An Extension for Feature Algebra

[Extended Abstract]

Peter Höfner
Institut für Informatik
Universität Augsburg
86135 Augsburg, Germany
hoefner@informatik.uni-augsburg.de

Bernhard Möller
Institut für Informatik
Universität Augsburg
86135 Augsburg, Germany
moeller@informatik.uni-augsburg.de

ABSTRACT

Feature algebra was introduced as an abstract framework for feature oriented software development. One goal is to provide a common, clearly defined basis for the key ideas of feature orientation. We first present concrete models for the original axioms of feature algebra which represent the main features of feature oriented programs. However, these models show that the axioms of the feature algebra do not reflect some aspects of feature orientation properly. Hence we modify the axioms and introduce the concept of an *extended feature algebra*. Since the extension is also a generalisation, the original algebra can be retrieved by a single additional axiom. Last but not least we introduce more operators to cover concepts like overriding in the abstract setting.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Object-oriented design methods*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Logics of programs, Mechanical verification, Specification techniques*

General Terms

Design, Languages, Verification

Keywords

feature oriented software development, feature algebra, algebraic characterisation of FOSD

1. INTRODUCTION

Over the last few years *Feature-Oriented Software Development* (FOSD) (e.g. [7]) has been established in computer science as a general programming paradigm that provides formalisms, methods, languages, and tools for building variable, customisable, and extensible software.

Feature algebra [3] is a formal framework that captures many of the common ideas of FOSD such as introductions, refinements, or quantification and hides differences of minor importance. It abstracts from the details of different programming languages and environments used in FOSD. Moreover, alternative design decisions in the algebra reflect variants and alternatives in concrete programming language mechanisms; for example, certain kinds of feature composition may be allowed or disallowed.

In one of the standard models of feature algebra, the structure of a feature is represented as a tree, called a *feature structure tree* (FST) [2]. An FST captures the essential, hierarchical module structure of a given program. Based on that, feature combination can be modelled as superimposition of FSTs, i.e., as recursively merging their corresponding substructures.

Feature algebra serves as a formal foundation of architectural metaprogramming [6] and automatic feature-based program synthesis [10]. Both paradigms emerged from FOSD and facilitate the treatment of programs as values manipulated by metaprograms, e.g., in order to add a feature to a program system. This requires a formal theory that precisely describes which manipulations are allowed.

In the present extended abstract we first derive a concrete model for feature algebra that is based on FSTs. It was already sketched in [3]; however we define it in a precise way. After introducing the abstract notion of a feature algebra, we present another concrete model. Next we show that the models are fine as long as one does not consider feature oriented programming on code level. If manipulation of code and not only of the overall program structure is explicitly included in the model some aspects of feature orientation cannot be reflected properly. In particular, we show that merging, overriding or extending bodies of methods yields problems. To overcome this deficiency, we relax the axioms and introduce the concept of an extended feature algebra. To clarify the idea and to underpin the relaxation we also extend the introduced model which can then handle code explicitly. Finally we discuss how additional operators can be introduced to capture even more properties of feature oriented programming formally. In particular, we present operators for merging, overriding and updating as they arise in code modification.

2. A STANDARD MODEL

Based on *feature structure trees* (FSTs), we give a first concrete model for feature algebra. The formal definition of feature algebras will be given in the next section. Fea-

ture structure trees capture the essential hierarchical module structure of a given program system (e.g. [3]). An example is given in Figure 1, where a simple Java class Base is described. For the present extended abstract we restrict ourselves to Java; examples of other feature oriented programming languages can easily described in a similar way.

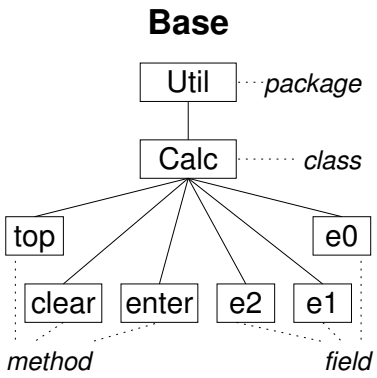

Figure 1: A simple JAVA-class as FST ([6, 3])

It is well known that certain labelled forests can be encoded using strings of node labels (e.g., [5]). We use forests rather than single trees in our description, since, in general, we deal with several classes.

