DeltaJ 1.5: Delta-oriented Programming for Java 1.5

Jonathan Koscielny Sönke Holthusen
Ina Schaefer Sandro Schulze
Institut für Softwaretechnik und Fahrzeuginformatik,
Technische Universität Braunschweig
{j.koscielny, s.holthusen, i.schaefer,
sanschul}@tu-braunschweig.de

Lorenzo Bettini Ferruccio Damiani
Dipartimento di Informatica,
Università degli Studi di Torino
{bettini, damiani}@di.unito.it

Abstract

Delta-oriented programming (DOP) is a modular, yet flexible approach to implement software product lines. In DOP, a product line is implemented by a set of deltas, which are containers of modifications to a program. A delta-oriented product line is specified by its code base, i.e., the set of delta modules, and a product line declaration specifying the set of possible product variants. In this paper, we present DOP for JAVA 1.5 extending previous proof-of-concept realizations of DOP for simple core JAVA-like languages. The novel prototypical implementation DELTAJ 1.5 provides full integrated access to the object-oriented features of JAVA. The extensions include delta operations to fully integrate the JAVA package system, to declare and modify interfaces, to explicitly change the inheritance hierarchy, to access nested classes and enum classes, to alter field declarations, and to unambiguously remove overloaded methods. Furthermore, we improve the specification of the product line declaration by providing a separate language. We have evaluated DELTAJ 1.5 using a case study.

1. Introduction

A *software product line (SPL)* [10, 27] is a set of software systems (the products) with well-defined commonalities and variabilities. An SPL realizes a set of products by relying on a single *code base* describing software artifacts that have to be assembled to generate the possible products. Each product is described by a set of features. A feature [4] is an abstract description of functionality. The set of *valid feature configurations* determining the set of possible product variants can be described by a feature model [18].

Delta-oriented programming (DOP) is a flexible paradigm to implement SPLs. In the original formulation of DOP, called *Core DOP* [29], the product-line code base consists of: *(i)* a *core module* which contains the implementation of the base product variant (a JAVA program); and *(ii)* a set of *delta modules* that expresses modifications to the product introduced by the core module, for adding or removing features. A *product line declaration* connects the delta modules to the features. Additionally, a partial order on the delta modules (called the *application order*) is provided to capture the necessary dependencies between the delta modules (which are usually semantic requires relations) and to ensure that for every feature configuration a uniquely defined product is obtained. A product is generated by selecting the delta modules associated to the product features and incrementally applying them to the core module according to the application order. A subsequent formulation of DOP, called *Pure DOP* [28], introduced a conceptual simplification by dropping the notion of core module. Every product variant is generated only by applying delta modules where the first delta module that is applied can only contain additions.

Pure DOP for JAVA has been first formalized by using core calculi for JAVA: in [28] by using LIGHTWEIGHT JAVA (LJ) [34], and then in [8, 30] by using an imperative version of FEATHERWEIGHT JAVA (FJ) [17]. The prototypical language DELTAJ 1.1[1] used the constructs described in [8, 28, 30] for implementing DOP for a small subset of JAVA that roughly corresponds to the union of FJ and LJ augmented with a selection of primitive types (`int` and `boolean`) and the `String` type. DELTAJ 1.1 does not provide any API access (importing types) and does not support interfaces, packages and visibility modifiers for classes, methods or fields and many other JAVA constructs.

In this paper, we present DELTAJ 1.5, a prototypical language supporting DOP for full JAVA 1.5. The design of DELTAJ 1.5 goes beyond the notion of delta module developed in [8, 28, 30] (and adopted by DELTAJ 1.1) in order to support the integration of full JAVA 1.5 syntax within the change operations specified in the delta modules. This allows access to the object-oriented features of JAVA 1.5 within DOP which makes DELTAJ 1.5 one of the first programming languages realizing a modular programming paradigm that is fully integrated into JAVA. In particular, we developed a notion of delta module that supports program transformations involving the JAVA 1.5 package system, JAVA interfaces, the inheritance hierarchy, nested classes, enum classes and field qualifiers. We developed a new unified code removal operation (integrated into the delta module construct) and an improved language to express the product line declaration. We have built an Eclipse plug-in for DELTAJ 1.5 that provides tool support with syntax highlighting, syntax checking and product generation. The results presented in this paper are based on a preliminary investigation carried out in the first author's Bachelor thesis [22] by exploiting a case study developed in [14].

The paper is structured as follows. In Section 2, we recall the DOP constructs formalized [8, 28, 30]. The integration of DOP with full JAVA 1.5 is discussed in Section 3. An overview of the implementation of DELTAJ 1.5 is given in Section 4. The evaluation is presented in Section 5 and related work in Section 6. We conclude the paper by an outlook to future work.

[1] `http://deltaj.sourceforge.net/`

2. Delta-oriented programming

In this section, we recall the notion of Pure DOP as described in [8, 28, 30], present the currently available implementations of DOP for JAVA, and briefly illustrate one of them by an example.

2.1 Delta-oriented software product lines

According to [8, 28, 30], a delta-oriented product line consists of a *code base* and a *product line declaration*. The code base consists of a set of delta modules (which are the software artifacts to be assembled to generate the various products), while the product line declaration expresses the connection between the delta modules and the product line variability specified in terms of product features. A *delta module* is a container of modifications to a JAVA-like program. Three kinds of operations are provided to implement modifications.

1. Addition of a new class definition.
2. Modification of an existing class definition. Within the modifies class operation the following operations can be used.
   (a) Change of the immediate superclass.
   (b) Addition of a new field to the class.
   (c) Addition of a new method to the class.
   (d) Modification of an method defined in the class. This operation replaces the body of the method by a new method body. The new method body may contain calls of the form `original(...)`, that represent calls to the original version of the method. If the original version of the method is recursive, the recursive calls in the original method body are interpreted as calls to the new version of the method.
   (e) Removal of a field defined in the class.
   (f) Removal of a method defined in the class.
3. Removal of an existing class definition.

The product line declaration specifies:

1. The set of valid feature configurations.
2. A mapping that associates to each valid feature configuration a set of delta modules that must be used to generate the corresponding product.
3. A partial order of the delta modules (called the *application order*) that captures the necessary dependencies between the delta modules (which are usually semantic requires relations) and ensures that for every feature configuration a uniquely defined product is generated.

A product for a given feature configuration is generated by selecting the delta modules associated to the feature configuration and incrementally applying them to the empty program according to the application order.

The application of a delta module to a program will fail if a class to be removed or modified does not exist or, if for some modified class, some method or field to be removed does not exist, or if a method to be modified does not exist, or if some class, method or field to be added already exist. The generation of a product fails if the application of one of the required delta modules fails. The first delta module that is applied must only contain additions.

Specifying the application order as a partial (rather than a total) order on the set of delta modules, so that a product can be generated applying the selected delta modules in any total order that extends the application order, opens the possibility for automatic optimization of the product checking and product generation processes (see, e.g., [11]). However, it introduces the issue of ensuring *unambiguity* of product generation. That is, for every feature configuration, all the total orders of the selected delta modules that extend the application order generate the same product. In [28], an effective approach for ensuring unambiguity is introduced:

- Specify the application order as a total order on the sets of a partition of the delta modules.
- Ensure that: *(i)* if a delta module in a partition adds or removes a class, then no other delta module in the same partition may add, remove or modify the same class; and *(ii)* if a delta module in a partition adds, removes or modifies the `extends` clause or a field or a method of a class, then no other delta module in the same partition may add, remove or modify that `extends` clause, field or method.

2.2 Prototypical implementations of DOP for Java

In [29], the name DELTAJAVA has been used to refer to a language for Core DOP that was under development. Subsequently, three prototypical implementations of DOP have been made available and referred to by the generic name DELTAJ.

