A Programming Language for Software Product Lines *

Lorenzo Bettini¹, Viviana Bono¹, Ferruccio Damiani¹, and Ina Schaefer²**

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

Abstract. A software product line is a set of software systems with well-defined commonalities and variabilities defined in terms of product features. In order to implement feature-based variability within the object-oriented programming paradigm, we present a novel linguistic approach based on the concept of program deltas. The implementation of a product line is divided into a core module and a set of delta modules. The core module comprises a set of classes that implement a valid product of the product line. Delta modules specify changes to the core module in order to incorporate specific product features. A product implementation is obtained by changing the core module according to the applicable delta modules. We propose a constraint-based type system for core and delta modules that provides static guarantees that the resulting product implementations are safe without having to generate all possible products. In order to show the feasibility of our approach, we use Featherweight Java for the implementation of product classes.

1 Introduction

A software product line (SPL) is a set of software systems with well-defined commonalities and variabilities [7, 20]. The variabilities of the products can be defined in terms of product features [11], which are important product characteristics. A feature model describes the set of possible products of the product line by the set of valid feature configurations, i.e., the set of features a product implements.

In order to represent feature-based variability in the implementation of product lines, two main approaches can be distinguished. First, syntactic approaches, e.g., [3, 12], mark the code of the product line syntactically with product features. A product for a specific feature configuration is obtained by modifying the code on a syntactic level allowing fine-grained modifications of code. Linguistic approaches, most prominently feature-oriented programming [5, 2, 8], associate features with programming language constructs. The code for a product is obtained by linguistic operations. In feature-oriented programming, feature modules are assembled to products by feature composition. While linguistic constructs associate features with more coarse-grained modifications of product code, they can provide static guarantees about the resulting products on the linguistic level, e.g. by type systems [2, 8].

In this paper, we present a novel linguistic approach to implementing feature-based variability of software product lines within the object-oriented programming paradigm.

* The authors of this work have been partially supported by the Ateneo Italo-Tedesco / Deutsch-Italienisches Hochschulzentrum.
** This author has been supported by the Deutsche Forschungsgemeinschaft (DFG).
We transfer the concept of delta modeling [21, 22], a general approach to represent feature-based variability during all phases of product line development, to the programming language level. The implementation of a software product line is divided into a **core module** and a set of **delta modules**. The core module comprises a set of classes that implement a valid product of the product line for a specific set of product features. Delta modules specify changes to the core module in order to implement other products. A delta module can add classes to a product implementation and remove classes from a product implementation. Furthermore, existing classes can be modified. Modification of a class comprises the change of the parent class, the change of the constructor, and additions, removals and renamings of fields and methods. A delta module contains an application condition determining for which feature configuration the specified modifications are to be carried out. In order to obtain a product implementation for a particular feature configuration, an incremental composition of the core module and the delta modules is performed by applying the modifications of all delta modules with valid application condition to the core module. In order to ensure that there are no conflicting modifications, the ordering of delta module application can be explicitly defined. Static guarantees that the implementations of the resulting products are safe are provided by a constraint-based type system that supports the analysis of the product line implementation without having to generate all products. The concept of core and delta modules is independent from a concrete programming language. In order to show the feasibility of our approach, we use FJ (**FEATHERWEIGHT JAVA**) [10] for the implementation of product classes.

The presented programming language combines the possibility to flexibly represent feature-based variability with the ability to effectively provide static guarantees on the resulting products. The core product is a complete product for any valid feature configuration. This allows developing it with well-established techniques from single application engineering. The choice of the core product is not fixed such that a different product line designs are supported within the same approach. The application conditions attached to delta modules do not necessarily refer to a single feature, but can refer to any combination of features. The implementations for combinations of features can be specified separately, thus increasing the expressiveness of code modifications induced by features and avoiding the optional feature problem [16]. The presented programming language supports the modular and evolutionary development of product lines. If additional features should be included in the products of the product line, new delta modules dealing with the effect of these features can be added to the implementation.

**Organization of the Paper.** In Section 2, we present the concepts of the proposed programming language. In Section 3, we demonstrate its flexibility at different implementations of the same product line. Section 4 introduces preliminary concepts for the formal calculus. In Section 5, we present the formal calculus and the semantics for product generation. Section 6 describes the constraint-based type system. In Section 7, we compare with presented language with related approaches. Section 8 summarizes the paper with an outlook to future work.

## 2 Implementing Feature-Based Variability with Delta Modules

As a running example to illustrate our approach, we use a SPL of bank accounts [8]. Figure 1 shows the feature model of the bank account SPL determining the different products by possible combinations of features. The mandatory **Base** feature represents the
basic functionality of any bank account allowing to store the current balance and to update it. This functionality can be extended by the optional Sync(hronized) feature guaranteeing synchronized access to the account. The features Retirement and Investment that provide the possibility to store an additional bonus for the account are optional and mutually exclusive. The optional feature With Holder adds a reference to the holder of the account and requires the presence of either the Retirement or the Investment feature.

Core modules. In our approach, feature-based variability on the implementation level is captured by a core module and a set of delta modules. A core module corresponds to the implementation of a product for a valid feature configuration. It defines the starting point for generating all the other products by delta module application. The core module depends on the underlying programming language used to implement the products. In the context of this work, the core module contains a set of Java classes. To give a structure to the SPL implementation, we enclose this set of classes inside a core block and specify the features of the product implemented by the core module:

```java
core <Feature names> { <Java classes> }
```

As we will show in Section 3, the product represented by the core module can be any valid product. Listing 1 contains a core module for the bank account SPL. It implements the account with only the Base feature by the Account class. In this and the following examples, we will use the full Java syntax, while the calculus presented in Section 5.1 is based on FJ.

Delta modules. Delta modules specify changes to the core module in order to implement other products. The alterations inside a delta module act on a class level (by adding, removing and modifying a class) and on a class structure level (by modifying the internal structure of a class by changing the super class, by changing the constructor and by adding, removing and renaming fields and methods). An application condition is attached to every delta module in its when clause determining for which feature configurations the specified alterations are to be carried out. The application condition creates the link from the features in the feature model to the implementation. Application conditions are Boolean constraints over features. This allows specifying delta modules for combinations

Listing 1: Core module implementing Base feature

```java
core Base {
    class Account extends Object {
        int balance;
        void update(int x) { balance += x; }
    }
}
```
delta Dinvestment when Investment {
  modifies class Account extending WaMu {
    adds int 401balance;
    renames update to originalUpdate;
    adds void addBonus (int x) { 401balance += x;}
    adds void update (int x) { x = x/2; originalUpdate(x); addBonus(x);}
  }
}

Listing 2: Delta module for Investment feature

of features and also to handle explicitly the absence of features which makes the implementation of features very flexible.

Listing 2 shows the delta module defining the changes of the Account in the core module to incorporate the Investment feature. First, the delta module changes the superclass of Account to WaMu (not shown here). It specifies to add a field 401balance. It redefines the method update by renaming the existing version to originalUpdate and by adding a new definition of the method update. Furthermore, a new method addBonus is added. The application condition in the when clause specifies that this ∆-module is applied for all feature configurations in which the Investment feature is present.

Delta application. In order to obtain a product for a particular feature configuration, the changes specified by delta modules with valid application conditions are applied to the core (delta application). The alternations specified in one delta module are applied simultaneously. In order to ensure, for instance, that a class to be modified exists or that a modification of the same method by different delta modules does not cause a conflict, an ordering on the application of the delta modules can be defined by means of the after clause. This ordering implies that a delta module is only applied to the core module after all delta modules with a valid application condition mentioned in the after clause have been applied. With the after clauses, a partial ordering on the set of delta modules is defined that captures only the necessary dependencies. Note that specifying that a delta module A has to be applied after the delta module B does not mean that A requires B: it only specifies that if a feature configuration satisfies the when clause of both A and B, then B must be applied before A (this is illustrated for the feature configuration with both Base and Sync, later in this section).

The delta module implementing the Sync feature of the Bank Account SPL is presented in Listing 3. It specifies to change the class Account by adding a lock field (whose class Lock is not shown here) and by wrapping the code for synchronization around the method update. The original method update is renamed into unsync_update, and a method update is introduced which calls unsync_update in a synchronized way (locking before the call, and unlocking afterwards). Note that this delta module must be applied after the delta module for the Investment or the Retirement feature (if any of them has a valid application condition), since the latter modify the update method themselves. With the after clause, we ensure that the synchronization takes place on the correct version of the update method.

The generation of a product for a given feature configuration consists of the following steps, performed automatically by the system:
delta DsyncUpdate after Dretirement, Dinvestment when Sync {
    modifies class Account {
        adds Lock lock;
        renames update to unsync_update;
        adds void update(int x) { lock.lock(); unsync_update(x); lock.unlock(); }
    }
}

Listing 3: Delta module for Sync feature

class Account extends WaMu {
    int balance;
    int 401balance;
    void original_update(int x) { balance += x; }
    void update(int x) { x = x/2; original_update(x); addBonus(x); }
    void addBonus(int x) { 401balance += x; }
}

Listing 4: Account with Base and Investment features

1. Find all delta modules with a valid application condition according to the feature configuration (specified in the when clause); and
2. Apply the selected delta modules to the core module in any linear ordering respecting the partial order induced by the after clauses.

