Requirements in Feature Algebra

Peter Höfner
NICTA, Sydney, Australia
peter.hoefner@nicta.com.au

Sven Mentl
sven.mentl@googlemail.com

Bernhard Möller
Universität Augsburg, Germany
moeller@informatik.uni-augsburg.de

Wolfgang Scholz
University of Passau, Germany
scholz@uni-passau.de

***Abstract*—Feature Algebra is intended to capture the commonalities of feature oriented software development (FOSD), such as introductions, refinements and quantification. It allows denoting systems composed of features by algebraic terms and transforming the systems by manipulating the terms using the laws of the algebra. The algebraic view abstracts from differences of minor importance and leads to more compact and effective reasoning. While the existing Feature Algebra covers most of the main aspects of FOSD, so far requirements have not been integrated into it. They naturally arise in connection with different aspects of feature orientation, such as feature elicitation, feature dependence, mutual feature exclusion and feature interaction. This paper presents a possibility for integrating requirements into Feature Algebra.**

I. Introduction

Over the last years *feature orientation* (FO) (e.g. [13], [14]) has been established in computer science as a general programming paradigm. It provides formalisms, methods, languages, and tools for building variable, customisable, and extensible software. In particular, FO summarises feature-oriented software development and feature-oriented programming. FO has widespread applications from network protocols [13] and data structures [15] to software product lines [31]. It arose from the idea of incremental software, i.e., every program, design or product can successively be built up by adding more and more features, like methods, documentations or properties. Roughly spoken, a *feature* reflects an increment in functionality or in the software development.

In the literature, there are different definitions of a feature. A common one describes a feature as "a structure that extends and modifies a given program in order to satisfy a stakeholder's requirement, to implement a design decision, and to offer a configuration option." [6]. Elementary features may be composed, even in a nested way, into more complex ones. Features may depend on each other: If a feature is added to a given program, other features may also be required and have to be added to get a proper (valid) program. These additional features are part of the requirements of a given feature. Other requirements might be that a product/program must have some property like a `toString()`-method or a requirement given by a stakeholder.

A feature itself can be viewed at different levels of abstraction. In the most concrete view, a feature consists of a bunch of artefacts, maybe even each in a different language (e.g., makefile, source code in Java and documentation in XML). Feature Algebra provides a more abstract view. It covers the common concepts of feature oriented software development (FOSD) such as introductions, refinements and quantification. In particular, it abstracts from differences of minor importance. The standard model (instance) of Feature Algebra is based on feature structure forests (FSFs). FSFs focus on the hierarchical structure of the features' implementations and hide details from a certain depth on. In [23], [24] FSFs have been encoded in lists and sets, so that the composition of two features (FSFs) can easily be determined.

So far, requirements are not covered directly by Feature Algebra. In this paper we show how they can be introduced into it, even without using fundamentally new concepts. In particular, we define a new model that satisfies the axioms of Feature Algebra and allows the formulation of all common types of requirements. It is able to reflect simple checks whether requirements are met. Hence, in some sense, this model can be seen as another contribution to a general theory of type systems for features (e.g. [2]).

On one hand this shows that Feature Algebra captures one more important trait of FO. On the other hand, having a solid mathematical foundation for requirements yields a better understanding of the theoretical concepts of FO.

As stated in our earlier papers on Feature Algebra [7], [24], the main achievement of Feature Algebra is to provide a single, mathematically sound foundation for the formulation and handling of additional information about software product lines, as it is, e.g., necessary to describe the admissible feature combinations. The integration of feature requirements adds to the details covered by the algebraic approach. Feature combinations no longer are either valid or invalid, but invalid combinations better reflect their conflicts. With this contribution we aim to extend Feature Algebra into a framework that is more widely applicable and better suited for automatic reasoning (and by that more able to deal with feature interactions) than previous versions.

II. Preliminaries

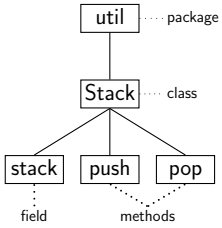
A. Feature Structure Forests

Feature structure forests (FSFs) capture the essential hierarchical module structure of a given system (e.g. [6]). An example is given in Figure 1b, where a simple Java class BaseStack is described as an FSF consisting of a single tree.

```
package util;

class Stack{
  Object[] stack;
  void push(Object o){
    stack[stackpointer++] = o; }
  Object pop(){
    return stack[stackpointer--]; }
}
```

(a) Implementation



(b) FSF

Fig. 1: Feature BaseStack

Each node of an FSF has a name and a kind[1]. For example, the **class** Stack is represented by the node Stack of kind **class** in Figure 1b. For the present paper we are mostly not interested in the kind information, hence we will skip this information when appropriate.

Formally, a *feature structure forest* is a labelled forest (collection of trees); the labels correspond to, e.g., directory names in a large structured system and the tree structure expresses their hierarchical interdependence.

In Java and in most other programming languages, the inner nodes correspond to modules like packages and classes, while the leaves contain the components of the modules. This structure is captured in a language-independent way by the branching structure of the trees in an FSF.

The transformation of a feature's implementation (real code) into a feature structure forest and hence towards an algebraic notion has been thoroughly investigated and is well known [6], [4], not least since feature structure forests are closely related to abstract syntax trees.

To combine features and FSFs the concept of tree superimposition was introduced [5], [32]. Superimposition is a composition technique that is able to deal with a wide range of artefacts, like source code, documentation or makefiles. Problems arise in the composition of leaf nodes if they represent the actual source code of the feature: it is still an open question whether implementations of the same method name should be merged, override one another, be inlined, etc.

