Pure Delta-oriented Programming [*]

Ina Schaefer[1] [†] Ferruccio Damiani[2]

[1]Chalmers University of Technology, 421 96 Gothenburg, Sweden

[2]Dipartimento di Informatica, Università di Torino, C.so Svizzera, 185 - 10149 Torino, Italy

Abstract

Delta-oriented programming (DOP) is a modular approach for implementing software product lines. Delta modules generalize feature modules by allowing removal of functionality. However, DOP requires to select one particular product as core product from which all products are generated. In this paper, we propose *pure delta-oriented programming* (Pure DOP) that is a conceptual simplification of traditional DOP. In Pure DOP, the requirement of one designated core product is dropped. Instead, program generation only relies on delta modules comprising program modifications such that Pure DOP is more flexible than traditional DOP. Furthermore, we show that Pure DOP is a true generalization of FOP and supports proactive, reactive and extractive product line engineering.

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 Software Product Line, Feature-oriented Programming, Delta-oriented Programming, Program Generation

1. Introduction

A *software product line* (SPL) is a set of software systems with well-defined commonalities and variabilities [12, 27]. The approaches to implementing SPL in the object-oriented paradigm can be classified into two main directions [19]. First, annotative approaches (e.g., [4, 17]) mark the source code of all products with respect to product features and remove marked code for particular feature configurations. Second, *compositional approaches* [23], associate code fragments to product features that are assembled to implement a given feature configuration.

Feature-oriented programming (FOP) [7] is a prominent approach for implementing SPLs by composition of *feature modules*. A feature module directly corresponds to a product feature. In the context of object-oriented programming, feature modules can introduce new classes or refine existing ones by adding fields and methods or by overriding existing methods. In *delta-oriented programming* (DOP) [29], feature modules are generalized to delta modules that additionally allow the removal of classes, fields and methods and that can refer to any combination of features. DOP requires selecting one particular product as designated core product. The core product is implemented in the core module. From this core module, all other products are generated by delta module application. However, the requirement of the core product makes it difficult to deal with product line evolution, for instance, if the product line evolves such that the original core product is no longer a valid product. Furthermore, the uniquely determined core product prevents a true generalization of FOP by DOP, since feature module composition in FOP may start from several different base feature modules that may not correspond to valid products.

In this paper, we propose *pure delta-oriented programming* (Pure DOP) as a conceptual simplification of traditional DOP [29], which we will call Core DOP in the following. In Pure DOP, the requirement to chose one product as core product is dropped. Instead, only delta modules are used for product generation. Thus, we call the approach *Pure DOP*. A delta module can specify additions, removals classes or modifications of classes. In order to define a product line over a set of delta modules, each delta module is attached an application condition determining for which feature configurations the modifications of the delta module have to be applied. This creates the connection between the modifications of the delta modules and the product features [16]. Additionally, the delta modules can be partially ordered to ensure that for every feature configuration a uniquely defined product is generated.

The contribution of this work is twofold. First, Pure DOP relaxes the requirement of a single valid core product. This makes Pure DOP more flexible than Core DOP [29]. Pure DOP is a true generalization of FOP since every FOP product line can be understood as a Pure DOP product line which is not obvious for Core DOP. Further, Pure DOP supports proactive, reactive and extractive product line development [22] by allowing program generation from any set of existing legacy product implementations which is not directly possible with Core DOP. Second, in the presentation of (Pure) DOP given in this paper, the application conditions for delta modules, as well as the delta module ordering, are only defined when a product line is specified. In contrast, in the traditional presentation of (Core) DOP [29], application conditions and ordering are fixed for each delta module. The separation of application conditions and application ordering from the specification of the modifications in a delta module increases the reusability of delta modules and allows developing different product lines over the same set of delta modules.

The paper is organized as follows: In Section 2, we present Pure DOP of JAVA programs and show its formalization LPΔJ using LJ (LIGHTWEIGHT JAVA) [32] as base language for the generated products in Section 3. We show that Pure DOP is a

[*] Work partially supported by the German-Italian University Centre (Vigoni program) and by MIUR (PRIN 2009 DISCO).

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

FOSD'10 October 10, 2010, Eindhoven, The Netherlands
Copyright © 2010 ACM 978-1-4503-0208-1/10/10. . . $10.00

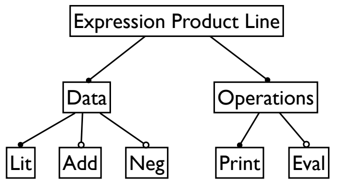

Figure 1. Feature model for Expression Product Line

true generalization of FOP by providing an embedding of LFJ (LIGHTWEIGHT FEATURE JAVA) [13] into LPΔJ in Section 4. We demonstrate that Pure DOP supports proactive, extractive and reactive SPLE in Section 5. We show that Pure DOP is a conceptual simplification of Core DOP in Section 6.

2. Pure Delta-oriented Programming

In order to illustrate the main concepts of Pure DOP, we use the *expression product line* (EPL) as described in [23]. The EPL is based on the *expression problem* [35], an extensibility problem, that has been proposed as a benchmark for data abstractions capable to support new data representations and operations. We consider the following grammar:

```
Exp ::= Lit | Add | Neg
Lit ::= <non−negative integers>
Add ::= Exp "+" Exp
Neg ::= "-" Exp
```

Two different operations can be performed on the expressions described by this grammar: printing, which returns the expression as a string, and evaluation, which computes the value of the expression. The products in the EPL can be described by two feature sets, the ones concerned with data Lit, Add, Neg and the ones concerned with operations Print and Eval. Lit and Print are mandatory features. The features Add, Neg and Eval are optional. Figure 1 shows the feature model [16] of the EPL.

