields being absent. Moreover, the rewriting relation states that when we have two consecutive actions for the same delta `d` (typically coming from two operators used in the definition of `d`), we can combine them using the operator $\triangleright$, thus computing the action of `d`. Finally, let $\mathit{Norm} \vdash TD$ denote when the annotations J in TD have been fully reduced with the rewriting relation, i.e. when the actions of the deltas have been computed.

Typing Rules. The rules defining our type system are structured in three parts: classes and core products; operators on classes and products; and product lines, i.e. deltas, core definitions and products.

$$\text{T:AddField}\qquad \frac{J=\gamma \qquad J'=\gamma;(\mathtt{d},\mathbf{add})}{\mathtt{d}\vdash \mathbf{adds}\ T\ \mathtt{f}\ \mathit{def} : \forall\rho,\gamma.\{\mathtt{f}:\mathbf{Abs}_J\,;\rho_\gamma\}\to\{\mathtt{f}:\mathbf{Pre}_{J'}\,;\rho_\gamma\}}$$

$$\text{T:DelField}\qquad \frac{J=\gamma \qquad J'=\gamma;(\mathtt{d},\mathbf{rem})}{\mathtt{d}\vdash \mathbf{removes}\ \mathtt{f} : \forall\rho,\gamma.\{\mathtt{f}:\mathbf{Pre}_J\,;\rho_\gamma\}\to\{\mathtt{f}:\mathbf{Abs}_{J'}\,;\rho_\gamma\}}$$

$$\text{T:ModField}\qquad \frac{J=\gamma \qquad J'=\gamma;(\mathtt{d},\mathbf{mod})}{\mathtt{d}\vdash \mathbf{modifies}\ \mathtt{f} : \forall\rho,\gamma.\{\mathtt{f}:\mathbf{Pre}_J\,;\rho_\gamma\}\to\{\mathtt{f}:\mathbf{Pre}_{J'}\,;\rho_\gamma\}}$$

$$\text{T:DelClass}\qquad \frac{J=\gamma \qquad J'=\gamma;(\mathtt{d},\mathbf{rem})}{\mathtt{d}\vdash \mathbf{removes\ class}\ \mathtt{c} : \forall\rho,\gamma,\rho',\gamma'.\{\mathtt{c}:\mathbf{Pre}_J(\{\rho_{\gamma'}\})\,;\rho'\}\to\{\mathtt{c}:\mathbf{Abs}_{J'}(\mathbf{Abs}_{\gamma'})\,;\rho'\}}$$

$$\text{T:AddClass}\qquad \frac{J=\gamma \qquad J'=\gamma;(\mathtt{d},\mathbf{add}) \qquad \mathtt{d},\gamma'\vdash FL:TC}{\mathtt{d}\vdash \mathbf{adds\ class}\ \mathtt{c}\ FL : \forall\rho,\gamma,\gamma'.\{\mathtt{c}:\mathbf{Abs}_J(\mathbf{Abs}_{\gamma'})\,;\rho\}\to\{\mathtt{c}:\mathbf{Pre}_{J'}(TC)\,;\rho\}}$$

$$\text{T:ModClass}\qquad \frac{J=\gamma \qquad J'=\gamma;(\mathtt{d},\mathbf{mod}) \qquad \mathtt{d}\vdash FOL:TC_1\to TC_2 \qquad \rho,\gamma\ \text{fresh}}{\mathtt{d}\vdash \mathbf{modifies\ class}\ \mathtt{c}\ FOL : \forall\rho,\gamma.\{\mathtt{c}:\mathbf{Pre}_J(TC_1)\,;\rho\}\to\{\mathtt{c}:\mathbf{Pre}J'(TC_2)\,;\rho\}}$$

$$\text{T:Seq}\qquad \frac{\mathtt{d}\vdash E:T_1\to T_2 \qquad \mathtt{d}\vdash E':T_2\to T_3}{\mathtt{d}\vdash E;E':T_1\to T_3} \qquad\qquad \text{T:Rew}\qquad \frac{\Lambda\vdash E:TP\to TP' \qquad TP'\blacktriangleright TP''}{\Lambda\vdash E:TP\to TP''}$$