B. The Problem

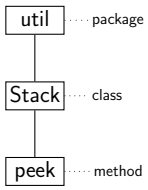
We now present a minimalistic example that shows our core ideas concerning requirements.

We assume that for some user the functionality provided by BaseStack is not sufficient and that we want another feature PeekStack that allows looking at the element on the top of the stack without popping it off. Figure 2 illustrates the implementation and the FSF of the feature PeekStack.

```
package util;

class Stack{
  public Object peek(){
    return stack[stackpointer]; }
}
```

(a) Implementation



(b) FSF

Fig. 2: Feature PeekStack

To get an implementation of the stack with the functionality of PeekStack and BaseStack, both features have to be composed into a new version of stack; the result of the composition of PeekStack and BaseStack is given in Figure 3. It is clear that the feature PeekStack cannot be compiled in isolation, since the method peek() requires an object stack.

That means that the FSF alone is expressive enough for reasoning in an abstract and general fashion about the structure of features, but it cannot express requirements on features: none of the standard models of Feature Algebra is able to characterise requirements.

III. Types of Requirements

We now describe all types of requirements[2] that occur in FOSD and which Feature Algebra should cover. We focus on requirements arising from object-oriented programming languages with feature support and use feature-oriented Java for our accompanying examples. Although there may be different types of requirements for different programming paradigms, we hope that the requirements listed here cover most of them, or, at least, the most relevant ones. There has been recent research on the capabilities of other languages for feature orientation, which indicates that issues like crosscutting features and feature interactions also exist in different programming domains, like functional programming [3] and specification languages [8]. We expect that this also applies to requirements.

To structure the list of requirements we distinguish between three different types: low-level and high-level requirements as well as non-functional criteria.

- *Low-level requirements* stem from the code level. An example are *dependences* like "feature A builds on feature B".
- *High-level requirements* mainly originate from the feature model or the domain. A typical example are optional features.
- An example of a *non-functional requirement* is that "the product must run on a mobile phone and has to have less than 1 Mb of compiled source code".

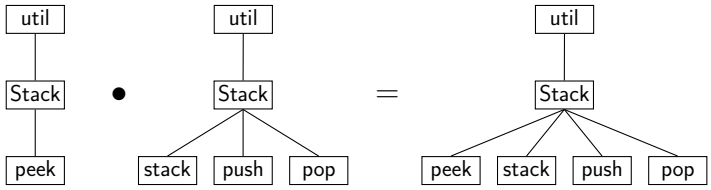
[1] Often the word "type" is used instead of "kind". In the present paper we use "type" for other concepts.

[2] at least all we are aware of

```
package util;

class Stack{
  Object[] stack;
  void push(Object o){
    stack[stackpointer++] = o; }
  Object pop(){
    return stack[stackpointer--]; }
  public Object peek(){
    return stack[stackpointer]; }
}
```

(a) Implementation



(b) FSF

Fig. 3: Feature composition

In this paper, we show how low- and high-level requirements can be formalised in Feature Algebra; although there are useful applications for non-functional requirements, we will not embed them in the algebraic model.

A. Low-Level Requirements

Low-level requirements are inferred from the code level; they describe all constraints that are based on implementation details. Some of them were discussed w.r.t. jak code by Thaker et al. [37].

References describe all code-level requirements where a method, a class, etc. refers to another object. In particular, the code cannot be compiled and/or executed if the referenced part is not contained in the overall product. An example was already given in the previous section; the feature PeekStack (cf. Figure 2 *refers* to stack). To focus on the essential parts and to obtain a uniform style we give a minimal example. In Figure 4, the feature F2 uses an element, the object o, which it does not define. Thus F2 implicitly contains the requirement that (at least one) feature in the overall system has to define an Object o.

```
//feature F2
class F1{
  void foo(){
    o.something();
    ... }
}
```

```
//feature F1
class F1{
  Object o;
}
```

Fig. 4: Low-level Requirement: Reference

Refinement is quite similar to the previous type of requirement. In jak, the keyword **refines** indicates that a feature F3 builds on (refines) a feature F4 (cf. Figure 5); hence to produce an executable program that has feature F3 as a component, F4 must be integrated, too. The same effect is produced by the keyword **extends** in Java.

```
//feature F4
refines class F3{
  ...
}
```

```
//feature F3
class F3{
  ...
}
```

Fig. 5: Low-level Requirement: Refinement

Abstract Class Constraints and **Interface Constraints** are two other types of requirements presented in [37]. 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 **class** or **interface** A or may introduce new abstract methods into A. If new descendants of A are introduced by a feature, only the introduced code is to be checked. Yet, if a feature introduces new abstract methods to the supertype A, we have to investigate all other features in the product whether they introduce a concrete subclass and, if so, that all these classes implement the newly introduced abstract method, as all descendants of **class** A now have a new implementation obligation.

Since the abstract class constraint and the interface constraint are very similar, we only give a code example for the former. Looking at Figure 6, we see that Feature F5 is an executable program without any requirement. In contrast to that, Feature F6 has a requirement, since it extends **class** F5. Moreover, the definition of **class** F5 forces the existence of the method foo in feature F6.

```
//feature F6
class F6 extends F5{
  void foo(){};
}
```

```
//feature F5
abstract class F5{
  abstract void foo();
}
```

Fig. 6: Abstract class constraint

This concludes our list of low-level requirements.