Let Σ be an alphabet of node labels and, as usual, Σ^+ the set of all nonempty finite strings over Σ. Every such word can be thought of as the sequence of node labels along the unique path from a root in the forest to a particular node. In the sequel we will just write "path" instead of the lengthy "string of labels along a path". Note that this approach does not allow different roots with identical labels and no identical labels on the immediate descendants of a node. However, this is not a restriction.

A first model now represents a forest by all possible paths from roots to nodes. Since every prefix of the path leading to a node x corresponds to a path from the respective root to an ancestor of x, with a path also all its non-empty prefixes are paths in the forest. Therefore the set of all possible paths is prefix-closed. Note, however, that a *set* of paths forgets about the relative order of child nodes of a node, i.e., this model is suitable only for unordered trees.

EXAMPLE 2.1. *The FST of Figure 1 is encoded as the following prefix-closed set:*

$$\begin{aligned} Base =_{def} \{\ & Util,\ Util :: Calc,\ Util :: Calc :: top,\\ & Util :: Calc :: clear,\ Util :: Calc :: enter,\\ & Util :: Calc :: e0,\ Util :: Calc :: e1,\\ & Util :: Calc :: e2\ \}\ , \end{aligned}$$

where :: *is used to separate the elements of* Σ. *Of course all occurring names must be elements of the underlying alphabet, i.e.,* $Util, Calc, top, clear, enter, e0, e1, e2 \in \Sigma$. □

Conversely, one can (uniquely up to the order of branches) reconstruct a forest from the prefix-closed set of its paths.

We define $P\Sigma$ as the set of all prefix-closed subsets of Σ^+. Note that $P\Sigma$ is closed under set union. Based on this, feature tree superimposition can simply be defined as set union. Hence the order of combination does not matter and therefore addition is commutative and idempotent.

It is easy to show that $(P\Sigma, \cup, \emptyset)$ forms a monoid, i.e., $\cup$ is associative with $\emptyset$ as its neutral element, and, because of commutativity and idempotence of its addition operator $\cup$, also satisfies the axiom of distant idempotence, namely $A \cup B \cup A = B \cup A$ for $A, B \in P\Sigma$.

In addition to feature superimposition, feature algebra also comprises modifications which in the concrete model are tree rewriting functions.

It is easy to see that such functions can be used to model many different aspects of feature oriented programming and development. With respect to FSTs a modification might be the action of adding a new child node (adding a method to a class), of deleting a node (removing a method) or of renaming a node (renaming a class). As we will see, altering the contents of a leaf node (overriding and extending a method body) may lead to a problem.

A concrete tool for performing the operations of a feature algebra is *FeatureHouse* [1, 2]. It allows composing features written in various languages such as Java, C#, C, Haskell, and JavaCC. With the help of this tool we will show in Section 5 that the implementation and the axioms of feature algebra given below do not coincide when modifications are allowed to touch the code level. As long as the code is ignored, i.e., only the method interfaces or the like are considered to be modifiable things works fine.

This observation implies that, to achieve full congruence, either the theory has to be adapted or the implementation of *FeatureHouse* has to be changed. In Section 6 we introduce an extension of feature algebra that is designed to cover also features at code level.

3. FEATURE ALGEBRA

We now abstract from the concrete model of FSTs and introduce the structure of feature algebra. It was first presented by Apel, Lengauer, Möller and Kästner in [3]. There a number of axioms is selected that have to be satisfied by languages suitable for feature oriented software development. For the present paper we compact them and come up with the following definition. To focus on the main aspects we omit a discussion of the variants and alternatives described in the same paper.

A feature algebra comprises a set I of *introductions* that abstractly represent feature trees and a set M of *modifications* that allow changing the introductions. The central operations are the summation $+$ that abstractly models feature tree superimposition, the operator $\cdot$ that allows application of a modification to an introduction and the modification composition operator $\circ$.

