Delta-oriented Programming of Software Product Lines

Ina Schaefer[1], Lorenzo Bettini[2], Viviana Bono[2],
Ferruccio Damiani[2], and Nico Tanzarella[2]

[1] Chalmers University of Technology, 421 96 Gothenburg, Sweden
schaefer@chalmers.se
[2] Dipartimento di Informatica, Università di Torino, C.so Svizzera, 185 - 10149 Torino, Italy
{bettini,bono,damiani}@di.unito.it nicotanz@libero.it

Abstract. Feature-oriented programming (FOP) implements software product lines by composition of feature modules. It relies on the principles of stepwise development. Feature modules are intended to refer to exactly one product feature and can only extend existing implementations. To provide more flexibility for implementing software product lines, we propose delta-oriented programming (DOP) as a novel programming language approach. A product line is represented by a core module and a set of delta modules. The core module provides an implementation of a valid product that can be developed with well-established single application engineering techniques. Delta modules specify changes to be applied to the core module to implement further products by adding, modifying and removing code. Application conditions attached to delta modules allow handling combinations of features explicitly. A product implementation for a particular feature configuration is generated by applying incrementally all delta modules with valid application condition to the core module. In order to evaluate the potential of DOP, we compare it to FOP, both conceptually and empirically.

1 Introduction

A *software product line* (SPL) is a set of software systems with well-defined commonalities and variabilities [13, 29]. The variabilities of the products can be defined in terms of *product features* [16], which can be seen as increments of product functionality [6]. *Feature-oriented programming* (FOP) [10] is a software engineering approach relying on the principles of *stepwise development* [9]. It has been used to implement SPLs by composition of *feature modules*. In order to obtain a product for a feature configuration, feature modules are composed incrementally. 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. Feature modules cannot remove code from an implementation. Thus, the design of a SPL always starts from a base feature module which contains common parts of all products. Furthermore, a feature module is intended to represent exactly one product feature. If the selection of two optional features requires additional code for their interaction, this cannot be directly handled leading to the optional feature problem for SPLs [20].

In this paper, we propose *delta-oriented programming* (DOP) as a novel programming language approach particularly designed for implementing SPLs, based on the

concept of program deltas [32, 31]. The goal of DOP is to relax the restrictions of FOP and to provide an expressive and flexible programming language for SPL. In DOP, the implementation of a SPL is divided into a *core module* and a set of *delta modules*. The core module comprises a set of classes that implement a complete product for a valid feature configuration. This allows developing the core module with well-established single application engineering techniques to ensure its quality. Delta modules specify changes to be applied to the core module in order to implement other products. A delta module can add classes to a product implementation or remove classes from a product implementation. Furthermore, existing classes can be modified by changing the super class, the constructor, and by additions, removals and renamings of fields and methods. A delta module contains an application condition determining for which feature configuration the specified modifications are to be carried out. In order to generate a product implementation for a particular feature configuration, the modifications of all delta modules with valid application condition are incrementally applied to the core module. The general idea of DOP is not restricted to a particular programming language. In order to show the feasibility of the approach, we instantiate it for JAVA, introducing the programming language DELTAJAVA.

DOP is a programming language approach especially targeted at implementing SPLs. The delta module language includes modification operations capable to remove code such that a flexible product line design and modular product line evolution is supported. The application conditions attached to delta modules can be complex constraints over the product features such that combinations of features can be handled explicitly avoiding code duplication and countering the optional feature problem [20]. The ordering of delta module application can be explicitly defined in order to avoid conflicting modifications and ambiguities during product generation. Using a constraint-based type system, it can be ensured that the SPL implementation is well formed. This yields that product generation is safe, which means that all resulting products are type correct. In order to evaluate the potential of DOP, we compare it with FOP, both on a conceptual and on an empirical level using case examples studied for FOP.

2 Delta-oriented Programming

In order to illustrate delta-oriented programming in DELTAJAVA, we use the *expression product line* (EPL) as described in [25] as running example. We consider the following grammar for expressions:

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: first, printing, which returns the expression as a string, and second, evaluation, which computes the value of the expression. The set of products in the EPL can be described with a feature model [16], see Figure 1. It has 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.

