Constraints in Feature Algebra

Andreas Zelend

Institut für Informatik, Universität Augsburg, Germany
zelend@informatik.uni-augsburg.de

Abstract. Feature Algebra captures the commonalities of feature oriented software development (FOSD), such as introductions, refinements and quantification. So far constraints have not been integrated into this algebraic framework. They arise in feature elicitation, feature dependence, feature exclusion, feature interaction etc. This paper presents a possibility for integrating such constraints into Feature Algebra.

1 Introduction

The paradigm of *feature orientation* (FO) (e.g. [2]) provides formalisms, methods, languages, and tools for building variable, customisable and extensible software. Applications include network protocols [2], database systems and software product lines [6]. Informally, a *feature* reflects an increment in functionality or in the software development. A *product* is an executable collection of features.

Complex features are composed of simpler ones. Hence there may be constraints that further features need to be added to get a product. Other constraints might be that a product/program must satisfy a requirement given by a stakeholder, etc.

Abstract reasoning about FO can be done in *Feature Algebra* [1]. The algebra abstracts from the detailed definition of a feature, in particular, from concrete programming languages, and covers the common concepts of FO such as introductions, refinements and quantification. A standard model (instance) is based on *feature structure forests (FSFs)* that reflect the hierarchical structure of the features' implementations and hide details from a certain depth on.

In this paper we show how constraints can be introduced into Feature Algebra, even without using fundamentally new concepts: we define an enrichment that satisfies the axioms of Feature Algebra and allows the formulation of all common types of constraints. It is able to reflect whether constraints are met. The approach strongly builds upon results from [3] and [7].

2 Types of Constraints

There is a wide variety of constraints to be considered in FO. In this section we classify them by focusing on the formalization of low- and high-level constraints in Feature Algebra.

Low-level constraints are inferred from the code; they describe all constraints that are based on implementation details. Some are discussed w.r.t. jak in [10].

References describe all code-level constraints where a method, a class, etc. refers to another object. The code can neither be compiled nor executed when the referenced part is not included in the overall product. The same type of constraint occurs when program parts are **import**ed.

Refinements are similar to references. In jak, the keyword **refines** indicates that a feature builds on another. The Java keyword **extends** has the same effect.

Abstract Class Constraints and **Interface Constraints** are two other types presented in [10]. A concrete subclass **class** C of an **abstract class** or **interface** A must implement all inherited abstract methods. Features may introduce new classes inheriting from A or may introduce new abstract methods into A. If new descendants of A are introduced by a feature, only the new code has to be checked. Yet, if a feature introduces new abstract methods into the supertype A, all descendants of A now have a new implementation obligation.

We call references and refinements, which behave similarly, *structural dependences* and represent them by a special kind of abstract constraints in Section 3.

High-level constraints are *semantic dependences* that can only partially be inferred from the implementation. For example, they can be derived from the given domain model [5], are specified by a customer who wants a particular product or product line or are given as invariants. Others, sometimes called primitives, can be encoded in the corresponding feature model (e.g. [5,9]):

Mandatority means that a certain feature has to be present in a product. An example is given by a user requiring a toString()-method for each class.

Optionality refers to an entire product line. The optional feature F may be part of product P, whereas another product Q does not have F.

Alternative provides a choice from a given set of features. It can be seen as "exactly one of m different features".

Exclusion describes that two features are not allowed to be within the same product. For example, if product P has a 64-bit implementation of foo(), P is not allowed to have a 32-bit implementation of the same method.

Implication describes the contrary of an exclusion: a second feature is not forbidden, but required. For example, if P provides a method (feature) to allocate memory, another feature for deallocation has to be provided.

Requirements are similar to the reference constraints of the previous subsection and to implications. But this time the dependence is given by the feature model or the user. For example, a customer demands the implementation of a printer driver whenever a function print() is implemented.

3 Constraints in Feature Algebra

We briefly recapitulate the formal definitions of Feature Algebra [1]. A Feature Algebra comprises a set I of *introductions* that abstractly represent features and a set M of *modifications* that allow changing the introductions. The central operations are feature addition $+$, application $\cdot$ of a modification to an introduction and modification composition $\circ$.