Formally, a *feature algebra* is a tuple $(M, I, +, \circ, \cdot, 0, 1)$ such that

- $(I, +, 0)$ 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 binary 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., $m \cdot (i + j) = (m \cdot i) + (m \cdot j)$

for all $m, n \in M$ and all $i, j \in I$.

On the introductions of a feature algebra, the *natural preorder* or *subsumption preorder* is defined by $i \leq j \Leftrightarrow_{def} i + j = j$; it is closely related to the subtyping relation $<:$ in the DEEP calculus of [15]. The *introduction equivalence* by $i \sim j \Leftrightarrow_{def} i \leq j \wedge j \leq i$. Finally, we define the *application equivalence* $\approx$ of two modifications m, n by $m \approx n \Leftrightarrow_{def} \forall i : m \cdot i = n \cdot i$. This is clearly an equivalence relation.

The model introduced in the previous section forms a feature algebra if a suitable set of tree rewriting functions is chosen as the set of modifications. The set has to be chosen carefully, since otherwise the functions might, e.g., violate the uniqueness conditions imposed on forests. The axiom $(m \circ n) \cdot i = m \cdot (n \cdot i)$ is satisfied by the usual definition of function composition: applying a composed function is equivalent to applying the single functions in sequence. Then the operator $\cdot$ coincides with function application and $\circ$ with function composition. Because of commutativity and idempotence of $\cup$ (which instantiates $+$ in that model), the natural preorder there actually is an order and coincides with the subset relation $\subseteq$.

An advantage of this particular abstract algebraic definition is that it contains only first-order equational axioms, i.e., it is predestined for automatic theorem proving. Since we have encoded feature algebra in Waldmeister [9][1], we skip the proofs. They can be found at a website [14].

LEMMA 3.1. *Assume i, j to be introduction sums and assume m, n, o to be modifications of a feature algebra.*

1. $0 \leq i$ *and* $i \leq 0 \Rightarrow i = 0$.
2. $+$ *is idempotent; i.e.,* $i + i = i$.
3. $\leq$ *is a preorder, i.e.,* $i \leq i$ *and* $i \leq j \wedge j \leq k \Rightarrow i \leq k$.
4. $i \leq i + j$ *and* $j \leq i + j$.
5. $i \leq k \wedge j \leq k \Rightarrow i + j \leq k$.
6. $+$ *is quasi-commutative w.r.t.* $\sim$, *i.e.,* $i + j \sim j + i$.
7. $(m \circ (n \circ o)) \cdot i = ((m \circ n) \circ o) \cdot i$.
8. $(m \circ 1) \cdot i = (1 \circ m) \cdot i = m \cdot i$.

Meanings and relevance of Parts (1)–(3) are straightforward. Part (4) says that addition determines an upper bound with respect to the natural preorder. Part (5) shows that the sum is even a least upper bound. Parts (7) and (8) show that, up to application equivalence, $\circ$ is associative and 1 is its neutral element, i.e., $(m \circ (n \circ o)) \approx ((m \circ n) \circ o)$ and $(m \circ 1) \approx (1 \circ m) \approx m$.

4. ANOTHER STANDARD EXAMPLE

Since in certain applications the relative order of the immediate successor nodes in a tree matters, we now present a second model that reflects forests of ordered labelled trees. It uses the fact that all paths in a tree can be recovered from the maximal ones that lead from roots to leaves by forming their prefix closure. It should be noted here that the maximal paths can be viewed as atomic introductions in the sense of [3]. This could have been exploited already in the previous model, but would have led to a much more complicated definition of tree superimposition. While an unordered forest can be represented as the finite *set* of its maximal paths, for an ordered one we use finite *lists* of such paths. To make the representation unique, we have to restrict ourselves to lists that are prefix-free, i.e., lists l that with a path p do not contain a proper or improper prefix of p elsewhere in l. In particular, such lists are repetition-free. Like the previous model, this does not admit different roots with identical labels and no identical labels on immediate descendants of a node.