Core Module In DELTAJAVA, a software product line is implemented by a core module and a set of delta modules. A *core module* corresponds to the implementation of a

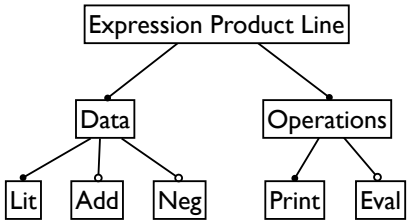

Fig. 1: Feature Model for Expression Problem Product Line

```
core Print, Lit {
  interface Exp { void print(); }

  class Lit implements Exp {
        int value;
        Lit(int n) { value=n; }
        void print() { System.out.print(value); }
    }
}
```

Listing 1: Core module implementing Lit and Print features

product for a valid feature configuration. It defines the starting point for generating all other products by delta module application. The core module depends on the underlying programming language used to implement the products. In the context of this work, the core module contains a set of JAVA classes and interfaces. These are enclosed in a `core` block additionally specifying the implemented features by listing their names:

core <Feature names> { <Java classes and interfaces> }

The product represented by the core module can be any valid product. Thus, it has to implement at least the mandatory features and a minimal set of required alternative features, if applicable. Note that the core module is not uniquely determined, as illustrated in Section 3. Choosing the core module to be a valid product allows to develop it with standard single application engineering techniques to ensure its quality and to validate and verify it thoroughly with existing techniques. Listing 1 contains a core module for the EPL. It implements the features Lit and Print.

Delta Modules Delta modules specify changes to the core module in order to implement other products. The alterations inside a delta module act on the class level (by adding, removing and modifying classes) and on the class structure level (by modifying the internal structure of a class by changing the super class or the constructor, or by adding, removing and renaming fields and methods).[3] Furthermore, a delta module can add, remove or modify interfaces by adding, removing or renaming method signatures. An application condition is attached to every delta module in its when clause determining for which feature configurations the specified alterations are to be carried out.

[3] Renaming a method does not change calls to this method. The same holds for field renaming.

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

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

Listing 2: Delta modules for Eval and Add features

Application conditions are propositional constraints over features. This allows specifying delta modules for combinations of features or to explicitly handle the absence of features. The number of delta modules required to implement all products of a product line depends on the granularity of the application conditions. In general, a delta module has the following shape:

```
delta <name> [after <delta names>] when <application condition> {
    removes <class or interface name>
    adds class <name> <standard Java class>
    adds interface <name> <standard Java interface>
    modifies interface <name> { <remove, add, rename method header clauses> }
    modifies class <name> { <remove, add, rename field clauses> <remove, add, rename method clauses> }
}
```

The left of Listing 2 shows the delta module corresponding to the Eval feature of the EPL. It modifies the interface Exp by adding the method signature for the eval method. Furthermore, it modifies the class Lit by adding the method eval. The when clause denotes that this delta module is applied for every feature configuration in which the Eval feature is present. The right of Listing 2 shows the delta module required for the Add feature adding the class Add. The when clause specifies that the delta module is applied if the features Add is present.

Product Generation In order to obtain a product for a particular feature configuration, the alterations specified by delta modules with valid application conditions are applied incrementally to the core module. The changes specified in the same delta module are applied simultaneously. In order to ensure, for instance, that a class to be modified exists or that a modification of the same method by different delta modules does not cause a conflict, an ordering on the application of delta modules can be defined by means of the after clause. This ordering implies that a delta module is only applied to the core module after all delta modules with a valid application condition mentioned in the after clause have been applied. The partial ordering on the set of delta modules defined by the after clauses captures the necessary dependencies between the delta modules, which are usually semantic requires relations. As an example, consider the left of Listing 3 which depicts the delta module introducing the evaluation functionality for the addition expression. Its when clause specifies that it is applied, if both the Add and the Eval features are present. In the after clause, it is specified that this delta module has to be applied after the delta module DAdd because the existence of the class Add is assumed. The implementation of the delta module DAddPrint is similar. Note that specifying that a delta module *A* has to be applied after the delta module *B* does *not* mean that *A* requires *B*: it only denotes that if a feature configuration satisfies the when clause of both *A* and *B*, then *B* must be applied before *A*.

```
delta DAddEval after DAdd when Add && Eval {
    modifies class Add {
        adds int eval()
            { return expr1.eval() + expr2.eval(); }
    }
}
```

```
delta DAddPrint after DAdd when Add && Print {
    modifies class Add {
        adds void print()
            { expr1.print();
              System.out.print(" + "); expr2.print(); }
    }
}
```