Pure Delta Modules The main concept of pure DOP are delta modules which are containers of modifications to an object-oriented program. The modifications inside a delta module act on the class level by adding, removing and modifying classes. A class can be modified by changing the super class, by adding and removing fields and methods and by modifying methods. The modification of a method can either replace the method body by another implementation, or wrap the existing method using the **original** construct. The **original** construct expresses a call to the method with the same name before the modifications and is bound at the time the product is generated. Before or after the **original** construct, other statements can be introduced wrapping the existing method implementation. The **original** construct (similar to the **Super**() call in AHEAD [7]) avoids a combinatorial explosion of the number of delta modules in case the original method has to be wrapped differently for a set of optional features. Listing 1 contains the delta module for introducing the Lit feature. Listing 2 contains the delta modules for incorporating the Print and Eval features by modification of the class `Lit`.

Pure Delta-oriented Product Lines The delta-oriented specification of a product line comprises the set of product features, the set of valid feature configurations and the set of delta modules necessary to implement all valid products. Furthermore, the specification of a product line in Pure DOP associates each delta module with the set of features configurations in which the delta modules has to be applied by attaching an application condition in a **when** clause. The application condition is a propositional constraint over the set

```
delta DLit{
  adds interface Exp {
  }
  adds class Lit implements Exp {
    int value;
    Lit(int n) { value = n; }
  }
}
```

Listing 1: Delta module for Lit feature

```
delta DLitPrint{
  modifies interface Exp {
    void print();
  }
  modifies class Lit implements Exp {
   adds void print() { System.out.println(value); }
  }
}

delta DLitEval{
  modifies interface Exp {
    adds eval();
  }
  modifies class Lit {
    adds int eval() { return value; }
  }
}
```

Listing 2: Delta modules for Print and Eval features

```
delta DAdd  {
  adds class Add implements Exp {
    Exp expr1;
    Exp expr2;
    Add(Exp a, Exp b) { expr1 = a; expr2 = b; }
  }
}

delta DAddPrint {
  modifies class Add {
    adds void print() { expr1.print(); System.out.print(" + "); expr2.print(); }
  }
}

delta DAddEval {
  modifies class Add {
    adds int eval() { return expr1.eval() + expr2.eval(); }
  }
}
```

Listing 3: Delta modules for Add, Print and Eval features

```
delta DNeg  {
  adds class Neg implements Exp {
   Exp expr;
   Neg(Exp a) { expr1 = a; }
  }
}

delta DNegPrint {
  modifies class Neg {
    adds void print() { System.out.print("-("); expr.print(); System.out.print(")");}
  }
}

delta DNegEval{
   modifies class Neg {
     adds int eval() { return (−1) ∗ expr.eval(); }
   }
}
```

Listing 4: Delta modules for Neg, Print and Eval features

```
features Lit, Add, Neg, Print, Eval
configurations Lit & Print
deltas
  [ DLit,
    DAdd when Add,
    DNeg when Neg ]

  [ DPrint,
    DEval when Eval,
    DAddPrint when Add,
    DAddEval when (Add & Eval),
    DNegPrint when Neg,
    DNegEval when (Neg & Eval) ]
```

Listing 5: Pure DOP specification of the EPL

of features. Since only feature configurations which are valid according to the feature model are used for program generation, the application conditions attached to delta modules have to be understood as a conjunction with the formula describing the set of valid feature configurations.[1] The application condition creates the link from the features in the feature model to the delta modules. In this way, we can specify delta modules for combinations of features to solve the optional feature problem [20].

In order to obtain a product for a particular feature configuration, the modifications specified in the delta modules with valid application conditions are applied incrementally to the previously generated product. The first delta module is applied to the empty product. All other delta modules are applied to the respective intermediate product. The modifications of a delta model are applicable to a (possibly empty) product if each class to be removed or modified exists and, for every modified class, if each method or field to be removed exists, if each method to be modified exists and has the same header as the modified method, and if each class, method or field to be added does not exist. During product generation, every delta module must be applicable. Otherwise, the resulting product is undefined. In particular, the first delta module that is applied can only contain additions.

In order to ensure that each delta module is applicable during product generation, the delta modules are ordered in the specification of a pure delta-oriented product line. The order of delta module application is defined by a total order on a partition of the set of delta modules. Deltas in the same partition can be applied in any order to the previous product, but the order of the partitions is fixed. The ordering captures semantic requires relations that are necessary for the applicability of the delta modules.

Listing 5 shows a delta-oriented specification of the EPL. In this specification, application conditions are attached to the delta modules that are required to implement the different products of the EPL. The used delta modules are depicted in Listings 1, 2, 3 and 4. The order of delta module application is defined by an ordered list of the delta module sets which are enclosed by `[ .. ]`.

Product Generation The generation of a product for a given feature configuration consists of two steps, performed automatically:

1. Find all delta modules with a valid application condition; and
2. Apply the selected delta modules to the empty product in any linear ordering that is consistent with the total order on the partitioning of the delta modules.

If two delta modules add, remove or modify the same class, the ordering in which the delta modules are applied can influence the resulting product. However, for a product line implementation, it is essential to guarantee that for every valid feature configuration exactly one product is generated. This property is called *unambiguity* of the product line. For unambiguity, the delta modules in each partition must be compatible. This means that if one delta module in a partition adds or removes a class, no other delta module in the same partition may add, remove or modify the same class, and the modifications of the same class in different delta modules in the same partition have to be disjoint. Defining the order of delta module application by a total ordering on a delta module partition provides an efficient way to ensure unambiguity, since only the compatibility of each partition has to be checked.