1. **DELTAJ 1.0 for Core DOP**. This prototype, available at `http://deltaj.sourceforge.net` is based on the notion of Core DOP as presented in [29].
2. **DELTAJ 1.1 for Pure DOP**. This prototype, available at `http://deltaj.sourceforge.net/new-version/`, is based on the notion of Pure DOP as presented in [8, 28, 30].
3. **DELTAJ 1.1 with Refactorings**. This prototype, available at `https://www.tu-braunschweig.de/isf/research/deltas/`, is a version of DELTAJ 1.1 for Pure DOP with an improved tool support which provides source code refactorings, described in [32].

The above implementations adopt the notion of delta module and product line declaration introduced in [8, 28, 30] (cf. Sect. 2.1) and consider products written in a small subset of JAVA that roughly corresponds to the union of FJ and LJ augmented with a selection of primitive types and the `String` type. That is, they consider only classes, fields and methods (without qualifiers), class-based inheritance with method overriding, field assignment, type casts, the `if` statement, the types `int`, `boolean` and `String`.

2.3 An example of DELTAJ 1.1 SPL

In order to illustrate the main concepts of DOP, as introduced in [8, 28, 30] and adopted by DELTAJ 1.1, we use a variant of the *expression product line* (EPL) [25]. The EPL is based on the *expression problem* [37], an extensibility problem that has been proposed as a benchmark for data abstractions' capability 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 the above grammar: print, which prints the textual representation the expression, and eval, which returns the value of the expression. The products in the EPL are described by two set of features, one concerning the data — `Lit, Add, Neg` — and another concerning the operations — `Eval` and `Print`. The features `Lit` and `Print` are mandatory, while the features `Add, Neg` and `Eval` are optional. Figure 1 shows the feature model of the EPL by means of a feature diagram [18].

The example illustrates the practical situation where an existing product has to be used as a basis for developing a family of prod-

ucts [23]. So, it does not provide an elegant implementation of the EPL from scratch.

Figure 2 contains a delta module for introducing an existing product (let us call it the *legacy* product), realizing the features `Lit`, `Add` and `Print`. Figure 3 contains the delta modules for adding the evaluation functionality to the classes `Lit` and `Add` of the legacy product. Figure 4 contains the delta modules for incorporating the `Neg` feature by adding and modifying the class `Neg` and for adding glue code required by the two optional features `Add` and `Neg` to cooperate properly—namely, when printing a sum of two expressions, a couple of surrounding parentheses is used to prevent ambiguities.[2] Figure 5 contains the delta module for removing the `Add` feature from the legacy product.

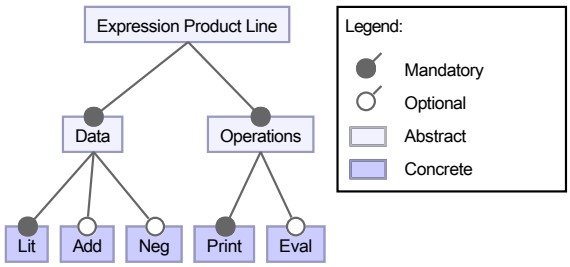

Figure 1: Feature model for Expression Product Line

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

Figure 2: DELTAJ 1.1 delta module for the product with features Lit, Add and Print

```
delta DLitEval {
  modifies Exp { adds int eval() {return 0;} }
  modifies Lit  {
    adds int eval() {return value;}
  }
}
delta DAddEval {
  modifies Add  {
    adds int eval(){return expr1.eval()+expr2.eval();}
  }
}
```

Figure 3: DELTAJ 1.1 delta modules for feature Eval in combination with Lit and Add

Figure 6 shows a DELTAJ 1.1 product line declaration for the EPL. It:

- Lists the product features.
- Describes the set of valid feature configurations by a propositional formula over the set of features (cf. the feature diagram in Figure 1).

[2] This example shows that DOP provides an elegant way to counter the optional-feature problem [19], where two optional features require additional glue code to cooperate properly.

```
delta DNeg {
  adds class Neg extends Exp {
    Exp expr;
    Neg(Exp a) {expr1 = a;}
  }
}
delta DNegPrint {
  modifies Neg {
    adds void print() {
      System.out.print("-"); expr.print();
    }
  }
}
delta DNegEval {
  modifies class Neg  {
    adds int eval() {return (-1) * expr.eval();}
  }
}
delta DOptionalPrint {
  modifies Add  {
    modifies void print() {
      System.out.print("("); original();
      System.out.println(")");
    }
  }
}
```

Figure 4: DELTAJ 1.1 delta modules for feature Neg in combination with Print and Eval

```
delta DremAdd { removes Add; }
```

Figure 5: DELTAJ 1.1 delta module for removing feature Add

```
spl EPL {
  features Lit, Add, Neg, Print, Eval
  configurations Lit && Print
  deltas
    [ DLitAddPrint,
      DNeg when Neg ]
    [ DremAdd when !Add ]
    [ DLitEval when Eval,
      DAddEval when (Add && Eval),
      DNegEval when (Neg && Eval),
      DNegPrint when Neg,
      DptionalPrint when (Add && Neg)  ]
  product Basic from EPL : {Lit, Print}
  product Full from EPL : {Lit, Add, Neg, Print, Eval}
}
```

Figure 6: DELTAJ 1.1 declaration of the EPL

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

Figure 7: JAVA code for product `Basic` (generated by DELTAJ 1.1)

- Attaches to each delta module an application condition (represented by a propositional constraint over the set of features) specifying for which feature configurations the delta module has to be applied. Since only feature configurations that are valid according to the feature model are used for product generation, the application conditions, given in the `when` clauses, are understood as a conjunction with the formula describing the set of valid feature configurations.
- Specifies the application order of the delta modules by defining a total order on the sets of a partition of the delta modules. Deltas in the same set of the partition can be applied in any order, but the order of the sets must be respected. The ordering is specified by writing an ordered list of the delta module sets which are enclosed by `[ .. ]` after the keyword `deltas`.

```
1  class Exp {
2    void print() {}
3    int eval() {return 0;}
4  }
5  class Lit extends Exp {
6    int value;
7    Lit(int n) {value = n;}
8    void print() {System.out.println(value);}
9    int eval() {return value;}
10 }
11 class Neg extends Exp {
12   Exp expr;
13   Neg(Exp a) {expr1 = a;}
14   void print() {
15     System.out.print("(-"); expr.print();
16     System.out.println(")");
17   }
18   int eval() {return (-1) * expr.eval();}
19 }
20 class Add extends Exp {
21   Exp expr1; Exp expr2;
22   Add(Exp a, Exp b) {expr1 = a; expr2 = b}
23   void print$DoptionalPrint() {
24     expr1.print(); System.out.print("␣+␣");
25     expr2.print();
26   }
27   void print() {
28     System.out.print("("); print$DoptionalPrint();
29     System.out.println(")");
30   int eval() {return expr1.eval() + expr2.eval();}
31 }
```

Figure 8: JAVA code for product `Full` (generated by DELTAJ 1.1)

- Declares two products: the product `Basic` that contains only the mandatory features, and the product `Full` that contains all the features.

Figure 7 and 8 show the `Basic` and the `Full` generated products, respectively. Note that, in the class `Add` in the code of `Full` product, the name `print$DOptionalPrint` corresponds to the original version of the method `print` modified by the application of the delta module `DOptionalPrint`.

3. Towards DELTAJ 1.5

The DOP language constructs, as introduced in [8, 28, 30] and adopted by DELTAJ 1.1, are unfit for full JAVA 1.5. This section illustrates the shortcomings of the DELTAJ 1.1 language constructs and introduces the new or revised DOP constructs that we have developed for integrating DOP with JAVA 1.5. JAVA 1.5 knows four different kinds of user-defined types: classes, interfaces, enum classes and annotations. Hence, we refer to *type* where operations affect more than one type, otherwise we refer to the specific type.