As an example of a product implementation resulting from delta application, the implementation of an account with the Base and Investment features is shown in Listing 4. It is the result of applying the delta module Dinvestment in Listing 2 to the core module in Listing 1. As stated earlier, the delta module for the Sync feature in Listing 3 must come after the delta module for the Investment feature, in case the feature configuration contains both features, which is not the case here. Indeed in Listing 5, the result of applying the delta module DsyncUpdate to the core module is depicted, representing the product for the feature configuration Base and Sync.

Note that the automatic generation of products by delta application is only performed if the implementation of the SPL is well-formed (a notion which is formally defined in Section 5). Well-formedness requires that all delta modules associated to a valid feature configuration must be applicable to the core module in any order compatible with the partial order provided by the after clauses, which implies, for instance, that all renamed or modified classes, methods and fields exists. Furthermore, the delta modules non comparable with respect to the after partial order must be compatible, i.e., not causing a

class Account extends Object {
    int balance;
    Lock lock;
    void unsync_update(int x) { balance += x; }
    void update(int x) { lock.lock(); unsync_update(x); lock.unlock(); }
}

Listing 5: Account with Base and Sync features
conflict. This means that all potential conflicts between modifications targeting the same class, method or field are resolved by the ordering specified with the after clauses.

3 Implementing Software Product Lines

In this section, we show how the programming language constructs presented in the previous section can be used to implement the bank account SPL. We show the flexibility of the introduced concepts to support different product line implementations starting from different core products. Furthermore, we illustrate the design freedom for the creation of delta modules.

Starting from a Simple Core. The core module has to contain an implementation of a valid product of the product line. One possibility is to take only the mandatory features and a minimal number of required alternative features, if applicable. In our example, the Base feature is the only mandatory feature. Listing 1 shows the respective core module if the SPL implementation is started with a product only containing the Base feature.

In order to represent all possible products of the product line, delta modules have to be defined that modify the core product to incorporate further product features. The delta modules for the Investment and the Sync feature are already shown in Listings 2 and 3. In addition to that, we need a delta module for the Retirement feature shown in Listing 6 (the class Lehman serving as new super class is not shown here). The features Investment and Retirement are optional and mutually exclusive (see Figure 1) which is not expressed directly in the when clauses of their delta modules (Listing 2 and 6, respectively). Because only valid feature configurations according to the constraints of the feature model are used for delta application, this will be ensured at delta application level.

Both delta modules act on the method update. By the after clause of the delta module DsyncUpdate in Listing 3, it is ensured that the correct version of the update method is synchronized. Moreover, if we want to have the Sync feature in presence of the Retirement or Investment features, we need to synchronize also the addBonus method. Thus, we have to implement the additional delta module DsyncBonus shown in Listing 7, which must be applied after the delta modules Dretirement and Dinvestment. This example shows that code required to connect the behavior of two optional features can be introduced by an additional delta module solving the optional feature problem [16]. In Listing 8 we show the Account with Base, Sync and Investment. This is the result of applying first delta Dinvestment, then DsyncUpdate and DsyncBonus.

\[\text{Listing 6: Delta module for Retirement feature}\]

\[
\text{delta Dretirement when Retirement \{ }
\text{ modifies class Account extending Lehman \{ }
\text{ adds int } 401\text{balance; }
\text{ removes int balance; }
\text{ removes void update(int x);} \\
\text{ adds void update(int x) \{ addBonus(x); \}} \\
\text{ adds void addBonus(int x) \{ 401\text{balance }+= x; \}}
\text{ \}}
\text{ \}}
\]

Note that our example relies on the fact that if the same thread calls lock() on the same Lock instance twice it will not deadlock.
Listing 7: Delta module for Sync feature in presence of Retirement or Investment feature

default DsyncBonus after Dretirement, Dinvestment when Sync && (Retirement || Investment) {
    modifies class Account {
        renames addBonus to unsync_addBonus;
        adds void addBonus(int x) { lock.lock(); unsync_addBonus(x); lock.unlock(); }
    }
}

Listing 8: Account with Base, Sync and Investment features

class Account extends WaMu {
    int balance;
    int 401balance;
    Lock lock;
    void original_update(int x) { balance += x; }
    void unsync_update(int x) { x = x/2; original_update(x); addBonus(x); }
    void unsync_addBonus(int x) { 401balance += x; }
    void update(int x) { lock.lock(); unsync_update(x); lock.unlock(); }
    void addBonus(int x) { lock.lock(); unsync_addBonus(x); lock.unlock(); }
}

Finally, the delta module for the With Holder feature, shown in Listing 9, adds a class Client with an Account field and a method acting on it to a product implementation. Note that the payday method relies on the fact that Account provides the method addBonus. This is ensured by the feature model that specifies that the With Holder feature requires the Retirement or the Investment feature such that the addBonus method is provided by delta application.

The delta modules for the Retirement and Investment features have some similarities: they both add the field 401balance and the addBonus method. Thus, an alternative implementation for these delta modules is to write a delta module DaddBonus comprising the common code, and to rewrite the delta modules for Retirement and Investment as it is shown in Listing 10. It is straightforward to check that the resulting products, e.g., the one depicted in Listing 8, do not change. The additional delta module allows reusing code, but increases the total number of delta modules. Depending on the size of the feature model, this is a tradeoff a programmer can take into consideration, thanks to the flexibility offered for the design of delta modules.

Starting from a Complex Core. The bank account SPL can also be implemented by starting with a core product containing, besides the mandatory Base feature, also an optional feature, like Sync. This is illustrated in Listing 11. Thus, we need the delta module DunsSyncUpdate, which removes the synchronization functionalities in order to generate

Listing 9: Delta module for With Holder feature

default DwithHolder when WithHolder {
    adds class Client {
        Account a;
        void payday(int x, int bonus) { a.addBonus(bonus); a.update(x); }
    }
}
delta DaddBonus when (Retirement || Investment) {
    modifies class Account {
        adds int 401balance;
        adds void addBonus(int x) { 401balance += x; }
    }
}
delta Dretirement after DaddBonus when Retirement {
    modifies class Account extending Lehman {
        removes balance;
        removes update;
        adds void update(int x) { addBonus(x); }
    }
}
delta Dinvestment after DaddBonus when Investment {
    modifies class Account extending WaMu {
        renames update to original_update;
        adds void update(int x) { x = x/2; original_update(x); addBonus(x); }
    }
}

Listing 10: Alternative implementation with delta module DaddBonus

a product with only the Base feature. The renaming of method unsync_update to update and the removal of the update method do not generate problems, since the modification operations are applied simultaneously.

In order to be able to generate all possible products, the delta modules of the previous implementation need some modifications. For instance, the delta module for the Investment feature must be changed to rename the method unsync_update instead of update. The delta module for the Retirement feature has to be modified similarly. In the delta module DsyncBonus of Listing 7, the after clause can simply be removed. Again, it is straightforward to check that the resulting products are the same as the ones which are generated by the previous core and delta modules.

4 Preliminary Concepts

FJ (FEATHERWEIGHT JAVA) [10] is a minimal calculus for Java, which focuses on a few basic concepts: mutually recursive class definitions, inheritance, object creation, field access, method invocation, method recursion through this, subtyping and type casts. The minimal syntax, typing and semantics make the type safety proof simple and compact, in such a way that FJ is a handy tool for studying the consequences of extensions with respect to Java [19].

The abstract syntax of FJ constructs is given in Figure 2. Following [10], we use the overline notation for possibly empty sequences. We write “œ” as short for a possibly empty sequence of expressions “e1,...,en” and “MD” as short for a possibly empty sequence of method definitions “MD1,...,MDn” (without commas). The empty sequence is denoted by •. The length of a sequence œ is denoted by #(œ). We abbreviate operations on sequences of pairs in similar way, e.g., we write “œ ˜f” as short for “œ1 f1,...,œn fn”, “œ ˜f;” as short for “œ1 f1;,...,œn fn;” and “this.œ ˜f = ˜f;” as short for “this.f1 = f1;,...,this.fn = fn;”.

Sequences of named elements (field, method or parameter names, field, method or class definitions,...) are assumed to contain no duplicate names (that is, the names of the elements of the sequence must be distinct).
```java
core Base, Sync {
    class Account extends Object {
        int balance;
        Lock lock;
        void unsync_update(int x) { balance += x; }
        void update(int x) { lock.lock(); unsync_update(x); lock.unlock(); }
    }
}
delta DunSyncUpdate after Dretirement, Dinvestment when !Sync {
    modifies class Account {
        removes Lock lock;
        renames unsync_update to update;
        removes update;
    }
}
delta Dinvestment when Investment {
    modifies class Account extending WaMu {
        adds int 401balance;
        renames unsync_update to originalUpdate;
        adds void addBonus (int x) { 401balance += x; }
        adds void unsync_update (int x) { x = x/2; originalUpdate(x); addBonus(x); }
    }
}
```

**Listing 11:** Alternative implementation starting from a different core.

```
CD ::= class C extends C { FB; KD }   class definitions
K ::= C(C k) { super(k); this.k = k; } constructor definitions
FD ::= C f                           field definitions
MD ::= C m(C k) { return e; }       method definitions
e ::= x | e.f | e.m(e) | new C(e) | (C)e expressions

Fig. 2. FJ syntax
```

The set of variables includes the special variable `this` (implicitly bound in any method declaration), which cannot be used as the name of a method’s formal parameter. This restriction is imposed by the FJ typing rules (see [10]). Note that no special syntax for `this` is required since, in method bodies, it is treated as an ordinary variable.