EXAMPLE 4.1. *The FST of Figure 1 is encoded as the following prefix-free list:*

$$\begin{aligned} Base =_{def} [\ & Util :: Calc :: top,\ Util :: Calc :: clear, \\ & Util :: Calc :: enter,\ Util :: Calc :: e2, \\ & Util :: Calc :: e1,\ Util :: Calc :: e0\ ]\ . \end{aligned}$$

□

Superimposition $+$ is now defined inductively over the length of the first list:

- The empty list does not effect another list of paths:

$$[\,] + [q_1, \ldots, q_n] =_{def} [q_1, \ldots, q_n]$$

- A singleton list $[p]$ is added to an existing list by replacing existing prefixes of it:

$$[p] + [q_1, \ldots, q_n] =_{def} \begin{cases} [q_1, \ldots, q_n] \\ \quad \text{if } p \text{ is a prefix of some } q_i \\ [q_1, \ldots q_{i-1}, p, q_{i+1}, \ldots, q_n] \\ \quad \text{if } q_i \text{ is a prefix of } p \\ [p, q_1, \ldots, q_n] \\ \quad \text{otherwise} \end{cases}$$

- For longer lists we set

$$\begin{aligned} & [p_1, \ldots, p_m, p_{m+1}] + [q_1, \ldots, q_n] =_{def} \\ & \quad [p_1, \ldots, p_m] + ([p_{m+1}] + [q_1, \ldots, q_n]) \end{aligned}$$

We define $L\Sigma$ as the set of all prefix-free lists of elements of Σ^+. It is easy to show that $(L\Sigma, +, [\,])$ forms a (non-commutative) monoid that additionally satisfies the axiom of distant idempotence; its natural preorder reflects list inclusion and the associated equivalence relation is permutation equivalence, i.e., equality up to a permutation of the list elements. Also in this model, modifications are just rewriting functions with the same operations as before.

In both algebras $P\Sigma$ and $L\Sigma$ distant idempotence models the fact that duplicating a feature has no effect. Hence idempotence seems of central interest in feature algebra. However, in the next section we will show that the axiom of distant idempotence yields problems in a model that considers more details.

5. THE LOST IDEMPOTENCE

As mentioned in the previous sections, distant idempotence (and hence idempotence), i.e., the fact that duplicating a feature has no effect, was of central interest in feature

[1] In contrast to [4], we use Waldmeister instead of Prover9 since it can handle multiple sorts. For feature algebra we use the two sorts M and I.

algebra. In [4], it is stated that languages and tools for feature combination usually have the idempotence property.

This works fine as long as a feature only contains the name and not its implementation. At the code level this property does not hold any longer. We illustrate this behaviour by a Java program.

EXAMPLE 5.1. *Consider a Java method* `foo` *given by*

```
void foo(int a) {
  a++;
  original(a);
}
```

When used in a feature superimposition, it updates a previous definition of `foo`*; the pseudo-statement* `original(a)` *inserts the original body. We assume further that* `foo` *is a method of the class* `Bar`. □

To integrate code into an FST, each terminal node has to be extended. According to this, we have also to extend the prefix-closed elements of the set $P\Sigma$. This is done as follows: Each letter (element of Σ) at the end of a maximal path is extended with code. This extension preserves that prefixes of paths are legal paths again. The following example should clarify the main idea; an abstract and more precise definition will be given below.

EXAMPLE 5.2. *With this explanation, the code of the previous example can be written as*

Bar::foo
```
void foo(int a) {
  a++;
  original(a);
}
```

To shorten the notation we write `Bar` :: `foo`$[A]$ *where* A *is an abbreviation for the complete code contained in the box.* □

As mentioned before, feature algebra was introduced as a formal treatment of FOSD and is intended to model *FeatureHouse* at an abstract level. If, however, two occurrences of the same feature appear in one program the code parts have to be merged and hence the order of combination does matter, since code parts of the are overwritten and/or updated. We skip the details how *FeatureHouse* merges code and applies overriding. Instead we illustrate the situation by an example.