DELTAJ 1.5 supports the basic operations of DOP (see Section 2.1). It can *add* methods and fields to classes with the operation `adds member`. The operation for adding classes is replaced by an operation for adding *Java Compilation Units* (see below). Types and methods can be *modified* with the operation `modifies member` and types, methods and fields can be *removed* from the source code base with the operation `removes member`. The `original(...)` call is also part of DELTAJ 1.5. It can be used to access the original method body when a method is modified.

Packages. JAVA 1.5 provides a package system to organize types in packages for structuring the source code of an application or an API. DOP provides an orthogonal mean to organize source code: the delta module, which groups source code fragments that are related to program features. The challenge is to develop a DOP construct for dealing with the package system. The DELTAJ 1.1 syntax provides no syntactic constructs to add a package or import declaration.

Assuming that each delta module is stored in a separate source file, our first attempt to add the JAVA package system to DOP could be that each delta module may contain a package declaration and several import declarations at the beginning of the delta module source file—this would look as shown in Figures 9 and 10. However, then, it would not be possible to modify or remove the package declaration of a type or the import clauses. Additionally, this would severely restrict the possibilities for modularizing code with delta modules, as we can see from the following scenarios:

1. Assume two classes that should naturally be organized in different packages (e.g., for the data structure and one for the user interface) are both needed for a feature. Then, it would not be possible to modify them within a single delta module.
2. Assume that only one class added by a delta module (e.g., class `C2` in Figure 9) needs to import the interface `List`. If we remove this class by another delta module (Figure 11), the result would be an unused import statement.
3. Assume a class (`C1` in Figure 9) is added by a delta module belonging to a particular package and another delta module modifies or removes this class, but belongs to another package. Then, there is no opportunity to identify this class by its qualified name (e.g., see the modification of the class `C1` in `delta2` in Figure 10).
4. Furthermore, it would be unclear to which package a type belongs in the generated product variant if it is added by a delta module in one package and modified by a delta module in another package. One possibility would be to generate new package declarations for the product variants (see Figure 11), but this would destroy the packaging information after the generation process.

To address the above issues, we have revised the delta operation `adds` to also handle package and import declarations. The new `adds` operation allows to add a complete *Java Compilation Unit* with the package declaration, the import list and the type definition. Imports and package declarations are distinct for each type declaration. This makes it possible to add types to different packages within one delta module. The new `adds` operation is illustrated by the example in Figure 13. Moreover, in DELTAJ 1.5, we are able to introduce *generic types* with the new adds operation.

With the JAVA package declaration, it is possible to identify a type distinctly by its qualified name. Following the extension of the `adds` operation, this leads to extensions of the `modifies` and `removes` operations. We provide the qualified name of a type to modify or remove it. These extended operations are presented in Figure 14. We also introduce the opportunity to define a type's package implicitly by writing the qualified name in the type declaration (see `C2` in `d1`).

To get a delta-oriented access to the import list and the package declaration, we provide a syntax extension to the adds, modifies and removes operations inside the modifies block. To modify the package declaration, we provide the operation `modifies package new.pkg`. To add or remove an import, we provide the operations `{adds | removes} import qualified.Name`. Within these new operations, we are able to modify the package system. A type with modified package declaration may lead to type errors, because types from the original package will not be found after this modification. The opportunity to modify the import list avoids this error. We do not allow types with and types without a package declaration side by side in the source code base. Hence, we do not provide a `removes` package operation. When generating the product variants, we extend the qualified names of generated types with a prefix, containing the product line's and product variant's

```
package org.pkg1;

import java.util.List;

delta delta1 {
  adds class C1 {
    private int i;
  }
  adds class C2 implements List { ... }
}
```

Figure 9: Adding a class with an import

0.45

```
package org.pkg2;

delta delta2 {
  modifies C1 {
    adds private int j;
  }
  removes C2;
}
```

Figure 10: Removing a class, but leaving the import

0.45

```
package spl.variante;

import java.util.List;

class C1 {
  private int i;
  private int j;
}
```

Figure 11: Resulting class with unused import

Figure 12: First attempt to add packages to DOP

```
delta d1 {
  adds {
    package org.pkg1;
    class C1 {...}
  }
  adds {
    import java.util.List;
    class org.pkg2.C2<T extends Number>
        implements List<Number> { ... }
  }
}
```

Figure 13: Delta adding classes in separate packages

```
delta d2 {
  modifies org.pkg2.C2 {

    modifies package com.pkg2;

    adds import java.util.MyList;
    removes import java.util.List;

    removes interfaces List<Number>;
    adds interfaces MyList:
  }
  removes org.pkg1.C1;
}
```

Figure 14: Delta changing the package and import structure

```
package spl.variante.org.pkg2;

import java.util.MyList;

class C2 implements MyList { ... }
```

Figure 15: Resulting variant

Figure 16: The new adds operation of DELTAJ 1.5 holding the complete Java Compilation Unit

```
delta delta1 {
  modifies my.pkg.MyInterface {
      adds import my.other.pkg.MyList;
      removes import my.other.pkg.MyList2;
      adds interfaces MyInterface2:
      adds public static final int MY_INT_CONST = 3;
      adds int methodSingature1(...);
      removes methodSignature2(...);
    }
  }
}
```

Figure 17: Modification of interface declarations

name. The extension of the qualified names prevents ambiguities of type names in the source code of generated variants.

Interfaces. Interfaces can be declared and added within the new adds operation. In order to be able to alter interfaces, we also introduce modification operations for interfaces. Figure 17 gives an overview of DELTAJ 1.5's capabilities to deal with interfaces. Within an interface declaration, we can add and remove methods declared by an interface as well as constants. Furthermore, we can add and remove imports as well as super-interfaces.

Inheritance. JAVA provides single inheritance to extend functionality of a class. To overcome limitations of single inheritance, JAVA also provides interfaces. A class has a unique name, an optional superclass and can contain a list of interfaces which are implemented. A complete class definition can look like this: `class A extends B implements C, D { ... }` where A is the classname, B is the superclass and C and D are interfaces. For DELTAJ 1.1, the keyword `extending` was defined to modify the superclass. Figure 18 shows the usage of this operator.

However, in DELTAJ 1.1, we do not have operations to add or remove a list of interfaces which have to be implemented in a class. Furthermore, DELTAJ 1.1, does not have keywords for explicitly adding or removing superclasses, instead this is encoded by the `extending` keyword. JAVA allows to define classes without an explicit superclass implicitly inheriting form `java.lang.Object`. The keyword `extending` changes the implicitly defined superclass `java.lang.Object`, if no superclass was defined. For removing the superclass definition, this can be encoded by the construct `modifies A extending Object { ... }`.

In order to provide more explicit control for superclass and interface implementations, we remove the keyword `extending` in DELTAJ 1.5. Instead, we introduce a new keyword `superclass`. It can be combined with the already known keywords `adds`, `modifies` and `removes`. We now can add, modify or remove the superclass definition of a class directly. For delta-oriented manipulation of the interface list, we provide a new keyword `interfaces`. It can be combined with the `adds` and `removes` operations. Thus, we can add and remove interfaces implemented by a class or superinterfaces of interfaces. There is no syntactical possibility to modify the interface list as this can be expressed by addition and removal

```
delta delta1 {
  adds class C1 {...}
  adds class C2 extends C1 {...}
  adds class C3 implements AnInterface {...}
}

delta delta2 {
  modifies C1 extending C3 {...}
  modifies C2 extending C3 {...}
}
```

Figure 18: The `extending` keyword in DELTAJ 1.1

of particular interfaces in the list. This syntax for superclasses and interfaces unifies the usage of the existing delta operations and is presented in Figure 19.

```
delta delta1 {
  modifies my.pkg.ClassName {

    adds superclass qualified.ClassName1;
    modifies superclass qualified.ClassName2;
    removes superclass;

    adds interfaces qualified.InterfaceName1, ...;
    removes interfaces qualified.InterfaceName2, ...;
  }
}
```

Figure 19: Superclasses and Interfaces in DELTAJ 1.5.

