Parametric DeltaJ 1.5: Propagating Feature Attributes into Implementation Artifacts

Tim Winkelmann[1], Jonathan Koscielny[1], Christoph Seidl[1],
Sven Schuster[1], Ferruccio Damiani[2], Ina Schaefer[1]
{t.winkelmann, j.koscielny, c.seidl, s.schuster, i.schaefer}@tu-bs.de, ferruccio.damiani@unito.it

[1]Technische Universität Braunschweig, Germany, [2]Università di Torino, Italy

Abstract

Systematic reuse of software artifacts can be achieved with software product lines, which represent a family of similar software systems. A variability model, e.g., feature model, describes their commonalities and variabilities and serves as a basis for a product configuration, i.e., the selection of features according to constraints defined in the model. These variability models can contain additional information, such as attributes, which enrich features with typed values for various purposes (e.g., optimization, simplified readability). Typically, these attributes are not directly reusable in code artifacts as the variability model is only used to assemble or change code artifacts according to a product configuration. Furthermore, there are many languages for implementing software product lines such as DeltaJ which do not support the direct propagation of feature attributes to the associated code artifacts. In this paper, we present parametric DeltaJ, an adaptation of the programming language DeltaJ for delta-oriented software product lines in Java. parametric DeltaJ allows the propagation of typed attributes from an attributed feature model to Java code artifacts. We perform a case study to show that introducing parameters reduces the number of variables, delta modules and lines of code for delta-oriented software product lines.

1 Introduction

The concept of (hardware) product lines from industries such as automotive, avionics, or mobile phones were adapted to software product line engineering (SPLE) [24, 9, 1] in order to reduce the drawbacks of maintaining and developing multiple software systems for one domain for different customers. A typical way to describe the commonalities and variabilities in software product lines (SPLs) are feature models (FMs) [15]. In particular, FMs describe configurable elements (e.g., mandatory, optional or mutually exclusive) of variable software systems in terms of features using a hierarchical tree-structure. A feature is usually defined as an increment in functionality [3]. Kang et al. [15] proposed additional information to be assigned to features. This information is typically captured in the form of *feature attributes*, which extend FMs to attributed feature models (AFMs) [19]. Most implementation languages for SPLs use these attributes for configuration purposes only. However, feature attributes are not propagated into the solution space to configure realization artifacts of concrete variants. In

Submission to: 8. Arbeitstagung Programmiersprachen, Dresden, Germany, 18-Mar-2015, to appear at http://ceur-ws.org

this paper, we propose and present an approach to propagate feature attributes of AFMs into the solution space of an SPL based on Delta-Oriented Programming (DOP).

DOP is a flexible programming paradigm to implement SPLs [7] by source code transformation. DOP can also be used directly to support evolution of SPLs, which is in contrast to other approaches, which allow either flexible, fine-grained code manipulation or modularization of code and are not directly capable of handling SPL evolution [27]. The language DeltaJ 1.5 [17] is designed to combine the object-oriented programming language Java with the DOP programming paradigm. DeltaJ 1.5 supports full Java 1.5.[1] Delta modules in DeltaJ modularize a Java code base according to the feature selection of a concrete product configuration. A delta module in DeltaJ is included for product derivation depending on the value of a Boolean application condition determining whether a combination of features is present in the configuration.

If the SPL uses an AFM to describe possible products, attributes may have other types than Boolean. Hence, attribute values cannot be used directly for configuration and can also not be propagated to the implementation artifacts which are configured. As a workaround, the AFM could be transformed into an FM. However, this leads to an explosion in the number of features and possible delta modules, i.e., possibly one for each attribute value. For instance, transforming a single integer attribute with a range of 1000 possible domain values leads to 1000 features and according number of delta modules.

To remedy this problem, in this paper, we introduce an extension of the language DeltaJ called parametric DeltaJ, to integrate attributes of AFMs such that attribute values can be used for configuration as well as propagated as parameters to implementation artifacts. We enhance delta modules with parameters, similar to methods in Java. These parameters are assigned values of feature attributes to manipulate the source code. This increases the capabilities of delta modules regarding reuse for AFM. With this approach, we demonstrate a solution for systematic reuse of feature attributes in the software artifacts of an SPL.

Section 2 shows the details of AFM based SPLs and gives a recap of DeltaJ 1.5. Section 3 presents our extension of DeltaJ and introduces necessary extensions to the product line declaration and delta modules. Section 4 presents the challenges of implementation the extension of DeltaJ 1.5. Section 5 contains our case study. Section 6 revisits other approaches for implementing SPLs as related work. Section 7 concludes the paper and presents further ideas to extend DeltaJ.

2 Foundations

A software product line (SPL) can be divided into three parts, the *problem space*, the *configuration space* and the *solution space* [11, 15]. The problem space contains an abstract description of the products that can be derived from the SPL. A typical model for such a description are feature models (FMs), which describe features that all products have in common and the variability between the products. The solution space provides the language-dependent code artifacts for the SPLs. The configuration space maps features to solution space artifacts such that *products* can be derived by selecting features according to the constraints of the FM. In FMs and AFMs, each feature may only be selected once.

2.1 Feature Models and Attributed Feature Models

Features describe configurable elements in an SPL. Dependencies between features are expressed in a FM, where features are structured in form of a tree. The tree structure consist of optional and mandatory features as well as of feature groups like alternative and or-groups. Additional constraints can be expressed in from of propositional formulas, so called cross-tree constraints (CTCs). Usually, CTCs are *require* and *exclude* constraints between two features, but also more complex CTCs are possible up to general Boolean formulas over features. In AFMs [10], feature attributes are added to features. An attribute is a pair of the attribute's name and its type and is coupled to a feature [10]. The type of the attribute defines the possible domain values that can be selected for the attribute during product configuration. Attributes in AFMs are used to further describe the properties of a feature. They can be used, e.g., to capture product optimization (such as lowest power consumption) and improve the readability of the constraints in the FM. For decidability of the CTCs, the domains of all involved feature attributes must be finite [16]. Feature attributes so far have only impact on the problem space, i.e., the feature model representation, and the configuration space, but their impact on the solution space (containing the code artifacts) has to be specified with an appropriate variability realization mechanism.