$$\text{T:Inst}\ \frac{\Lambda\vdash E:\forall\rho.T}{\Lambda\vdash E:T} \qquad \text{T:Gen}\ \frac{\Lambda\vdash E:T}{\Lambda\vdash E:\forall\rho.T} \qquad \text{T:Subst}\ \frac{\Lambda\vdash E:T}{\Lambda\vdash E:\sigma(T)} \qquad \text{T:Equiv}\ \frac{\Lambda\vdash E:T \qquad T\equiv T'}{\Lambda\vdash E:T'}$$

Fig. 6: Typing Operators

Core Products. The typing rules for core products are presented in Figure 5. The rules for cores and classes have the form $\Phi\vdash E:T$, where Φ is a delta context, either a pair $(\mathtt{d},\gamma)$ or $\bot$, E is the typed term and T its type. When Φ is a pair $(\mathtt{d},\gamma)$, we are currently typing the addition of some classes performed by delta $\mathtt{d}$, where γ is the previous actions done on the elements. the annotation J is $\gamma;(\mathtt{d},\mathbf{add})$. Otherwise, when Φ is $\bot$, we are typing a core and the annotation is $\bot$. The rule T:FL types the body of a class with a mapping stating that all the fields of the class are present. The rule T:CL types a core product with a mapping stating that all the classes of the core are present, with their bodies typed with the previous rule.

Operators. The typing rules for operators are presented in Figure 6. Our typing statements for operators have the form $\mathtt{d}\vdash E:T$ where $\mathtt{d}$ is the delta in which the operators are declared, E is the typed operator and T is its type. The typing rules for operators on fields and core products are almost identical to the ones presented in our previous work [16], with the addition of the annotations J describing the actions performed by the operator on the typed element. For

$$\text{T:D}\qquad \frac{\mathtt{d} \vdash COL : TD \qquad Norm \vdash TD \qquad \Gamma; \mathtt{d} : TD \vdash PL : \Pi}{\Gamma \vdash \textbf{delta}\ \mathtt{d}\ \textbf{after}\ DL\ \{COL\}\ PL : \Pi}$$

$$\text{T:K}\qquad \frac{\bot \vdash CL : TP \qquad \Gamma; \mathtt{k} : TP \vdash PL : \Pi}{\Gamma \vdash \textbf{core}\ \mathtt{k}\ \{CL\}\ PL : \Pi}$$

$$\text{T:P}\qquad \frac{\Gamma \vdash DL : TP \to TP' \qquad \Gamma \vdash \mathtt{k} : TP \qquad \Gamma \vdash PL : \Pi}{\Gamma \vdash \textbf{product}\ \mathtt{p} = \{DL\}\ K\ PL : \Pi; \mathtt{p} : TP'}$$

$$\text{T:DL-D}\qquad \frac{\begin{array}{c}\Gamma \vdash \mathtt{d} : TP_2 \to TP_3 \qquad \Gamma \vdash D_1\ \ldots\ D_n : TP_1 \to TP_2 \\ \forall \mathtt{d}' \in D_1\ \ldots\ D_n,\ \mathtt{d} \not\prec_{PL} \mathtt{d}' \qquad \mathtt{detect}(\mathtt{d}, TP_3) = TP_4\end{array}}{\Gamma \vdash D_1\ \ldots\ D_n\ \mathtt{d} : TP_1 \to TP_4}$$

$$\text{T:N}\textsc{ame}\quad \Gamma \vdash \mathtt{n} : \Gamma(\mathtt{n}) \qquad\qquad \text{T:DL-E}\quad \Gamma \vdash \epsilon : \forall \rho.\{\rho\}\{\rho\} \qquad\qquad \text{T:E}\textsc{q}\quad \frac{PL \equiv_{dl} PL' \qquad PL : \Pi}{PL' : \Pi}$$

Fig. 7: Typing Product Lines

instance, the addition of a field `f` by a delta `d`, typed with the rule T:AddDield, states that the operator expects: as input, a class with the field `f` absent and annotated with a variable γ to capture manipulation done by previous deltas; as output, the same class with field `f` added (i.e. present) and with $(\mathtt{d}, \textbf{add})$ added to γ, thus capturing the addition action performed by `d` after the previous manipulations stored in γ.