[1] In the examples the valid feature configurations are represented by a propositional formula over the set of features. Other representations are possible (see, e.g., [5] for a discussion of possible representations).

$$\begin{array}{lcll}
\mathrm{DMD} & ::= & \textbf{delta}\ \delta\ \{\overline{\mathrm{DC}}\} & \text{delta module} \\
\mathrm{DC} & ::= & \textbf{adds}\ \mathrm{cd}\ \mid & \text{delta clause} \\
 & & \textbf{modifies}\ \mathrm{C}\ [\textbf{extending}\ \mathrm{C}]\ \{\ \overline{\mathrm{DS}}\ \}\ \mid & \\
 & & \textbf{removes}\ \mathrm{C} & \\
\mathrm{DS} & ::= & \textbf{adds}\ \mathrm{fd}\ \mid & \text{delta subclause} \\
 & & \textbf{adds}\ \mathrm{md}\ \mid & \\
 & & \textbf{modifies}\ \mathrm{md}\ \mid & \\
 & & \textbf{modifies}\ \mathrm{wmd}\ \mid & \\
 & & \textbf{removes}\ \mathrm{a} & \\
\mathrm{wmd} & ::= & \mathrm{ms}\ \{\bar{\mathrm{s}};\ \textbf{original}();\ \bar{\mathrm{s}};\ \textbf{return}\ \mathrm{y};\} & \text{method wrapper}
\end{array}$$

Figure 2. LPΔJ: syntax of delta modules

3. A Kernel Calculus for Pure Delta Modules

In this section, we introduce the syntax and the semantics of LPΔJ (LIGHTWEIGHT PURE DELTA JAVA), a kernel calculus for Pure DOP of product lines of JAVA programs. LPΔJ is based on LJ (LIGHTWEIGHT JAVA) [32]. Thus, it is particularly suitable for comparison with the formalization of FOP in LFJ (LIGHTWEIGHT FEATURE JAVA) [13].

LPΔJ *Syntax* The syntax of LPΔJ, as an extensions to LJ, is given in Figure 2. Following [15], we use the overline notation for possibly empty sequences. For instance, we write "$\bar{\mathrm{s}}$;" as short for a possibly empty sequence of statements "$\mathrm{s}_1;\ldots\mathrm{s}_n;$" and "$\overline{\mathrm{DC}}$" as short for a possibly empty sequence of delta clause definitions "$\mathrm{DC}_1\ldots\mathrm{DC}_n$". Sequences of named elements (like delta clause or delta subclause definitions) are assumed to contain no duplicate names (that is, the names of the elements of the sequence must be distinct). The constructs for class definitions cd, field definitions fd, method definitions md, method signatures ms and statement s are those of LJ [32] (and of LFJ [13]). The metavariable δ ranges over delta module names.

A delta module definition DMD for a delta module with the name δ can be understood as a mapping from class names to delta clause definitions. A delta clause definition DC can specify the addition, removal or modification of a class. The adds-domain, the removes-domain and the modifies-domain of a delta module definition DMD are defined as follows:

$$\begin{array}{lcl}
\mathit{addsDom}(\mathrm{DMD}) & = & \{\mathtt{C} \mid \mathrm{DMD}(\mathtt{C}) = \textbf{adds class}\ \mathtt{C}\ \cdots\} \\
\mathit{removesDom}(\mathrm{DMD}) & = & \{\mathtt{C} \mid \mathrm{DMD}(\mathtt{C}) = \textbf{removes}\ \mathtt{C}\} \\
\mathit{modifiesDom}(\mathrm{DMD}) & = & \{\mathtt{C} \mid \mathrm{DMD}(\mathtt{C}) = \textbf{modifies}\ \mathtt{C}\ \cdots\}
\end{array}$$

The modification of a class is defined by possibly changing the super class and by listing a sequence of delta subclauses DS defining modifications of methods and additions/removals of fields and methods. A delta modifies clause DC can be understood as a mapping from the keyword **extending** to an either empty or singleton set of class names and from field/method names to delta subclauses. The adds-, removes- and modifies-domain of a delta modifies-clause DC are defined as follows:

$$\begin{array}{lcl} \mathit{addsDom}(\mathrm{DC}) & = & \{\mathrm{a} \mid \mathrm{DMT}(\mathrm{a}) = \textbf{adds} \cdots \mathrm{a} \cdots\} \\ \mathit{removesDom}(\mathrm{DC}) & = & \{\mathrm{a} \mid \mathrm{DMT}(\mathrm{a}) = \textbf{removes}\ \mathrm{a}\} \\ \mathit{modifiesDom}(\mathrm{DC}) & = & \{\mathrm{m} \mid \mathrm{DMT}(\mathrm{m}) = \textbf{modifies} \cdots \mathrm{m} \cdots\} \end{array}$$

The modification of a method, defined by a delta modifies subclause, can either replace the method body by another implementation, or wrap the existing method using the **original**() call. In both cases, the modified method must have the same header as the unmodified method. The **original**() call may only occur in the body of the method provided by a delta modifies subclause **modifies** wmd. The occurrence of **original**() represents a call to the unmodified method where the formal parameters of the modified method are passed implicitly as arguments. In LFJ [13], the `Super()` construct of AHEAD [7] is modeled in the same way.

After we have defined the notion of delta modules over LJ, we can formalize LPΔJ product lines. We use the metavariables φ and ψ to range over feature names. We write $\overline{\psi}$ as short for the set $\{\overline{\psi}\}$, i.e., the feature configuration containing the features $\overline{\psi}$. A *delta module table* DMT is a mapping from delta module names to delta module definitions. A LPΔJ SPL is a 5-tuple $\mathrm{L} = (\overline{\varphi}, \Phi, \mathrm{DMT}, \Gamma, <_{\mathrm{DMT}})$ consisting of:

1. the features $\overline{\varphi}$ of the SPL,
2. the set of the valid feature configurations $\Phi \subseteq \mathscr{P}(\overline{\varphi})$,[2]
3. a delta module table DMT containing the delta modules,
4. a mapping $\Gamma : \mathit{dom}(\mathrm{DMT}) \to \Phi$ determining for which feature configurations a delta module must be applied (which is denoted by the **when** clause in the concrete examples),
5. a total order $<_{\mathrm{DMT}}$ on a partition of $\mathit{dom}(\mathrm{DMT})$, called the application partial order, determining the order of delta module application.

To simplify notation, in the following we always assume a *fixed* SPL $\mathrm{L} = (\overline{\varphi}, \Phi, \mathrm{DMT}, \Gamma, <_{\mathrm{DMT}})$. We further assume that the SPL L satisfies the following sanity conditions.