EXAMPLE 5.3. *Using FeatureHouse leads to the following result:*

`Bar` :: `foo`$[A] \oplus$ `Bar` :: `foo`$[A]$

$=$

Bar::foo
```
void foo(int a) {
  a++;
  a++;
  original(a);
}
```

$\neq$ `Bar` :: `foo`$[A]$

where $\oplus$ *is the feature combination of FeatureHouse. In particular,* $\oplus$ *is not idempotent and therefore the axiom of distant idempotence does not hold.* □

This short example and this short application of *FeatureHouse* show that idempotence is not satisfied in general in the setting of FOSD. Moreover, either feature algebra is not the formal model for *FeatureHouse* or *FeatureHouse* does not follow the theoretical foundations introduced by the algebraic structure.

This section provided only a brief description and focussed on some parts of *FeatureHouse* and feature algebra which lead to discrepancies. It was not the intention to explain every fact of *FeatureHouse* and feature algebra in detail. The interested reader is referred to the references [1, 3].

6. EXTENDED FEATURE ALGEBRA

We have shown that the axioms of distant idempotence and hence also standard idempotence do not hold when arguing at code level. The remainder of the paper presents some ideas how to solve the described problems.

To model code-level behaviour at an abstract level we extend feature algebra by a third type C of *code fragments*.

We define the structure of an *extended feature algebra* as a tuple $(M, I, C, +, \circ, \cdot, |, 0, 1)$ with the following properties for all $m, n \in M$, $i, j \in I$ and $a, b, c \in C$:

- We consider pairs (i, c) where i is an introduction corresponding to a maximal path in the forest under consideration and c is the code fragment contained in the leaf at the tip of that path. We denote (i, c) by $i[c]$.[2] The set of all these pairs is denoted by $I[C]$.
- $(C, |)$ is a semigroup in which $|$ is an update or override operation (see below),
- $(I[C], +, 0)$ is a monoid satisfying $i[a] + j[c] + i[b] = j[c] + i[a|b]$,
- $(M, \circ, 1)$ is a groupoid operating via $\cdot$ on $I[C]$,
- 0 is a right-annihilator for $\cdot$ and
- $\cdot$ distributes over $+$.

The original definition of a feature algebra can be retrieved by choosing C as a set containing only one single element (the empty code fragment).

The operation $|$ can be seen as an update. In the previous section, $|$ merged code fragments. In the next section we will discuss this operation in our concrete models. Note that we have modified the axiom of distant idempotence: adding a feature a second time updates the earlier instance of that feature rather than just ignoring it.

Unfortunately, we cannot define a natural preorder on an extended feature algebra. This is due to the lack of idempotence. Hence the counterpart of Lemma 3.1 reduces to

LEMMA 6.1. *Assume* i, j *to be introductions,* m, n, o *to be modifications and assume* c *to be a code fragment of an extended feature algebra. Then*

1. $(m \circ (n \circ o)) \cdot i[c] = ((m \circ n) \circ o) \cdot i[c]$,
2. $(m \circ 1) \cdot i[c] = (1 \circ m) \cdot i[c] = m \cdot i[c]$.

[2]This fits well with the notation of the example of the previous section.

To overcome the deficiency of the missing preorder we can define two different relations:

$$i[a] \leq_r j[b] \quad =_{def} \quad \exists\, k[c] \in I[C] : i[a] + k[c] = j[b] \ ,$$
$$i[a] \leq_l j[b] \quad =_{def} \quad \exists\, k[c] \in I[C] : k[c] + i[a] = j[b] \ .$$

This implies immediately the following

LEMMA 6.2.

1. $\leq_l$, $\leq_r$ *are preorders*
2. $0 \leq_l i$, $0 \leq_r i$

Up to now we do not know which of the orders should be preferred. A further investigation of properties as well as the interaction of both preorders will be part of future research (cf. Section 8).

7. EXTENDING THE MODELS

In Section 5 we pointed out that *FeatureHouse* merges and updates code. Therefore a formal model should also reflect this behaviour. Unfortunately this does not hold for the models presented in Section 2, which led to our extension by code fragments.

In this section, we show how to define the update operator $|$ in our concrete models.