To avoid duplication of typing rules, in the six last typing rules, we use E for any language term, T for any type and Λ for either a delta name `d` or a typing environments Γ that map delta names and core names to their types (these environments are used to type product lines). The rule T:Seq types the sequential composition of operators. The rules T:Equiv and T:Rew introduce the usage of the structural equivalence $\equiv$ and rewriting relation $\prec$ in our type system. Finally, the rules T:Inst, T:Gen and T:Subst deal with type generalization and substitution, which can be freely applied.

Product Lines. The type rules for product lines are presented in Figure 7. The typing judgements have the form $\Gamma \vdash E : T$, where Γ is the typing environment storing the type of deltas and cores, E is the typed term and T is its type in the context Γ. The type T can either be the type of a delta TD or a mapping Π between product names and their type. The rule T:D types delta declarations by: i) computing the type TD of the delta's body COL; ii) computing the action of the delta: the statement $Norm \vdash TD$ means that the annotations in TD have been completely rewritten by $\blacktriangleright$ (by construction of $\blacktriangleright$, every annotation thus corresponds to the action performed by the delta on the annotated element); and iii) continuing the typing of the product line with the environment Γ extended with a mapping between the delta and its type. The rule T:K types core declarations by typing the declared classes and continuing the typing of PL with the extended typing environment. The rule T:P types a product by typing the list of deltas DL, ensuring that the core `k` is a valid input for DL, and adding the type of the product to the rest of the mapping Π. The rule T:Name is used to type names, where `n` is either a core or a delta name.

The type of a list of delta names DL is constructed using the four last typing rules; it works as follows. Using the typing rule T:Eq based on the relation $\equiv_{dl}$, we sort the list of delta names to match $<_{PL}$: DL is thus replaced by DL', corresponding to a valid computation of the product. This sorting is enforced by the statement $\forall \mathtt{d}' \in D_1 \ \ldots \ D_n$, $\mathtt{d} \not<_{PL} \mathtt{d}'$ in rule T:DL-D. Then, the list is typed inductively using T:DL-E for the empty list and T:DL-D to add new deltas to the list. Finally, the rule T:DL-D types the deltas in sequence, with the additional application of the function `detect(d,`TP_3`)` to detect conflicts added or resolved by `d` using the annotations in TP_3. This function traverses type TP_3, applying `detectClass` on all annotations at the class level, and `detectField` on all annotations at the field level. We present in Figure 8 these two functions `detectClass` and `detectField` (we omit the straightforward presentation of `detect`) together with an helper function `check` which is used to check for hard conflicts between the delta in input and all the previous actions done on the element. Note that because there are soft conflicts only at the field level, the function `detectField` returns a modified annotation or an error `ERR` when an hard conflict is detected; `detectClass` returns only either `OK` when there are no conflicts, or `ERR` otherwise.

Properties The type system combines the properties of the classic row types system and the conflict detection performed using annotations.

Definition 9. *A type Π is said to be* conflict free *if there are no product name* `p`*, type context* $\mathbb{T}$ *and annotation J of the form* $J'; (\mathtt{d}_1 \ldots \mathtt{d}_n)$ *such that* $\Pi(\mathtt{p}) = \mathbb{T}[J]$*.*

Definition 10. *A product line PL has a* delta application error *iff during its reduction, either: i) a delta* `d` *that adds an element E is applied on a core that already contains E; ii) a delta* `d` *that removes or modifies an element E is applied on a core that does not contain E.*

Theorem 2 (Soundness). *Given a closed product line PL and type Π such that $\emptyset \vdash PL : \Pi$ holds, there exists PL' with $PL' \equiv_{dl} PL$ such that PL' does not have any delta application error.*

Proof. Consider the derivation K of $\emptyset \vdash PL : \Pi$. K can be transformed it into a derivation K' where we apply the rule T:Eq only once, at the end. Consider the term PL' typed by K' just before the application of T:Eq: this term will not have an error as our type system without T:Eq is more restrictive than [19].