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

```
interface Exp { void print(); int eval(); }

class Lit implements Exp {
    int value;
        Lit(int n) { value=n; }
        void print()
            { System.out.print(value); }
        int eval()
            { return value; }
}
```

```
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(); }
    int eval()
        { return expr1.eval() + expr2.eval();}
}
```

Listing 4: Generated implementation for Lit, Add, Print, Eval features

The generation of a product for a given feature configuration consists of the following steps, performed automatically by the system: (i) Find all delta modules with a valid application condition according to the feature configuration (specified in the when clause); and (ii) Apply the selected delta modules to the core module in any linear ordering respecting the partial order induced by the after clauses.

As an example of a product implementation generated by delta application, consider Listing 4 which shows the implementation of the Lit, Add, Print, Eval features of the EPL. It is an ordinary JAVA program containing the interface Exp and the classes Lit and Add. The implementation is obtained by applying the delta modules depicted in Listings 2 and 3 to the core module (cf. Listing 1) in any order in which DAddEval and DAddPrint are applied after DAdd.

Safe Program Generation The automatic generation of products by delta application is only performed if the DELTAJAVA product line implementation is well-formed. Well-formedness of a product line means that all delta modules associated to a valid feature configuration are well-formed themselves and applicable to the core module in any order compatible with the partial order provided by the after clauses. The partial order ensures that all compatible application orders generate the same product. A delta module is well-formed, if the added and removed classes are disjoint and if the modifications inside a class target disjoint fields and methods. A delta module is applicable to a product if all the classes to be removed or modified exist, all methods and fields to be removed or renamed exist and if classes, methods and fields to be added do not exist. Furthermore, all delta modules applicable for the same valid feature configuration that are not comparable with respect to the after partial order must be compatible. This means that no class is added or removed in more than one delta module, and for every class modified in more than one delta module the fields or methods added, modified and

renamed are disjoint. This implies that all conflicts between modifications targeting the same class have to be resolved by the ordering specified with the after clauses.

In order to ensure well-formedness, DELTAJAVA is accompanied by a constraint-based type system (not shown in this paper). For each delta module in isolation, a set of constraints is generated that refers to the classes, methods or fields required to exist or not to exist for the delta module to be applicable. Then, for every valid feature configuration, only by checking the constraints, it can be inferred whether delta module application will lead to a well-typed JAVA program. The separate constraint generation for each delta module avoids reinspecting all delta modules if only one delta is changed or added. Furthermore, if an error occurs, it can easily be traced back to the delta modules causing it.

3 Implementing Software Product Lines

The delta-oriented implementation of a SPL in DELTAJAVA comprises an encoding of the feature model, the core module and a set of delta modules necessary to implement all valid products. The feature model is described by its basic features and a propositional formula describing the valid feature configurations (other representations might be considered [6]). In this section, we show how SPLs are flexibly implemented in DELTAJAVA starting from different core products using the EPL as illustration.

Starting from a Simple Core The core module of a DELTAJAVA product line contains an implementation of a valid product. One possibility is to take only the mandatory features and a minimal number of required alternative features, if applicable. In our example, the Lit and Print features are the only mandatory features. Listing 1 shows the respective core module serving as starting point of a SPL implementation starting from a simple core. In order to represent all possible products, delta modules have to be defined that modify the core product accordingly. For the EPL starting from the simple core, this are the delta modules depicted in Listings 2 and 3 together with three additional delta modules implementing the Neg feature alone as well as in combination with the Print feature and the Eval feature. Their implementation is similar to the implementation of the DAdd, DAddPrint and DAddEval delta modules and, thus, not shown here. Listing 5 shows the complete implementation of the EPL containing the encoding of the feature model (cf. Figure 1), the core module and the delta modules. For space reasons, the concrete implementations of the core and delta modules are omitted.