A class definition `class C extends D { FB; KD }` consists of its name `C`, its superclass `D` (which must always be specified, even if it is `Object`), a list of field names `C f` with their types, a constructor `K` and a list of method definitions `MD`. The instance variables of `C` are added to the ones declared by `D` and its superclasses and are assumed to have distinct names.

The constructor definition `C(A k, B g) { super(k); this.g = g; }` specifies how to initialize the fields of an instance of `C`: there is one parameter for each field (including the inherited ones), with the same name of the field; its body consists of a call to the parent class constructor to initialize the inherited fields `k` with the parameters of the same name, followed by an assignment to the new fields `g` (defined in `C`) of the parameters of the same name. In FJ [10], the order of the constructor parameters `g` is determined by the order in which the corresponding fields are declared in the class `C`. In this paper, we use a slightly more liberal syntax (that will provides more flexibility in programming a SPL for FJ programs). Namely, the order of the constructor parameters `g` may be a permutation of
the order of the corresponding field definitions in C. For instance we admit the following

class D extends Object {
    A a; B b; C c;
    super(B b, A a, C c) { super(); this.b=b; this.a=a; this.c=c; }
}
class E extends D {
    F f; G g;
    E(B b, A a, C c, G g, F f) { super(b,a,c); this.g=g; this.f=f; }
}

A method definition $\mathcal{MD}$ specifies the name, the signature and the body of a method; a
body is a single return statement since FJ is a functional core of Java.

A class table $\mathcal{CT}$ is a mapping from class names to class definitions. The subtyping
relation $<: \; \text{on classes (types)}$ is the reflexive and transitive closure of the extends relation
(the immediate subclass relation, given by the extends clauses in $\mathcal{CT}$). The class $\text{Object}$
has no members and its definition does not appear in $\mathcal{CT}$. We assume that a class table $\mathcal{CT}$
satisfies the following sanity conditions: (i) $\mathcal{CT}(\text{C}) = \text{class C} \ldots$ for every $\text{C} \in \text{dom}(\mathcal{CT})$
(ii) for every class name $\text{C}$ (except $\text{Object}$) appearing anywhere in $\mathcal{CT}$, we have $\text{C} \in \text{dom}(\mathcal{CT})$; (iii) there are no cycles in the transitive closure of the extends relation.

A FJ program is a class table $\mathcal{CT}$ (containing all the class definitions of the program).

A class definition $\mathcal{CD}$ can be understood as a mapping from the keyword extends to
a class name, from the keyword constructor to class constructor definition and from
field/method names to field/method definitions. We use the metavariable $a$ to range over
field/method names, and the metavariable $\mathcal{AD}$ to range over field/method definitions. The lookup of the definition of the field/method $a$ in class $\text{C}$ is written $a\mathcal{Def}(\text{C})(a)$. Formally,
for every class $\text{C}$ in $\text{dom}(\mathcal{CT})$, the function $a\mathcal{Def}(\text{C})$ is defined as follows:

$$a\mathcal{Def}(\text{C})(a) = \begin{cases} \mathcal{CT}(\text{C})(a) & \text{if } a \in \text{dom}(\mathcal{CT}(\text{C})) \\ a\mathcal{Def}(\text{D})(a) & \text{if } a \notin \text{dom}(\mathcal{CT}(\text{C})) \text{ and } \mathcal{CT}(\text{C})(\text{extends}) = \text{D} \end{cases}$$

The type of a method is a pair, $\text{B} \rightarrow \text{B}$, of a sequence of argument types $\text{B}$ and a return
type $\text{B}$. Given a field definition $\mathcal{FD} = \text{C} \text{I}$ and a method definition $\mathcal{MD} = \text{C} \text{m}(\text{C} \times) \{ \cdots \}$, we
write $\text{signature}(\mathcal{FD})$ to denote the type $\text{C}$ of the field $\text{I}$ and $\text{signature}(\mathcal{MD})$ to denote the type
$\text{C} \rightarrow \text{C}$ of the method $\text{m}$. For a constructor definition $\text{K} = \text{C}(\text{A} \text{I}, \text{B} \text{G}) \{ \text{super}(\text{I}); \text{this}.\text{G} = \text{G} \}$ we write $\text{header}(\text{K})$ to denote the constructor header $\text{C} \{ \text{C} \text{I} \}$.

5 The FAJ Calculus

In this section, we present FAJ (FEATHERWEIGHT DELTA JAVA), a minimal calculus for
programming a product lines of FJ programs by defining a feature model, a core module
and a set of delta modules.

5.1 Syntax

The abstract syntax of FAJ constructs is given in Figure 3. The constructs for class defini-
tions $\mathcal{CD}$, field definitions $\mathcal{FD}$ and method definitions $\mathcal{MD}$ are taken from FJ as given in

---

4. In [10], a program is a pair $\langle \mathcal{CT}, e \rangle$ of a class table and an expression $e$ (the program’s main
entry point). Such a notion of program can be encoded by adding to $\mathcal{CT}$ a distinguished class

```java
class Main { C main() { return e; } }
```

where $\text{C}$ is a suitable type for $e$. 

---
Fig. 3. F∆J: Syntax of feature model, core-module and delta-module definitions

| features | Base, Sync, Retirement, Investment, With_Holder |
| configurations | Base & & !(Retirement & & Investment) & & (With_Holder -> (Retirement | Investment)) |

Listing 12: F∆J code for the feature model in Figure 1

Figure 2. The metavariables \( \varphi \), \( \chi \) and \( \psi \) range over feature names. We write \( \bar{\varphi} \) as short for the set \( \{ \varphi \} \), i.e., the feature configuration containing the features \( \bar{\varphi} \). A feature model definition \( FMD \) declares the features of a software product line, \( \bar{\varphi} \), and a propositional formula, \( \Pi \), describing the set of valid feature configurations \( \text{CONFIGURATIONS} \subseteq \mathcal{P}(\bar{\varphi}) \).

We assume that all feature names occurring in \( \Pi \) occur in \( \bar{\varphi} \). For example, the F∆J code for the feature model described by the feature diagram in Figure 1 is given in Listing 12.\(^5\)

A core module definition \( CMD \) consists of the feature configuration \( \bar{\varphi} \) implemented by the core product and the corresponding classes. A delta module definition \( DMD \) can be understood as a mapping from class names to delta-clause definitions, from the keyword \( \text{after} \) to a possibly empty set of delta module names and from the keyword \( \text{when} \) to an application condition. A delta clause definition \( DC \) can specify the addition, removal or modification of a class. The modification of a class is defined by potentially changing the super class and constructor and a sequence of delta subclauses \( DSC \) which can define additions, removals or renamings of fields and methods. The set of delta module names in the \( \text{after} \) clause are the delta modules that (when applicable) have to be applied before this delta module. The \( \text{when} \) clause determines for which feature configurations the delta module is applicable by a Boolean constraint over feature names. We write \( \text{DELTA}(\bar{\varphi}) \) to denote the set of names of delta modules whose application condition is satisfied by \( \bar{\varphi} \in \text{CONFIGURATIONS} \). A delta module table \( DMT \) is a mapping from delta module names to delta module definitions.

A F∆J SPL is a 4-tuple \( L = (FMD, \bar{\chi}, \text{CT}_\bar{\chi}, \text{DMT}) \) consisting of a feature model definition \( FMD \), a feature configuration \( \bar{\chi} \) (corresponding to the product implemented by the core-module), a class table \( \text{CT}_\bar{\chi} \) (containing the class definitions of the core module) and a delta-module table \( \text{DMT} \) (containing the delta modules of the SPL). To simplify the notation, in the following we always assume a fixed SPL \( L \), that is, fixed feature model definition \( FMD \), core feature configuration \( \bar{\chi} \), core class table \( \text{CT}_\bar{\chi} \), and delta-module table \( \text{DMT} \). We assume that:

\(^5\) The translation of a feature diagram into a propositional formula can be performed in a systematic way (see, e.g., [4]).
- The features $\mathcal{F}$ associated to the core module are defined in FMD.
- The class table of the core product $CT_\mathcal{F}$ satisfies the FJ sanity conditions.
- The delta-module table $DMT$ satisfies the following sanity conditions: (i) for every $\delta \in \text{dom}(DMT)$, $DMT(\delta) = \text{delta after } \vec{\delta}$ when $\vec{\delta} \in \text{dom}(DMT)$, all feature names occurring in $\Pi$ are defined in FMD, and for every $C \in \text{dom}(DMT(\delta))$, $DMT(\delta)(C) \in \{\text{adds class } C \ldots, \text{modifies } C \ldots, \text{removes } C\}$; (ii) for every class-name $C$ (except Object) appearing in DMT, we have $C \in CT_\mathcal{F} \cup (\cup_{\delta \in \text{dom}(DMT)} \text{dom}(DMT(\delta)))$; (iii) there are no cycles in the transitive closure of the after relation (given by the after clauses in the delta module table), thus, its reflexive and transitive closure is a partial order.