Since the first two of them behave similarly we will call them *structural dependences*. In the next section we will abstract structural dependences to a special kind of abstract requirements. All other requirements will be abstracted using a general concept of *constraints*.

B. High-Level Requirements

High-level requirements are semantic dependences that can only partially be derived from a product line description or specification. In general, they cannot be determined from a given implementation. They can either be derived from the given domain model [27] or are specified by a customer who wants a specific product or product line.

```
//features CONS
class List{
  ListNode first;

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

class ListNode{
  ListNode next;
}
```

(a) Implementation of cons

```
//feature name: SNOC
class List{
  ListNode last;

  void snoc(ListNode n){
    n.prev = last;
     last = n; }
}

class ListNode{
  ListNode prev;
}
```

(b) Implementation of snoc

```
//feature CONSSNOC
class List{
  ListNode first;
  ListNode last;

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

class ListNode{
  ListNode next;
  ListNode prev;
}
```

(c) Composition of cons and snoc

Fig. 7: Feature CONSSNOC

A customer may, for example, require a feature f in each product. Other constraints, such as invariants, are of a more semantic nature.

We start with requirements that can be encoded in the corresponding feature model (e.g., [27], [35]). Sometimes these requirements are also called primitives.

Mandatority describes the fact that a certain feature has to be within a product. An example is for example that a user requires a toString()-method for each class.

Optionality describes a requirement for a whole product line. It might be that a product A has the optional feature f, whereas another product B of the same product line does not have (implement) f.

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

Exclusion describes the situation that two features are not allowed to be inside the same product. For example, if a product has a 64-bit implementation of method foo, another implementation of the same method for a 32-bit system is not allowed.

Implication[3] describes the counterpart to an exclusion. A second feature is not forbidden, but required. For example, if a product provides a method (feature) to allocate memory, another feature for deallocation has to be provided.

User Requirements[4] are quite similar to the reference requirements of the previous subsection. But this time the dependence is given by the feature model or the user. For example, a costumer demands the implementation of a printer driver whenever a function print() is implemented. In other words, print() requires a printer driver.

[3]Implications are not given in the original literature. However we think that implications are the natural counterpart of exclusions. They are quite similar to references and refinement, but cannot be determined from the given code fragment.

[4]In the literature this primitive is only called "requirement". However, since we use the word "requirement" in a much more general setting, we added the word "user" to distinguish this special class from requirements in general.

Our examples are more or less simple; real-life requirements are more complicated, but base on the same concepts. We illustrate this by a cons/snoc example for lists. We assume two separate implementations: one for cons and another one for snoc (cf. Figure 7a–b). The composition (superimposition) of these features compiles without an error. However, depending on the intention of the programmer the result is erroneous. In Figure 7c the operations cons and snoc are not working on the same list.

To overcome this deficiency, the programmer can provide a further feature that should be added whenever both cons and snoc occur. Such a code fragment or derivative is given in Figure 8. Together with this code fragment one then has the requirement "if the features cons and snoc occur in a program, the derivative has also to be added."

```
refines class List{
  void cons(ListNode n){
    if (first == null)
      last = n;
    else
      first.prev = n;
    original(n); }
  void snoc(ListNode n){
    if (last == null)
      first = n;
    else
      last.next = n;
    original(n); }
}
```

Fig. 8: Code derivative for cons and snoc

This derivative enforces that the composition of cons and snoc work on the same list.

C. Non-Functional Requirements

Next to low-level and high-level requirements, there is another class of requirements. Examples are

- Since the implementation is intended to run on a mobile phone, the size of the compiled source code should be smaller than $1\,Mb$.

- Modern embedded systems save electricity by turning off temporarily unused components. Running on a contactless smart card, the code must be able to perform wireless authentication with a maximum power consumption of $7\,mW$.
- On the same smart card device, the code must be robust against differential power analysis, i.e. there are no means of reconstructing the cryptographic key used during authentication by an analysis of the device's power consumption.

Since these requirements can be *anything*, we will not integrate them into our abstract algebraic model. Rather, we will focus on the first two classes of requirements.

IV. Requirements in Feature Algebra

A. Feature Algebra

We briefly recapitulate the formal definitions relevant for Feature Algebra [6]. A Feature Algebra comprises a set I of *introductions* that abstractly represent FSFs and a set M of *modifications* that allow changing the introductions. The central operations are the addition $+$ that abstractly models feature tree superimposition, the operator $\cdot$ that allows application of a modification to an introduction and the modification composition operator $\circ$.

Definition 4.1: Formally, a *Feature Algebra* is a tuple $(M, I, +, \circ, \cdot, 0, 1)$ such that, for all $m, n \in M$ and $i, j \in I$,

- $(I, +, 0)$ is a monoid satisfying the additional axiom of distant idempotence, i.e., $i + j + i = j + i$,
- $(M, \circ, 1)$ is a groupoid operating via $\cdot$ on I, i.e., $\circ$ is a binary inner operation on M and 1 is an element of M such that furthermore
  - $\cdot$ is an external operation from $M \times I$ to I,
  - $(m \circ n) \cdot i = m \cdot (n \cdot i)$,
  - $1 \cdot i = i$,
- 0 is a right-annihilator for $\cdot$, i.e., $m \cdot 0 = 0$,
- $\cdot$ distributes over $+$, i.e., $m \cdot (i + j) = (m \cdot i) + (m \cdot j)$.