Theorem 3 (Freedom). *Given a closed product line PL and conflict free type Π such that $\emptyset \vdash PL : \Pi$ holds. Then PL is conflict free.*

```
detectClass(d,$J$) {
 if $J$ = $J'$;$(\mathtt{d}, A)$ then
  if $J'$ = $J''$;$(\mathtt{d}', A')$ then
   if $\mathtt{d}' <_{PL} \mathtt{d}$ then
    OK
   else if $A$ = $A'$ = mod then
    check(d,$J''$)
   else ERR
  else OK
 else OK
}

check(d,$J$) {
 if $J$ = $J'$;$(\mathtt{d}', A)$ then
  if $\mathtt{d}' <_{PL} \mathtt{d}$ then OK
  else if $A$ = mod then
   check(d,$J'$)
  else ERR
 else if $J$ = $J'$;$(\mathtt{d}_1 \ldots \mathtt{d}_n)$ then
  if $\exists i, \mathtt{d}_i < \mathtt{d}$ then OK
  else check(d,$J'$)
 else OK
}
```

```
detectField(d,$J$) {
 if $J$ = $J'$;$(\mathtt{d}, A)$ then
  if $J'$ = $J''$;$(\mathtt{d}', A')$ then
   if $\mathtt{d}' <_{PL} \mathtt{d}$ then $J$
   else if $A$ = $A'$ = mod then
    if check(d,$J''$) = OK then
     $J''; (\mathtt{d}_1, \mathtt{d}_2)$
    else ERR
   else ERR
  else if $J'$ = $J''$;$(\mathtt{d}_1 \ldots \mathtt{d}_n)$ then
   $S \leftarrow \{\mathtt{d}_j \mid 1 \leq j \leq n \wedge \mathtt{d}_j \not<_{PL} \mathtt{d}\}$
   if $S$ = $\emptyset$ then
    $J$
   else if $A$ = mod then
    if check(d,$J'$) = OK then
     $S' \leftarrow \{\mathtt{d}_j \mid 1 \leq j \leq n\} \setminus S$
     $J''; (S'); (\mathtt{d}, S)$
    else ERR
   else ERR
  else $J$
 else $J$
}
```

Fig. 8: Conflict Detection function

5 Related Work

The goal of type checking the code base of a software product line is to ensure that the generated products are type safe, up to the degree of type safety provided by the base language, *without* having to actually generate the products. Other static analysis techniques can instead be employed to check for other potential deficiencies, without aiming to be ensure complete type safety.

Thaker et al. [23] describe an informally specified approach to the safe composition of software product lines that guarantees that no reference to an undefined class, method or variable will occur in the resulting products. The approach is presented modulo variability given in the feature model and deals especially with the resulting combinatorics. The lack of a comprehensive formal model of the underlying language and type system was rectified with *Lightweight Feature Java* (LFJ) [8]. Underlying LFJ is a constraint-based type system whose constraints describe composition order, the uniqueness of fields and methods, the presence of field and methods along with their types, and feature model dependencies. The soundness of LFJ's type system was validated using theorem prover Coq.

A formal model of a feature-oriented Java-like language called *Featherweight Feature Java* (FFJ) [2] presents a similar base language that also formalizes Thaker et al. [23]'s approach to safe composition, although for this system type checking occurs only on the generated product. *Coloured Featherweight Java* [11], which employs a notion of colouring of code analogous to but more advanced than `#ifdef`s, lifts type checking from individual products to the level of the product line and guarantees that all generated products are type safe. More recent work [1] refines the work on FFJ, expressing code refinements as modules rather than low-level annotations. The resulting type system again works at the level of the product line and enjoys soundness and completeness results, namely, that a product line is well-typed if and only if all of its derived products are well-typed.