Starting from a Complex Core Alternatively, a SPL implementation in DELTAJAVA can start from any product for a valid feature configuration containing a larger set of features. The advantage of the more complex core product is that all included functionality can be developed, validated and verified with standard single application techniques. In order to illustrate this idea, we choose the product with the Lit, Add, Print and Eval features as core product whose implementation is contained in Listing 4. In order to provide product implementations containing less features, functionality has to be removed from the core. Listing 6 shows a delta module that removes the evaluation functionality from the complex core. It is applied to the core module if the Eval feature

```
features Lit, Add, Neg, Print, Eval
configurations Lit && Print && (Add | Neg | Eval )
core Lit, Print { [ ... ]  }
delta DEval when Eval { [...] }
delta DAdd when Add { [...] }
delta DAddPrint after DAdd when Add && Print { [...] }
delta DAddEval after DAdd when Add && Eval { [...] }
delta DNeg  when Neg { [...] }
delta DNegPrint after DNeg when Neg && Print { [...] }
delta DNegEval after DNeg when Neg && Eval { [...] }
```

Listing 5: Product Line Implementation in DELTAJAVA starting from Simple Core

```
delta DRemEval when !Eval && Add {
   modifies interface Exp { removes eval;}
   modifies class Lit { removes eval;}
   modifies class Add { removes eval;}
}
```

Listing 6: Removing Eval from Complex Core

is not included in a feature configuration, but the Add feature is selected. This means that only the eval method from the Add class is removed, but not the Add class itself.

Listing 7 shows the implementation of the EPL starting from the complex core module depicted in Listing 4. In addition to the delta module DRemEval, a delta module DRemAdd (not shown here) is required to remove the Add class if the Add feature is not selected, and a third delta module DRemAddEval (not shown here) is required to remove the eval method from the Lit class and the Exp interface, in case both the Eval and the Add features are not selected. Further, the delta modules DNeg, DNegPrint and DNegEval as in the previous implementation are required.

4 Comparing Delta-oriented and Feature-oriented Programming

In order to evaluate DOP of SPLs, we compare it with FOP [10]. Before the comparison, we briefly recall the main concepts of FOP.

```
features Lit, Add, Neg, Print, Eval
configurations Lit && Print && (Add | Neg | Eval )
core Lit, Print, Add, Eval  { [ ... ]  }
delta DRemEval when !Eval && Add { [...] }
delta DRemAdd when !Add { [...] }
delta DRemAddEval when !Add && !Eval { [...] }
delta DNeg when Neg { [...] }
delta DNegPrint after DNeg when Neg && Print { [...] }
delta DNegEval after DNeg when Neg && Eval { [...] }
```

Listing 7: Product Line Implementation in DELTAJAVA starting from Complex Core

```
interface Exp { String print(); }

class Lit implements Exp {
    int value;
    Lit(int n) { value=n; }
    void print() { System.out.print(value); }
}
```

(a) LitPrint feature module

```
class Add implements Exp {
    Exp x;
    Exp y;
    Add( Exp x, Exp y ) { this.x = x; this.y = y; }
    public String print() { return x + "+" + y; }
}
```

(b) AddPrint feature module

```
refines interface Exp { int eval(); }

refines class Lit {
    public int eval() { return value; }
}
```

(c) LitEval feature module

```
refines class Add {
    public int eval()
        { return x.eval() + y.eval(); }
}
```

(d) AddEval feature module

Listing 8: Feature Modules for EPL in JAK

4.1 Feature-oriented Programming

In FOP [10], a program is incrementally composed from feature modules following the principles of stepwise development. A feature module can introduce new classes and refine existing ones. The concept of stepwise development is introduced in GenVoca [9] and extended in AHEAD [10] for different kinds of design artifacts. For our comparison, we restrict our attention to the programming language level of AHEAD, i.e., the JAK language, a superset of JAVA containing constructs for feature module refinement.

In order to illustrate FOP in JAK, we recall the implementation of the EPL presented in [25]. The five domain features, shown in the feature model in Figure 1, are transformed into six feature modules. The difference between the number of domain features and the number of feature modules is due to the fact that combinations of domain features cannot be dealt with explicitly in JAK. Therefore, a suitable encoding of the domain features has to be chosen. This results in the feature modules for the feature combinations LitPrint, AddPrint and NegPrint combining every data feature with the Print feature. Furthermore, for each data feature, there is a feature module adding the evaluation operation, i.e., LitEval, AddEval and NegEval. The JAK code implementing the feature modules LitPrint, AddPrint, LitEval, and AddEval is shown in Listing 8. The generation of a product starts from the base feature module LitPrint. A program containing the Lit, Add, Print, and Eval features can be obtained by composing the feature modules as follows: LitPrint • AddPrint • LitEval • AddEval. The code of the resulting program is as shown in Listing 4.