The standard model (e.g. [6]) of Feature Algebra uses FSFs as elements. The operation $+$ coincides with superimposition (cf. Section II-A). Superimposition is denoted by $\bullet$. 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 [6], [7].

In [24], other instances of Feature Algebra are discussed. Later on, we will introduce yet another model of Feature Algebra that captures requirements.

Definition 4.2: Based on the introductions of a Feature Algebra, the *natural preorder* or *subsumption preorder* is defined by $i \leq j \Leftrightarrow_{df} i + j = j$; it is closely related to the subtyping relation $<:$ in the Deep calculus of [25].

In the above mentioned model based on FSFs the subsumption preorder can be understand as follows: An FSF i is less or equal than a forest j ($i \leq j$) if i is a subforest of j. By a subforest we mean that each node(edge) of i is also a node(edge) in j. Moreover, if m is a node of i, each predecessor of m in j as well as all connection have to be in i. Please note that the relative order of terminal nodes does not matter for the subsumption relation.

B. Representing Requirements Algebraically

In the original model of Feature Algebra, an introduction is an FSF that abstractly represents a feature's implementation (cf. Section II). Other models use lists and sets as introductions [24]. However, none of them is able to formalise requirements at the abstract level.

In this section we discuss how requirements can be represented algebraically. Based on that we will introduce a model capturing requirements. The main idea is to use triples (i, d, c) consisting of a (partial) implementation i, a collection d of structural dependences and a constraint c. Both i and d are given by the introductions of some Feature Algebra A; this is possible, since structural dependences have the same structure as implementations. The component c is a predicate on the set of introductions I, i.e., a mapping from I to the set of truth values.

Let us look at an example. We already showed that the Feature PeekStack (cf. Figure 2) requires an object stack. Such a requirement is now modelled by yet another FSF (presented in Figure 9).

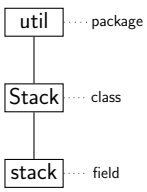


Fig. 9: A structural dependence of Feature PeekStack

To ease readability, we will denote the FSF of Figure 2b by i, the one of Figure 9 by d. Obviously, we do not have $d \leq i$. Hence i does not satisfy its requirements. In contrast to that the composed FSF of PeekStack and Basestack satisfies d, since it includes d as a subtree.

Let us now give some formal definitions.

Definition 4.3: Assume a Feature Algebra $A = (M, I, +, \circ, \cdot, 0, 1)$ and let $\mathcal{P}_I$ be the set of all predicates $p : I \to \{\mathsf{true}, \mathsf{false}\}$ over the introductions I of A.

1) A *design* over A is an element of $I \times I \times \mathcal{P}_I$. To select the components, we define, for a design $D = (i, d, c)$, $im(D) =_{df} i$, $sd(D) =_{df} d$ and $co(D) =_{df} c$.
2) Design D *satisfies its dependence* if $sd(D) \leq im(D)$.
3) Design D *satisfies its constraint* if $co(D)(im(D)) = \mathsf{true}$.
4) A design that satisfies both its dependence and its constraint is called a *product*.

Using designs it is now easy to model the classes of requirements identified in Section III. For most of these tasks it is useful to assume a predicate $has(f)$ that checks

whether a feature f is included in a given Feature Algebra element i. Frequently, but not always, this can be expressed in the form

$$has(f)(i) \Leftrightarrow_{df} f \leq i \ .$$

Moreover, we need some basic knowledge of predicates.

Definition 4.4: Assume an arbitrary set I, the set of predicates $\mathcal{P}_I$ over I and predicates $p, q \in \mathcal{P}_I$.

1) The *conjunction* of two predicates p, q is given by $(p \wedge q)(i) =_{df} p(i) \wedge q(i)$ for all $i \in I$. This operation is associative, commutative and idempotent. The corresponding subsumption order can be described by

$$p \leq q \Leftrightarrow \forall i \in I : q(i) \Rightarrow p(i) \ .$$

2) The predicate that maps every element of I to true is denoted by **true**.

The introductory discussion on designs has already shown that structural dependences (references and refinement) can easily be determined from a given implementation and formulated by in the concrete algebra of designs over FSFs. An example was already presented in the previous section.

Also, mandatory features, optionality, alternatives and user requirements can be encoded into constraints in a straightforward manner. If more than one requirement occurs they are combined by conjunction. We skip the details for these constructions and refer to the case study in [22].

Next, we show how exclusions and implications can be formalised. Since the third component of designs is based on predicates, we freely use the logical connectives $\wedge, \vee, \Rightarrow, \neg, \ldots$. All these operations are, like $\wedge$, defined on predicates via pointwise lifting.

- That feature f excludes feature g is expressed as

$$has(f)(i) \Rightarrow \neg has(g)(i)$$

for all FSFs i or,equivalently, by $\neg(has(f)(i) \wedge has(g)(i))$.

- Similarly, feature implication is expressed as

$$has(f)(i) \Rightarrow has(g)(i)$$

for all FSFs i or,equivalently, by $\neg has(f)(i) \vee has(g)(i)$.

Last, we have have a look at abstract class constraints; abstract interface constraints can be characterised in a similar way. An abstract class constraint has the form

If a **class** C extends the **class** D then it must provide a method foo().

Again this can be formalised as an implication:

$$extends_D(i) \Rightarrow has(\mathsf{foo}())(i) \ ,$$

where $extends_D$ is a predicate which looks at the provided code and decides whether the phrase **extends** D occurs.