In the above mentioned work the refinement mechanisms are monotonic, so no method/class removal or renaming is possible. Kuhlemann et al. [14] addresses the problem of non-monotonic refinements, though their approach does not consider type safety. They consider the presence of desired attributes depending upon which features are selected. Checking is implemented as an encoding into propositional formulas, which are fed into a SAT solver. Recent work addresses non-monotonic refinement mechanisms that can remove or rename classes and methods. An alternative approach due to Schaefer *et al.* [20] generate detailed dependency constraints for checking delta-oriented software product lines. The checking of the constraint is performed per product, rather than at the level of product lines. This approach to typing delta-oriented programs is complementary to our work, providing part of the checking we have omitted.

A number of static analysis techniques have been developed for the design models or code of software product lines. Heidenreich [10] describes techniques for ensuring that the correspondence between feature models, solution-space models, and problem-space models, which is realised in the FeatureMapper tool. In this tool, models are checked for well-formedness against their meta-model. Similarly, Czarnecki and Pietroszek [7] provide techniques for ensuring that no ill-structured instance of a feature-based model template will be generated from a correct configuration. Apel et al. [3] present a general, language independent, static analysis framework for reference checking—checking which dependencies are present and satisfied. This is one of the key tasks of type checking a software product line. Similar ideas are applied in a language-independent framework for ensuring the syntactic correctness of all product line variants by checking only the product line itself, again without having to generate all the variants [12]. Clarke et al. [4] present an abstract framework for describing about conflicts between code refinements and conflict resolution in the setting of delta-oriented programming. Padmanabhan and Lutz [18] describe the DECIMAL tool, which performs a large variety of consistency checks on software product line requirements specifications, in particular, when a new feature is added to an existing system. Techniques developed for the analysis and resolution of interference of aspects in AOP [13, 9] address similar problems to analyses of software product line conflicts, but they do not consider variability.

6 Conclusion

This paper presented a simple language for delta-oriented programming and defines notions soft and hard conflicts, a type system based on row polymorphism to capture errors and on a new concept of annotations to capture conflicts. This paper also shows that, in contrast to Clarke et al. [4], the notion of conflict is not simple and accurately detecting them is not easy. Much work remains to be done. First, we need to extend our type system to ensure type safety not only of delta application, but also of the generated products. Then, we need to extend our calculus and type system to include features models. Whether this can be done while keeping the time complexity of our type system polynomial in the number of deltas remains to be seen.

References

1. Sven Apel, Christian Kästner, Armin Größlinger, and Christian Lengauer. Type safety for feature-oriented product lines. *Autom. Softw. Eng.*, 17(3):251–300, 2010.
2. Sven Apel, Christian Kästner, and Christian Lengauer. Feature Featherweight Java: A calculus for feature-oriented programming and stepwise refinement. In Yannis Smaragdakis and Jeremy G. Siek, editors, *GPCE*, pages 101–112. ACM, 2008.
3. Sven Apel, Wolfgang Scholz, Christian Lengauer, and Christian Kästner. Language-independent reference checking in software product lines. In *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development*, FOSD '10, pages 65–71, New York, NY, USA, 2010. ACM.
4. Dave Clarke, Michiel Helvensteijn, and Ina Schaefer. Abstract delta modeling. In *Proceedings of the ninth international conference on Generative programming and component engineering*, GPCE '10, pages 13–22, New York, NY, USA, 2010. ACM.
5. Dave Clarke, Radu Muschevici, José Proença, Ina Schaefer, and Rudolf Schlatte. Variability modelling in the abs language. In Bernhard K. Aichernig, Frank S. de Boer, and Marcello M. Bonsangue, editors, *FMCO*, volume 6957 of *Lecture Notes in Computer Science*, pages 204–224. Springer, 2010.
6. Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.
7. Krzysztof Czarnecki and Michal Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In Robert Glück and Michael R. Lowry, editors, *GPCE*, volume 3676 of *Lecture Notes in Computer Science*, pages 422–437. Springer, 2005.
8. Benjamin Delaware, William R. Cook, and Don S. Batory. Fitting the pieces together: a machine-checked model of safe composition. In Hans van Vliet and Valérie Issarny, editors, *ESEC/SIGSOFT FSE*, pages 243–252. ACM, 2009.
9. Rémi Douence, Pascal Fradet, and Mario Südholt. A framework for the detection and resolution of aspect interactions. In Don S. Batory, Charles Consel, and Walid Taha, editors, *GPCE*, volume 2487 of *Lecture Notes in Computer Science*, pages 173–188. Springer, 2002.
10. Florian Heidenreich. Towards systematic ensuring well-formedness of software product lines. In *Proceedings of the 1st Workshop on Feature-Oriented Software Development*, pages 69–74, New York, NY, USA, oct 2009. ACM.