4.2 Comparison

Both delta modules and features modules support the modular implementation of SPLs. However, they differ in their expressiveness, the treatment of domain features, solutions for the optional features problem, guarantees for safe composition and support for evolution. Both techniques scale to a general development approach [10, 31].

Expressiveness Feature modules can introduce new classes or refine existing ones following the principles of stepwise development [10]. The design of a SPL always starts from the base feature module containing common parts of all products. In contrast, delta modules support additions, modifications and removals of classes, methods and fields. This allows choosing any valid feature configuration to be implemented in the core module and facilitates a flexible product line design starting from different core products, as shown in Section 3. The core module contains an implementation of a complete product. This allows developing and validating it with well-established techniques from single application engineering, or to reengineer it before starting the delta module programming to ensure its quality. For verification purposes, it might save analysis effort to start with a complex product, check the contained functionality thoroughly and remove checked functionality in order to generate other products. In JAK, an original method implementation before refinement can be accessed with a Super() call. Delta modules currently do not have an equivalent operation. However, a Super() call in delta modules could be encoded by renaming the method to be accessed and adding a corresponding call during program generation.

Domain Features In FOP, domain-level features are intentionally separated from feature modules in order to increase the reusability of the refinements. The mapping from domain features to the feature modules is taken care of by external tools. In the AHEAD Tool Suite [10], the external tool *guidsl* [6] supports this aspect of product generation. For a given domain feature configuration, *guidsl* provides a suitable composition of the respective feature modules.

In a DELTAJAVA implementation, the features of the feature model and the corresponding constraints are explicitly specified. This allows reasoning about feature configurations within the language. For product generation, it can be established that the provided feature configuration is valid, such that only valid products are generated. For each delta module, the application condition ranges over the features in the feature model such that the connection of the modifications to the domain-level features is made explicit. This limits the reusability of delta modules for another SPL, but allows static analysis to validate the design of the very product line that is implemented. It can be checked whether the application condition of a delta module can actually evaluate to true for any valid feature configuration. Otherwise, the delta module will never be applied and can be removed. Furthermore, for a given feature configuration, the set of applicable delta modules can be determined directly without help of external tools. If, for instance, a new delta module has to be added, it is easy to learn about the consequences and potential conflicts in the existing implementation.

The Optional Feature Problem The optional feature problem [20] occurs when two optional domain features require additional code for their interaction. Feature modules [10] are not intended to refer to combinations of features. Thus, one way to solve the optional feature problem is to move code belonging to one feature to a feature module for another feature, similar to the combination of domain features in the EPL. This solution violates the separation of concerns principle [20] and leads to a non-intuitive mapping between domain features and feature modules. In the Graph Product Line [24] implementation [5] (cf. Section 5), the optional feature problem is solved by multiple

implementations per domain feature which leads to code duplications. Alternatively, derivative modules [23] can be used. In this case, a feature module is split into a base module only containing introductions and a set of derivative modules only containing refinements that are necessary if other features are also selected. However, this may result in a large number of small modules and might not scale in practice [20].

In contrast, the optional feature problem can be solved in DOP within the language. A delta module does not correspond to one domain feature, but can refer to any combination of features. By the application condition of a delta module, the feature configurations the delta module is applied for are made explicit such that code only required for feature interaction can be clearly marked. In particular, delta modules can implement derivative modules. The implementation of the EPL in Listing 5 follows the derivative principle. Moreover, code duplication between two features modules can be avoided by factoring common code into a separate delta module that is applied if at least one of the respective features is selected.

Safe Composition Feature composition in FOP is performed in a fixed linear order. This linear ordering has to be provided before feature module composition to avoid conflicting modifications. In DOP, the partial order specified in the `after` clauses of delta modules captures only essential dependencies and semantic requirements between different modifications of the same class or method. Instead of specifying a partial order, conflicting modifications between delta modules could also be prohibited completely at the price of writing additional delta modules for the respective combinations. Thus, the partial order is a compromise between modularity and a means to resolve conflicting modifications without increasing the number of delta modules.