(*i*) For every class name `C` (except `Object`) appearing in DMT, we have $\mathtt{C} \in (\cup_{\delta \in \mathit{dom}(\mathrm{DMT})} \mathit{addsDom}(\mathrm{DMT}(\delta)))$, meaning that every class is added at least once.

(*ii*) The mapping $\Delta : \Phi \to \mathscr{P}(\mathit{dom}(\mathrm{DMT}))$, such that $\Delta(\overline{\psi})$, the set of names of delta modules whose application condition is satisfied by the feature configuration $\overline{\psi}$, is injective and such that $(\cup_{\overline{\psi} \in \Phi} \Delta(\overline{\psi})) = \mathit{dom}(\mathrm{DMT})$, i.e., for every feature configuration a different set of delta modules is applied and every delta module is applied for at least one feature configuration.

In the following, we write $\mathit{dom}(\delta)$ as short for $\mathit{dom}(\mathrm{DMT}(\delta))$, and we write $\delta(\mathtt{C})$ as short for $\mathrm{DMT}(\delta)(\mathtt{C})$.

LPΔJ *Product Generation* A LJ program can be represented by a *class table*. A class table CT is a mapping from class names to class definitions. A delta module is *applicable* to a class table CT if each class to be removed or modified exists and, for every delta modifies clause, if each method or field to be removed exists, if each method to be modified exists and has the same header specified in method modifies subclause, and if each class, method or field to be added does not exist.

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

[2] The calculus abstracts from the concrete representation of the feature model.

$$\begin{array}{llll} \mathrm{FMD} & ::= & \textbf{feature}\ \varphi\ \{\overline{\mathrm{cd}}\ \overline{\mathrm{rcd}}\} & \text{feature module} \\ \mathrm{rcd} & ::= & \textbf{refines class}\ \mathrm{C}\ \textbf{extending}\ \mathrm{C}\ \{\ \overline{\mathrm{fd}};\ \overline{\mathrm{md}}\ \overline{\mathrm{rmd}}\ \} & \text{class refinement} \\ \mathrm{rmd} & ::= & \textbf{refines}\ \mathrm{ms}\ \{\overline{\mathrm{s}};\ \textbf{Super}();\ \overline{\mathrm{s}};\ \textbf{return}\ \mathrm{y};\} & \text{method refinement} \end{array}$$

Figure 3. LFJ: syntax of feature modules

$$\mathrm{CT}'(\mathrm{C}) = \begin{cases} \mathrm{CT}(\mathrm{C}) & \text{if } \mathrm{C} \notin \mathit{dom}(\mathrm{DMT}(\delta)) \\ \mathrm{CD} & \text{if } \delta(\mathrm{C}) = \textbf{adds}\ \mathrm{CD} \\ \textsc{apply}(\delta(\mathtt{C}), \mathrm{CT}(\mathtt{C})) & \text{if } \mathtt{C} \in \mathit{modifiesDom}(\delta) \end{cases}$$

where $\textsc{apply}(\delta(\mathtt{C}), \mathrm{CT}(\mathtt{C}))$, the application of the delta clause $\delta(\mathtt{C}) = \mathrm{DC} = \textbf{modifies}\ \mathtt{C} \cdots \{\cdots\}$ to the class definition $\mathrm{CT}(\mathrm{C}) = \mathrm{CD}$, is the class definition CD' defined as follows:

$$\mathrm{CD}'(\textbf{extends}) = \begin{cases} \mathrm{CD}(\textbf{extends}) & \text{if } \mathrm{DC}(\textbf{extending}) = \emptyset \\ \mathrm{C}' & \text{if } \mathrm{DC}(\textbf{extending}) = \{\mathrm{C}'\} \end{cases}$$

$$\mathrm{CD}'(\mathrm{a}) = \begin{cases} \mathrm{CD}(\mathrm{a}) & \text{if } \mathrm{a} \notin \mathit{dom}(\mathrm{DC}) \\ \mathrm{AD} & \text{if } \mathrm{DC}(\mathrm{a}) = \textbf{adds}\ \mathrm{AD} \\ \mathtt{MD}[\overline{\mathrm{s}}/\textbf{original}()] & \text{if } \mathrm{DC}(\mathrm{a}) = \textbf{modifies}\ \mathtt{MD} \\ & \quad \text{and } \mathrm{CD}(\mathrm{a}) = \cdots \mathrm{a}(\cdots)\{\overline{\mathrm{s}};\ \textbf{return}\ \mathrm{y};\} \end{cases}$$

The semantics of the **original**() call is captured by replacing the occurrence of **original**() in the method body specified by the modifies subclause with the body of the unmodified method.

For any given total order of delta module application, a LPΔJ SPL 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 delta modules $\Delta(\overline{\psi})$ to the empty class table according to the given order. The product generation mapping can be partial since a non-applicable delta module may be encountered during product generation such that the resulting product is undefined.

Unambiguous and Type-Safe* LPΔJ *Product Lines A LPΔJ SPL is *unambiguous* if all total orders of delta modules that are compatible with the application partial order define the same product generation mapping. In an unambiguous SPL, for every feature configuration at most one product implementation is generated.

In order to find a criterion for unambiguity, we define the notion of compatibility of a set of delta modules. A set of delta modules is called *compatible* if no class added or removed in one delta module is added, removed or modified in another delta module contained in the same set, and for every class modified in more than one delta module, its direct superclass is changed at most by one delta clause and the fields and methods added, modified or removed are distinct. For a set of compatible delta modules, any order of delta module application yields the same class table since the alterations in compatible delta modules do not interfere with each other.

A SPL is *locally unambiguous* if every set S of delta modules in the partition of $\mathit{dom}(\mathrm{DMT})$ provided by the application partial order $<_{\mathrm{DMT}}$ is compatible. If the SPL L is locally unambiguous, then it is unambiguous. Local unambiguity can be checked by inspecting the delta modules in each partition only once.

A LPΔJ SPL is *type-safe* if the following conditions hold: (i) its product generation mapping is total, (ii) it is locally unambiguous, and (iii) all generated products are well-typed LJ programs.