The two predicates (*has* and *extends*) presented so far, are given in a very informal way (although it is clear how to define them properly). A more general form of constraints is specified in the form

$$P(i) \Rightarrow \exists j : Q(i, j) \ ,$$

where i, j are variables for introductions and P, Q are predicates. In particular, each tool based on DES_A has to provide such a simple query language, closely related to the query language that defines modifications on FSFs (cf. Section II). Examples for existing query languages are AspectJ (for defining pointcuts) or the modification language of FeatureHouse [17].

C. A Feature Algebra of Designs

We now want to make the set of designs over A into a Feature Algebra itself. Having again a Feature Algebra immediately implies that all knowledge about Feature Algebras also holds for the new structure. In particular, the new model violates none of the concepts Feature Algebra was intended for. Therefore the model can be seen as an algebra that covers "even more" concepts of FOSD at the abstract algebraic level.

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 requirements of a combined design should be the joined requirements 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, i.e., of functions that transform predicates into predicates. They are for example used in Dijkstra's wp-calculus [19] or in Back and von Wright's refinement calculus [10].

Definition 4.5: Assume an arbitrary set I, the set of predicates $\mathcal{P}_I$ over I and predicates $p, q \in \mathcal{P}_I$.

1) We denote by $\mathcal{PT}_I$ the set of all *predicate transformers* $m : \mathcal{P}_I \to \mathcal{P}_I$.
2) By id we denote the identity predicate transformer with $id(p) = p$ for all $p \in \mathcal{P}_I$.
3) The composition operator $\circ : \mathcal{PT}_I \times \mathcal{PT}_I \to \mathcal{PT}_I$ is defined, as usual, by $(m_1 \circ m_2)(p) =_{df} m_1(m_2(p))$.
4) The operator $\cdot : \mathcal{PT}_I \times \mathcal{P}_I \to \mathcal{P}_I$ applies a predicate transformer to a predicate; the result is again a predicate: $m \cdot p =_{df} m(p)$.
5) A predicate transformer m is *conjunctive* if for all $p, q \in \mathcal{P}_I$ we have $m(p \wedge q) = m(p) \wedge m(q)$. In particular, id is conjunctive.
6) The set of all conjunctive predicate transformers over A is denoted by $\mathcal{CT}_I$.

Lemma 4.6: Given a set I of introductions, the structure $(\mathcal{CT}_I, \mathcal{P}_I, \wedge, \circ, \cdot, \mathbf{true}, id)$ forms a Feature Algebra.

The proof is straightforward.

Now we are ready to construct a Feature Algebra of designs. The following result is immediate by universal algebra (e.g. [38]).

Theorem 4.7: Let $A = (M, I, +, \circ, \cdot, 0, 1)$ *be a Feature Algebra. Then 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 again a Feature Algebra if $\mathbf{0} =_{df} (0, 0, \mathbf{true})$, $\mathbf{1} =_{df} (1, 1, id)$[5] *and the operations as well as modifications are lifted pointwise. For example, the external operation* $\cdot$ *is defined by* $(m, n, l) \cdot (i, d, c) =_{df} (m \cdot i, n \cdot d, l \cdot c)$ *, where* $m, n \in M$ *are modifications of* A *and* $l \in \mathcal{CT}_I$ *is a predicate transformer (modification of* $\mathcal{P}_I$*).*

V. Propagating Type Information

Not all structural dependences can be represented as FSFs. This means that structural dependences and introductions are not equivalent in the concrete model. Informally, structural dependences can be viewed as introductions enriched by the additional concept of *replacement expressions*. Before a definition of structural dependences and replacement expressions is given, they are motivated by an example.

In the case of a typed language the type of an element is determining which function can be applied to or invoked on this element. For example in Java elements of type String can be concatenated, two elements of type Integer can be added.

In an FSF each node has a position which can be identified with the path to that node (as long as the ordering of children is irrelevant). Furthermore each node has a kind. According to [6], [7] the kind of a node in the case of Java can be **package**,**class**, field , method ,.... Furthermore classes in Java denote types. This means that a node labelled MyOwnClass is of kind **class** and introduces a new type MyOwnClass, which is important for the type system.

When identifying structural dependences it can occur that a function f() is invoked on an element e, but it is not possible to determine the type of element e. This means, in FSF terminology, the exact path and the type (types in the sense MyOwnClass) of the node labelled e are not known. A little Java example should clarify this.

Given the three features A, B, C of Figure 10. We try to find their structural dependences. For feature A it is quite easy since there are no structural dependences at all. In Feature B, the member b is of type D. Hence this feature requires a definition of a **class** D and the structural dependences of feature B are $sd(\mathsf{B}) = \{\mathsf{D}\}$.

For feature C things get more difficult. In the method foo(), a method e() is invoked on an element b. This means, first, that C requires an element b. There are two different possibilities to include it. The first one is that b is declared as a member of **class** A. The other one is that b is a local variable in method foo(). Since this affects the code level, we will not go into more detail at the moment and assume the first case. Hence A.b is a structural dependence of feature C.

Although it might seem plausible that A.b.e() is another structural dependence, this is not the case. First the type of the member A.b has to be determined and then it has to be checked whether an operation e() is defined on that type or not. Therefore a type evaluation operator τ is introduced and the structural dependence A.b.e() is replaced by τ(A.b, F).e(), which means "determine the type of A.b within F and insert it". Here F is the overall FSF in which we are working. Then we can write $sd(\mathsf{C}) = \{\mathsf{A}, \mathsf{A.b}, \tau(\mathsf{A.b},\mathsf{F}).\mathsf{e}()\}$.