During feature module composition, it is not guaranteed that the resulting program is correct, e.g., that each referenced field or method exists. Such errors are only raised during compilation of the generated program. Recently, there have been several approaches to guarantee safety of feature module composition [2, 3, 14, 35] by means of external analysis or type systems. DELTAJAVA has an integrated constraint-based type system guaranteeing that the generated program for every valid feature configuration is type correct and that all conflicts are resolved by the partial order. Constraints are generated for each delta module in isolation such that an error can be traced back to the delta modules where it occurred. Additionally, changed or added delta modules do not require re-checking the unchanged delta modules.

Product Line Evolution Product lines are long-lived software systems dealing with changing user requirements. For example, if in the EPL printing should become an optional feature, the JAK implementation has to be refactored to separate the printing functionality from the implementation of the data. In the DELTAJAVA implementation, only one delta module has to be added to remove the printing functionality from the simple as well as from the complex core, while all other delta modules remain unchanged. To this end, the expressivity of the modification operations in delta modules supports modular evolution of SPL implementations.

Scaling Delta-oriented Programming The AHEAD methodology [10] for developing software by stepwise development is not limited to the implementation level and has

	Feature-oriented Programming	Delta-oriented Programming
Expressiveness	Design from Base Module	Design from Any Product
Domain Features	Bijection between Features and Feature Modules	Delta Modules for Feature Combinations
Optional Feature Problem	Rearrange Code, Multiple Impl., Derivative Modules	Direct Implementation of Interaction
Safe Composition	External Tools, Type Systems	Partial Order for Conflict Resolution, Type System
Evolution	Refactoring	Addition of Delta Modules

Table 1: Summary of Comparison

been instantiated to other domain-specific languages as well as to XML. Similarly, the concepts of DOP can be applied to other programming or modeling languages. In [32], a seamless delta-oriented model-driven development process is proposed. The variability structure in terms of core and delta modules has to be determined only once for an initial delta-oriented product line representation on a high level of abstraction. By stepwise refinement of the delta-oriented product models without changing the variability structure, a DELTAJAVA implementation of a SPL can eventually be obtained. In this way, product variability can be managed in the same manner on all levels during SPL development.

Summary FOP is a general software engineering approach based on the principles of stepwise development that has been used to implement SPLs. In contrast, DOP is specifically designed for this task such that it differs from FOP as summarized in Table 1.

5 Evaluation

In order to evaluate DOP in practice, we have implemented a set of case studies in DELTAJAVA that have also been studied in the context of JAK [10]. These case studies include two versions of the EPL [1, 25], two smaller case examples [7], and the Graph Product Line (GraphPL), suggested in [24] as a benchmark to compare SPLs architectures. The first implementation of the EPL in AHEAD [1] follows the derivative module principle. The second implementation of the EPL is the same as sketched in this paper and presented in [25]. In the corresponding DELTAJAVA implementations, the design of the delta modules has been chosen to mimic the AHEAD design. In order to evaluate the flexibility of DOP, we have implemented each example in DELTAJAVA starting from a simple core product and from a complex core product. For the EPL, we used the simple and the complex core products presented in Section 3.

The results of our evaluation are summarized in Table 2 containing the number of feature modules or delta modules and the corresponding lines of code required to implement the respective examples. The number of feature modules and delta modules does

	JAK		DELTAJAVA Simple Core		DELTAJAVA Complex Core	
	# feature modules	LOC	# delta modules	LOC	# delta modules	LOC
EPL [1]	12	98	7	123	6	144
EPL [25]	6	98	5	117	5	124
Calculator [7]	10	75	6	76	6	78
List [7]	4	48	3	58	2	59
GraphPL [24]	19	2348	20	1407	19	1373

Table 2: Evaluation Results (LOC is the number of lines of code)

not differ significantly in the considered examples. For the first version of the EPL [1], the only reason that 12 features modules are necessary is that also interfaces are implemented by separate modules which is a design decision taken in [1]. In the second version of the EPL [25], the number of delta modules plus the core module is actually the same as the number of features modules, since DELTAJAVA encodes the same modular SPL representation. In the Calculator and List examples, less delta modules are required because several feature modules could be combined into one delta module, whereas in the GraphPL example with the simple core, additional delta modules are used to factor out common code for combinations of features. In the considered examples, the differences between the number of delta modules required to implement a SPL starting from a simple core or starting from a complex core are marginal. A more conceptual analysis on how the choice of the core product influences the SPL design is subject to future work.