Nested types. In Java, it is possible to nest a type into another type. With the existing syntax of DELTAJ 1.1, we are not able to deal with nested types. To overcome this, we provide a syntax extension for nested types (see Figure 20). A nested type is identified by its name. To avoid a collision with fields, we introduce a new keyword `nested` which identifies a nested type by its name. This keyword can be combined with the existing keywords for adding, modifying and removing class members. For the definition and modification of the nested type itself, we can use the same syntax as for types.

```
delta delta1 {
  adds {
    class C1 {
      ...
      public class Inner1 {...}
    }
  }
  modifies C2 {
    adds nested{
      public class Inner2 {...}
    }
    modifies nested Inner3 {...}
    removes nested Inner4;
  }
}
```

Figure 20: Nested classes in DELTAJ 1.5

Enumerations. An enumeration class defines an enumeration type. In JAVA, we are able to define enumeration types, but in DELTAJ 1.1, there are no operations to alter an enumeration type. Hence, we need a syntax extension to use enumeration types in DELTAJ 1.1. Our goal is to add items to an enumeration type and to remove items. We do not need to modify enumeration items, because an enumeration type can be interpreted as a list of identifiers. Modifying a list means to add or to remove items of the list. Thus, we provide the keywords `adds` and `removes` to add and remove a list of enumeration items. Figure 21 shows the new syntax for modifying an enumeration type.

```
delta delta0 {
  adds {
    public enum E { SEND, RECEIVE }
  }
}
delta delta1 {
  modifies E {
    removes SEND;
    adds ENCRYPTED, PLAIN;
  }
}
```

Figure 21: Enumerations in DELTAJ 1.5

Fields and Methods. In DELTAJ 1.1, fields only have a type and a name. Thus, there is no `modifies` operation for fields—a field's type can be changed by removing the field and adding it again with the other type. In JAVA 1.5, however, fields can have access modifiers which may have to be changed in a delta. The above workaround for modifying a field is not practical, since it requires to use two different delta operations, e.g., when a field should be modified to become final, while the type and the other modifiers should not be changed. Therefore, in DELTAJ 1.5, we introduce a `modifies` operation for fields, which flexibly supports to modify both the type and the different access modifiers. With the new `modifies field` operation, we are able to refer only to the modifiers we want to change. All other modifiers, set when adding the field, are not touched.

```
delta delta1 {
  adds {
    class C1 {
      private int i, j;
      public static int k, l;
      private List<Number> list;
    }
  }
 }
delta delta2
  modifies C1 {
    modifies i {public};
    modifies j {static};
    modifies k {!static final};
    modifies l {static !final Integer};
    modifies list {protected List<Integer>};
  }
}
```

Figure 22: Deltas introducing and modifying fields

```
class C1 {
  public int i;
  private static int j;
  private final int k;
  private static Integer l;
}
```

Figure 23: Resulting class after applying delta1 and delta 2

Figure 24: Modification of fields in DELTAJ 1.5

Figure 24 shows the usage of this operation. The syntax of the new field modification operation consists of the keyword `modifies`, the field's name and a block, surrounded by curly

braces. Between the braces one can override the access modifier and type property of the field or remove modifiers. To remove a modifier, it is marked with an exclamation mark. Every modifier mentioned in the modifies clause overwrites the original modifier, but modifiers that are not mentioned will not be overwritten. Figure 24 shows different `modifies field` operations: In line 12, the visibility is changed from `private` to `public`. Line 13 gives the field `j` the additional property of being static. The field `k` looses the property static in line 14 and is set to final. In the case, that one does not know if a property is set or unset, we allow to set this property twice (see lines 5 and 15): `l` is introduced as static, but this property is set again. Furthermore, the type modification can deal with generic types. In line 16, the type `List<Number>` of field `list` is changed to `List<Integer>`. It is also possible to introduce a generic type to a field which had a non generic type and vice versa. The instantiation (`<...>`) of a generic type cannot be modified directly. As a workaround one can exchange the complete type, e.g., from `List<Number>` to `List<Integer>`.

The modification of methods in DELTAJ 1.5 replaces the methods body with a new body like in DELTAJ 1.1. Within the new body the original body can be accessed by calling it with its special method call `original(...)`. The modification of the methods properties, i.e., parameters, modifiers and return type is currently not possible. However, a similar modification operation as for fields is possible and currently being designed.

JAVA 1.5 allows to overload method names. Because a method is identified by its signature, it is possible to define several methods with the same name, but with a different signature. DELTAJ 1.1 provides the operation, `removes methodName` to remove a method. This operation will not work safely in a context where methods are overloaded. A problem also occurs if a class has a field and a method with the same name. Because the keyword `removes` is only followed by the name of the member to be removed in DELTAJ 1.1, it is ambiguous which member will be removed.

To overcome this issue, we extend the removal operation for methods in DELTAJ 1.5 by its signature and fields by its name. We propose the syntax `removes methodName(...)` and `removes fieldName` to remove methods and fields, respectively, shown in Figure 25. The removal of methods just needs to include the types of the method parameters, for fields it just needs the name.

```
1  delta delta1 {
2    adds {
3      class C1 {
4        private int i, j, k;
5        public void doSomething() {...};
6        public void doSomething(int i){...};
7        public void doSomething(Number n){...};
8      }
9    }
10 }
11 delta delta2 {
12   modifies C1 {
13     removes i;
14     removes k;
15     removes doSomething();
16     removes doSomething(int);
17   }
18 }
```

Figure 25: Removal operations in DELTAJ 1.5

4. Implementation

There is an existing Eclipse plug-in for DELTAJ 1.1[3]. It was implemented with Xtext[4], an Eclipse framework to implement DSLs and its corresponding Eclipse plug-ins. Because of the existing implementation, we decided to use Xtext again to implement a new Eclipse plug-in for DELTAJ 1.5. This section will give an overview of the main ideas of implementing DELTAJ 1.5 with Xtext.

4.1 Xtext

Xtext [1, 7] is a *language workbench* (such as MPS [38] and Spoofax [20]): it takes as input a grammar definition, specified in a DSL similar to EBNF, and it generates a parser, an abstract syntax tree, and Eclipse-based tooling features (e.g., editors with syntax highlighting, code completion and static error highlighting). For this reason, it is much more powerful than traditional parser generators (such as Flex/Bison [24] or ANTLR [26]) which only deal with the syntax of a language.

The first task in Xtext is to write the grammar of the language using an EBNF-like syntax. The grammar of our language defined in Xtext is partially shown later in Figure 28-(b). Using this grammar, Xtext generates an ANTLR parser [26]. During parsing, the AST is automatically generated by Xtext in the shape of an EMF model (Eclipse Modeling Framework [33]). Thus, we can manipulate the AST using all mechanisms provided by EMF itself.

Most of the code generated by Xtext can already be used as it is, since the main aim of Xtext is to infer good and sensible default implementations from the grammar definition itself. However, most constraint checks, like type checking, have to be adapted by customizing some classes used in the framework. The customizations are "merged" into Xtext's classes using Google-Guice, a *dependency injection* framework [13].

Xbase [12] is an extendable and reusable expression language that is part of Xtext and can be embedded in a DSL. Xbase integrates completely with the JAVA platform and JDT (Eclipse JAVA development tools). In particular, Xbase reuses the JAVA type system (including generics) without modifications; this means that a language that uses Xbase will automatically and transparently access any JAVA type. In order to reuse Xbase's JAVA type system, we have to map the concepts of our language into the *Java model elements* of Xbase (e.g., classes, fields, methods, etc.). Such mapping will let Xbase automatically implement type checking for the expressions.