If, for example, features B and C are added we have $sd(\mathsf{B} + \mathsf{C}) = \{\mathsf{D}, \mathsf{A}, \mathsf{A.b}, \tau(\mathsf{A.b},\mathsf{F}).\mathsf{e}()\}$ and $im(\mathsf{B} + \mathsf{C}) = \{\mathsf{A}, \mathsf{A.b}, \mathsf{A.foo}()\}$. Now it is possible to evaluate the type of A.b since $\{\mathsf{A.b}\} \subseteq im(\mathsf{B} + \mathsf{C})$ and to rewrite the structural dependences as $sd(\mathsf{B} + \mathsf{C}) = \{\mathsf{D}, \mathsf{A}, \mathsf{A.b}, \mathsf{D.e}()\}$

```
//feature A
class D{
 void e(){}
}
```

```
//feature B
class A {
 D b;
}
```

```
//feature C

class A{
 void foo(){
  b.e(); }
}
```

```
//composition of B and C
class A{
 D b;
 void foo(){
  b.e(); }
}
```

Fig. 10: Type Evaluation

This example is strongly influenced by Java. Nevertheless the idea of type evaluation can be expressed in an abstract, language independent way.

Definition 5.1 (replacement): Replacement is a function $\tau : I \times I \to I$ which maps an introduction together with a context to another introduction.

A replacement expression is of the form $\tau(i, j)$, where i, j are elements of I. It is supposed to represent the type of i as provided by the context j. With a view on implementation, the function τ can be viewed as abstracting references from applied occurrences to defining occurrences (and extracting information from there).

In terms of the FSF model, an introduction represents an FSF and therefore encodes each path of an FSF in an abstract way. Then replacement maps an FSF to another FSF as seen in the previous example. In the example the FSF with the set-based representation {A, A.b} was mapped to the FSF with the set-based representation {D}.

Like for the function sd, we need a language dependent implementation of the function τ. In the case of Java it is called type evaluation. Note that we are now not talking at the abstract level of introductions but at the level of the concrete model.

[5]Please note that we overload the symbols $+, \circ$ and $\cdot$.

In the concrete model, replacement expressions have to be of a certain (well-formed) kind to be usable as structural dependences. Therefore we will revise the identification of the replacement expression in the above example.

Let again F be the overall FSF in which all the features subsequently mentioned occur. The replacement expression τ(A.b,F).e() is found in feature C. Looking at the implementation of method foo() a first structural dependence is A.b. A method e() is invoked on the element b, that has just been identified as an structural dependence. Whether such a method e() exists depends on the type of the element b. Hence the method e() itself is a structural dependence. Consequently the second structural dependence depends on the structural dependence A.b. Furthermore, translating the second structural dependence into an FSF we know that there must be a leaf labelled e. So we only have to evaluate the FSF excluding the leaf itself.

Assume a path $p = a.b.c$ and an introduction d. The following two properties must be satisfied for a type evaluation expression to be well-formed.

1) Type evaluation may only be applied to a proper prefix of the path leading to the feature in question. Thus τ(a.b.c,F) is not well-formed whereas τ(a.b,F).c is.
2) Furthermore, for every design D and every path p in F the implication $\tau(p,\mathsf{F}).\mathsf{c} \in sd(D) \Rightarrow p \in sd(D)$ must hold.

The second property is necessary since a type evaluation expression cannot be applied to an argument p if the argument itself is not given.

Since replacement expressions evaluate to introductions, they have the same algebraic properties. Hence replacement expressions also form an FA. To formalise this, we slightly change the definitions used in Theorem 4.7. We distinguish between introductions representing pure FSFs and introductions enriched with replacement expressions representing structural dependences; the set of the latter ones is called SD. Now, instead of working on triples $(i, d, c) \in I \times I \times \mathcal{P}_I$ designs, we use designs $(i, d, c) \in I \times \mathrm{SD} \times \mathcal{P}_I$.

With this modification, adding τ to the set of structural dependences does not violate the axioms of Feature Algebra.

Lemma 5.2: The structure $(M, SD, +, \circ, \cdot, 0, 1)$ forms a feature algebra, when modification application on path replacement expressions is defined as:

$$m \cdot \tau(p, q).e = \tau(m \cdot p, q).e$$

This is intuitive, since a modification in the FSF model is applied to paths.

To check whether a feature satisfies its dependence (in the sense of Definition 4.3), all type evaluations must have been replaced. If that is not possible, the design cannot satisfy its dependence, since a type evaluation expression is never part of an introduction.

VI. Related Work

A main goal of our Feature Algebra is to formally model conflicts between features in a language-independent way. To our knowledge we are the first to provide an algebra to model these conflicts. Yet, there are several approaches like ours dealing with formal aspects of feature composition. For clarity, we want to discern them into four areas:

- feature-oriented type systems
- constraint-based approaches
- hybrid approaches
- approaches with focus on feature models

A. Feature-oriented Type Systems

Feature-oriented type systems resemble what we call the low-level requirements in terms of Feature Algebra. Type system rules apply to all elements of a given programming language. In Feature Algebra, these rules are modelled as requirements, which have to be applied to every fragment of feature of a certain language. Type systems cannot model domain-specific constraints, which are a major source of feature interactions. This is why type systems are necessary to handle conflicts between features, but are not enough to serve as a device to avoid them.

gDeep is a language-independent, feature-oriented type calculus [2]. New languages can be adopted via a plug-in system, which promotes a (nearly) arbitrary type system into a feature-oriented one. Our Feature Algebra is also situated on top of a language-dependent type system. Like in *gDeep*, these not feature-oriented type rules have effect on the feature-oriented requirements imposed by the algebra. For example, an *interface constraint* is the feature-oriented form of a corresponding type rule.