In the smaller case examples, the lines of code in DELTAJAVA exceed the lines of code required in JAK, because in DELTAJAVA the feature model encoding and the application conditions have to be specified. Furthermore, as DELTAJAVA currently has no call to the original variants of modified methods, the required renaming has to be done manually, leading to additional lines of code, in particular for the EPL. This can be avoided if DELTAJAVA is extended with an operation similar to the JAK Super() call as pointed out in Section 4. In the larger case example of the GraphPL [24], delta modules require much less code, because they can represent product functionality more flexibly. First, common code for two features can be factored out into one delta module, and second, combinations of features can be treated directly by designated delta modules instead of duplicating code for feature combinations. This shows that DOP can be beneficial in terms of code size for larger SPLs in which the optional feature problem [20] arises. However, tool support has to be provided to deal with the complexity that is introduced by the flexibility of DOP, e.g., for visualizing dependencies between delta modules applicable for the same feature configuration.

6 Related Work

The approaches to implementing SPLs in the object-oriented paradigm can be classified into two main directions [19]. First, annotative approaches, such as conditional compilation, frames [36] or COLORED FEATHERWEIGHT JAVA (CFJ) [17], mark the source code of the whole SPL with respect to product features on a syntactic level. For a particular feature configuration, marked code is removed.

Second, compositional approaches, such as DELTAJAVA, associate code fragments to product features that are assembled to implement a particular feature configuration. In [25], general program modularization techniques, such as aspects [18], framed aspects [26], mixins [33], hyperslices [34] or traits [15, 11], are evaluated with respect to their ability to implement features. Furthermore, the modularity concepts of recent languages, such as SCALA [28] or NEWSPEAK [12], can be used to represent product features. Although the above approaches are suitable to express feature-based variability, they do not contain designated linguistic concepts for features. Thus, DOP is most closely related and compared to FOP which considers features on a linguistic level. Apart from JAK [10], there are various other languages using the FOP paradigm, such as FEATUREC++ [4], FEATUREFST [5], or Prehofer's feature-oriented JAVA extension [30]. In [27], CAESARJ is proposed as a combination of feature modules and aspects extending FOP with means to modularize crosscutting concerns.

The notion of program deltas is introduced in [25] to describe the modifications of object-oriented programs. In [32], DOP is used to develop product line artifacts suitable for automated product derivation and implemented with frame technology [36]. In [31], delta-oriented modeling is extended to a seamless model-based development approach for SPLs where an initial product line representation is stepwise refined until an implementation, e.g., in DELTAJAVA, can be generated. The ordering of delta modules within the after clause resembles the precedence order on advice used in aspect-oriented programming, e.g., in ASPECTJ [21]. The constraints that are generated for delta modules in order to ensure safe product generation require the existence and non-existence of classes, methods or fields which is similar to the constraints used in [22]. Delta modules are one possibility to implement arrows in the category-theoretical framework for program generation proposed by Batory in [8].

7 Conclusion and Future Work

We have presented DOP, a novel programming approach particularly designed to implement SPLs. It allows the flexible modular implementation of product variability starting from different core products. Because core products are complete product implementations, they can be developed with well-established single application engineering techniques to ensure their quality. DOP provides a solution to the optional feature problem [20] by handling combinations of features explicitly.

For future work, we will extend DELTAJAVA with a Super() call as in JAK to directly express the access to methods that are modified by delta modules applied later during product generation in order to avoid a combinatorial explosion for combinations of optional features. Furthermore, we will improve the tool support for DELTAJAVA

with IDE functionalities, e.g., to show the set of applicable delta modules for a given feature configuration. In order to propose a process for the selection of core products, we are investigating how to the choice of the core product influences the design of the delta modules. Finally, we are aiming at efficient verification techniques of SPLs implemented by core and delta modules without generating the products. This work will use the information available in the delta modules to determine unchanged parts between different products to reuse verification results.