[1] We use DeltaJ as a synonym for DeltaJ 1.5 in the following.

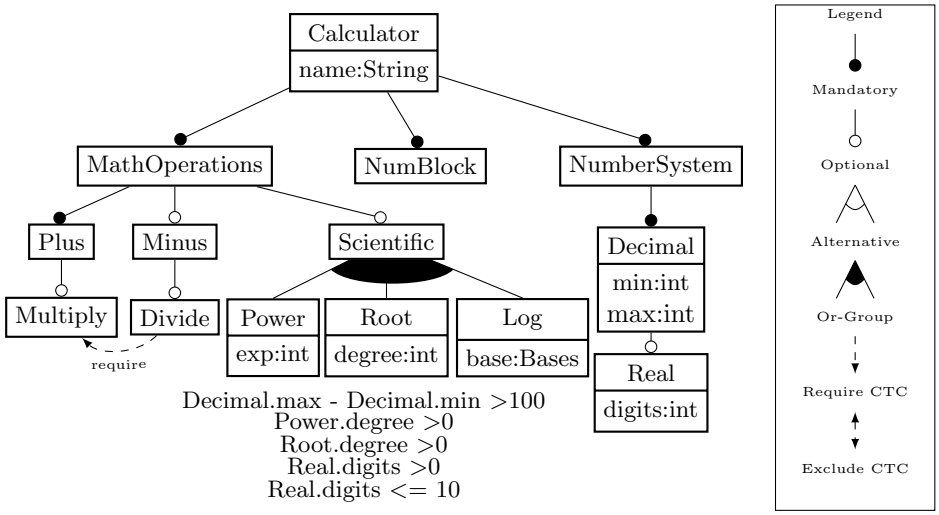

Figure 1: Example attributed feature model of Calculator SPL

Figure 1 shows the feature model of an example calculator SPL with feature attributes, that have an impact on implementation artifacts. The example contains the most basic elements of an FM, such as mandatory/optional features, a feature group, CTCs and attributes of different types (e.g., the enum type `Bases` of feature `Log`). Feature `Calculator` is the root feature of the SPL and must be present in all derived products. Also present in all derivable products are the core features, which are also known as full mandatory features.

- `MathOperations` to structure the mathematical operations of the calculator SPL
- `Plus` as the simplest mathematical operation that all our derivable calculators support
- `NumBlock` the number block as input for numbers
- `NumberSystem` with its child feature `Decimal` to define the number system, which we are using for all calculators

Mandatory features are indicated by a filled circle at the top of the feature. Optional features are indicated by an empty circle, for example, the other mathematical operations `Minus`, `Multiply` and `Divide`. We further restrict the feature `Divide` by a CTC to include the feature `Multiply` if `Divide` is selected. The additional mathematical operations are combined below the feature `Scientific` in an or-group, where at least one of the mathematical operations `Power`, `Root`, `Log` must be selected if the parent feature `Scientific` is selected. The features `Power` and `Root` each have one attribute of the type *int*. For the feature `Power`, the attribute `exp` is used to configure the exponent of the operation. The same for the attribute `degree` of feature `Root`. For the feature `Log`, we use an attribute with the *enum* type Bases to distinguish the base for the operation. The enum contains the values {`Ten`, `Two`, `E`} as common bases. Additional CTCs in a textual representation define the range of attributes and other constraints, such as that the difference between the minimal decimal number and the maximal decimal number is greater than 100.

2.2 DeltaJ

DOP is a transformational approach for implementing software product lines [26] where a core variant is successively transformed into other variants by applying so called *delta modules*. Delta modules contain addition, modification and removal operations for existing artifacts [26]. Delta modules are selected based on a product configuration containing a selection of features. The connection of delta models to features is expressed by application conditions ranging over feature combinations in a *product line declaration*, which additionally specifies a partial order for the delta modules to later determine their sequence of application (called *application order* [27]). The partial order is necessary to capture dependencies between the delta modules. The first delta module in the product generation process contains only additions to create a first code base to operate on [27].

DOP for JAVA is realized in the programming language DELTAJ. An SPL in DELTAJ consists of two parts, first, a product line declaration and, second, set of delta modules. The DELTAJ product line declaration file specifies the used delta modules, the partial order of the delta modules and the mapping of features to delta modules. It further describes which products of the SPL are to be generated. For this purpose, it needs the FM as a basis to ensure that no constraint of the FM is violated for product configuration. The structure of a product line declaration file is displayed in Listing 1. It contains clauses for features, delta modules, constraints, partitions and products. The hierarchy and CTCs of the FM are described as a propositional formula in the Constraints clause. The Partitions clause introduces a partial order of delta modules for the product generation process where the `when` clause maps delta modules to features by specifying the application condition. The `generate` keyword of the Products clause marks the concrete product definitions for the generation process.

A delta module declaration starts with the keyword `delta` followed by the name of the delta module. It contains operations to add, modify or remove classes. Modifying a class can contain further operations to add, modify or remove fields and methods of the class. In case of modifying a method, the keyword `original` is introduced to reuse earlier versions of the method.

```
SPL Calculator {
Features { //Set of features
 Calculator, MathOperations, NumBlock, ...
}
Deltas { //Set of delta modules
 dCalculator, dNumberSystem, dMinus, dMultiply, dDivide,...
}
Constraints { //Set of constraints over features
 MathOperations & ... & Log => (LogBaseTen | LogBaseTwo | LogBaseE);
}
Partitions { //List of partitions
 {dCalculator} when (Calculator);
 ...
 {dLogBaseE} when (LogBaseE);
}
Products { //Set of product configurations
 Minimal = {Calculator, MathOperations, NumBlock, NumberSystem, ...};
 generate  Basic = {Calculator, ..., Decimal, Real};
}
}
```