In the following, instead of $DMT(\delta) = \text{delta } \delta \ldots$, we write $\text{delta } \delta \ldots$; instead of $\text{dom}(DMT(\delta))$ we write $\text{dom}(\delta)$; and instead of $DMT(\delta)(C)$, $DMT(\delta)(\text{after})$ and $DMT(\delta)(\text{when})$ we write $\delta(C)$, $\delta(\text{after})$ and $\delta(\text{when})$, respectively.

The adds-domain, the modifies-domain and the remove-domain of a delta module definition $DMD$ are defined as follows:

$$
\text{addsDom}(DMD) = \{ C \mid \text{DMD}(C) = \text{adds class } C \ldots \}
$$
$$
\text{modifiesDom}(DMD) = \{ C \mid \text{DMD}(C) = \text{modifies } C \ldots \}
$$
$$
\text{removesDom}(DMD) = \{ C \mid \text{DMD}(C) = \text{removes } C \}
$$

A delta modifies-clause $DC$ can be understood as a mapping from the keyword extending to an either empty or singleton set of class names, from the keyword constructor to an either empty or singleton set of class constructors and from field/method names to delta subclauses. The adds-domain, the rename-from-domain, the rename-to-domain and the remove-domain of a delta modifies-clause $DC$ are defined as follows:

$$
\text{addsDom}(DC) = \{ a \mid \text{DMD}(a) = \text{adds } \ldots \}
$$
$$
\text{renamesFromDom}(DC) = \{ a \mid \text{DMD}(a) = \text{renames } a \ldots \}
$$
$$
\text{renamesToDom}(DC) = \{ a \mid \text{DMD}(a) = \text{renames } \ldots \text{to } a \}
$$
$$
\text{removesDom}(DC) = \{ a \mid \text{DMD}(a) = \text{removes } a \}
$$

### 5.2 Well-Formed FJ SPL

In order to define the application of a set of delta modules to a core module for generating product implementations, we define the notion of well-formed FJ SPL. The first three requirements for well-formedness deal with the structure of the core module and the delta modules. First, the core module has to represent a valid feature configuration. Second: there must not be a delta module that has to be applied for the core feature configuration; each delta module must be applied for at least one configuration; and for all configurations, the sets of delta modules that have to be applied must be distinct and contained in the delta module table. Third, the fields and methods added and renamed-to in every delta module must be disjoint. Thus, given the SPL $L = (\text{FMD}, \mathcal{F}, CT_\mathcal{F}, DMT)$, these requirements are formalized as follows:

**Requirement 1.** The feature configuration of the core product satisfies the configuration proposition in FMD (i.e., $\mathcal{F} \in \text{CONFIGURATIONS}$).

**Requirement 2.** The mapping $\Delta : \text{CONFIGURATIONS} \rightarrow \mathcal{P}(\text{dom}(DMT))$ is injective and such that $\Delta(\mathcal{F}) = \emptyset$ and $(\bigcup_{\mathcal{F} \in \text{CONFIGURATIONS}} \Delta(\mathcal{F})) = \text{dom}(DMT)$.
Requirement 3. For every \( \delta \in \text{dom}(\text{DMT}) \) and for every \( C \in \text{modifiesDom}(\text{DMT}(\delta)) \), we have that \( \text{addsDom}(\delta(C)) \cap \text{renamesToDom}(\delta(C)) = \emptyset \).

The fourth well-formedness requirement ensures that there are no conflicting modifications if a set of delta modules is applied to a core module. In order to check conflict-freedom, it suffices to consider only class namespace tables (an abstraction of class tables). A class namespace \( CN \) is a set of fields/methods names. A class namespace table \( CNT \) is a mapping from class names to class namespaces. We denote the namespace of a class definition \( CD \) by \( \text{namespace}(CD) \) and the class namespace table of the classes in the class table \( CT \) by \( \text{namespace}(CT) \). We write \( \text{namespace}(C) \) as short for \( \text{namespace}(\text{CT}(C)) \).

A conflict occurs if two (or more) delta modules are not compatible or if a delta module is not applicable to a class table. A set of delta modules is compatible if the sets of added, removed and modified classes are disjoint. A set of (compatible) delta-modules is applicable to a class table \( CT \) if: modified/removed classes exist and added classes do not exist in \( \text{dom}(CT) \); and for every delta modifies-clause, the removed or renamed-from fields/methods exist (in the modified class) and the added or renamed-to fields/methods do not exist (in the modified class). Formally, a delta module \( \delta \) is applicable to a class namespace table \( CNT \) if:

- \( \text{modifiesDom}(\delta) \cup \text{removesDom}(\delta) \subseteq \text{dom}(CNT) \).
- \( \text{addsDom}(\delta) \cap \text{dom}(CNT) = \emptyset \).
- for every \( C \in \text{modifiesDom}(\delta) \) the delta-clause \( DC = \delta(C) \) and the class namespace \( CN = CNT(C) \) are such that:
  - \( (\text{addsDom}(DC) \cup \text{renamesToDom}(DC)) \cap \text{dom}(CS) = \emptyset \), and
  - \( \text{removesDom}(DC) \cup \text{renamesFromDom}(DC) \subseteq \text{dom}(CS) \).

The class namespace table \( CNT' \) obtained by applying an applicable delta module \( \delta \) to the class namespace table \( CNT \), denoted by \( \text{APPLY}(\delta, CNT) \), is defined as follows:

\[
CNT'(C) = \begin{cases} 
  CNT(C) & \text{if } C \notin \text{dom}(\text{DMT}(\delta)) \\
  \text{namespace}(CD) & \text{if } C \in \text{addsDom}(\delta) \text{ and } \delta(C) = \text{adds CD} \\
  \text{APPLY}(\delta(C), CNT(C)) & \text{if } C \in \text{modifiesDom}(\delta)
\end{cases}
\]

where, for every \( C \in \text{modifiesDom}(\delta) \), the application of the delta-clause \( DC = \delta(C) = \text{modifies } \cdots \{ \cdots \} \) to the class namespace \( CN = CNT(C) \), denoted by \( \text{APPLY}(DC, CN) \), is the class namespace

\[
(\text{dom}(CN) - (\text{removesDom}(DC) \cup \text{renamesFromDom}(DC)) \\
\cup \text{addsDom}(DC) \cup \text{renamesToDom}(DC))
\]

Note that, if the delta-module \( \delta \) is applicable to the class namespace table \( CNT \), then

\[
\text{dom}(\text{APPLY}(\delta, CNT)) = (\text{dom}(CNT) - (\text{removesDom}(\delta) \cup \text{renamesFromDom}(DC))) \\
\cup \text{addsDom}(\delta) \cup \text{renamesToDom}(DC)
\]

Given a sequence of delta-modules \( \vec{\delta} = \delta_1 \cdots \delta_n \ (n \geq 0) \) and a class namespace table \( CNT \) we will write \( \text{APPLY}(\vec{\delta}, CNT) \) as short for \( \text{APPLY}(\delta_1, \text{APPLY}(\cdots, \text{APPLY}(\delta_n, CNT) \cdots)) \). We write \( CNT_{\vec{\delta}} \) to denote the class namespace table obtained by application of \( \text{DELTAS}(\vec{\delta}) \) to the core module for the feature configuration \( \vec{\delta} \).
Proposition 1. For every class table \( \text{CT} \), if the delta-modules \( \delta_1, \ldots, \delta_n \) \( (n \geq 0) \) are compatible and applicable to the class namespace table \( \text{CNT} \), then \( \text{APPLY}(\delta_1 \cdots \delta_n, \text{CNT}) = \text{APPLY}(\delta_1 \cdots \delta_n, \text{CNT}) \), for every permutation \( i_1, \ldots, i_n \) of \( 1, \ldots, n \).

Conflicting modifications in a set of delta modules should not arise if the delta modules are applied in an order respecting the after partial order (that is, the partial order induced by the after clauses). The after partial order can be represented by a direct acyclic graph, that we will call the after-DAG, such that there is an arrow from \( \delta_1 \) to \( \delta_2 \) if and only if \( \delta_1 \) after \( \cdots \) \( \delta_2 \cdots \). The after-level of a delta-module \( \delta \), denoted by \( \text{afterLevelOf}(\delta) \), is the length of the longest path in the after-DAG ending in \( \delta \). A set of delta-modules \( \text{DELTAS}(\phi) \) for feature configuration \( \phi \) is conflict-free if the partition of \( \text{DELTAS}(\phi) \) into the sets of the delta-modules with the same after-level, \( \{\phi^{(1)}\}, \ldots, \{\phi^{(k)}\} \) (where \( k \geq 0 \) and the sets are listed in increasing order with respect to the after-level) is such that, for all \( i \in \{1, \ldots, k\} \), the delta-modules in \( \phi^{(i)} \) are compatible, and applicable to the class namespace table \( \text{APPLY}(\phi^{(i+1)} \cdots \phi^{(k)}, \text{CNT}_\phi) \), where \( \text{CNT}_\phi = \text{namespace}(\phi, \text{CT}_\phi) \) is the class namespace table of the core product.