Acknowledgements We are grateful to Sven Apel, Don Batory and Roberto E. Lopez-Herrejon for many insightful comments on a preliminary version of this paper. We also thank the anonymous SPLC referees for detailed suggestions for improving the paper. This work has been partially supported by MIUR (PRIN 2009 DISCO) and by the German-Italian University Centre (Vigoni program 2008-2009). Ina Schaefer's work has been partially supported by the Deutsche Forschungsgemeinschaft (DFG) and by the EU project FP7-ICT-2007-3 HATS.

References

1. Expression Problem Product Line, Webversion. Available at `http://www.cs.utexas.edu/users/schwartz/ATS/EPL/`.
2. S. Apel, C. Kästner, A. Grösslinger, and C. Lengauer. Type safety for feature-oriented product lines. *Automated Software Engineering An International Journal*, 2010.
3. 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.
4. S. Apel, T. Leich, M. Rosenmüller, and G. Saake. Featurec++: On the symbiosis of feature-oriented and aspect-oriented programming. In *GPCE*, volume 3676 of *LNCS*, pages 125–140. Springer, 2005.
5. S. Apel and C. Lengauer. Superimposition: A language-independent approach to software composition. In *Software Composition*, volume 4954 of *LNCS*, pages 20–35. Springer, 2008.
6. D. Batory. Feature Models, Grammars, and Propositional Formulas. In *SPLC*, volume 3714 of *LNCS*, pages 7–20. Springer, 2005.
7. D. Batory. A Tutorial on Feature Oriented Programming and the AHEAD Tool Suite (ATS). In *GTTSE*, volume 4143 of *LNCS*, pages 3–35. Springer, 2006.
8. D. Batory. Using modern mathematics as an FOSD modeling language. In *GPCE*, pages 35–44. ACM, 2008.
9. D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Trans. Softw. Eng. Methodol.*, 1(4):355–398, 1992.
10. D. Batory, J. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Trans. Software Eng.*, 30(6):355–371, 2004.
11. L. Bettini, F. Damiani, and I. Schaefer. Implementing Software Product Lines using Traits. In *SAC, OOPS Track*, pages 2096–2102. ACM, 2010.
12. G. Bracha. Executable Grammars in Newspeak. *ENTCS*, 193:3–18, 2007.
13. P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison Wesley Longman, 2001.
14. B. Delaware, W. Cook, and D. Batory. A Machine-Checked Model of Safe Composition. In *FOAL*, pages 31–35. ACM, 2009.
15. 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.

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. M. Kuhlemann, D. Batory, and C. Kästner. Safe composition of non-monotonic features. In *GPCE*, pages 177–186. ACM, 2009.
23. J. Liu, D. Batory, and C. Lengauer. Feature oriented refactoring of legacy applications. In *ICSE*, pages 112–121. ACM, 2006.
24. R. Lopez-Herrejon and D. Batory. A standard problem for evaluating product-line methodologies. In *GCSE*, volume 2186 of *LNCS*, pages 10–24. Springer, 2001.
25. 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.
26. N. Loughran and A. Rashid. Framed aspects: Supporting variability and configurability for aop. In *ICSR*, volume 3107 of *LNCS*, pages 127–140. Springer, 2004.
27. M. Mezini and K. Ostermann. Variability management with feature-oriented programming and aspects. In *SIGSOFT FSE*, pages 127–136. ACM, 2004.
28. M. Odersky. The Scala Language Specification, version 2.4. Technical report, Programming Methods Laboratory, EPFL, 2007.
29. K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering - Foundations, Principles, and Techniques*. Springer, 2005.
30. C. Prehofer. Feature-oriented programming: A fresh look at objects. In *ECOOP*, volume 1241 of *LNCS*, pages 419–443. Springer, 1997.
31. I. Schaefer. Variability Modelling for Model-Driven Development of Software Product Lines. In *Intl. Workshop on Variability Modelling of Software-intensive Systems*, 2010.
32. I. Schaefer, A. Worret, and A. Poetzsch-Heffter. A Model-Based Framework for Automated Product Derivation. In *Proc. of MAPLE*, 2009.
33. 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.
34. 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.
35. S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe Composition of Product Lines. In *GPCE*, pages 95–104. ACM, 2007.
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.