Listing 1: Structure of a product line declaration file

```
delta dNumberSystem{
 adds {
  package calc;
  public class Number {
   private int number;
   public void set(int i) {
    number = i;
   }
   public void plus(int i){
    number += i;
   }
   public void print(){
    System.out.print(number);
   }
  }
 }
}
```

Listing 2: Delta Module with adds operation

```
delta dDecimalMin0{
 modifies calc.Number {
  modifies set(int i) {
   if(i >= 0)) {
    original(i);
   }
  }
  modifies plus(int i) {
   if((number + i) >= 0) {
    original(i);
   }
  }
 }
}
```

Listing 3: Delta Module with modifies operation

The delta module `dNumberSystem` in Listing 2 adds the class `Number`. The class contains a setter method to store the value in the object and the `plus` method for the mandatory arithmetic operation. Assume that the Calculator SPL contained a feature `DecimalMin0` that requires that no numbers lower than 0 can be used. The according delta module `dDecimalMin0` to implement this requirement is shown in Listing 3 and applied if the feature `DecimalMin0` is selected. The delta module modifies each input method, verifies the input and

calls the original version of the method with the `original` keyword. Operations to remove classes or methods use the keyword `removes` and work similar to the `modifies` operation.

2.3 Encoding Attributes in DeltaJ

In the existing version of DeltaJ, feature attributes can only handled by encoding all attribute values as features. An example of such a Features clause with encoded attribute values is shown in Listing 4, which shows the set of features for the calculator SPL. Features displayed here are simple literals. As DeltaJ does not support attributes in features, we transformed every domain value of every attribute to a literal which demonstrates the shortcomings of DeltaJ regarding attributes. One of the first things to be noticed is that the String attribute of the root feature `calculator` can not be transformed into a set of features. Only finite domains can be transformed into a feature by this encoding for every domain value. This transformation results in many features that are otherwise easily expressed in the AFM. This can be seen in Listing 4 in line 5, where we express every integer value as a feature. To reduce the number of features we decreased the range of the variables `DecimalMax` and `DecimalMin`.

```
1 SPL Calculator {
2 Features {
3  Calculator, MathOperations, NumBlock, NumberSystem, Plus, Minus, Scientific,
4  Divide, Decimal, Multiply, Power, Root, Log, Real,
5  DecimalMin-100, ... , DecimalMin100, DecimalMax-100, ... , DecimalMax100,
6  PowerExp1, ... , PowerExpN, RootDegree1, ... , RootDegreeN,
7  RealDigits1, ... , RealDigitsN, ...
8 }
9 ...
```

Listing 4: Product line declaration file features

The Deltas clause for the calculator SPL is shown in Listing 5. Here, delta modules are represented by their names. In this case, we need as many delta modules as features to capture the impact of the feature on the implementation artifacts, which also leads to an explosion of the number delta modules most of which are very similar. We also have an explosion of constraints when we transform the CTCs for the attributes in order to require that only one value for an attribute can be selected. This phenomenon can be observed in Listing 6. A constraint is represented by a Boolean formula over features and ends by a semicolon. All constraints must be satisfied in a product configuration, otherwise no product can be generated. We put the single domain values for every attribute into an XOR relation of the feature comprising the attribute is selected (if not full mandatory). Transforming the constraint $max - min < 100$ in Boolean logic is possible but a lot of constraints are necessary to express it.

```
1 ...
2 Deltas {
3  dCalculator, dNumberSystem, dMinus, dMultiply, dDivide, dPower, dRoot, dLog,
4  dReal, dDecimalMin-N, ... dDecimalMinN, dDecimalMax-N, ... dDecimalMaxN,
5  dPowerExp1, ... , dPowerExpN, dRootDegree1, ... , dRootDegreeN,
6  dRealDigits1, ... , dRealDigits10, dLogBaseTen, dLogBaseTwo, dLogBaseE
7 }
8 ...
```

Listing 5: Product line declaration file deltas

```
...
 Constraints {
  MathOperations & ...;
  DecimalMin-N ^ ... ^ DecimalMinN; DecimalMax-N ^ ... ^ DecimalMaxN;
  DecimalMin-N => !(DecimalMax-N | ... | DecimalMax100); ...
  DecimalMin1 => !(DecimalMax-N | ... | DecimalMax101); ...
  DecimalMinN => !(DecimalMax-N | ... | DecimalMax2_147_483_547);
  //Decimal.max - Decimal.min > 100;
  Power => (PowerExp1 ^ ... ^ PowerExpN); //Power.exp;
  Root => (RootDegree1 ^ ... ^ RootDegreeN); //Root.degree;
  Real => (RealDigits1 ^ ... ^ RealDigits10);//Real.digits;
  Log => (LogBaseTen | LogBaseTwo | LogBaseE);
 }
...
```

Listing 6: Product line declaration file constraints

3 Parametric DeltaJ

In this section, we present an extension to DeltaJ to incorporate feature attribute as parameters. The following steps have to be performed towards the extension of DeltaJ which are either addressed in the product line declaration or in the delta modules.

1. Introduce features attributes to the product line declaration. This includes access in constraints, partitions, delta modules and product definition.
2. Introduce parameters to delta modules, so that they can be used in classes and methods.
3. Update the validation of the attributed feature model and the definied products.
4. Include feature attributes and parameters of delta modules in the product generation process.

3.1 DeltaJ Product Line Declaration with attributed feature models

For the product line declaration, we first extend the syntax of DeltaJ for feature attributes. For this purpose, our feature declaration now contains an optional *Formal Parameter Declaration (FPD)* after the feature name to specify feature attributes. Syntactically, this declaration is similar to the FPD of a regular Java method parameter declaration.

```
SPL Calculator {
 enum Bases {Ten, Two, E};
 Features {
  Calculator(String name), MathOperations, NumBlock, NumberSystem,
  Plus, Minus, Scientific,  Divide, Decimal(int min, int max), Multiply,
  Power(int exp), Root(int degree), Log(Bases base), Real(int digits)
 }
 ...
}
```

Listing 7: Features with attributes in the product line declaration file of the Calculator SPL

The adapted example from Listing 4 is shown in Listing 7. The number of defined literals for features in parametric DeltaJ is significantly smaller as no attributes have to be transformed into simple features anymore. As parameter types, we provide Java `primitive` types, the `java.lang.String` type and Java-like `Enumeration` types, which can be defined in the product line declaration. With the type `String` we also support infinite domains. Listing 7 also shows the definition of the enum type `Bases` for the feature `Log`. The enums are defined before the `Features` clause. In the product generation process, each defined `enum` is generated as a Java enumeration type, which is used in the implementation if needed.