In particular, we will identify the "common part" of two given implementations of the same feature oriented program. Based on the common part one can determine which part of a method body has to be overridden and which part has to be preserved. Of course these calculations highly depend on the respective language and have to follow exact rules. In Java, for example, *FeatureHouse* simply overrides declarations and functions as long as the keyword `original` does not occur in the code.[3] For a detailed description we refer again to [1].

To model such behaviour we define *abstract interfaces* for each Java method. Whereas a general Java element may contain arbitrary (legal) programming constructs, abstract interfaces may contain only the types of the corresponding Java parts and "forgets" the remaining bodies, initialisations etc. We illustrate this behaviour by an example.

EXAMPLE 7.1. *On the left hand side there is a simple Java method while its abstract interface appears on the right hand side.*

```
int min5(int a) {
  int b=5;
  if(a<b) return a;
  else return b;
}
```

```
int min5(int a) {
  int b;
}
```

The typing of the local variable `b` *appears, since its declaration may be overwritten during a feature combination.* □

A precise definition of the abstract interface will need to reflect also nested scopes etc. The use of abstract interfaces may yield invalid Java code (e.g., `return` statements are omitted). This does not matter, though, since it will only be employed to identify the "common part".

Let again C be the set of possible code fragments and $T \subseteq C$ the set of the corresponding abstract interfaces. The function that determines the abstract interface for a given Java code is denoted by $ai : C \to T$. Next we define two functions

$$\begin{aligned} \ddagger, - \quad &: \quad \mathcal{P}(C) \times \mathcal{P}(T) \to \mathcal{P}(C) \\ X \ddagger U \quad &= \quad \{x \in X \mid ai(x) \in U\} \\ X - U \quad &= \quad \{x \in X \mid ai(x) \notin U\} \ . \end{aligned}$$

The restriction operator $\ddagger$ determines for a set X of code fragments the ones whose corresponding abstract interfaces lie in the given subset $U \subseteq T$, while the operator $-$ selects its relative complement.

To define the update function $|$ we need to lift the function ai to sets of code fragments by

$$ai(X) \ =_{def} \ \{ai(x) \mid x \in X\} \ .$$

Then

$$X|Y \ =_{def} \ (Y - ai(X)) \cup X \ .$$

This means that all "old" definitions of elements in Y that are redefined in X are discarded and replaced by the ones in X; moreover, all elements of X not mentioned in Y are added. It should be noted that ai and $|$ are closely related to the interface operator $\uparrow$ and the asymmetric composition $\&*$ in the DEEP calculus of [15].

It turns out that this rather concrete definition can be lifted to the same level of abstraction as that of our feature algebra, which again opens the possibility for automated verification. The key is the observation that $ai(X)$ is the least set that leaves X unchanged under the selection operation $\ddagger$:

$$X = X \ddagger U \ \Leftrightarrow \ ai(X) \subseteq U$$

In fact, since $X \ddagger U \subseteq U$ holds anyway by definition, this can be relaxed to

$$X \subseteq X \ddagger U \ \Leftrightarrow \ ai(X) \subseteq U \ .$$

Mathematically, this is known as a *Galois connection* (e.g. [8, 11]). Also, ai behaves in many respects like the abstract codomain operator of [12]. The definition of $|$ is similar to the ones based on relations or semirings with domain (e.g. [16, 13]). These correspondences allow us to re-use a large body of well-known theory — another advantage of an abstract algebraic view.

Let us detail this a bit more. We may abstract the set C of code fragments to a Boolean algebra L and the set T of abstract interfaces to a subalgebra N of L. Then the above functions can be characterised and generalised using the following axioms:

$$\begin{array}{l|l} (a+b) \ddagger p = a \ddagger p + b \ddagger p & (a+b) - p = (a-p) + (b-p) \\ a \ddagger 0 = 0 & a - 0 = a \\ a \ddagger (p+q) = a \ddagger p + a \ddagger q & a - (p+q) = (a-p) - q \\ & \qquad\quad = (a-q) - p \\ 0 \ddagger p = 0 & 0 - p = 0 \end{array}$$

$$\begin{array}{rcl} a \leq a \ddagger p & \Leftrightarrow & \overrightarrow{a} \leq p \\ a|b & = & (b - \overrightarrow{a}) + a \end{array}$$

where $a, b \in L$, $p, q \in N$ and $+$ denotes the supremum of L, $\leq$ its order, 0 the least element and $\overrightarrow{a}$ is the abstract counterpart of $ai(a)$.