We have considered to use Xbase in our implementation for the body of JAVA methods, but we decided against it for two main reasons. First of all, Xbase expressions are not JAVA expressions, since one of the main goals of Xbase is removing most of the "syntactic noise" from JAVA (e.g., types of variable declarations can be inferred by Xbase itself) and providing more advanced features (e.g., lambda expressions). Thus, using Xbase would have implied to implement deltas for a language that is similar to JAVA, but not strictly compliant with it from the syntactic point of view. Most of all, Xbase strictly requires that each Xbase expression (i.e., in our case, each method body) is mapped exactly into one JAVA model expression. This is not possible in our language since we need to generate different versions of the same method according to the modification operations. In the future, we might consider to further investigate if we can bypass these issues in order to benefit from the automatic JAVA integration offered by Xbase.

4.2 Extending DELTAJ 1.1 to DELTAJ 1.5

This section gives an overview of the implementation of the language plug-ins for DELTAJ 1.5.

[3] `https://www.tu-braunschweig.de/isf/research/deltas`

[4] `http://www.eclipse.org/Xtext`

4.2.1 DELTAJ 1.5 Grammar

As described above, the implementation starts with the language's grammar. To implement DELTAJ 1.5, we took a JAVA 1.5 ANTLR grammarto implement the JAVA core, translated it into the Xtext grammar language and refactored some grammar rules to have better access to model elements. As an example of the refactorings, Figure 28 shows the `statement` rule of both grammars. In the ANTLR grammar (Figure 28-(a)), the statement rule is concrete and implements all possible statements. The keyword decides which kind of statement it is and which other keywords and elements have to occur in the source code (e.g., `'for' '(' forControl ')' statement` in line 6). This implementation of the statement rule is ambiguous. To identify the concrete statement, a complex *if-else if-. . . -else statement* has to check all child elements, if information from a concrete statement has to be extracted or modified. To overcome this, we have refactored this rule. In our Xtext grammar (Figure 28-(b)), the statement rule is an abstract rule which can be one of a number of subrules. Each JAVA statement occurs as a unique rule (e.g., `For: 'for' '(' control=ForControl ')' statement=Statement;` in ll. 16-18). The refactoring leads to less code, as the decision which kind of statement is used is made with one `instanceof`-operation. The delta-oriented part of the grammar was extended by a set of new rules. These rules implement the syntax extensions we have presented in the previous section. Some of the existing rules of DELTAJ 1.1 were refactored to implement the modified operations. As a result, we are able to provide a grammar which allows to add classes with full JAVA syntax and most of a class' properties can be modified or removed within DOP.

4.2.2 Product line declaration

To specify the product line declaration in DELTAJ 1.5, we provide a novel *domain specific language (DSL)* (see Figure 29 for the corresponding product line declaration of the expression product line of Section 2.3). This allows a separate specification of the product line declaration in DELTAJ 1.5 which in DELTAJ 1.1 was still integrated with the definition of the delta modules. The DSL in DELTAJ 1.5 contains

1. the set of all features of the product line (l. 2),
2. the set of all delta names which implement the features (ll. 3-4),
3. a set of constraints which describes the valid configurations of the feature model (ll. 5 - 7),
4. a set of partitions which contains the mappings between deltas and features (ll. 8 - 17) and
5. the set of all products which can be generated in the product line (ll. 18 - 21).

A feature model constraint is a propositional formula which describes dependencies between features. We provide the operators *and* (`AND` or `*` or `&`), *or* (`OR` or `+` or `|`), *exclusive or* (`XOR` or `^`), *not* (`!` or `-` or `~`) and *implies* (`IMPLIES` or `=>`) to build a constraint. The constraints block is followed by the partitions block (`Partitions {...}`). A partition consists of a set of *when clauses*. A when clause begins with a list of deltas followed by the keyword `when` and a propositional formula like in the constraints block. The `when` clause determines for which feature combinations a delta is applied. This allows saving to copy the when clauses for several deltas. The end of a partition is marked by a semicolon. The order of the partitions implies a partial order of the deltas. The deltas in a single partition have no special application order, but deltas in an earlier partition are applied before the deltas of a later partition. The last block is the products block (`Products {...}`). It contains a set of products of the product line. A product starts with the product's

```
1  statement:
2    block |
3    ASSERT expression (':' expression)? ';' |
4    'if' parExpression statement
5      (options {k=1;}:'else' statement)? |
6    'for' '(' forControl ')' statement |
7    'while' parExpression statement |
8    'do' statement 'while' parExpression ';' |
9    'try' block ( catches 'finally' block |
10     catches |
11     'finally' block) |
12   'switch' parExpression '{'
13     switchBlockStatementGroups '}' |
14   'synchronized' parExpression block |
15   'return' expression? ';' |
16   'throw' expression ';' |
17   'break' Identifier? ';' |
18   'continue' Identifier? ';' |
19   ';' |
20   statementExpression ';' |
21   Identifier ':' statement;
```

Figure 26: Statement rule of JAVA 1.5 (ANTLR)

```
1  Statement:
2    Block | Assert | If | For | While | Do | Try |
3    Switch | Synchronized | Return | Throw | Break |
4    Continue |  {Empty} ';' | StatementExpression1 |
5    Identifier;
6
7  Assert:
8    'assert' expression+=Expression
9    (':' expression+=Expression)? ';';
10
11 If:
12   'if' expression=ParExpression
13   ifStatement=Statement (=>
14   'else' elseStatement=Statement)?;
15
16 For:
17   'for' '(' control=ForControl ')'
18   statement=Statement;
```

Figure 27: Statement rule of JAVA 1.5 (Xtext)

Figure 28: Grammar translation an refactoring

name followed by an assignment and the set of the features which should be implemented by this product.

4.2.3 Product generation

We re-designed the process of product generation based on a second plugin implementing the *product line declaration* explained in the previous subsection. The functionality of this plug-in helps the developer to easily specify a valid delta-oriented SPL. The plug-in contains validation of the feature model and of the mapping of deltas to feature combinations (and vice versa). For the defined deltas, it validates, if these deltas exists in the source code If not, these deltas will be automatically generated (with an empty body). The plug-in provides information if a feature or delta is never used.

To generate a product for a given feature configuration, we need to find the necessary deltas to be applied. First, we create a conjunction of all feature names. If a feature is not selected in the given feature configuration, it is negated. This Boolean expression is as follows for a feature configuration of the EPL (see Figure 29): $\texttt{MyEPL} = \texttt{Lit} \wedge \texttt{!Add} \wedge \texttt{Neg} \wedge \texttt{Print} \wedge \texttt{Eval}$. Then, for each when-clause `W` of each partition, a Boolean expression is constructed which states that the `when` clause is implied by the feature configuration, e.g., $\texttt{ToApply} = \texttt{MyEPL} \rightarrow \texttt{W}$. The expression `ToApply`

```
SPL EPL {
  Features = {Lit, Add, Neg, Print, Eval}
  Deltas = {DLitAddPrint, DLitEval, DAddEval, DNeg,
      DNegPrint, DNegEval, DOptionalPrint, DremAdd}
  Constraints {
    Lit & Print;
  }
  Partitions {
    {DLitAddPrint, DNeg} when (Neg);

    {DremAdd} when (!Add);

    {DLitEval} when (Eval),
    {DAddEval} when (Add & Eval),
    {DNegPrint} when (Neg),
    {DOptionalPrint} when (Add & Neg);
  }
  Products {
    Basic = {Lit, Print};
    Full = {Lit, Add, Neg, Print, Eval};
  }
}
```

Figure 29: Product line declaration of the EPL in DELTAJ 1.5

is evaluated with a *SAT Solver*[5]. If the expression `ToApply` is satisfiable, the deltas of this partition have to be applied to build the product. Internally, we add the names of these deltas to a list of applicable deltas. The order of this deltas is given by the order of their occurrence in the partitions.

After deltas that have to be applied have been determined, the source code for this product is generated. To do that, we extract the implementation of all necessary deltas and apply the contained delta operations. The generation process is a model-to-model transformation. The input model is an instance of the EMF meta-model which is created by Xtext during the parsing. The root element of this model is `DeltaJUnit`. It holds elements of the type `Delta` from the list of delta modules that have to be applied. The output model is an instance of this EMF meta-model, too. Here, the root element is the `JavaCompilationUnit`, i.e., the root element of the JAVA part of the DELTAJ 1.5 grammar.