Parameters of delta module declarations use the same syntax as attributes declared for features. With these parameters, we reduce the number of necessary delta modules significantly as it is no longer needed to write one delta module for every feature attribute domain value. Listing 8 shows the parametric DeltaJ variant of Listing 5 with delta module parameters.

```
...
 Deltas {
  dCalculator(String name),
  dNumberSystem(int min, int max),
  dMinus, dMultiply, dDivide,
  dPower(int exp), dRoot(int degree),
  dLog(Bases base), dReal(int digits)
 }
...
```

Listing 8: Delta module definition with parameters in the product line declaration file for the Calculator SPL

For the constraints, we extended the possible operations by basic equality (=), basic inequality (>,<,≥,≤,≠) and arithmetic (+,−,∗,/) operations. The syntax extension allows expressing constraints such as "any min and max value is positive and its range is at least 100", but doesn't allow expressing constraints such as "a string must not contain any spaces and has to start with an upper case character". This is because constraints are validated using a CSP Solver, but we do not validate them by executing JAVA code contained in expressions. A more detailed description is presented in Section 4. The updated example of Listing 6 is shown in Listing 9 where we use the basic inequality and arithmetic operations to set the minimal range for our decimal numbers and constrain the range of our scientific operations.

```
...
 Constraints {
  MathOperations & ... ;
  Divide => Minus & Multiply;
  Power | Root | Log => Scientific;
  Scientific => Power | Root | Log;
  Decimal.max - Decimal.min > 100;
  ...
 }
...
```

Listing 9: Constraints in the product line declaration file for the Calculator SPL

We extend the partition declaration to allow delta module selection to also consider attributes in the `when` clause. This is because a delta module can occur multiple times in any partition, but it can only be applied once for a product, which is ensured by the `when` clause. As the `when` clause defines the connection of features and delta modules, we extend the delta module to be able to receive a delta parameter list. An argument value for a parameter can either be a type-compatible parameter of a corresponding feature, a type-compatible literal (constant value) or an expression over those values. Listing 10 shows the partitions for the Calculator SPL.

```
...
 Partitions {
  {dNumberSystem (Decimal.min, Decimal.max)},
  {dCalculator(Calculator.name)};
  ...
  {dPower(Power.exp)} when (Power);
  {dRoot(Root.degree)} when (Root);
  {dLog(Log.base)} when (Log);
  {dReal(Real.digits)} when (Real);
 }
...
```

Listing 10: Partitions of delta modules in the product line declaration file

```
...
 Products {
  ...
  Scientific = {Calculator("Scientific Calculator"), MathOperations, NumBlock,
   NumberSystem, Plus, Minus, Multiply, Divide, Real(9), Scientific,
   Decimal(-100_000_000, 1_000_000_000), Power(2), Root(2), Log(Two)};
 }
}
```

Listing 11: Product definition in the product line declaration file

Products can be defined in the products block. Within the feature selection of a product, concrete values are defined for a feature attributes. In Listing 11, we define a set of products in the Calculator SPL.

3.2 DeltaJ Parameterized Delta Modules

In this section, we present the changes to the syntax for defining delta modules. As in the product line declaration file, we extended the signature of delta modules with parameters. The concrete value of each parameter is propagated from the configuration to each delta module by the product line declaration (see Listing 10). In Listing 12, we show how parameters are used in delta modules. The listing displays the code for the delta module `dNumberSystem` with its two parameters `min` and `max` as range for the decimal numbers used by our calculator. To reference a delta parameter, the name of the parameter is used. As JAVA allows name overloading, a parameter can be shadowed by either a class field or a local variable. To be able to qualify a parameter, we introduce a new keyword `dm` (for "delta module"), which can be used to build a parameters qualified name. Listing 12 demonstrates how to reference parameters. In line 7, the delta parameter `min` is referenced by its simple name and parameter `max` is referenced with its qualified name in lines 13 and 16.

```
1  delta dNumberSystem(int min, int max) {
2   adds {
3    class Decimal{
4     private int number;
5     ...
6     public void setNumber(int n) {
7      if(n < min) {
8       this.number = n;
9      }
10    }
11    ...
12    public int plus(Decimal addend) {
13     if(addend.getNumber() + number < dm.max) {
14      number += addend.getNumber();
15     } else {
16      number = dm.max;
17     }
18    }
19   }
20  }
21  ...
22 }
```

Listing 12: Delta module `dNumberSystem`

3.3 Checking Configuration Validity

To make sure that the defined products in the product clause only contain valid configurations of the FM, in DELTAJ, this problem was modeled internally as *satisfiability problem*. The *satisfiability problem* is checked by a SAT solver, that validates, that no constraint of the feature model is violated by the feature configuration of the product.

With introducing feature attributes, the complexity of validating the selected feature and feature attribute configuration for product generation against the AFM is increased. Because a valid parameter type in PARAMETRIC DELTAJ is not necessarily Boolean, the problem evolves from SAT to a general constraint satisfaction problem (CSP) problem [16]. Since a CSP solver can only handle finite domains, we prohibit the use of `String` attributes in CTCs. The CSP solver replaces the satisfiability (SAT) solver in the validation of the selected product configuration against the feature. A transformation of attributed feature models to CSP problems can be found in [19]. In principle, it is checked that the selection of features and feature attribute values for a particular product does not violate the feature model constraints.