There are other formal approaches to type systems, which are not explicitly feature-oriented, but have a notion of components, for example *FeatherTraitJ* [30] and *Java Mixins* [1]. These are language-dependent.

As our Feature Algebra is not bound to any specific language syntax, it is able to model programming paradigms instead of language constructs in terms of feature conflicts. This way, similar languages are treated similarly.

B. Constraint-Based Approaches

There have been attempts to cover the issue of feature interactions by model-checking [21], [20], [16], [36]. Like us, these approaches use additional information to argue about properties of feature combinations. These properties map to high-level requirements and abstract from code. We aim at covering all these requirements in one formal algebra and hope to encapsulate fundamental concepts of FOSD which do not arise when considering either low or high-level requirements alone.

Distributed feature composition by Zave et al. [26] uses component instance rules to determine the environment a feature can cooperate with . These instance rules try to be modular, complete and comprehensible service specifications. Unlike our Feature Algebra, the approach is limited

to the domain of telecommunications. Yet, similarities to our approach are the formal basis and the idea to reduce the number of interactions by a set of rules.

C. Hybrid Approaches

Hybrid approaches unite low-level as well as high-level requirements.

Delta-oriented programming represents a product line by one core module and several delta modules [34]. These delta modules are mostly similar to optional features, with the difference that they even may remove code. Delta modules include application conditions, which have to be checked if a product is generated. Delta-oriented programming both has measures to automatically assert type correctness and application conditions allow for constraints at the level of features. This information is used to automatically determine if the composition of certain features is commutative. It is yet unknown whether our Feature Algebra complies with delta-oriented programming.

Multiple approaches dealing with requirements were introduced by Batory et al. Conditions and obligations are used for Features in *GenVoca* to express high-level requirements [12], formality is provided by the *Inscape* approach by Perry [33] and *Safe Composition* asserts low-level requirements [29], [18], [11], [37]. In sum these approaches handle all kinds of requirements present in our Feature Algebra, but there is no evidence that a combination is possible. Our Feature Algebra is able to cover all these approaches in one comprehensive model.

D. Feature Models

Feature-oriented domain analysis aims to discern valid and invalid feature combinations. The set of all valid combinations inside a product line is described by a propositional formula, the *feature model* [27]. Traditionally, a feature model is constructed by hand utilizing the knowledge of a domain expert. While there are recent efforts to automatically build feature models out of given software product lines, these approaches can only handle low-level requirements [9], [28].

A prominent goal of our Feature Algebra is to be able to specify a high-level requirement inside a feature, which then has effects on the whole product line. Thus, it would be possible to generate feature models out of both low and high-level requirements.

Additionally, a requirement can be formulated in terms of one feature's domain characteristics (like a specification of the environment the feature needs for proper function). This can help to avoid combinations of interacting features. However, the feasibility of this goal is yet to be showed, as we expect this to be the more difficult the more intricate a given feature interaction is.

VII. Conclusion

In this paper we have shown how requirements can be embedded into the abstract structure of Feature Algebra. So far Feature Algebra was used to describe main aspects of FOSD. Feature Algebra itself is intended to describe FOSD at a very abstract level. Its purpose is to provide a mathematically precise foundation for this kind of software development. The main model of Feature Algebra is based on feature structure forests, structures that describe the main components of a given program and that are closely related to abstract syntax trees.

However, none of the earlier models was able to describe requirements. Feature interactions are an immanent problem of FOSD, so measures helping to handle them are requisite. Providing proper instruments for the formulation of requirements are one step in this direction, that is why we think that requirements are a major construct of FOSD and should be included in the abstract model. They have various forms: either they can be determined from a given program (like references) or they are explicitly given by the stakeholder/programmer.

As the main contribution of this paper we presented a model of Feature Algebra that covers all aspects of requirements. It is based on a product construction over different kinds of feature algebras. The fact that the outcome forms again a feature algebra is especially remarkable and important, since all knowledge about feature algebras immediately holds for the new structure. In particular, the new model does not invalidate any of the concepts of the original Feature Algebra. Therefore it can be seen as an algebra that covers "even more" concepts of FOSD at the abstract algebraic level. Based on this model, we have shown how different kinds of requirements can be formalised algebraically. Moreover we have presented a small case study that underpins our approach.

Future work will include much larger case studies. They will help to further understand the structure of Feature Algebra and requirements. Moreover, by implementing a language-independent algebra-based tool set we hope to get incentives for further development of Feature Algebra and see how it can serve as a formal basis for the fundamental concepts of FOSD.

References

[1] D. Ancona, G. Lagorio, and E. Zucca. Jam—designing a Java extension with mixins. *ACM Trans. Program. Lang. and Syst.*, 25(5):641–712, 2003.

[2] S. Apel and D. Hutchins. A calculus for uniform feature composition. *ACM Trans. Program. Lang. and Syst.*, 32(5):1–33, 2010.

[3] S. Apel, C. Kästner, A. Größinger, and C. Lengauer. Feature (de)composition in functional programming. In A. Bergel and J. Fabry, editors, *Software Composition*, volume 5634 of *LNCS*, pages 9–26. Springer, 2009.