4. Generalization of FOP

In this section, we show that Pure DOP is generalization of FOP [7] by providing a mapping from LFJ [13] into LPΔJ.

4.1 Recalling LFJ

The syntax of the LFJ extensions to LJ is given in Figure 3. It is taken from [13]. A feature module definition FMD contains the

$$
\begin{aligned}
&[\![\textbf{feature}\ \varphi\ \{\overline{\mathrm{cd}}\ \overline{\mathrm{rcd}}\}]\!] = \\
&\qquad \textbf{delta}\ \varphi\ \{\ \overline{\textbf{adds}\ \mathrm{cd}}\ \ \overline{[\![\mathrm{rcd}]\!]}\ \} \\
&[\![\textbf{refines class}\ \mathrm{C}\ \textbf{extending}\ \mathrm{C}\ \{\ \overline{\mathrm{fd}};\ \ \overline{\mathrm{md}}\ \ \overline{\mathrm{rmd}}\ \}]\!] = \\
&\qquad \textbf{modifies}\ \mathrm{C}\ \textbf{extending}\ \mathrm{C}\ \{\ \overline{\textbf{adds}\ \mathrm{fd}}\ \ \overline{\textbf{adds}\ \mathrm{md}}\ \ \overline{[\![\mathrm{rmd}]\!]}\ \} \\
&[\![\textbf{refines}\ \mathrm{ms}\ \{\overline{\mathrm{s}};\ \textbf{Super}();\ \overline{\mathrm{s}};\ \textbf{return}\ \mathrm{y};\}]\!] = \\
&\qquad \textbf{modifies}\ \mathrm{ms}\ \{\overline{\mathrm{s}};\ \textbf{original}();\ \overline{\mathrm{s}};\ \textbf{return}\ \mathrm{y};\}
\end{aligned}
$$

Figure 4. Translation of a feature module to a delta module

feature φ and a set of class definitions $\overline{\mathrm{cd}}$ and class refinement definitions $\overline{\mathrm{rcd}}$. Class definitions are given according to the syntax of LJ. A class refinement definition can change the superclass, add fields $\overline{\mathrm{fd}}$, provide new method definitions $\overline{\mathrm{md}}$ and refine existing method definitions $\overline{\mathrm{rmd}}$. A method refinement can wrap the existing method body using the **Super**() construct.

A *feature module table* FMT is a mapping from feature names to feature module definitions. A LFJ product line can be described by a 3-tuple $\mathrm{L} = (\mathrm{FMT}, \Phi, <_{\mathrm{FMT}})$ consisting of:

1. a feature module table FMT with a feature module for each feature of the SPL,
2. the set of the valid feature configurations $\Phi \subseteq \mathscr{P}(dom(\mathrm{FMT}))$,
3. a total order $<_{\mathrm{FMT}}$ on the set of features $dom(\mathrm{FMT})$.

The product associated to a feature configuration $\overline{\psi}$ is generated by composing (see Section 3.1 of [13]) the feature modules associated to the features in $\overline{\psi}$ according to the total order $<_{\mathrm{FMT}}$. During feature module composition, newly defined classes, fields and methods are added and class and method refinements are carried out. According to [13], a LFJ product line is *type-safe* if all generated products are well-typed LJ programs.

4.2 Mapping LFJ into LPΔJ

A product line in FOP can be represented as a product line in Pure DOP. The set of features and the set of valid feature configurations in both product lines is the same. Every feature module in a LFJ product line is mapped to a delta module where additions are translated to adds clauses and refinements to modifies clauses. The application condition of the delta module denotes all configurations in which the respective feature is contained. The ordering of delta module application is the total ordering of the feature modules.

Formally, the mapping from LFJ product lines to LPΔJ product lines is defined as follows: for a LFJ product line $\mathrm{L} = (\mathrm{FMT}, \Phi, <_{\mathrm{FMT}})$, $[\![\mathrm{L}]\!]$ denotes the corresponding LPΔJ product line $(\overline{\varphi}, \Phi, \mathrm{DMT}, \Gamma, <_{\mathrm{DMT}})$ where

- $\overline{\varphi} = dom(\mathrm{FMT}) = dom(\mathrm{DMT})$,
- The delta module table DMT is obtained by translating each feature module in FMT to a delta module with the same name, according to the clauses in Figure 4,
- $\Gamma : dom(\mathrm{DMT}) \rightarrow \Phi$, where $\Gamma(\varphi) = \{\overline{\psi} \mid \overline{\psi} \in \Phi \text{ and } \varphi \in \overline{\psi}\}$,
- $<_{\mathrm{DMT}}$ is the total order on $\{\{\varphi\} \mid \varphi \in \overline{\varphi}\}$ defined by: $\{\varphi_1\} <_{\mathrm{DMT}} \{\varphi_2\}$ if and only if $\varphi_1 <_{\mathrm{FMT}} \varphi_2$.

The following theorem states that the LPΔJ product lines generates the same products as the LFJ product line. Hence, Pure DOP is a true generalization of FOP.

THEOREM 4.1. *If* L *is a type safe* LFJ *product line, then* $[\![\mathrm{L}]\!]$ *is a type safe* LPΔJ *product line such that, for every valid feature configuration* $\overline{\psi}$, *the product for* $\overline{\psi}$ *generated by* L *is the same as the product for* $\overline{\psi}$ *generated by* $[\![\mathrm{L}]\!]$.

Although it is possible in principle to encode FOP in Core DOP, a straightforward embedding as for Pure DOP is not possible. This is because a feature-oriented SPL may have several base feature modules, while Core DOP requires exactly one core module as starting point for product generation.