3.4 Propagating Parameter Values into the Generated Product

For a particular product, the defined concrete values for the selected feature attributes must be propagated into the product's generated source code. To this end, multiple possibilities exist:

1. Place the attribute value as a (local) constant directly into the particular locations, where the attribute is required, e.g., into a method body.
2. Encapsulate all attribute values that are relevant for a particular class as constant fields within that class.
3. Encapsulate *all* existing attribute values as constants within *one* additionally generated class or interface.

We decided for the third strategy of using one additional interface for *all* existing attribute values as it has two major advantages above the other strategies: prohibiting code scattering and avoiding name clashes. As a delta module can introduce or modify multiple classes and methods, both the first strategy (i.e., a local creation

of constants) as well as the second strategy (i.e., creating constant fields in each relevant class) could lead to the same constant being created multiple times across the product. This would prohibit the possibility of tracing the attribute values within the generated source code. Moreover, with both strategies, name clashes with existing variables and constants could occur. The third strategy (i.e., encapsulating *all* attribute values within *one* interface) avoids these problems. Each attribute value is created once as a static constant within this unified interface, which allows tracing the feature attributes within the generated source code. Moreover, name clashes are prohibited because a static reference to this interface is necessary to reference the attribute value.

4 Implementation

In this section, we describe relevant implementation decisions of PARAMETRIC DELTAJ as an extension to our previous work on DELTAJ 1.5 [17]. We provide an Eclipse plug-in[2] for DELTAJ that is implemented using the Xtext framework[3]. To incorporate JAVA types, we employed XBASE [13], an expression language to automatically integrate rich expressions into a language and to integrate a custom language seamlessly with JAVA types.

4.1 Syntax Extensions

In the following, we provide implementation details on the necessary syntactic extensions to DELTAJ to incorporate feature attributes in the product line declaration as well as in the delta modules. As DELTAJ is developed using Xtext, there are grammar files defining the syntax of both the product line declaration as well as the delta modules.

4.1.1 Product Line Declaration Extensions

In the product line declaration, multiple extensions to incorporate feature attributes and propagating these into particular products are required:

- Features declarations and delta module declarations need to allow defining attributes as parameters (see Listing 7 and Listing 8).
- Constrains need to be extended by enabling expressions (e.g., equals, greater than) on feature attributes rather than only supporting propositional logic (see Listing 9).
- The `when` clauses in the partitions require a mapping of feature attributes to corresponding delta module parameters (see Listing 10).
- For product definitions, concrete values need to be provided as arguments of the selected features (see Listing 11).
- Enumeration declarations need to be integrated into the product line declaration to allow for using custom enumeration types as parameters (see Listing 7).

In the following, we briefly explain the steps taken to integrate the described extensions into the product line declaration, categorized by the clause in the product line declaration, which is extended.

Features & Deltas. To integrate parameters into feature declarations and delta module declarations, the grammar of the product line declaration is extended by adding a parameter rule, whose syntax is similar to JAVA method parameters. The grammar is illustrated in 2. The rules for feature declaration and delta module declaration both use an optional rule to define attributes or parameters.

Constraints. The constraints in DELTAJ are defined by propositional formulas over feature names. To incorporate constraints for feature attributes, complex expressions over booleans, integers and strings are required within the constraints clause (see Listing 6).

Partitions. The delta application constraints, which now also define the mapping of feature attributes to delta module parameters, are extended. By propagating feature parameter names to delta modules as an argument (see Listing 10) within the partitions clauses, a mapping is established. Such arguments are defined as a non-empty list of expressions that is surrounded by parentheses. Using a list of expressions allows combining several feature arguments to a single argument of a corresponding delta module (e.g., `dDelta(Feature.a + Feature.b)` for integer attributes a and b).

[2] `http://tu-braunschweig.de/isf/research/deltas`
[3] `http://eclipse.org/Xtext`

FC	::=	**Features** { $\overline{\text{FD}}$ }	*features clause*
FD	::=	F | F ($\overline{\text{PD}}$)	*feature declaration*
DC	::=	**Deltas** { $\overline{\text{DD}}$ }	*deltas clause*
DD	::=	D | D ($\overline{\text{PD}}$)	*delta module declaration*
PD	::=	PT P | $\bar{\text{E}}$ P	*parameter declaration*
PT	::=	**int** | **boolean** | ...	*primitive type*
ED	::=	**enum** E { $\bar{\text{L}}$ }	*enum declaration*

Figure 2: Syntax of feature and delta module declarations with F ∈ feature names, D ∈ delta module names, P ∈ parameter names, L ∈ enum literals

Products. The product definitions need to allow giving concrete values for feature attributes. To this end, we extend the grammar for product definitions by allowing to specify concrete values for feature attributes as arguments (see Listing 11). Arguments are defined as a non-empty list of expressions that is surrounded by parentheses.

Enumerations. To integrate enumeration declarations (see Listing 7) into the product line declaration, we extend the grammar by adding an enumeration declaration which is syntactically similar to JAVA enum type declarations.

4.1.2 Delta Module Extensions

To incorporate delta parameters within the realization artifacts, i.e., delta modules, the syntax for delta modules is extended in the following ways.

Parameters. To integrate delta parameters into DELTAJ (see Listing 12), we extend the Delta declaration with an optional parameter declaration similar to the one in the product line declaration.

Parameter references. To reference a delta parameter within a JAVA statement (see Listing 12), we extend the JAVA syntax by an additional keyword `dm`. Similar to the `this`, it defines the scope of the following identifier reference to only reference the delta parameters.

4.2 Semantic Extensions

In the following, we provide implementation details on the necessary semantic extensions to DELTAJ to incorporate feature attributes in the configuration validation as well as in the product generation.

4.2.1 Product Validation and Delta Module Selection

The CSP solver we selected for validating product selections [18], only allows the variable types `IntVar`, `FloatVar` and `BooleanVar`. To be able to use all JAVA primitive types and user-defined enumeration types, we map them to the closest CSP type and encode the values.