Requirement 4. For every feature configuration \( \phi \) in \( \text{CONFIGURATIONS} \), the set of delta modules \( \text{DELTAS}(\phi) \) is conflict-free.

Definition 1 (Well-formed SPL). A FAJ SPL \( L \) is well-formed iff \( L \) satisfies the Requirements 1, 2, 3 and 4 above.

5.3 Generating FAJ Programs from a Well-Formed FAJ SPL

Given a delta-module \( \delta \) and a class table \( \text{CT} \) such that \( \delta \) is applicable to \( \text{namespace}(\text{CT}) \), the application of \( \delta \) to \( \text{CT} \), denoted by \( \text{APPLY}(\delta, \text{CT}) \), is the class table \( \text{CT}' \) defined as follows:

\[
\text{CT}'(\text{C}) = \begin{cases} 
\text{CT}(\text{C}) & \text{if } \text{C} \not\in \text{dom}(\text{DMT}(\delta)) \\
\text{CD} & \text{if } \text{C} \in \text{addsDom}(\delta) \text{ and } \delta(\text{C}) = \text{adds CD} \\
\text{APPLY}(\delta(\text{C}), \text{CT}(\text{C})) & \text{if } \text{C} \in \text{modifiesDom}(\delta)
\end{cases}
\]

where, for every \( \text{C} \in \text{modifiesDom}(\delta) \), the application of the delta-clause \( \text{DC} = \delta(\text{C}) = \text{modifies C} \cdots \{\cdots\} \) to the class definition \( \text{CD} = \text{CT}(\text{C}) \), denoted by \( \text{APPLY}(\text{DC}, \text{CD}) \), is the class definition \( \text{CD}' \) defined as follows:

\[
\text{CD}'(\text{extends}) = \begin{cases} 
\text{CD}(\text{extends}) & \text{if } \text{DC}(\text{extending}) = \emptyset \\
\text{C}' & \text{if } \text{DC}(\text{extending}) = \{\text{C}'\}
\end{cases}
\]

\[
\text{CD}'(\text{constructor}) = \begin{cases} 
\text{CD}(\text{constructor}) & \text{if } \text{DC}(\text{constructor}) = \emptyset \\
\text{K} & \text{if } \text{DC}(\text{constructor}) = \{\text{K}\}
\end{cases}
\]

\[
\text{CD}'(\text{a}) = \begin{cases} 
\text{AD} & \text{if } \text{DC}(\text{a}) = \text{adds AD} \\
\text{CD}(\text{a}') & \text{if } \text{DC}(\text{a}) = \text{renames a' to a}
\end{cases}
\]

Note that, for every class table \( \text{CT} \), if the delta-module \( \delta \) is applicable to \( \text{namespace}(\text{CT}) \), then it holds that \( \text{dom}(\text{APPLY}(\delta, \text{CT})) = \text{dom}(\text{APPLY}(\delta, \text{namespace}(\text{CT}))) \), and for every \( \text{C} \in \text{dom}(\text{APPLY}(\delta, \text{CT})) \), \( \text{namespace}(\text{APPLY}(\delta, \text{CT})(\text{C})) = \text{APPLY}(\delta, \text{namespace}(\text{CT}))(\text{C}) \).
Proposition 2. For every class table \( CT \), if the delta-modules \( \delta_1, \cdots, \delta_n \) \( (n \geq 0) \) are compatible and applicable to \( \text{namespace}(CT) \), then \( \text{APPLY}(\delta_1 \cdots \delta_n, CT) = \text{APPLY}(\delta_1, \cdots, \delta_n, CT) \), for every permutation \( i_1, \ldots, i_n \) of \( 1, \ldots, n \).

Thus, a well-formed FJ SPL defines a mapping from each feature configuration \( \overline{\psi} \) in \( \text{CONFIGURATIONS} \) to the class table obtained by applying the delta modules \( \text{DELTAS}(\overline{\psi}) \) to the class table of the core module according to the after partial order. We write \( CT_{\overline{\psi}} \) to denote the class table generated for the feature configuration \( \overline{\psi} \) and write \( \prec \overline{\psi} \) and \( a\text{Def}_{\overline{\psi}} \) to denote the subtype relation and the field/method lookup function associated to the class table \( CT_{\overline{\psi}} \), respectively (see Section 4). Note that, for every well-formed FJ SPL and for every feature configuration \( \overline{\psi} \) in \( \text{CONFIGURATIONS} \), it holds that \( \text{namespace}(CT_{\overline{\psi}}) = \text{CNT}_{\overline{\psi}} \).

5.4 Type-Safe, Redundant-Def-Free and Type-Uniform Well-Formed FJ SPL

A SPL is type-safe if all its products are well-typed programs. Formally, the notion of type-safe FJ SPL is as follows (we refer to [10] for the notion of well-typed FJ programs).

Definition 2. A well-formed FJ SPL is type-safe iff for every \( \{ \overline{\psi} \} \) \( \in \text{CONFIGURATIONS} \), the class table \( CT_{\overline{\psi}} \) represents a well-typed FJ program.

Type-safety of a FJ SPL can be checked by generating and typechecking each product according the FJ type system. In Section 6, we present a technique for checking whether a FJ SPL is type-safe without having to generate all products.

A FJ SPL \( L \) is redundant-def-free if it does not contain unused parts. This means that every class definition \( CD \), extending clause, constructor definition and field/method definition contained in any delta module of \( L \) is used in at least one product. For example, both the variants of the bank account SPL illustrated in Section 3 are redundant-def-free. This property of a FJ SPL can be checked only using the class namespace tables (by including the extending and constructor keywords in the class namespaces) without generating the products. Formally, the notion of redundant-def-free FJ SPL is defined as follows.

Definition 3. A well-formed FJ SPL is redundant-def-free iff for every delta-module name \( \delta \) and every class name \( C \)

- \( C \in \text{addsDom}(\delta) \) implies that
  - \( CT_{\overline{\psi}}(C)(\text{extends}) = \delta(C)(\text{extends}) \) (for some \( \overline{\psi} \in \text{CONFIGURATIONS} \)),
  - \( CT_{\overline{\psi}}(C)(\text{constructor}) = \delta(C)(\text{constructor}) \) (for some \( \overline{\psi} \in \text{CONFIGURATIONS} \))
  - for all \( a \in \text{dom}(\delta(C)) \) there exists \( \overline{\psi} \in \text{CONFIGURATIONS} \) such that \( CT_{\overline{\psi}}(C)(a) = \delta(C)(a) \).

- \( C \in \text{modifiesDom}(\delta) \) implies that
  - if \( \delta(C)(\text{extending}) \neq 0 \), then \( \{CT_{\overline{\psi}}(C)(\text{extends})\} = \delta(C)(\text{extends}) \) (for some \( \overline{\psi} \in \text{CONFIGURATIONS} \)),
  - if \( \delta(C)(\text{constructor}) \neq 0 \), then \( \{CT_{\overline{\psi}}(C)(\text{constructor})\} = \delta(C)(\text{constructor}) \) (for some \( \overline{\psi} \in \text{CONFIGURATIONS} \))
  - for all \( a \in \text{addsDom}(\delta(C)) \) there exists \( \overline{\psi} \in \text{CONFIGURATIONS} \) such that \( CT_{\overline{\psi}}(C)(a) = \delta(C)(a) \).
We say that a SPL is type-uniform to mean that (i) if there are two (or more) products containing a class of name \(C\) with a field/method of name \(a\), then the type of \(a\) must be the same; and (ii) if a class \(C\) is a subtype of a class \(D\) in some product, then there are no products where a class \(D\) is a subtype of a class \(C\). Type-uniformity enforces a communality in the family of the products that helps the design of the SPL. For example, both bank account SPLs shown in Section 3 are type-uniform. However, if a class \(Investor\) extending the class \(Client\) was added to the product with the Retirement and With Holder features, and the product with the Investment and With Holder features was modified by making the class \(Investor\) a subclass of the class \(Investor\), this would break type-uniformity. Formally, the notion of type-uniform FΔJ SPL is defined as follows.

**Definition 4.** A well-formed FΔJ SPL is type-uniform iff, for every pair of products \(CT_\varphi\) and \(CT_\psi\) it holds that:

- for all \(C \in \text{dom}(CT_\varphi) \cap \text{dom}(CT_\psi)\) and for all field/method \(a\), if both \(a\text{Def}_\varphi(C, a)\) and \(a\text{Def}_\psi(C, a)\) are defined, then \(\text{signature}(a\text{Def}_\varphi(C, a)) = \text{signature}(a\text{Def}_\psi(C, a))\), and
- for all \(C, C' \in \text{dom}(CT_\varphi) \cap \text{dom}(CT_\psi)\), if \(C \neq C'\) and \(C \prec \varphi C\) then \(C' \neq \psi C\).