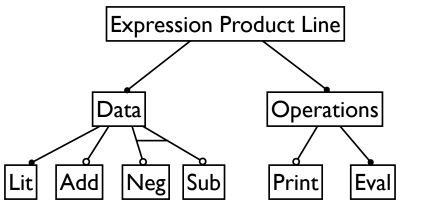

Figure 5. Feature model for evolved Expression Product Line

5. Pure DOP for Product Line Development

Pure DOP supports proactive, extractive and reactive product line development [22]. In the proactive approach, the scope of the product line, i.e., the set of products to be developed, is analyzed beforehand. All reusable artifacts are planned and developed in advance. The example for Pure DOP presented in Section 2 can be seen as proactive product line development, since we start from the feature model defining the scope of the product line and develop delta modules and a Pure DOP SPL for these products. However, proactive development requires a high upfront investment to define the scope of the product line and to develop reusable artifacts.

Hence, in order to reduce the adoption barrier for product line engineering, Krueger [22] proposes the usage of reactive and extractive approaches. In reactive product line engineering, only a basic set of products is developed. When new customer requirements arise, the existing product line is evolved. The extractive approach allows turning a set of existing legacy application into a product line. Development starts with the existing products from which the other products of the product line are derived.

FOP [7, 13] supports proactive product line development well. However, since feature modules are restricted to add or refine existing classes, FOP does not support extractive development and only partially supports reactive development. It is not possible to start from an existing legacy application comprising a larger set of features and to remove features. Moreover, in order to deal with new requirements following the reactive approach, feature modules might have to be refactored to remove functionalities. Also, in Core DOP, extractive product line development is not straight forward, since one product has to be chosen as designated core product. In contrast, Pure DOP is flexible and expressive enough to cover all three product line engineering approaches directly.

5.1 Reactive Product Line Engineering

In reactive product line engineering, development starts with an initial product line that is evolved in order to deal with changing customer requirements. Consider as initial product line the example depicted in Listing 5. Assume now that a new feature Sub should be introduced for representing subtraction expressions. In the new EPL, the Sub feature should be an alternative to the Neg feature. Additionally, the Print feature should become optional and the Eval feature mandatory. The feature diagram for the evolved product line is given in Figure 5.

In order to realize the new Sub feature, we have to add delta modules that introduce the corresponding data structure for subtraction and the associated print and the evaluation functionalities. The respective delta modules are shown in Listing 6. The specification for the evolved SPL is shown in Listing 7, where the operator **choose1**$(P_1, \ldots, P_n)$ means at most one of the propositions $P_1, \ldots, P_n$ is true (see [5]).

```
delta DSub {
  adds class Sub implements Exp {
    Exp expr1;
    Exp expr 2:
    Sub(Exp a, Exp b) { expr1 = a;  expr2= b; }
  }
}

delta DSubPrint {
  modifies Sub {
    adds void print() { expr1.print(); System.out.print("-");  expr2.print();}
  }
}

delta DSubEval{
   modifies class Sub {
     adds int eval() { return expr1.eval() − expr2.eval(); }
   }
}
```

Listing 6: Delta modules for Sub feature

```
features Lit, Add, Neg, Sub, Print, Eval
configurations Lit & Eval & choose1(Neg,Sub)
deltas
  [ DLit,
    DAdd when Add,
    DNeg when Neg,
    DSub when Sub ]

  [ DLitPrint when Print,
    DLitEval,
    DAddPrint when (Add & Print),
    DAddEval when Add,
    DNegPrint when (Neg & Print),
    DNegEval when Neg,
    DSubPrint when (Sub & Print),
    DSubEval when Sub ]
```

Listing 7: Pure DOP specification of the evolved EPL

As we can see in this example, Pure DOP supports reactive product line development, first, by adding new delta modules to implement new product features or to deal with new feature combinations, and, second, by reconfiguring the application conditions and the delta module order in the product line configuration to capture changes in the feature model.

5.2 Extractive Product Line Engineering

Extractive product line engineering starts with a set of existing legacy applications from which the other products of the product line are generated. Assume that we have already developed a product containing the Lit, Neg and Print features and a product containing the Lit, Add and Print features. Now, we want to transform these existing legacy applications into a product line according to the feature model in Figure 1.

First, the existing applications have to be transformed into delta modules that are applied initially. Listing 8 shows two delta modules adding the implementation of the two existing products, respectively. Second, in order to provide product implementations with less features, delta modules have to be specified that remove functionality from the existing products. Listing 9 shows the delta module that removes the feature Add.

Listing 10 shows the extractive implementation of the product line described by the feature model in Figure 1 starting from a product with features Lit, Neg and Print and a product with features Lit, Add, and Print introduced by the delta modules `DLitNegPrint` and `DLitAddPrint` in the first and second partitions, respectively. Their application conditions are exclusive such that for any feature configuration product generation starts with one of them. If the Add feature is not selected and the Neg feature is selected, we start with the existing product in delta module `DLitNegPrint`. Otherwise, we start with the existing product in delta module `DLitAddPrint`. If both features Add and Neg are selected, we add the Neg feature by the delta modules `DNeg` and `DNegPrint` of Listing 4. If both the Add feature and the Neg feature are not selected, we remove the Add feature by the delta module `DremAdd` of Listing 9. Finally, we add the evaluation functionality if the feature Eval is selected.

This example shows that Pure DOP supports extractive product line engineering by introducing the existing products in initial delta modules, by delta modules removing functionality, and by specifying the product line to generate the products from the existing products by suitable delta module application.

```
delta DLitNegPrint{
  adds interface Exp {
    void print();
  }
  adds class Lit implements Exp {
    int value;
    Lit(int n) { value = n; }
    void print() { System.out.println(value); }
  }
  adds class Neg implements Exp {
   Exp expr;
   Neg(Exp a) { expr1 = a; }
   void print() { System.out.print("-("); expr.print(); System.out.print(")");}
  }
}

delta DLitAddPrint{
  adds interface Exp {
    void print();
  }
  adds class Lit implements Exp {
    int value;
    Lit(int n) { value = n; }
    void print() { System.out.println(value); }
  }
  adds class Add implements Exp {
    Exp expr1;
    Exp expr2;
    Add(Exp a, Exp b) { expr1 = a; expr2 = b; }
    void print() { expr1.print(); System.out.print(" + "); expr2.print(); }
  }
}
```