[4] S. Apel, C. Kästner, and C. Lengauer. FeatureHouse: Language-independent, automated software composition. In *31th International Conerence on Software Engineering(ICSE)*, pages 221–231. IEEE Press, 2009.

[5] S. Apel and C. Lengauer. Superimposition: A language-independent approach to software composition. In C. Pautasso and É. Tanter, editors, *Software Composition*, volume 4954 of *LNCS*, pages 20–35. Springer, 2008.

[6] S. Apel, C. Lengauer, B. Möller, and C. Kästner. An algebra for features and feature composition. In *Algebraic Methodology and Software Technology (AMAST 2008)*, volume 5140 of *LNCS*, pages 36–50. Springer, 2008.

[7] 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.

[8] S. Apel, W. Scholz, C. Lengauer, and C. Kästner. Detecting dependences and interactions in feature-oriented design. In *International Symposium on Software Reliability Engineering (ISSRE 2010)*, pages 161–170. IEEE Computer Society, 2010.

[9] S. Apel, W. Scholz, C. Lengauer, and C. Kästner. Language-independent reference checking in software product lines. In *Workshop on Feature-Oriented Software Development (FOSD)*, pages 65–71. ACM Press, 2010.

[10] R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer, 1998.

[11] D. Batory. Using modern mathematics as an FOSD modeling language. In *Conference on Generative Programming and Component Engineering (GPCE 08)*, pages 35–44. ACM Press, 2008.

[12] D. Batory and B. Geraci. Composition validation and subjectivity in GenVoca generators. *IEEE Trans. Software Engineering*, 23(2):67 –82, 1997.

[13] 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.

[14] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling stepwise refinement. In *Conference on Software Engineering (ICSE 03)*, pages 187–197. Proc. IEEE, 2003.

[15] D. Batory, V. Singhal, M. Sirkin, and J. Thomas. Scalable software libraries. *ACM SIGSOFT Software Engineering Notes*, 18(5):191–199, 1993.

[16] L. D. Bousquet. Feature interaction detection using testing and model-checking—experience report. In *In World Congress on Formal Methods*, pages 622–641. Springer, 1999.

[17] B. Boxleitner, S. Apel, and C. Kästner. Language-independent quantification and weaving for feature composition. In A. Bergel and J. Fabry, editors, *Software Composition*, volume 5634 of *LNCS*, pages 45–54. Springer, 2009.

[18] B. Delaware, W. Cook, and D. Batory. A machine-checked model of safe composition. In *Workshop on Foundations of Aspect-oriented Languages (FOAL 09)*, pages 31–35. ACM Press, 2009.

[19] E. Dijkstra. *A discipline of programming*. Prentice Hall, 1976.

[20] N. Griffeth, R. Blumenthal, J. Gregoire, and T. Ohta. Feature interaction detection contest of the fifth international workshop on feature interactions. *Computer Networks*, 32(4):487 – 510, 2000.

[21] O. Grumberg and D. Long. Model checking and modular verification. *ACM Trans. Program. Lang. and Syst.*, 16(3):843–871, 1994.

[22] P. Höfner, S. Mentl, B. Möller, and W. Scholz. Requirements in feature algebra. Technical Report 2010-12, Institut für Informatik, Universität Augsburg, 2011. (to appear).

[23] 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.

[24] P. Höfner and B. Möller. An extension of feature algebra. Technical Report 2010-9, Institut für Informatik, Universität Augsburg, 2010.

[25] D. Hutchins. Eliminating distinctions of class: Using prototypes to model virtual classes. In P. Tarr and W. Cook, editors, *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2006)*, pages 1–20. ACM Press, 2006.

[26] M. Jackson and P. Zave. Distributed feature composition: A virtual architecture for telecommunications services. *IEEE Trans. Software Engineering*, 24:831–847, 1998.

[27] 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.

[28] K. Kästner, S. Apel, S. Trujillo, M. Kuhlemann, and D. Batory. Guaranteeing syntactic correctness for all product line variants: A language-independent approach. In *Conference on Objects, Models, Components, Patterns (TOOLS EUROPE)*, volume 33 of *Lecture Notes in Business and Information Processing*, pages 174–194. Springer, 2009.

[29] M. Kuhlemann, D. Batory, and C. Kästner. Safe composition of non-monotonic features. *ACM SIGPLAN Notices*, 45(2):177–186, 2010.

[30] L. Liquori and A. Spiwack. FeatherTrait: A modest extension of Featherweight Java. *ACM Trans. Program. Lang. and Syst.*, 30(2):1–32, 2008.

[31] 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.

[32] H. Ossher and H. Harrison. Combination of inheritance hierarchies. In *Object-Oriented Programming, Systems, Languages, and Applications*, pages 25–40. ACM Press, 1992.

[33] D. E. Perry. The inscape environment. In *Conference on Software Engineering (ICSE 89)*, pages 2–11. ACM Press, 1989.

[34] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. Delta-oriented programming of software product lines. In J. Bosch and J. Lee, editors, *Software Product Lines: Going Beyond*, volume 6287 of *LNCS*, pages 77–91. Springer, 2010.

[35] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps. Generic semantics of feature diagrams. *Computer Networks*, 51:456–479, 2007.

[36] B. Stepien and L. Logrippo. Feature interaction detection using backward reasoning with LOTOS. In *Protocol Specification, Testing and Verification XIV (PSTV 94)*, pages 71–86. Chapman & Hall, 1995.

[37] 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.

[38] W. Wechler. *Universal Algebra for Computer Scientists*. Springer, 1992.