Note that this section only gives the main ideas how to model the update operation in the abstract setting of an extended feature algebra. The work presented is part of ongoing work and will be investigated in much more detail (see the next section).

[3] There are some exceptions.

8. CONCLUSION AND OUTLOOK

The present paper is based on earlier work by Apel, Lengauer, Möller and Kästner [3]. They introduced a formal model to capture the commonalities of feature oriented software development such as introductions, refinements and quantification. We have defined a concrete model for feature algebra and have illustrated that the axioms of feature algebra are fine as long as one does not consider feature oriented programming at code level. Otherwise not all aspects of feature orientation can be modelled. To remedy this, we have introduced the structure of an extended feature algebra which generalises the original definition. To clarify the idea we have also extended the introduced models correspondingly. Finally we sketched how additional operators can be introduced to capture even more properties of feature oriented programming like updating or overriding.

This extended abstract is a further step towards an algebraic theory that covers all aspects of FOSD. Of course, all introduced operators like update need further investigation; in particular Section 7 reports about ongoing work. On the one hand more properties need to be derived; on the other hand it has to be checked whether the extension adequately covers the essential properties of FOSD, in particular, the merging of code fragments. If this turns out not to be the case, the extended feature algebra will need further modification.

Acknowledgement.
We are grateful to Han-Hing Dang, Roland Glück and the anonymous referees for fruitful comments.

9. REFERENCES

[1] 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.

[2] 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 *Lecture Notes in Computer Science*, pages 20–35. Springer, 2008.

[3] S. Apel, C. Lengauer, B. Möller, and C. Kästner. An algebra for features and feature composition. In *AMAST 2008: Proceedings of the 12th international conference on Algebraic Methodology and Software Technology*, volume 5140 of *Lecture Notes in Computer Science*, pages 36–50. Springer, 2008.

[4] S. Apel, C. Lengauer, B. Möller, and C. Kästner. An algebraic foundation for automatic feature-based program synthesis and architectural metaprogramming. *Science of Computer Programming*, 2009. (to appear).

[5] F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1999.

[6] D. Batory. From implementation to theory in product synthesis. *ACM SIGPLAN Notices*, 42(1):135–136, 2007.

[7] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions Software Engineering and Methodology*, 1(4):355–398, 1992.

[8] G. Birkhoff. *Lattice Theory*. American Mathematical Society, 3rd edition, 1967.

[9] A. Buch, T. Hillenbrand, and R. Fettig. Waldmeister: High Performance Equational Theorem Proving. In J. Calmet and C. Limongelli, editors, *Proceedings of the International Symposium on Design and Implementation of Symbolic Computation Systems*, number 1128 in Lecture Notes in Computer Science, pages 63–64. Springer, 1996.

[10] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. 2000.

[11] B. A. Davey and H. A. Priestley. *Introduction to lattices and order,*. Cambridge University Press, 2nd edition, 2002.

[12] J. Desharnais, B. Möller, and G. Struth. Kleene algebra with domain. *ACM Transactions on Computational Logic*, 7(4):798–833, 2006.

[13] T. Ehm. Pointer Kleene algebra. In R. Berghammer, B. Möller, and G. Struth, editors, *RelMiCS*, volume 3051 of *Lecture Notes in Computer Science*, pages 99–111. Springer, 2004.

[14] P. Höfner. Database for automated proofs of Kleene algebra. `http://www.dcs.shef.ac.uk/~georg/ka` (accessed October 1, 2009).

[15] D. Hutchins. Eliminating distinctions of class: Using prototypes to model virtual classes. In P. L. Tarr and W. R. Cook, editors, *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006*, pages 1–20. ACM Press, 2006.

[16] B. Möller. Towards pointer algebra. *Science of Computer Programming*, 21(1):57–90, 1993.