Definition 3.1 Formally, a *Feature Algebra* is a tuple $(M, I, +, \circ, \cdot, 0, 1)$ satisfying the following properties for $m, n \in M$ and $i, j \in I$. Addition $+$ is associative and commutative and satisfies the additional axiom of *distant idempotence*, i.e., $i + j + i = j + i$. This means that adding a feature that is already present has no effect. Moreover, 0 is required to be the neutral element of $+$. Modification composition $\circ$ is a binary inner operation on M and 1 is an element of M representing the identity modification. M operates via $\cdot : M \times I \to I$ on I, satisfying $(m \circ n) \cdot i = m \cdot (n \cdot i)$, $1 \cdot i = i$, $m \cdot (i + j) = (m \cdot i) + (m \cdot j)$ and $m \cdot 0 = 0$.

For additional properties that follow from these axioms, see e.g. [1,4]. We now present a fundamental notion for comparing features.

Definition 3.2 [1] The associative and distantly idempotent operation $+$ induces a *subsumption preorder* by $i \leq j \Leftrightarrow_{df} i + j = j$.

A relation $i \leq j$ means that all elementary features of i are contained in j.

Lemma 3.3 *The modification application $\cdot$ is isotone w.r.t $\leq$.*

Proof. Assume i, j with $i \leq j$ and a modification m. Then we have $i \leq j \Leftrightarrow i + j = j \Rightarrow m \cdot (i + j) = m \cdot j \Leftrightarrow m \cdot i + m \cdot j = m \cdot j \Leftrightarrow m \cdot i \leq m \cdot j$. □

```
//feature CONS
package util;

class List{
  ListNode first;

  void cons(ListNode n){
    n.next = first;
    first = n; }
}

class ListNode{
  ListNode next;
}
```

(a) Implementation of CONS

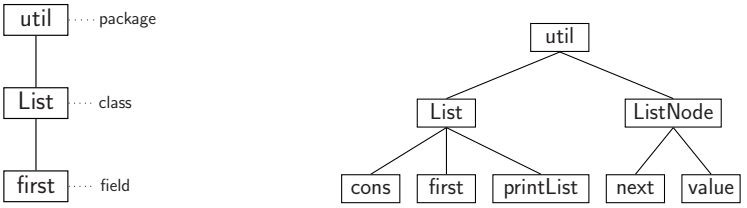
```
//feature PRINT_LIST
package util;

class List{

  public void printList(){
    ListNode iter = first;
    while (iter != null){
      System.out.print(iter.toString()+",");
      iter = iter.next; }
  }
}

class ListNode{
  Object value;
}
```

(b) Implementation of PRINT

Fig. 1: Features CONS and PRINT

The standard model of Feature Algebra uses *feature structure forests (FSFs)* as introductions. FSFs capture the essential hierarchical structure of a given system (e.g. [1]) and therefore contain less details than abstract syntax trees.

An example for a FSF is given in Fig. 2. It represents a feature PRINT, an extension of CONS. The feature offers a method to print the content of a list constructed using the method cons. The corresponding code is given in Fig. 1b. Formally, a *feature structure forest* is a labelled forest; the labels correspond to, e.g., directory names or packages and classes, while the leaves contain the components of the modules. The tree structure expresses the hierarchical structure in a language-independent way. It is well known that FSFs can be expressed by prefix-closed sets of tree paths (e.g. [4]).



(a) Dependence (b) Composition of CONS and PRINT

Fig. 3: Structural Dependence of Feature PRINT

In that model $\leq$ coincides with inclusion of path sets. In this standard model, addition + coincides with *tree superimposition* [8], i.e., recursive merging of FSFs. Each modification consists of a *query* that selects a subset of introductions and a *change function* that specifies how to modify the selected introductions. More details concerning this model can be found in [1].

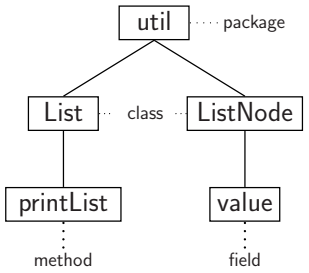


Fig. 2: FSF of PRINT

Now we discuss how constraints can be represented algebraically. The main idea is to use triples (i, d, c) consisting of an introduction i, a collection d of structural dependences and a condition c. Since structural dependences essentially have the same nature as introductions, both i and d are represented by elements of some Feature Algebra A. The component c is a predicate on the set of introductions I.