For each class, which is added, the class elements are copied from the input to the output model, and the copy is referenced by an instance of a wrapper class for the Java Compilation Unit (JCU). This wrapper class provides a method to apply modify actions, which are elements of the input model. This method identifies the modifies or removes actions, extracts the information that has to be modified and identifies the element in the JCU where this modification has to be made. Then, it modifies the identified element. After all deltas are applied, the generated classes are serialized and written into the JAVA source files in the file system.

Since the JAVA code is generated in source folders of an Eclipse project, Eclipse will automatically compile such generated files. Thus, the programmer does not need to invoke the JAVA compiler manually on the generated JAVA code.

5. Evaluation

With DELTAJ 1.5, extended to full JAVA 1.5 syntax, we can now exploit the full power of DOP for large-scale software development. As a first step, we provide an initial case study, where we compare DELTAJ 1.5 with a plugin-based approach, that is, the ECLIPSE RCP. The plugin-based approach is widely used for flexible and modular software development and thus aims at supporting variant-rich systems, similarly as DELTAJ 1.5. To guide our case study, we formulate the following research questions:

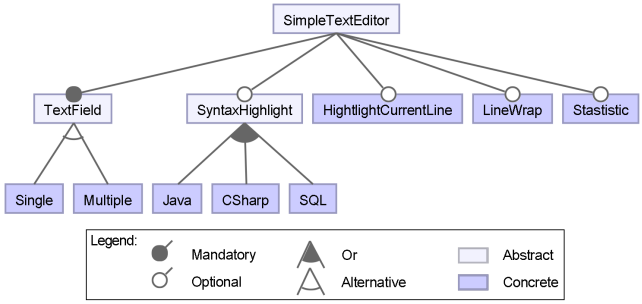

Figure 30: Feature diagram of the STE

RQ1 Is DELTAJ 1.5 *expressive enough* to implement a real world, variant-rich software system?

RQ2 Is the architecture of a variant-rich software system *less redundant* and *more intuitive* than the ECLIPSE RCP approach?

With RQ 1, we want to demonstrate that, with the new DELTAJ 1.5, we unleash the full power of Java to the delta-oriented approach and thus, we are capable to support large-scale development similarly to well-established approaches. Moreover, with RQ 2, we show that implementing variable software systems is more flexible (in terms of granularity) and less redundant with our language (e.g., no boilerplate code).

SimpleTextEditor SPL. Next, we present the subject system of our case study, *SimpleTextEditor SPL* (STE) [14], which we designed as SPL and implemented with DELTAJ 1.5 and as a set of plug-ins for the ECLIPSE RCP. The STE product line consists of 11 features, resulting in 128 valid variants. In Figure 30, we show the corresponding feature diagram. Basically, the STE provides a text area, which is mandatory (feature `TextField`), but has alternatively a single area (`Single`) or a tabbed multiple areas (`Multiple`). Moreover, different optional features exist for extended capabilities, such as syntax highlighting for different programming languages, line wrapping or statistics about active documents within the editor.

As mentioned earlier, two versions of the STE product line exists (for DELTAJ 1.5 and ECLIPSE RCP, respectively). Both are based on an implementation with plain JAVA, which we decomposed and migrated manually to the respective implementation approach. This way, we want to ensure comparability between both (variable) implementations. With DELTAJ 1.5, each feature is mapped to a *delta module* that encompasses all code artifacts, related to this feature. Similarly, the ECLIPSE RCP implementation provides a plug-in for each feature. Dependencies between features are mapped accordingly as plug-in dependencies. Product variants are generated in DELTAJ 1.5 by applying the delta operations to the variant's source code base. An ECLIPSE RCP variant is created by installing its corresponding plug-ins.

The implementation of the STE uses a lot of widely used language concepts of JAVA 1.5 including *abstract classes*, *annotated methods*, *anonymous classes*, *arrays*, *exception handling*, *generic types*, *imported types*, *interfaces*, *loops*, *modifiers* and *packages*. This is of particular importance for RQ 1. The STE also uses types from the JAVA API (`java.util` and `java.io`) and the ECLIPSE SWT API[6]. An observation we made during decomposition is that

[5] We use the Prop4J (part of the FeatureIDE project [36]) and Sat4J [6] APIs to evaluate this expression.

[6] `http://www.eclipse.org/swt/`

DELTAJ 1.5	Full variant		Basic variant		AVG
	M	S	M	S	
Files	10	10	8	8	9
Classes	37	34	24	21	28.75
LOC	1150	1088	616	554	852
MPC	2.81	2.94	2.42	2.57	2.61
MC	440	401	286	247	343.5

ECLIPSE RCP	Full variant		Basic variant		AVG
	M	S	M	S	
Files	25	25	16	16	20.5
Classes	56	54	36	34	45.25
LOC	1275	1215	792	732	1003.5
MPC	2.25	2.26	2.31	2.32	2.29
MC	455	415	324	284	369.5

Table 1: Result of the measurement of the STE. It shows the number of files (*Files*), number of classes (*Classes*), lines of code (*LOC*), average methods per class (*MPC*) and method calls (*MC*) for the four products of the STE.

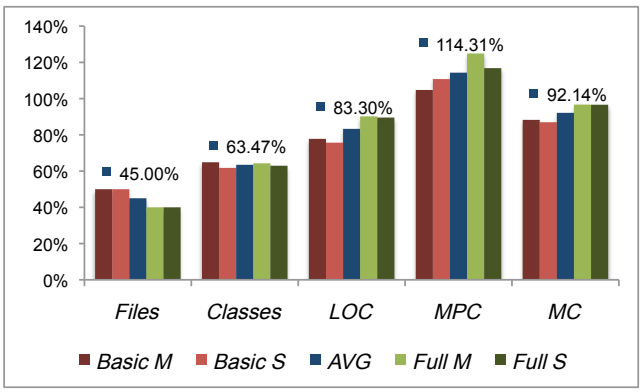

Figure 31: Results of the quantitative evaluation. Each bar represents the DELTAJ 1.5 measurement relatively to the ECLIPSE RCP measurement.

with DELTAJ 1.5, it is possible to keep the source code structure of the original implementation. In contrast, for the ECLIPSE RCP implementation, we had to adapt the structure so that it fits the extension points mechanism, needed to access a feature's implementation.

Metrics. The analysis and evaluation of DELTAJ 1.5 source code is difficult, because of missing tool support to measure variability-aware metrics. To overcome this problem, we generate products, measure the generated JAVA source code and transfer the results to the DELTAJ 1.5 source code. To cover all features, we measure four variants for each implementation, respectively: A basic (minimal) variant and a full variant for each alternative feature `Single` and `Multiple` .

We used different source code measures for answering our research questions and show the results in Table 1. To evaluate the expressiveness (RQ1), we are looking for JAVA 1.5 language features used in the source code. As qualitative measure, we count the method calls (row *MC*), because we consider methods as the most natural building blocks (even more than classes) of JAVA programs. To evaluate the complexity (RQ2) of the source code, we measure the number of files (row *Files*) and classes (row *Classes*). Moreover, we measured size-related metrics, in particular lines of code (row *LOC*) and the average number of methods implemented in a class (row *MPC*).