Listing 8: Delta modules introducing the two legacy products

```
delta DremAdd {
  remove Add
}
```

Listing 9: Delta module removing the Add feature

```
features Lit, Add, Neg, Print, Eval
configurations Lit & Print
deltas
  [ DLitNegPrint when (!Add & Neg) ]

  [ DLitAddPrint when (Add | !Neg) ]

  [ DNeg when (Add & Neg),
    DremAdd when (!Add & !Neg) ]

  [ DNegPrint when (Add & Neg),
    DLitEval when Eval,
    DAddEval when (Add & Eval),
    DNegEval when (Neg & Eval) ]
```

Listing 10: Pure DOP specification of the extractive EPL

```
delta DremPrintLit {
  modifies interface Exp { removes print }
  modifies class Lit { removes print }
}

delta DremPrintAdd {
  modifies class Add { removes print }
}

delta DremPrintNeg {
  modifies class Neg { removes print }
}
```

Listing 11: Delta modules removing the Print feature

```
features Lit, Add, Neg, Sub, Print, Eval
configurations Lit & Eval & choose1(Neg,Sub)
deltas
  [ DLitNegPrint when (!Add & Neg),
    DSub when Sub ]

  [ DLitAddPrint when (Add | !Neg) ]

  [ DNeg when (Add & Neg),
    DremAdd when (!Add & !Neg)]

  [ DNegPrint when (Add & Neg & Print),
    DLitEval,
    DAddEval when Add,
    DNegEval when Neg,
    DremPrintLit when !Print,
    DremPrintAdd when (!Print & Add),
    DremPrintNeg when (!Print & Neg),
    DSubPrint when (Sub & Print),
    DSubEval when Sub ]
```

Listing 12: Pure DOP specification of the evolved extractive EPL

5.3 Combining Extractive and Reactive PL Engineering

Extractive and reactive product line engineering can be combined. An initial product line is developed from a set of existing legacy applications and evolved when new requirements arise. Consider, the product line developed using the extractive approach in Listing 10. Assume, that now the Sub feature should be added and the product line should be changed to implement the feature diagram in Figure 5. Since in this product line, the feature Print is optional, we have to provide delta modules that remove the printing functionality from the `Lit`, `Add` and `Neg` classes. These delta modules are depicted in Listing 11.

Listing 12 shows the specification of the evolved product line depicted in Listing 10. The generation starts again from the two delta modules `DLitNegPrint` and `DLitAddPrint` introducing the existing products. Additionally, the product line contains delta modules for adding the Sub feature (cf. Listing 6) and delta modules for removing the Print feature (cf. Listing 11).

6. Comparison with Core DOP

In the traditional presentation of DOP [29], which we refer to as Core DOP, program generation always starts from a core module containing the implementation of a selected valid product of the product line. Then, delta modules specify the changes to the core module in order to implement the other products. Moreover, in the presentation of Core DOP given in [29]:

- the feature configuration corresponding to the product implemented by the core module is specified in the code of the core module,
- the application condition of a delta module is specified in the code of the delta module by a clause of the form "**when** γ", where γ is a propositional constraint specifying the feature configurations in which the delta module has to be applied, and
- the application partial order for the delta modules is specified in the code of the delta modules using a clause of the form "**after** $\overline{\delta}$", which specifies that the delta module must be applied after all applicable delta modules in $\overline{\delta}$ have been applied.

Pure DOP and Core DOP are indeed equivalent:

- A Pure DOP product line can be expressed as a Core DOP product line by adding an empty product to the product line and choosing it as the product implemented by the core module.
- A Core DOP product line can be expressed as a Pure DOP product line by transforming the core module into a delta module that has to be applied before any other delta module for all the valid feature configurations.

Pure DOP is a conceptual simplification of Core DOP dropping the notion of the core module and separating the specification of the application conditions and of the application ordering from the delta modules. This presents the following advantages:

- Pure DOP allows reusing delta modules for implementing different product lines (cf. Sections 2 and 5).
- Every delta module in Pure DOP containing only adds clauses can play the role of the core module. Thus, product lines with multiple base modules, that may not correspond to valid products, are possible. As a consequence, Pure DOP is a true generalization of FOP (cf. Section 4).
- Pure DOP supports the evolution of product lines. If a product line evolves such that the core product of a Core DOP product line is no longer a valid product, the core module and potentially all delta modules have to be refactored. In contrast, in pure DOP, existing delta modules can be reused for the specification of the evolved product line (cf. Section 5).

7. Related Work

The notion of program deltas is introduced in [23] to describe the modifications of object-oriented programs. In [30], delta-oriented modeling is used to develop product line artifacts suitable for automated product derivation and implemented with frame technology [36]. This approach is extended in [28] to a seamless delta-oriented model-based development approach for SPLs. In [11], an algebraic representation of delta-oriented product lines is presented. The main focus in [11] is to reason about conflicting modifications and to devise a general criterion to guarantee the unambiguity of product lines using conflict-resolving deltas. The unambiguity property presented in this work is an instance of the criterion presented in [11], but it is more restrictive since it requires to order all potential conflicts. Delta modules are one possibility to implement arrows in the category-theoretical framework for program generation proposed by Batory in [6].

Feature-oriented programming (FOP) [2, 7, 13, 34], Core DOP [29] and Pure DOP are compositional approaches [19] for implementing SPLs. For a detailed comparison between FOP and Core DOP, the reader is referred to [29]. Other compositional approaches used to implement product lines rely on aspects [18], framed aspects [24], combinations of feature modules and aspects [3, 25], mixins [31], hyperslices [33] or traits [8, 14]. In [23], several of these modern program modularization techniques are compared with respect to their ability to represent feature-based variability. Furthermore, the modularity concepts of recent languages, such as SCALA [26] or NEWSPEAK [10], can be used to represent product features.