Most JAVA primitive types can be modeled with the integer domain as they utilize a smaller domain. However, `long` and `double` cannot be modeled without losing information. To handle this shortcoming, a warning informs the user that these values are treated as `int` and `float` values, respectively. Expressions are modeled using the constraint API of the CSP solver. To check whether a product is valid, the chosen feature selection is captured by allocating the associated CSP variables with `true`, the feature attribute allocation is represented by encoding their value and allocating the associated CSP variables. Iff the CSP Solver is able to find an allocation of all variables, the product is valid.

4.2.2 Generating a Product's Source Code

As we add FPD to delta modules, all statements referencing a delta parameter have to be modified in the generated JAVA source code. To provide the delta parameter instantiation in the product's source code, we first generate an additional JAVA `interface` containing a constant for each delta parameter, with the name `DELTA_PARAM`. An example is shown in Listing 13. Afterwards, each reference to a delta parameter is replaced by a reference to the corresponding constant.

```
package spl.product;
public interface Parameters {
 public static final String
  CALCULATOR_NAME = "Adder";
 public static final int DECIMAL_MIN = 0;
 public static final int
  DECIMAL_MAX = 1000000;
}
```

Listing 13: Generated parmeter interface of product Adder (cf. Listing 11).

5 Case Study

In this section, we present our case study, which is based on the *BattleOfTanks*[4] software product line (SPL). The *BattleOfTanks* SPL is a two player game. Each player controls a tank in a variable environment (obstacles, power ups, sound, color, ...). With this case study we want to show the feasibility of our language extension and show that it is possible to reduce redundant code artifacts. For our case study, we require a reference for comparison with PARAMETRIC DELTAJ regarding potential improvements. Hence, for comparison the subject of the case study must be written in plain DELTAJ and as an SPL with an attributed feature model (AFM) in PARAMETRIC DELTAJ. Unfortunately, no such case studies exist, as DELTAJ 1.5 is quite recent and existing JAVA based SPL case studies only rely on Boolean FMs. Therefore, we chose *BattleOfTanks*, one of the biggest (w.r.t. number of features) JAVA based SPLs with a Boolean FM freely available and adapted the FM towards an AFM. The SPL consists of 144 features, 1975 lines of code (LOC), 11 classes and 2206 possible products. The original SPL is implemented in plain JAVA using *Antenna*[5], a preprocessor for the JAVA programming language, to implement variability.

Since we introduced constructs to eliminate the state explosion in PARAMETRIC DELTAJ, we need less redundant code to implement the case study than in plain DELTAJ. Hence, in the case study, we measure the number of necessary delta modules and LOC for the SPL realized with DELTAJ and with our extension PARAMETRIC DELTAJ. We compare these values to quantify the efficiency of our improvement compared to plain DELTAJ.

5.1 BattleOfTanks in DeltaJ

The DELTAJ version of BattleOfTanks has the same number of features as the original *Antenna* implementation. It contains 124 delta modules with 2484 LOC. The difference in the number of features to the number of delta modules is due to the fact that not all features from the original *Antenna* version are implemented. In consequence, the resulting number of delta modules is less than the number of features from the feature module. We had five delta modules that were related to feature combinations (a.k.a. feature interactions). As the preprocessor allowed a more fine grained code manipulation (on statement level), we had to introduce many new methods to the original implementation to encapsulate the product specific variability in the source code base of DELTAJ. Using these new methods, we were able to define the DOP modify operations to implement variability of the SPL. Hence, we were able to implement the alternative features. To implement the different combinations of the *PowerUp* features, which are organized in an or group, we used the `original` call to extend the method according to the different feature selections. In this case, we were able to mimic code manipulation on statement level.

5.2 BattleOfTanks in parametric DeltaJ

The transformed FM of the BattleOfTanks SPL contains 51 features, 3 distinct enumerations for the shape of the tanks, for the background color and for the sound. With overall 33 enumeration literals, we were able to reduce the number of features to 89. Each enumeration literal represents a media file that can be used several times (e.g., shape of tanks for player 1 and player 2). They are now modeled with seven feature attributes. But not all alternative feature groups can be transformed to enum attributes as they differ significantly in their implementation. For instance, the alternative feature group containing the move operations of a tank, could not be improved by using an enumeration. The PARAMETRIC DELTAJ version of the BattleOfTanks SPL contains 43 delta modules and a total of 1927 LOC. All seven feature attributes belong to mandatory features that are

[4] `http://spl2go.cs.ovgu.de`
[5] `http://antenna.sourceforge.net`

present in all possible products of the SPL. This made it possible to introduce these seven attributes to a single delta module and decreased the total number of delta modules to 81. We introduced one new delta module, which contains only one class, that maps the product line enumeration literals to String values that represent the used media files. With this delta module, we modularized the input files.

5.3 Results and Discussion

We showed that by construction in case of PARAMETRIC DELTAJ, the number of necessary delta modules and LOC are significantly smaller than in plain DELTAJ, if we consider an AFM used in the SPL. The fewer number of necessary delta module is possible, because an AFM is by design more concise than an equivalent boolean FM. This suggests that PARAMETRIC DELTAJ has a similar level of reduction to DELTAJ as an AFM to an FM. If we count the features needed in plain DELTAJ and PARAMETRIC DELTAJ including the literals of the enumerations, we arrive at a difference of 60 features. That is a reduction alone in the literals for the AFM of 41.66%. If we compare the number of delta modules, we come to a reduction of 65.32%. For LOC, we arrive at a reduction of 22.42%.

Comparing the LOC of the original *Antenna* implementation, with the PARAMETRIC DELTAJ implementation, there is no significant improvement.

We also discovered a use case to allow a delta module to be used multiple times for one configuration. The *PowerUp* feature group could be implemented with only one delta module with parameters. That delta module would extend the method to handle the *PowerUps* by using the `original` clause based on the parameter value from the configuration. For this case, it is theoretically possible to apply the delta module multiple times.