Result. For a better comparison of DELTAJ 1.5 and ECLIPSE RCP, in Figure 31, we show the ratio between the DELTAJ 1.5 and the ECLIPSE RCP metrics. Each bar shows the relative value of DELTAJ 1.5 compared to ECLIPSE RCP. For instance, the source code's base size of the DELTAJ 1.5 variant is only 75.68% of the corresponding ECLIPSE RCP variant's source code size. Next, we present the concrete results and interpret them with respect to the research questions:

1. DELTAJ 1.5 *is expressive enough* (RQ1) to implement a real world, variant-rich software system. An inspection of the source code (paragraph *implementation details*) revealed that the STE makes use of many JAVA 1.5 language concepts, which are only available with DELTAJ 1.5 but not within DELTAJ 1.1. Moreover, the DELTAJ 1.5 implementation demonstrates that these concepts integrate seamlessly with the delta-oriented concepts such as `modifies` or `removes`. Finally, based on the fact that we reused most of the original structure of STE, we argue that DELTAJ 1.5 is highly expressive in the sense that it can be easily adopted in existing implementations without a migration overhead.
2. DELTAJ 1.5 source code *is less redundant* (RQ2). Our data reveal that implementing variability (w.r.t. to our STE product line) with DELTAJ 1.5 is less ambiguous than with ECLIPSE RCP. First, for DELTAJ 1.5, only half of the files (45%) is needed, compared to ECLIPSE RCP. Thinking of large software systems, we argue that this causes less ambiguity, and searching for concrete feature implementation is not hindered or obfuscated by meaningless files. Second, DELTAJ 1.5 implements considerably less classes (63.47%) compared to ECLIPSE RCP, while providing the same functionality for the analyzed variants. The main reason might be the fact that with ECLIPSE RCP additional code is needed especially for implementing interfaces, which are necessary for interaction between plug-ins. Particularly, an interface provides access to the implementation of a feature, which allows to extend this feature, but results in a certain amount of boilerplate code. In contrast, DELTAJ 1.5 directly extends existing classes due to the concept of delta module, which encompass everything related to such a module. There is no glue code needed to connect delta module and thus, boilerplate code is completely omitted. As a consequence, the DELTAJ 1.5 implementation *has a smaller code base* (83.3% in average), especially for variants that to not make use of all features. We argue that this also indicates a less complex design (no interface overhead, no boilerplate code) and thus, implementing variability with DELTAJ 1.5 is more intuitive. Nevertheless, common design principles are still needed and should be applied where necessary.

Threats to Validity: Although we conducted our evaluation with care, it exhibits some limitations. First, our case study consists of only one system that is small in size and has been implemented for this purpose only. Hence, our findings are not generalizable to other (large-scale) systems. Second, we compared DELTAJ 1.5 with one other approach for implementing variant-rich systems. Hence, we cannot generalize our findings (w.r.t. expressiveness and redundancy) to other variability mechanisms such as feature-oriented programming or preprocessor annotations. Third, we analyzed concrete variants rather than the complete code base. Especially for quality-related measures, such as complexity or cohesion, this may be limited. However, the main focus of this paper was rather the technical realization of DELTAJ 1.5, whereas the evaluation was complementary in order to demonstrate the applicability of DELTAJ 1.5 for implementing SPLs. We will address the aforementioned shortcomings in a comprehensive case study in future work.

6. Related Work

For implementing software product lines, there are three main approaches [31]. First, *annotative approaches*, such as conditional compilation, mark parts of the source code as belonging to a feature. These parts are only used in the final product when the respective feature is selected, all other parts are removed. This leads to large and complex programs if many features are considered. Second, *compositional approaches* encapsulate the artifacts of a feature into a distinct module which are composed to form the final product. Examples are feature-oriented (FOP) [5] and aspect-oriented (AOP) [21] programming. Third, *transformational approaches*, such as DOP [8, 28, 30] construct further program variants by modifying an existing program.

Delta modeling, the variability modeling technique underlying DOP, is not only used to express variability on the programming language level, but is also applied to express variability of modeling languages, such as software architectures [15] or Matlab/Simulink models [16]. Recently, the concept of delta modeling has been applied to express variability in the abstract behavioral modeling language ABS [9].

Comparing* DELTAJ 1.5 *with FOP and AOP. As the main contribution of this paper is the extension of DOP to JAVA 1.5, we compare other programming languages for modular programming paradigms with respect to their capabilities to access the object-oriented features of JAVA. Although FOP and AOP do not allow the removal of source code, they are the modular programming paradigms closest to DOP. Thus, we compare AHEAD and FeatureHouse (FOP) and AspectJ (AOP) with DELTAJ 1.5.

AHEAD. An early approach to implement feature-oriented SPLs is the AHEAD tool suite [5]. It facilitates the stepwise refinement of classes and supports *jampack* (i.e., the class modifications are merged into the final class) and *mixins* (i.e., refined classes become abstract with a generated name and are extended by the final class). AHEAD extends the JAVA syntax by the keywords `Super`, referencing a refined method (similar to the `original` call), and `refines`, used for refining interfaces and classes.

An implementation of AHEAD for JAVA is available[7]. Concerning JAVA's object-oriented features, it is possible to change the package of a class when it is first introduced, but not afterwards. Imports can be added to a class or interface, but an import can not be removed. The superclass may not be changed, but interfaces can be added to a class or interface. Interfaces can also be refined. Fields can be added. Methods can be added or overridden using the `Super` keyword to call the overridden method. Nested classes and enumerations are not supported.

FeatureHouse. FeatureHouse is a language independent-concept for composing different kinds of software artifacts (e.g., JAVA source code) using the concept of FOP [3]. The artifacts of a feature module are transformed into *feature structure trees* (FST) and subsequently superimposed with the corresponding FSTs of another feature module. The terminal nodes (in the case of JAVA, methods and fields) are overridden. The `original` keyword can be used to call the overridden method.

Concerning JAVA's object-oriented features, the package of a class in FeatureHouse cannot be changed after it is introduced. Imports can be added to a class, but they can not be removed. It is not possible to change the superclass, but additional interfaces can be added. The type of a field and the return type of a method can be changed by overriding a method, but the modifiers of fields and methods cannot be changed. A method can be overridden and the original method can be called from inside the overriding method using the `original` keyword. Besides the new keyword `original`, FeatureHouse uses the syntax of plain JAVA classes. Nested classes are supported and can be changed the same way a normal class can. Enumerations cannot be changed.

FeatureHouse does not have a technical documentation describing the exact behavior of the composition. We used FeatureIDE [8] to observe how refinements are handled.

AspectJ. AspectJ[9] is an extension to JAVA which allows enhancing functionality by weaving additional code in the form of aspects into a program. AspectJ is intended to implement crosscutting concerns, but not for realizing software product lines, although it is also used in this way [2]. AspectJ uses joint points to define where extensions, called advices, are executed. A point cut is set of join points and can be defined over method or constructor calls, exceptions or field accesses. Point cuts can be combined by operators (not, and, or) and restricted, e.g., by the class the control flow currently is in. Aspects can have their own variables and methods, but they are not added to a class. The syntax is similar to that of JAVA with the addition of point cuts and advices.

Concerning other object-oriented features of JAVA, a class itself cannot be changed by an aspect. Therefore the package, superclass and imports cannot be changed either. It is not possible to add or modify fields of a class, but an aspect can introduce fields and methods and associate them with a class to be used inside advices. An advice can be executed before, after or instead of a method which works the same as using the `original` keyword in DELTAJ 1.5. It is not possible to add nested classes.

Discussion. AHEAD and FeatureHouse try to minimize the changes to the JAVA syntax to make it easy for the developer to implement feature-oriented SPLs. This leaves some of JAVA's object-oriented features uncovered. Implementing SPLs is not the goal of AspectJ. This makes it necessary to use a more complex syntax for expressing point cuts and advices, but by construction, this does not cover JAVA's object-oriented features completely. In DELTAJ 1.5, in addition to adding and modifying elements, elements like classes, methods or fields can also be removed. This makes it necessary to provide additional delta operations. The explicit removal operation leads to a much more expressive language for software reuse compared to FeatureHouse or AHEAD. Furthermore, compared to the above languages, DELTAJ 1.5 is the first language for modular implementation of software product lines that allows integrated access to the object-oriented features of JAVA 1.5.