The type-uniformity of a well-formed FΔJ SPL can be checked, without generating the products, by relying on the notions of class signature and class signature table. A class signature \(CS\) is a mapping from the keyword extends to a class name, from the keyword constructor to a class constructor header, from field names to class names and from method names to method types. A class signature table \(CST\) is a mapping from class names to class signatures. The signature of a class definition \(CD\) is denoted by \(\text{signature}(CD)\). The class signature table of the signatures of the classes in the class table \(CT\) is denoted by \(\text{signature}(CT)\). We write \(\text{signature}(C)\) as short of \(\text{signature}(CT(C))\).

The class signature tables of the products of a well-formed FΔJ SPL are generated (without generating the products) by applying the transformations described by the delta modules \(\text{DELTAS}(\varphi)\) to the class signature table of the core module according to the after partial order, for each feature configuration \(\varphi\) in \text{CONFIGURATIONS}. The formal definition for the application of a delta module to a class signature table can be straightforwardly obtained by mimicking the definition for the application of a delta module to a class table. We write \(CST_\varphi\) to denote the class signature table \(\text{signature}(CT_\varphi)\) of the product for configuration \(\varphi\). The subtyping relation \(\prec\) can be read off from the class signature table \(CST_\varphi\) (so that it is possible to check whether there are no cycles in the transitive closure of the extends relation). The lookup of the type of the field/method \(a\) in class signature in the class \(C\) is denoted by \(a\text{Type}(C)(a)\). Formally, for every class \(C\) in \(\text{dom}(CST)\), the function \(a\text{Type}(C)\) is defined as follows:

\[
a\text{Type}(C)(a) = \begin{cases} 
\text{CST}(C)(a) & \text{if } a \in \text{dom}(\text{CST}(C)) \\
\text{aType}(\text{D})(a) & \text{if } a \notin \text{dom}(\text{CST}(C)) \text{ and } \text{CST}(C)(\text{extends}) = \text{D}
\end{cases}
\]

### 6 A Type System for FΔJ

In this section, we present how to check that a type-uniform FΔJ SPL \(L\) is type safe without generating all products. We present a constraint-based type system for typechecking the core module with respect to the aggregate class signature table \(ACST_\ell\) of \(L\) (capturing the signature information for all products of \(L\)) in order to infer a set of class constraints.
hasPCP($\mathcal{C}, \mathcal{I}$) is used to ensure that

### 6.1 Aggregate Class Signature Tables

The class signature tables of all the products of a well-formed FJ SPL can be generated without generating the products (see the explanation at the end of Section 5.4). By inspecting the class signature table $\text{CST}_\mathcal{F}$ it is possible to check, for every class $\mathcal{C}$ in $\text{dom}(\text{CST}_\mathcal{F})$, whether the names of the fields defined in $\mathcal{C}$ are distinct from the names of the fields inherited from its superclasses and whether the formal parameters of the constructor of $\mathcal{C}$ correspond to the (defined or inherited) fields of $\mathcal{C}$ (as required by FJ, see Section 4). In the following we assume that the class signature tables of all the products satisfy these conditions.

An aggregate class signature $\text{ACS}$ is a mapping from the keyword extends to a finite non-empty set of class names, from the keyword constructor to a finite non-empty set of constructor headers, from field names to class names and from method names to method types. An aggregate class signature table $\text{ACST}$ is a mapping from class names to aggregate class signatures such that the subtyping relation $\prec_{\text{ACST}}$ induced by the extends clauses in $\text{ACST}$ is acyclic. The aggregation of the class signature tables of all the products of a type-uniform SPL $L$, denoted by $\text{ACST}_L$, is the aggregate class signature defined as follows:

$$\text{ACST}_L(\mathcal{C}) = \text{aggregate}(\{\text{CST}_\mathcal{F}(\mathcal{C}) \mid \mathcal{F} \in \text{CONFIGURATIONS and } \mathcal{C} \in \text{dom(\text{CST}_\mathcal{F})}\})$$

where the aggregation $\text{aggregate}(\{\mathcal{CS}_1, \ldots, \mathcal{CS}_k\})$ of a set of class signatures $\{\mathcal{CS}_1, \ldots, \mathcal{CS}_k\}$ ($k \geq 1$), is the aggregate class signature $\mathcal{CS}$ defined as follows:

- $\mathcal{CS}($extends$) = \bigcup_{i \in \{1, \ldots, k\}} \{\mathcal{CS}_i($extends$)\}$
- $\mathcal{CS}($constructor$) = \bigcup_{i \in \{1, \ldots, k\}} \{\mathcal{CS}_i($constructor$)\}$
- $\mathcal{CS}(a) = \mathcal{CS}_h(a)$ for any $h \in \{j \mid j \in \{1, \ldots, k\}$ and $a \in \text{dom(\mathcal{CS}_j)}\}$

The type-uniformity of the SPL $L$ ensures that $\text{ACST}_L$ is an aggregate class signature table. Note that $\text{dom}(\text{ACST}_L) = \bigcup_{\mathcal{F} \in \text{CONFIGURATIONS}} \text{dom}(\text{CT}_\mathcal{F})$ and $\text{dom}(\text{aggregate}(\{\mathcal{CS}_1, \ldots, \mathcal{CS}_k\})) = \bigcup_{i \in \{1, \ldots, k\}} \text{dom}(\mathcal{CS}_i)$.

We write $\text{aType}_L(\mathcal{C})(a)$ to denote the type of the field/method $a$ in class $\mathcal{C}$ according to $\text{ACST}_L$. Formally, for every class $\mathcal{C} \in \text{dom(\text{ACST}_L)}$, the lookup function $\text{aType}_L(\mathcal{C})$ is defined as follows:

$$\text{aType}_L(\mathcal{C})(a) = \begin{cases} \text{ACST}_L(\mathcal{C})(a) & \text{if } a \in \text{dom(\text{ACST}_L(\mathcal{C}))} \\ A & \text{if } a \notin \text{dom(\text{ACST}_L(\mathcal{C}))} \\ \text{and } A = \text{aType}_L(\mathcal{D})(a) & \text{for some } \mathcal{D} \in \text{ACST}_L(\mathcal{C})(\text{extends}) \end{cases}$$

### 6.2 Basic Constraints and Basic Constraint Checking Rules

The syntax of basic constraints, together with an informal explanation of their meaning, is given in Figure 4. A constraint of the form $\text{hasPCP}(\mathcal{C}, \mathcal{I})$ is used to ensure that
Constraint checking:

- `hasField(C, f)` means class `C` must define or inherit field `f`.
- `hasMethod(C, m)` means class `C` must define or inherit method `m`.
- `subtype(C, D)` means `C` must be a subtype of `D`.
- `hasCAT(C, C_1, ..., C_n)` means class `C` must have a Constructor accepting `n` Arguments of Type `C_1, ..., C_n`.
- `cast(C, D)` means `D` must be castable to type `C`.
- `hasPCP(C, f)` means the Parent class of `C` must have a Constructor with Parameters `f`.

The rules for checking the satisfaction of a set of basic constraints with respect to the class signature table of a product `CST_ϕ` are given in Figure 5. Most of the rules are self-explanatory. Note that, as in the type system for FJ (see [10]), there are three rules for type casts, corresponding to `upcast` (when the subject is a subtype of the target), `downcast` (when the target is a supertype of the subject) and `stupid cast` (when subject and target are unrelated), respectively. We say that a set of basic constraints `B` is cast safe with respect to a class signature table `CST` to mean that the judgement `CST_ϕ |= B` has been established without downcasts or stupid casts (uses of rules with a `downcast` or `stupid cast` premise).

### 6.3 Typing Rules for the Core Module and their Correctness/Completeness

In order to support the application of a set of constraints `B` inferred for a delta module to the set of constraints `C` inferred for the core module (thus making it possible to generate the constraints for a product without having to generate the product itself), the typing rules for the core module organize the inferred constraints in a two level hierarchy, corresponding to the structure of the class table of the core module. Namely: (i) each typing rule infers a set of class constraints (one for each class definition in the core module); (ii) each class constraint consists of the name of the subject class `C` and of a set containing a basic constraint of the form `hasPCP(C, ...)` (inferred for the constructor of the class) and method constraints inferred for the methods defined in the class; and (iii) each method constraint consists of the name of the subject method and of the set of basic constraints.

---

**Fig. 4.** FΔJ: Syntax and (informal) meaning of basic constraints

\[
\text{Constraint checking:}
\begin{align*}
\forall f \in \text{dom}(\text{aType}_ϕ(C)) & \quad \text{CST}_ϕ \models \text{hasField}(C, f) \\
\forall m \in \text{dom}(\text{aMethod}_ϕ(C)) & \quad \text{CST}_ϕ \models \text{hasMethod}(C, m) \\
\forall C \prec ϕ D & \quad \text{CST}_ϕ\models \text{hasCAT}(C, C_1, ..., C_n) \\
C \prec ϕ D & \quad \text{CST}_ϕ\models \text{hasPCP}(C, f)
\end{align*}
\]

**Fig. 5.** FΔJ: Checking rules for satisfaction of basic constraints
Method constraints:

\[ m \text{ with } \mathcal{B} \]

method \( m \) enforces the basic constraints \( \mathcal{B} \)

(where \( \mathcal{B} \) is a set of basic constraints not containing \( \text{hasPCP}(\cdots, \cdots) \))

Class constraints:

\[ \mathcal{C} \text{ with } \mathcal{K} \]

class \( \mathcal{C} \) enforces the constraints \( \mathcal{K} \)

(where \( \mathcal{K} \) is a set containing a constraint \( \text{hasPCP}(\mathcal{C}, \cdots) \)

and some method constraints)

Fig. 6. F\( \Delta \)J: Syntax of class constraints

inferred for the body of the subject method. Thus, a set of class constraints \( \mathcal{C} \) can be understood as a function from class names to class-constraints, and a class-constraint can be understood as a function from method names to method-constraints and from the keyword constructor to a basic constraint of the form \( \text{hasPCP}(\mathcal{C}, \cdots) \). The syntax of class constraints and method constraints is given in Figure 6. The constraint-based typing rules for FJ expressions, methods and classes and for the core module are given in Figure 7.