Definition 3.4 Let $A = (M, I, +, \circ, \cdot, 0, 1)$ be a Feature Algebra and $\mathcal{P}_I$ be the set of all predicates over the introductions I of A. A *design* over A is an element of $I \times I \times \mathcal{P}_I$. A design (i, d, c) *satisfies* its *dependence* d iff $d \leq i$. A design (i, d, c) *satisfies* its *condition* c iff $c(i) = \mathsf{true}$. A design that satisfies its dependence and its condition is called a *product*.

Definition 3.5 The *conjunction* of predicates $p, q \in \mathcal{P}_I$ is defined by $(p \wedge q)(i) =_{df} p(i) \wedge q(i)$ for all $i \in I$. The predicate that maps every element of I to true is denoted by **true**.

Let us look at an example. We will derive a design D_{pl} for the feature PRINT presented in Fig. 1. The introduction of PRINT (*impl*) is the FSF given in Fig. 2. Its structural dependence (*sdpl*) is the FSF given in Fig. 3a. The feature PRINT does not impose any condition, i.e., its condition is the constant predicate **true**. The design D_{pl} can now be defined as $(impl, sdpl, \mathbf{true})$. Obviously, *sdpl* is not contained in *impl*, i.e., $sdpl \not\subseteq impl$. Hence D_{pl} does not satisfy its dependence. In contrast, the composed FSF of PRINT and CONS (cf. Fig. 3b) satisfies *sdpl* , since it includes *sdpl* as a subtree. In the set-based representation this is expressed by $sdpl \subseteq \mathsf{CONS} + \mathsf{PRINT}$.

Using designs it is now easy to model the classes of constraints identified in Section 2. For most of them it is useful to assume a predicate $has(F)(i)$ that checks whether a feature F is included in a given Feature Algebra element $i \in I$. When F can be represented as an introduction f, this can be

expressed as the condition $has(f)(i) \Leftrightarrow_{df} f \leq i$. We now show how exclusions and implications can be formalised. Since the third component of designs is based on predicates, we freely use the logical connectives $\wedge, \vee, \Rightarrow, \neg$, etc. These operations are, like $\wedge$, defined by pointwise lifting. Feature exclusion is expressed as $has(F)(i) \Rightarrow \neg has(G)(i)$. Similarly, feature implication is expressed as $has(F)(i) \Rightarrow has(G)(i)$. Last, we have a look at abstract class constraints; interface constraints can be characterised in a similar way. These constraints have the form "If a **class** C extends the **class** D then it must provide a method foo()." This can be formalised as an implication: $extends_D(i) \Rightarrow has(\mathsf{foo()})(i)$.

We now make the set of designs over A into a Feature Algebra itself. Since designs are elements of a Cartesian product, the idea is to define the necessary operations componentwise. For the first two components we can simply use the corresponding operations from A. For the third component we have to decide how to define the sum and the set of modifications together with application and composition. Since the constraints of a combined design should be the joined constraints of the parts, it seems reasonable to use conjunction of predicates here; the neutral element is **true**. For modifying predicates, we can use the well-known concept of predicate transformers.

Definition 3.6 Assume an arbitrary set I, the set of predicates $\mathcal{P}_I$ over I and predicates $p, q \in \mathcal{P}_I$. By $\mathcal{PT}_I$ we denote the set of all *predicate transformers*. A predicate transformer t is *conjunctive* if $t(p \wedge q) = t(p) \wedge t(q)$ holds for all $p, q \in \mathcal{P}_I$. In particular id, $id(p) = p$ for all $p \in \mathcal{P}_I$, is conjunctive. The set of all conjunctive predicate transformers over A is denoted by $\mathcal{CT}_I$. A predicate p is called *stable* iff $p(i) \Rightarrow p(i + j)$ for all j.

Stable predicates are closed under conjunction and disjunction. A design with a stable condition can be composed with another design while the condition does not change its value. For example optionality and all $has(f)(i)$ conditions are stable.