7. Conclusion and Future Work

DOP [8, 28, 30] is a modular programming paradigm for implementing software product lines. In this paper, we have presented DELTAJ 1.5 extending previous proof-of-concept realizations of DOP for java-like core languages. DELTAJ 1.5 allows integrated access to the object-oriented features of JAVA 1.5 as one of the first programming languages for a modular programming paradigm. Although, DELTAJ 1.5 integrates most of JAVA 1.5's object-oriented features, there is still room for improvements. We are currently working on language support for delta operations on JAVA annotations and an extension of the modification operations for JAVA generics and for method signatures, including alterations of access modifiers. Our final goal is to support all JAVA language features up to JAVA 8 in DOP, such as lambda expressions, anonymous methods, default interface methods, switch-case-statements with strings and multi-catch-blocks. The prototype for DELTAJ 1.5[10] implements an editor with syntax highlighting, a customized outline view

[7] `http://www.cs.utexas.edu/users/schwartz/ATS/fopdocs/`

[8] `http://fosd.de/fide`

[9] `http://eclipse.org/aspectj/`

[10] `https://www.tu-braunschweig.de/isf/research/deltas`

and product generation. In order to avoid the generation of products which are not well-typed, we have developed a light-weight type checking approach [22] which adapts the concepts of [35] to DOP. The main idea of this type checking approach is to collect the dependencies between language elements defined in delta modules and elements accessed in delta modules. These dependencies are then checked against the product line declaration in order to ensure that all possible products are well-typed. This approach is currently implemented within the prototype of DELTAJ 1.5.

References

[1] Xtext 2.4 Documentation. PDF document from the frameworks website, 16. April 2013. Online at `www.eclipse.org/Xtext/documentation/2.4.0/Documentation.pdf`; last visited on 2013/05/13.

[2] Vander Alves, Pedro Matos, Leonardo Cole, Alexandre Vasconcelos, Paulo Borba, and Geber Ramalho. Extracting and evolving code in product lines with aspect-oriented programming. *T. Aspect-Oriented Software Development*, 4:117–142, 2007.

[3] Sven Apel, Christian Kastner, and Christian Lengauer. Language-independent and automated software composition: The featurehouse experience. *Software Engineering, IEEE Transactions on*, 39(1):63–79, 2013.

[4] Don Batory. Feature models, grammars, and propositional formulas. In *Proceedings of the 9th international conference on Software Product Lines*, SPLC'05, Berlin, Heidelberg, 2005. Springer-Verlag.

[5] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling stepwise refinement. *Software Engineering, IEEE Transactions on*, 30(6): 355–371, 2004.

[6] Daniel Le Berre and Anne Parrain. The Sat4j library, release 2.2. *JSAT*, 2010.

[7] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing, 2013. ISBN 9781782160304.

[8] Lorenzo Bettini, Ferruccio Damiani, and Ina Schaefer. Compositional type checking of delta-oriented software product lines. *Acta Informatica*, 50(2):77–122, 2013.

[9] Dave Clarke, Nikolay Diakov, Reiner Hähnle, Einar Broch Johnsen, Ina Schaefer, Jan Schäfer, Rudolf Schlatte, and Peter Y. H. Wong. Modeling spatial and temporal variability with the hats abstract behavioral modeling language. In *SFM*, pages 417–457, 2011.

[10] Paul Clements and Linda Northrop. *Software product lines: practices and patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[11] Ferruccio Damiani and Ina Schaefer. Family-based analysis of type safety for delta-oriented software product lines. In *Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change*, volume 7609 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 2012.

[12] Sven Efftinge, Moritz Eysholdt, Jan Köhnlein, Sebastian Zarnekow, Robert von Massow, Wilhelm Hasselbring, and Michael Hanus. Xbase: Implementing Domain-Specific Languages for Java. In *GPCE*, pages 112–121. ACM, 2012.

[13] Martin Fowler. Inversion of Control Containers and the Dependency Injection pattern. `http://www.martinfowler.com/articles/injection.html`, January 2004.

[14] Konstantin Friesen. Entwicklung einer Werkzeugunterstützung für DeltaJava und Evaluierung der Sprache anhand einer Fallstudie. Master thesis, Technische Universität Braunschweig, Institut für Softwaretechnik und Fahrzeuginformatik, September 2012.

[15] Arne Haber, Thomas Kutz, Holger Rendel, Bernhard Rumpe, and Ina Schaefer. Delta-oriented architectural variability using monticore. In *ECSA Companion Volume*, 2011.

[16] Arne Haber, Carsten Kolassa, Peter Manhart, Pedram Mir Seyed Nazari, Bernhard Rumpe, and Ina Schaefer. First-class variability modeling in Matlab/Simulink. In *VaMoS*, 2013.

[17] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, May 2001.

[18] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical report, DTIC Document, 1990.

[19] Christian Kästner, Sven Apel, Syed Saif ur Rahman, Marko Rosenmüller, Don Batory, and Gunter Saake. On the impact of the optional feature problem: analysis and case studies. In *Proceedings of the 13th International Software Product Line Conference*, SPLC '09, pages 181–190, 2009.

[20] Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench. Rules for declarative specification of languages and IDEs. In *OOPSLA*, pages 444–463. ACM, 2010.

[21] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997.

[22] Jonathan Koscielny. Typsicherheit in delta-orientierten Softwareproduktlinien. Bachelor thesis, Technische Universität Braunschweig, Institut für Softwaretechnik und Fahrzeuginformatik, May 2013.

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

[24] John Levine. *flex & bison*. O'Reilly Media, 2009.

[25] Roberto E Lopez-Herrejon, Don Batory, and William Cook. *Evaluating support for features in advanced modularization technologies*. Springer, 2005.

[26] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Programmers, May 2007. ISBN 978-0-9787392-5-6.

[27] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., 2005.

[28] Ina Schaefer and Ferruccio Damiani. Pure delta-oriented programming. In *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development*, FOSD '10, New York, NY, USA, 2010. ACM.

[29] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented Programming of Software Product Lines. In *Proc. of 15th Software Product Line Conference*, SPLC 2010, Korea, September 2010.

[30] Ina Schaefer, Lorenzo Bettini, and Ferruccio Damiani. Compositional type-checking of delta-oriented programming. In *Proceedings of the Tenth International Conference on Aspect-oriented Software Development*, AOSD '11, pages 43–56, 2011.

[31] Ina Schaefer, Rick Rabiser, Dave Clarke, Lorenzo Bettini, David Benavides, Goetz Botterweck, Animesh Pathak, Salvador Trujillo, and Karina Villela. Software diversity: state of the art and perspectives. *International Journal on Software Tools for Technology Transfer*, 14 (5):477–495, 2012. ISSN 1433-2779.

[32] Sandro Schulze, Oliver Richers, and Ina Schaefer. Refactoring delta-oriented software product lines. In *Proceedings of the 12th Annual International Conference on Aspect-oriented Software Development*, AOSD '13, pages 73–84, 2013.

[33] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison Wesley Professional, 2nd edition, 2008.

[34] Rok Strniša, Peter Sewell, and Matthew Parkinson. The Java module system: core design and semantic definition. In *Proc. of OOPSLA 2007*, pages 499–514. ACM, 2007.

[35] Sahil Thaker, Don Batory, David Kitchin, and William Cook. Safe composition of product lines. In *Proceedings of the 6th international conference on Generative programming and component engineering*, GPCE '07, New York, NY, USA, 2007. ACM.

[36] Thomas Thüm and Christian Kästner and Fabian Benduhn and Jens Meinicke and Gunter Saake and Thomas Leich. FeatureIDE: An Extensible Framework for Feature-Oriented Software Development. *Science of Computer Programming*, 2014.

[37] Mads Torgersen. The Expression Problem Revisited. In *ECOOP 2004*, pages 123–146. Springer, 2004. .

[38] Markus Voelter. Language and IDE Modularization and Composition with MPS. In *GTTSE*, volume 7680 of *Lecture Notes in Computer Science*, pages 383–430. Springer, 2011.