The hierarchical organization on a per-class and per-method basis of the constraints associated to a product is immaterial for the purpose of checking constraint satisfaction. The flattening function \( \text{FLAT} \), defined below, transforms a set of class-constraints \( \mathcal{C} \) into a set of basic constraints \( \text{FLAT}(\mathcal{C}) \).

\[
\text{FLAT}(\bigcup_{i \in \{1, \ldots, n\}} \{ \mathcal{C}_i \text{ with } \mathcal{X}_i \}) = \bigcup_{i \in \{1, \ldots, n\}} \text{FLAT}(\mathcal{X}_i)
\]

\[
\text{FLAT}(\{ \text{hasPCP}(\mathcal{C}, \bar{x}), m_1 \text{ with } \mathcal{B}_1, \ldots, m_n \text{ with } \mathcal{B}_n \}) = \{ \text{hasPCP}(\mathcal{C}, \bar{x}) \} \cup \bigcup_{i \in \{1, \ldots, n\}} \mathcal{B}_i
\]

We could use the FJ typing rules (see [10]) to check whether the core module is a well-typed FJ program. However (as we will see in Section 6.4), the constraint-based type system makes it possible to check the well-typedness of any other products without having to reanalyze the code that comes from the core module. The following theorem states that the constraint-based typing rules for the core module are correct and complete with respect to the FJ type system (we refer to [10] for the notions of well-typed FJ program and cast-safe FJ program).

**Theorem 1** (Correctness and Completeness of the F\( \Delta \)J typing rules for the core module). Let \( L \) be a type uniform F\( \Delta \)J SPL.

1. If \( \vdash \text{core } \mathcal{X} \{ \cdots \} : \mathcal{Y} \vdash \text{CST}_Y \models \text{FLAT}(\mathcal{X}) \), then the core product \( \text{CT}_Y \) is
   - a well-typed FJ program, and
   - if \( \text{FLAT}(\mathcal{X}) \) is cast safe with respect to \( \text{CST}_Y \), then \( \text{CT}_Y \) is cast safe.
2. If either the core module \( \text{core } \mathcal{X} \{ \cdots \} \) is not \( \vdash \)-typable or \( \text{CST}_Y \not\models \text{FLAT}(\mathcal{X}) \), then the core product \( \text{CT}_Y \) is not a well-typed FJ program.

6.4 Typing Rules for Delta Modules and their Correctness/Completeness

Also, the typing rules for delta modules organize the constraints inferred for a delta module in a two level hierarchy corresponding to the structure of the delta module in order to support the application of a set of constraints \( \mathcal{D} \) inferred for a delta module to the set of constraints \( \mathcal{C} \) inferred for the core module. Namely: (i) the typing rules infer a set of delta removes/adds/modifies-clause constraints (one for each delta clause in the delta module);
Expression typing:

\[ \Gamma \vdash x : \Gamma(x) \mid \emptyset \quad (\text{T-VAR}) \]

\[ \Gamma \vdash e : C \mid \mathcal{R} \quad aType_L(C)(\bar{f}) = D \quad (\text{T-FIELD}) \]

\[ \Gamma \vdash e_0 : C_0 \mid \mathcal{R}_0 \quad \forall i \in 1..n, \quad \Gamma \vdash e_i : C_i \mid \mathcal{R}_i \quad aType_L(C)(\bar{m}) = D_1 \cdots D_n \rightarrow C \quad (\text{T-INVK}) \]

\[ \Gamma \vdash e.m(e_1, \ldots, e_n) : C \mid \{ \text{hasField}(C, \bar{f}) \} \cup \mathcal{R} \quad (\text{T-NEW}) \]

\[ \Gamma \vdash e : D \mid \mathcal{R} \quad \Gamma \vdash (C)e : C \mid \{ \text{cast}(C, D) \} \cup \mathcal{R} \quad (\text{T-CAST}) \]

Method definition typing:

\[ \text{this} : C, \bar{x} : \mathcal{D} \vdash e : E \mid \mathcal{R} \quad \text{ACST}_L(C)(\bar{m}) = aType_L(C)(\bar{m}) = D \rightarrow D \quad (\text{T-METHOD}) \]

Class definition typing:

\[ D \in \text{ACST}_L(C) \quad \text{extends} \quad K = C(\bar{\mathcal{B}} \bar{\mathcal{I}} \bar{\mathcal{g}}) \{ \text{super}(\bar{f}) ; \text{this} \bar{g} = \bar{g} ; \} \]

\[ \forall i \in 1..q, \quad \text{this} : C \vdash \mathcal{M}D_i : \{ m_i \text{ with } \mathcal{R}_i \} \quad (\text{T-CLASS}) \]

Core module typing:

\[ \forall i \in 1..n, \quad \vdash \mathcal{C}D_i : \{ C_i \text{ with } \mathcal{X}_i \} \quad (\text{T-CORE}) \]

(ii) a delta removes-clause constraint is a removes-clause removes \( C \), a delta adds-clause constraint consists of the keyword adds followed by a class constraint (defined in Section 6.3), and each delta modifies-clause constraint consists of the name of the subject class \( C \) and of a set possibly containing a basic constraint of the form \( \text{hasPCP}(C, \cdots) \) (inferred for the constructor of the class) and some delta removes/renames/adds-subclause constraints; and (iii) a delta remove-subclause constraint is a delta subclause of the shape \( \text{removes} \ m \), a renames-delta-subclause is a delta subclause of the shape \( \text{renames} \ m \rightarrow m' \) and a delta adds-subclause constraint consists of the keyword adds followed by a method constraint (defined in Section 6.3). Thus, a set of delta clause constraints \( \mathcal{D} \) can be understood as a function from method names to delta subclause constraints and from the keyword constructor to either a singleton set (containing a basic constraint of the form \( \text{hasPCP}(C, \cdots) \)) or the empty set. The syntax of the delta clause constraints is given in Figure 8.

For each feature configuration \( \mathcal{F} \) in CONFIGURATIONS, the class constraints \( \mathcal{C}_\mathcal{F} \) for the product \( \mathcal{C} \mathcal{T}_\mathcal{F} \) are generated (without generating the products) by applying the sets of delta clause-constraints inferred for the delta modules \( \text{DELTAS}(\mathcal{F}) \) to the class constraints of the core module according to the after partial order. The result of the application of a set of delta clause constraints \( \mathcal{D} \) to a set of class constraints \( \mathcal{C} \), denoted by \( \text{APPLY}(\mathcal{D}, \mathcal{C}) \),
Delta subclause-constraints:

- **adds m with B**
  
  add constraint “method m enforces the basic constraints B”

- **removes m**
  
  remove constraint “m with …”

- **renames m to m’**
  
  change constraint “m with B” into “m’ with B”

Delta clause-constraints:

- **adds C with K**
  
  add constraint “class C enforces the constraints K”

- **removes C**
  
  remove constraint “C with …”

- **modifies C with M**
  
  change constraint “C with K” into “APPLY(D(C), C with K)”
  
  (where M is a set possibly containing hasPCP(C, ···) and some delta subclause constraints)

**Fig. 8. FΔJ: Syntax of delta clause constraints**

is the set of class constraints C’ defined as follows:

\[
C'(C) = \begin{cases} 
C(C) & \text{if } C \notin \text{dom}(D) \\
C \text{ with } K & \text{if } C \notin \text{dom}(E) \text{ and adds } C \text{ with } K \in D \\
\text{APPLY}(D(C), C(C)) & \text{if modifies } C \cdots \in D
\end{cases}
\]

where the application of the delta modifies-clause constraint \( dcc = C \text{ with } M = D(C) \) to the class-constraint \( cc = C \text{ with } K = C(C) \), denoted by \( \text{APPLY}(dcc, cc) \), is the class-constraint \( cc' = C \text{ with } K' \) defined as follows:

\[
cc'(\text{constructor}) = \begin{cases} 
cc(\text{constructor}) & \text{if } dcc(\text{constructor}) = \emptyset \\
C(\bar{C} \bar{f}) & \text{if } dcc(\text{constructor}) = \{C(\bar{C} \bar{f})\}
\end{cases}
\]

\[
cc'(m) = \begin{cases} 
cc(m) & \text{if removes } m \cdots \notin dcc \text{ and renames } m \cdots \notin dcc \\
m' \text{ with } B & \text{if } cc(m) = m \text{ with } B \text{ and renames } m \text{ to } m' \in dcc
\end{cases}
\]

The constraint-based typing rules for FΔJ delta subclauses, delta clauses and delta modules are given in Figure 9. Most of the rules are self explanatory. The rules (T-S-ADD-M) and (T-C-ADD-C) rely on the rules (T-METHOD) and (T-CLASS) in Figure 7, respectively. The rule (T-C-MOD-C) is presented with “optional parts” (enclosed in square brackets) to cope with the fact that the extending and constructor parts of a delta modifies-clause are optional.