11. Christian Kästner and Sven Apel. Type-checking software product lines - a formal approach. In *ASE*, pages 258–267. IEEE, 2008.
12. Christian Kästner, Sven Apel, Salvador Trujillo, Martin Kuhlemann, and Don S. Batory. Guaranteeing syntactic correctness for all product line variants: A language-independent approach. In Manuel Oriol and Bertrand Meyer, editors, *TOOLS (47)*, volume 33 of *Lecture Notes in Business Information Processing*, pages 175–194. Springer, 2009.
13. Emilia Katz and Shmuel Katz. Incremental analysis of interference among aspects. In Curtis Clifton, editor, *FOAL*, pages 29–38. ACM, 2008.
14. Martin Kuhlemann, Don S. Batory, and Christian Kästner. Safe composition of non-monotonic features. In Jeremy G. Siek and Bernd Fischer, editors, *GPCE*, pages 177–186. ACM, 2009.
15. Michael Lienhardt and Dave Clarke. Conflict detection in delta-oriented programming. Technical report, University of Bologna, 2012. `http://proton.inrialpes.fr/~mlienhar/reports/2012-Conflict-Detection.pdf`.
16. Michaël Lienhardt and Dave Clarke. Row types for delta-oriented programming. In *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS '12, pages 121–128, New York, NY, USA, 2012. ACM.
17. Michael Lienhardt, Claudio Antares Mezzina, Alan Schmitt, and Jean-Bernard Stefani. Typing component-based communication systems. In *11th Formal Methods for Open Object-Based Distributed Systems and 29th Formal Techniques for Networked and Distributed Systems*. Springer, 2009.
18. Prasanna Padmanabhan and Robyn R. Lutz. Tool-supported verification of product line requirements. *Autom. Softw. Eng.*, 12(4):447–465, 2005.
19. Didier Rémy. *Type inference for records in natural extension of ML*, pages 67–95. MIT Press, Cambridge, MA, USA, 1994.
20. Ina Schaefer, Lorenzo Bettini, and Ferruccio Damiani. Compositional type-checking for delta-oriented programming. In *Proceedings of the tenth international conference on Aspect-oriented software development*, AOSD '11, pages 43–56, New York, NY, USA, 2011. ACM.
21. Ina Schaefer, Lorenzo Bettini, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. In *Proceedings of the 14th international conference on Software product lines: going beyond*, SPLC'10, pages 77–91, Berlin, Heidelberg, 2010. Springer-Verlag.
22. Ina Schaefer and Ferruccio Damiani. Pure delta-oriented programming. In *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development*, FOSD '10, pages 49–56, New York, NY, USA, 2010. ACM.
23. Sahil Thaker, Don S. Batory, David Kitchin, and William R. Cook. Safe composition of product lines. In Charles Consel and Julia L. Lawall, editors, *GPCE*, pages 95–104. ACM, 2007.