Lemma 3.7 *For a set I of introductions the structure $(\mathcal{CT}_I, \mathcal{P}_I, \wedge, \circ, \cdot, \mathbf{true}, id)$ forms a Feature Algebra.*

Now the following result is immediate by universal algebra.

Theorem 3.8 *For a Feature Algebra $A = (M, I, +, \circ, \cdot, 0, 1)$ the structure $DES_A =_{df} (M \times M \times \mathcal{CT}_I, I \times I \times \mathcal{P}_I, +, \circ, \cdot, \mathbf{0}, \mathbf{1})$ forms a Feature Algebra of designs if $\mathbf{0} =_{df} (0, 0, \mathbf{true})$, $\mathbf{1} =_{df} (1, 1, id)$ and the operations as well as modifications are lifted pointwise.*

Lemma 3.9 *If the designs $f = (i, d, b)$ and $g = (j, e, c)$ are products and b and c are stable then the composition $f + g$ is also a product. Furthermore if f is a product and $(t \cdot b)(m \cdot i) = b(i)$ then $(m, m, t) \cdot f$ is a product.*

Proof. Since f and g are products, they both satisfy their dependences, i.e., $d \leq i$ and $e \leq j$. Since $+$ is isotone [4] and by transitivity of $\leq$, $d + e \leq i + j$ follows. Since b and c are stable we have $b(i) \wedge c(j) \Rightarrow b(i+j) \wedge c(i+j) \Leftrightarrow (b \wedge c)(i+j)$, i.e., $f + g$ satisfies its condition. In sum $(i + j, d + e, (b \wedge c)(i + j))$ is a product.

For the second part we have $(m, m, t) \cdot f = (m \cdot i,\ m \cdot d,\ t \cdot b)$. By Lemma 3.3 $m \cdot d \leq m \cdot i$ holds. The satisfaction of the condition is preserved by the additional assumption $(t \cdot b)(m \cdot i) = b(i)$. □

4 Conclusion

In this paper we have shown how constraints can be embedded into the abstract structure of Feature Algebra. This is important, since feature interactions are an immanent problem of FOSD, so that measures helping to handle them are requisite. Providing proper instruments for the formulation of constraints is one step in this direction, which is why we think that constraints are a major construct of FOSD and should be included in the abstract model. Future work will be directed towards representative case studies. They will help to further understand the structure of Feature Algebra and constraints.

Acknowledgements I thank P. Höfner and B. Möller who guided me in this direction. The research was partially sponsored by DFG, Project FeatureFoundation.

References

1. S. Apel, C. Lengauer, B. Möller, and C. Kästner. An algebraic foundation for automatic feature-based program synthesis. *Science of Computer Programming*, 75(11):1022–1047, 2010.
2. D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Trans. Softw. Engineering and Methodology*, 1(4):355–398, 1992.
3. P. Höfner, S. Mentl, B. Möller, W. Scholz, and A. Zelend. Constraints in feature algebra. Unpublished manuscript, 2012.
4. P. Höfner and B. Möller. An extension for feature algebra — Extended abstract. In *Workshop on Feature-Oriented Software Development (FOSD 09)*, pages 75–80. ACM Press, 2009.
5. 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 CMU/SEI-90-TR-21, Carnegie-Mellon University Software Engineering Institute, 1990.
6. R. Lopez-Herrejon and D. Batory. A standard problem for evaluating product-line methodologies. In J. Bosch, editor, *Conference on Generative and Component-Based Software Engineering (GCSE 01)*, volume 2186 of *LNCS*, pages 10–24. Springer, 2001.
7. S. Mentl. Requirements in feature algebra. Master's thesis, Institut für Informatik, Universität Augsburg, 2010.
8. H. Ossher and H. Harrison. Combination of inheritance hierarchies. In *Object-Oriented Programming, Systems, Languages, and Applications*, pages 25–40. ACM Press, 1992.
9. P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps. Generic semantics of feature diagrams. *Computer Networks*, 51:456–479, 2007.
10. S. Thaker, D. Batory, D. Kitchin, and W. R. Cook. Safe composition of product lines. In *Generative Programming and Component Engineering (GPCE 07))*, pages 95–104. ACM Press, 2007.