In [1], an approach is presented that combines reactive and extractive product line engineering [22] based on aspect-oriented program refactorings. The modification operations that can be specified in delta modules are sufficient to express before, after and around advice considered in aspect-oriented programming [21]. Delta modules do not comprise a specification formalism for modifications to be carried out at several places of a program (such as pointcuts), such that all program modifications have to be explicitly specified. Adding a pointcut-specification technique to delta modules would allow encoding AOP by DOP, which is a subject of future work. However, delta modules are more flexible than aspects by their ability to remove functionality, such that a program refactoring is not required to evolve a product line when functionality has to be removed.

8. Conclusions and Future Work

In this paper, we have proposed *pure delta-oriented programming* (Pure DOP) as a conceptual simplification of Core DOP [29]. An implementation of the Pure DOP programming language presented in this paper and a core calculus with a constraint-based type system are currently being developed. Following the conceptual comparison of FOP, Core DOP and Pure DOP in this paper, we are evaluating Pure DOP empirically at larger case examples and investigating the extraction of delta modules from version histories.

The concept of Pure DOP is not bound to a particular programming language. In this work, we have instantiated it for LFJ. For future work, we are aiming to use other languages for the underlying product implementations. A starting point is the trait-based calculus FEATHERWEIGHT RECORD-TRAIT JAVA (FRTJ) [8, 9]. In FRTJ, classes are assembled from interfaces, records (providing fields) and traits [14] (providing methods) that can be directly manipulated by designated composition operations. These operations make FRTJ a good candidate for implementing delta modules in an expressive way.

References

[1] V. Alves, P. Matos, L. Cole, A. Vasconcelos, P. Borba, and G. Ramalho. Extracting and evolving code in product lines with aspect-oriented programming. In *Transactions on aspect-oriented software development IV*, pages 117–142. Springer-Verlag, 2007.

[2] S. Apel, C. Kästner, and C. Lengauer. Feature Featherweight Java: A Calculus for Feature-Oriented Programming and Stepwise Refinement. In *GPCE*, pages 101–112. ACM, 2008.

[3] S. Apel, T. Leich, and G. Saake. Aspectual feature modules. *IEEE Trans. Software Eng.*, 34(2):162–180, 2008.

[4] P. G. Bassett. *Framing software reuse: lessons from the real world*. Prentice-Hall, Inc., 1997.

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

[6] D. Batory. Using modern mathematics as an FOSD modeling language. In *GPCE*, pages 35–44. ACM, 2008.

[7] D. Batory, J. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Trans. Software Eng.*, 30(6):355–371, 2004.

[8] L. Bettini, F. Damiani, and I. Schaefer. Implementing Software Product Lines using Traits. In *SAC, OOPS Track*, pages 2096–2102. ACM, 2010.

[9] L. Bettini, F. Damiani, I. Schaefer, and F. Strocco. A Prototypical Java-like Language with Records and Traits. In *PPPJ*. ACM, 2010.

[10] G. Bracha. Executable Grammars in Newspeak. *ENTCS*, 193:3–18, 2007.

[11] D. Clarke, M. Helvensteijn, and I. Schaefer. Abstract delta modeling. In *Proc. of GPCE*, 2010. (to appear).

[12] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison Wesley Longman, 2001.

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

[14] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. Black. Traits: A mechanism for fine-grained reuse. *ACM TOPLAS*, 28(2):331–388, 2006.

[15] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.

[16] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, Carnegie Mellon Software Engineering Institute, 1990.

[17] C. Kästner and S. Apel. Type-Checking Software Product Lines - A Formal Approach. In *ASE*, pages 258–267. IEEE, 2008.

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

[19] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *ICSE*, pages 311–320. ACM, 2008.

[20] C. Kästner, S. Apel, S. S. ur Rahman, M. Rosenmüller, D. Batory, and G. Saake. On the Impact of the Optional Feature Problem: Analysis and Case Studies. In *SPLC*. IEEE, 2009.

[21] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *ECOOP*, volume 2072 of *LNCS*, pages 327–353. Springer, 2001.

[22] C. Krueger. Eliminating the Adoption Barrier. *IEEE Software*, 19(4):29–31, 2002.

[23] R. Lopez-Herrejon, D. Batory, and W. Cook. Evaluating Support for Features in Advanced Modularization Technologies. In *ECOOP*, volume 3586 of *LNCS*, pages 169–194. Springer, 2005.

[24] N. Loughran and A. Rashid. Framed aspects: Supporting variability and configurability for aop. In *ICSR*, volume 3107 of *LNCS*, pages 127–140. Springer, 2004.

[25] M. Mezini and K. Ostermann. Variability management with feature-oriented programming and aspects. In *SIGSOFT FSE*, pages 127–136. ACM, 2004.

[26] M. Odersky. The Scala Language Specification, version 2.4. Technical report, Programming Methods Laboratory, EPFL, 2007.

[27] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering - Foundations, Principles, and Techniques*. Springer, 2005.

[28] I. Schaefer. Variability Modelling for Model-Driven Development of Software Product Lines. In *Intl. Workshop on Variability Modelling of Software-intensive Systems*, 2010.

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

[30] I. Schaefer, A. Worret, and A. Poetzsch-Heffter. A Model-Based Framework for Automated Product Derivation. In *Proc. of MAPLE*, 2009.

[31] Y. Smaragdakis and D. Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Trans. Softw. Eng. Methodol.*, 11(2):215–255, 2002.

[32] R. Strniša, P. Sewell, and M. Parkinson. The Java module system: core design and semantic definition. In *OOPSLA*, pages 499–514. ACM, 2007.

[33] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr. N degrees of separation: multi-dimensional separation of concerns. In *ICSE*, pages 107–119, 1999.

[34] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe Composition of Product Lines. In *GPCE*, pages 95–104. ACM, 2007.

[35] M. Torgersen. The Expression Problem Revisited. In *ECOOP*, volume 3086 of *LNCS*, pages 123–146. Springer, 2004.

[36] H. Zhang and S. Jarzabek. An XVCL-based Approach to Software Product Line Development. In *Software Engineering and Knowledge Engineering*, pages 267–275, 2003.