5.4 Threats to Validity

We only showed one case study were we came to the results that PARAMETRIC DELTAJ is capable of directly supporting feature attributes compared to DELTAJ. PARAMETRIC DELTAJ is designed to eliminate the feature explosion if we consider AFMs in DELTAJ, which we show in this case study. For plain Boolean FMs PARAMETRIC DELTAJ is as capable as DELTAJ, since it contains the complete grammar of DELTAJ.

The case study has the drawback, that we adapted the original *Antenna* implementation to DELTAJ ourselves. The *Antenna* implementation uses a Boolean FM as source, which we only needed to adapt for the PARAMETRIC DELTAJ implementation. Fortunately, the example used boolean features for colors, shapes and sound, which we could easily design as enum typed feature attributes for the PARAMETRIC DELTAJ implementation. For the implementation we used a minimized number of delta modules in the DELTAJ and PARAMETRIC DELTAJ implementation for the rest of the optional boolean features. This strategy minimized the delta modules needed in the DELTAJ implementation except for delta modules that implemented the feature interactions, which only occur if a combination of features results in code adaption operations. This was helpful for the DELTAJ implementation, since it simplifies the transformation to DELTAJ.

6 Related Work

There are other approaches to implement software product lines (SPLs). For example, preprocessors such as the C processor CPP, which activates code fragments enclosed by `#ifdef` and `#endif`, and approaches such as Feature-Oriented Programming (FOP) [25] and Aspect-Oriented Programming (AOP) [21] that compose the source code from different artifacts. Delta-Oriented Programming (DOP) transforms code artifacts to reach the necessary variability. We compare the concept of PARAMETRIC DELTAJ with implementations of these approaches.

Preprocessors use commands to implement variability at compile-time. A well known preprocessor is the C processor CPP [20].The configuration can deal with Boolean values (e.g., for features), but also with other types of values. In case of the *Linux Kernel*, such a configuration contains features and attributes and is configured by the tool *kconfig* [28]. Therefore, preprocessors are capable of supporting the implementation of SPLs with AFMs. Nonetheless, preprocessors can lead to complex and obfuscated code and do not support modularization and evolution [21].

FeatureHouse [4] is a language-independent FOP implementation for composing different kinds of software artifacts. FOP uses feature modules to encapsulate those artifacts. The feature modules used in *FeatureHouse* do not yet support the specification of parameters coming from an AFM. This again leads to an explosion of Boolean features representing every domain value of every feature attribute.

The AOP implementation *AspectJ* weaves additional code in form of aspects into a program. An extension for AOP on *AspectJ* was proposed by Alvarez [2], which introduces parameters on aspects. They go even further than our method as their parameters not only consider types and constants but also classes and methods. However, with their approach, it is not possible to change classes, packages, imports, super classes or remove and modify existing methods and fields, which PARAMETRIC DELTAJ is capable of.

CaesarJ [5]is an AOP JAVA based programming language with the aim to facilitate better modularity and development of reusable components. They are designing Aspects as reusable components with a clear abstraction. To improve the separation of concerns of *AspectJ*, they modularize components, which consist of multiple collaborating classes. The binding to the application is done in a separate module. As *AspectJ*, *CaesarJ* is not intentionally designed for SPLs and, therefore, does not support the direct propagation of selected feature attribute values to code artifacts.

Hyper/J [22] is a language for JAVA that supports multi-dimensional separation of concerns called Hyperspaces [23]. A composition is defined by three parts, the *hyperspace*, a mapping of concerns and a *hypermodule*. A *hyperspace* describes a set of units, which can either be files, methods and constructors, classes, interfaces or packages. A concern (e.g., a feature) is implemented by a *hyperslice*, which groups all units for a concern. Together, they define the mapping of concerns. A *hypermodule* integrates *hyperslices* to build larger *hyperslices* or even complete systems. This mechanism can be used for feature composition in SPLs, but so far fails to support the automatic forwarding of chosen domain values from the product configuration.

Clarke et al. [8] introduce a core language for abstract behavioral specification (ABS) for designing executable models of distributed object-oriented systems. To model variable systems, they use the *Micro Textual Variability Language* (μTVL). μTVL is a language to describe FMs with feature cardinalities, group cardinalities and feature attributes. They also support the DOP paradigm to model variability to their specification. Feature attributes can be propagated to the delta modules in a similar way compared to our approach. However, ABS is not a programming language that generates executable Java code in itself, but only by subsequent code generation steps from ABS.

Component-based development offers similar characteristics for software development as PARAMETRIC DELTAJ, i.e., reuse, composability and explicit component dependencies [29]. Invasive software composition even offers transformation-based composition of such components [6]. However, component-based software composition lacks explicit variability modeling that is required for SPL development. Moreover, components may be configured externally using, e.g., configuration files such as JAVA property files. Unfortunately dependencies between features and attributes cannot be expressed in such configuration files, which introduce inconsistencies for the desired software system. Elsner et al. [14] propose therefore a framework to check for inconsistencies in and between different configuration files. This is obsolete if system properties are written as feature attributes in PARAMETRIC DELTAJ, since constraints over the attributes are checked by the used CSP solver and only valid products can be generated.

7 Conclusion

In this paper, we extended the SPL programming language DELTAJ with parameters and called the language extension PARAMETRIC DELTAJ. Namely, we extended the DELTAJ product line declaration language and DELTAJ delta modules to deal with feature attributes and formal parameter declarations. With these, we were able to propagate feature attribute values from the feature model (FM) via product configuration directly into the JAVA source code. To express also constraints on the feature attribute values, we also extended the expression language of the DELTAJ product line declaration language. We showed in a case study that we can significantly reduce the number of necessary delta modules by using parameters.

In future work, we would like to extend our method to interoperate with other extensions of FMs, such as feature cardinalities and group cardinalities or multiple instances of the same feature (cloned features). Our case study has shown that it might be useful to allow application of delta modules multiple times, but we need to find restrictions that have to be observed to guarantee well-formed products with multiple applied delta modules. We will evaluate this idea by analyzing different use cases and scenarios. We also intend to increase the range of parameter types for delta modules to further increase the reuse potential. Another interesting direction of future work is to extend PARAMETRIC DELTAJ to support the realization of multi SPLs [12].