The constraint-based type system checks the well-typedness of all possible products (without generating them) by: analyzing the core module and each of the delta modules in isolation (by relying on the aggregate class signature table), generating the constraints associated to the products, and checking the constraints associated to each product with respect to the class signature table of the product. The following theorem states that the constraint-based typed system is correct and complete with respect to the FJ type system. Note that completeness require redundant-def-freedom, since a type error occurring in an unused part of the SPL would not affect the well-typedness of any product.

**Theorem 2 (Correctness and Completeness of the FΔJ type system).** Let \( L \) be a type-uniform FΔJ SPL.

1. If \( \vdash \text{core } \emptyset \{ \cdots \} : \text{FLAT}(\emptyset) \) and \( \vdash \text{delta } \delta \cdots : D_\delta \) (for all \( \delta \in \text{dom}(DMT) \)), then for every \( \emptyset \in \text{CONFIGURATIONS} \), if \( \text{CST}_\emptyset \models \text{FLAT}(\emptyset) \) then the FJ program \( \text{CT}_\emptyset \)
Delta-subclause typing:
\[
\begin{align*}
\text{ACST}_{\text{L}}(C)(f) = D \\
\forall \Delta \in \text{DEL}\text{TAS}(\overline{\phi}) \quad \text{this} : C \vdash \text{adds} Df : \emptyset \\
\text{this} : C \vdash \text{MD} : \{\text{DDF}\} & \quad (\text{T-DeltaF}) \\
\text{this} : C \vdash \text{MD} : \{\text{DDF}\} & \quad (\text{T-DeltaM})
\end{align*}
\]

Delta-clause typing:
\[
\begin{align*}
\vdash \text{CD} : \{C \text{ with } \phi\} \\
\vdash \text{adds} \text{CD} : \{\text{adds} C \text{ with } \phi\} & \quad (\text{T-ADD}C) \\
\text{this} : C \vdash \text{removes} \text{CD} & \quad (\text{T-REMC})
\end{align*}
\]

\[
\forall i \in 1..q, \text{this} : C \vdash \text{DSC}_i : \{\text{dscc}_i\} \\
\vdash \text{modifies} C \{\text{extending} D\} & \quad (\text{T-MOD}C)
\]

Delta-module typing:
\[
\forall i \in 1..n, \vdash \text{DC}_i : \{\text{dccc}_i\} \\
\vdash \text{delta} \delta : \{\text{dccc}_1 \cup \ldots \cup \text{dccc}_n\} & \quad (\text{T-DELTA})
\]

\[\text{Fig. 9. FAJ: Typing rules for delta subclauses, delta clauses and delta modules w.r.t. ACST}_{\text{L}}\]

1. If \(\text{FLAT}(\overline{\phi})\) is not \(\vdash\)-typable or \(\text{CST}_{\overline{\phi}}\) is not well-typed.
2. \(\text{CST}_{\overline{\phi}}\) is not well-typed.
3. \(\text{FLAT}(\overline{\phi})\) is not \(\vdash\)-typable or \(\text{CST}_{\overline{\phi}} \neq \text{FLAT}(\overline{\phi})\), then the FJ program \(\text{CST}_{\overline{\phi}}\) is not well-typed.

7 Related Work

The approaches to implementing the variability of SPL in the object-oriented paradigm can be classified into two main directions [15]. First, annotative approaches, such as conditional compilation, frames [3] or COLORED FEATHERWEIGHT JAVA (CFJ) [13], mark the source code of the whole SPL with respect to product features on a syntactic level. For a particular feature configuration, marked code is removed. Second, compositional approaches, such as the approach presented in this work, associate code fragments to product features that are assembled to implement a particular feature configuration. In [17], general program modularization techniques, such as aspects [14], mixins [23] or traits [9, 18], are evaluated with respect to their ability to implement features. Although the above approaches are suitable to express feature-based variability, they do not contain designated linguistic concepts for representing features.

The approach which is closest to the presented work is feature-oriented programming (FOP) [5, 25, 2, 8]. In FOP, product features are explicitly represented by a linguistic construct called feature module. Feature modules can introduce new classes or refine existing ones by adding fields and methods or by overriding existing methods. In order to obtain an implementation for a feature configuration, the feature modules are composed incrementally by carrying out the specified class refinements. Because feature modules can only
specify extensions of classes, development always starts from the mandatory features of all products. In contrast, delta modules support additions, modifications and removals of classes which allows to choose any valid product as the core module and facilitates flexible product line design starting from different core products. Feature composition in FOP is carried out in a linear order because a feature that refines a class can be added only after the class to be refined has been introduced. If an already modified method is modified by a later composed feature module, the first modifications are overridden. For delta modules, the partial order only captures essential dependencies to ensure that modified entities exist. Furthermore, it allows explicitly specifying the ordering in which conflicting modifications are carried out. A feature module in FOP is associated to a single product feature. Instead, complex application conditions over the product features can be attached to delta modules such that combinations of features can be handled explicitly. In this way, the optional feature problem [16] is solved by the flexible nature of delta modules (cf. Section 3).

The calculi Featherweight Feature Java (FFJ) [2] and Lightweight Feature Java (LFJ) [8] aim at a formalization of feature-oriented programming [5] with static guarantees by means of a type system. FFJ is based on FJ and extends it with class refinements, but the notion of feature does not appear in the syntax of the language. FFJ has a type system to check product specifications which is an extension of the FJ type system. It requires the generation of all products belonging to the product line in order to ensure their well-typedness.

LFJ extends LJ (Lightweight Java) [24] by a feature module construct representing class refinements of FOP. LFJ introduces a constraint-based type system that supports the type checking of feature modules in isolation and allows for compositional type checking, to guarantee that all products in a SPL are type correct without having to generate them. The type system infers a signature for each feature module which is a set of requirements to be satisfied by a composition including this feature. By the soundness of the LFJ type system, the satisfaction of each feature signature is enough to ensure that a composition of feature modules is a well-formed program with respect to LJ. Each signature can be translated to a propositional formula, whose conjunction ϕsafe specifies the requirements for all correct programs. The validity of the formula \( FM \land WF_{Spec} \rightarrow ϕ_{safe} \) endows that the set of programs satisfying the formula \( FM \), a logical representation of the feature model (according to [4]), and the formula \( WF_{Spec} \) are type correct. \( WF_{Spec} \) encodes the constraints on the precedence and on the subtyping relation which guarantee that the programs can be obtained by feature module composition. With the formula \( WF_{Spec} \), the LFJ type system mirrors the requirement of the FOP approach that a suitable linear composition ordering has to be provided. However, a feature model describes the possible products of a product line only by sets of features without any ordering. In this respect, the approach presented in this paper is closer to the variability expressed by feature models, because the constraints inferred from the aggregate signature table ACST (defined in Section 6) do not impose an ordering on delta application. The essential dependencies between delta modules are specified with the after clauses inducing a partial order. Although, this order could also be inferred automatically as in LFJ, the after clauses allow the programmer to explicitly control the order of delta application.
8 Conclusions and Future Work

In this paper, we presented a novel linguistic approach to implementing feature-based variability of software product lines within the object-oriented paradigm. The presented programming language combines the ability to represent flexibly feature-based variability by core and delta modules with means to effectively provide static guarantees on the resulting products by a constraint-based type system. This type system is designed to check type uniform SPLs since type uniformity is a valuable property for SPL implementations. The FJ constraint-based type system can also be adapted to deal with well-formed non-type uniform SPLs. The idea is to drop the notion of aggregate class signature tables and to enrich the syntax of the constraints to include variables (along the line of the constraint-base system in [1]). These variables are instantiated to class names when checking the constraints inferred for a product with respect to the class signature table of the product.

An implementation of the programming language presented in this paper is currently being developed. In addition, we want to provide an IDE to support the programmer with a global view on the feature model and the relationships between deltas modules for effective software product line development. Furthermore, we aim at investigating the impact of selecting the core product on the resulting product line implementation. We want to provide guidelines for product line development with core and delta modules and integrate it into an model-based development process [22].

The general concept of a delta modules is not bound to a programming language. We have instantiated it on FJ [10] to show the feasibility of our approach. For future work, we are aiming at using other languages for the underlying product implementations. A starting point is the trait-based calculus Featherweight Record-Trait Java (FRTJ) [6]. In FRTJ, classes are assembled from interfaces, records (providing fields) and traits (providing methods) that can be directly manipulated by record or trait composition or using operations such as renaming of fields and methods. The operations on interfaces, records and traits provided by FRTJ make it a good candidate for implementing delta modules in a very expressive way. Moreover, they can help improving code reuse, which is another goal in SPL engineering.

References

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