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable comments and suggestions to improve the quality of the paper. This work was partially supported by the DFG (German Research Foundation) under grant SCHA1635/2-2 and by the European Commission within the project HyVar (grant agreement H2020-644298).

References

[1] Software Product Line Hall of Fame, http://splc.net/fame.html.

[2] J. Alvarez. Parametric Aspects: A Proposal. In *Proc. of ECOOP'2004 Workshop on Reflection, AOP and Meta-Data for Software Evolution*, 2004.

[3] S. Apel and C. Kästner. An Overview of Feature-Oriented Software Development. *Journal of Object Technology*, 8(5):49–84, 2009.

[4] S. Apel, C. Kastner, and C. Lengauer. FEATUREHOUSE: Language-independent, Automated Software Composition. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 221–231, Washington, DC, USA, 2009. IEEE Computer Society.

[5] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. Transactions on aspect-oriented software development i. chapter An Overview of CaesarJ, pages 135–173. Springer-Verlag, Berlin, Heidelberg, 2006.

[6] U. Amann. Invasive software composition. In *Invasive Software Composition*, pages 107–145. Springer Berlin Heidelberg, 2003.

[7] L. Bettini, F. Damiani, and I. Schaefer. Compositional Type Checking of Delta-Oriented Software Product Lines. *Acta Inf.*, 50(2):77–122, 2013.

[8] D. Clarke, R. Muschevici, J. Proença, I. Schaefer, and R. Schlatte. Variability Modelling in the ABS Language. In *Proceedings of the 9th International Conference on Formal Methods for Components and Objects*, FMCO'10, pages 204–224, Berlin, Heidelberg, 2011. Springer-Verlag.

[9] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2002.

[10] K. Czarnecki, T. Bednasch, P. Unger, and U. Eisenecker. Generative Programming for Embedded Software: An Industrial Experience Report. In D. Batory, C. Consel, and W. Taha, editors, *Generative Programming and Component Engineering*, volume 2487 of *Lecture Notes in Computer Science*, pages 156–172. Springer Berlin Heidelberg, 2002.

[11] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[12] F. Damiani, I. Schaefer, and T. Winkelmann. Delta-Oriented Multi Software Product Lines. In *Proceedings of the 18th International Software Product Line Conference - Volume 1*, SPLC '14, pages 232–236, New York, NY, USA, 2014. ACM.

[13] S. Efftinge, M. Eysholdt, J. Köhnlein, S. Zarnekow, R. von Massow, W. Hasselbring, and M. Hanus. Xbase: Implementing Domain-specific Languages for Java. *SIGPLAN Not.*, 48(3):112–121, Sept. 2012.

[14] C. Elsner, D. Lohmann, and W. Schroder-Preikschat. Fixing Configuration Inconsistencies Across File Type Boundaries. In *Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on*, pages 116–123, Aug 2011.

[15] 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 University Software Engineering Institute, Nov. 1990.

[16] A. S. Karataş, H. Oğuztüzün, and A. Doğru. From Extended Feature Models to Constraint Logic Programming. *Science of Computer Programming*, 78(12):2295 – 2312, 2013.

[17] J. Koscielny, S. Holthusen, I. Schaefer, S. Schulze, L. Bettini, and F. Damiani. DeltaJ 1.5: Delta-Oriented Programming for Java 1.5. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '14, pages 63–74, New York, NY, USA, 2014. ACM.

[18] K. Kuchcinski and R. Szymanek. JaCoP-Java Constraint Programming Solver. In *CP Solvers: Modeling, Applications, Integration, and Standardization, co-located with the 19th International Conference on Principles and Practice of Constraint Programming*, 2013.

[19] U. Lesta, I. Schaefer, and T. Winkelmann. Detecting and Explaining Conflicts in Attributed Feature Models. In *Proceedings 6th Workshop on Formal Methods and Analysis in SPL Engineering, FMSPLE 2015, London, UK, 11 April 2015.*, pages 31–43, 2015.

[20] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An Analysis of the Variability in Forty Preprocessor-based Software Product Lines. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 105–114, New York, NY, USA, 2010. ACM.

[21] D. Lohmann, F. Scheler, R. Tartler, O. Spinczyk, and W. Schröder-Preikschat. A Quantitative Analysis of Aspects in the eCos Kernel. *SIGOPS Oper. Syst. Rev.*, 40(4):191–204, Apr. 2006.

[22] H. Ossher and P. Tarr. Hyper/J: Multi-dimensional Separation of Concerns for Java. In *Proceedings of the 22Nd International Conference on Software Engineering*, ICSE '00, pages 734–737, New York, NY, USA, 2000. ACM.

[23] H. Ossher and P. Tarr. Multi-Dimensional Separation of Concerns and the Hyperspace Approach. In M. Aksit, editor, *Software Architectures and Component Technology*, volume 648 of *The Springer International Series in Engineering and Computer Science*, pages 293–323. Springer US, 2002.

[24] K. Pohl, G. Böckle, and F. J. v. d. Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[25] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In M. Akşit and S. Matsuoka, editors, *ECOOP'97 Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 419–443. Springer Berlin Heidelberg, 1997.

[26] I. Schaefer, L. Bettini, F. Damiani, and N. Tanzarella. Delta-Oriented Programming of Software Product Lines. In *Proceedings of the 14th International Conference on Software Product Lines: Going Beyond*, SPLC'10, pages 77–91, Berlin, Heidelberg, 2010. Springer-Verlag.

[27] I. Schaefer and F. Damiani. Pure Delta-Oriented Programming. In *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development*, FOSD '10, pages 49–56, New York, NY, USA, 2010. ACM.

[28] S. She and T. Berger. Formal Semantics of the Kconfig Language. *Technical note, University of Waterloo*, page 24, 2010.

[29] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.