An Algebra for Features and Feature Composition

Sven Apel[†], Christian Lengauer[†], Bernhard Möller[‡], and Christian Kästner[⋆]

[†] Department of Informatics and Mathematics, University of Passau,
{apel,lengauer}@uni-passau.de
[‡] Institute of Computer Science, University of Augsburg,
moeller@informatik.uni-augsburg.de
[⋆] School of Computer Science, University of Magdeburg,
kaestner@iti.cs.uni-magdeburg.de

Abstract. *Feature-Oriented Software Development (FOSD)* provides a multitude of formalisms, methods, languages, and tools for building variable, customizable, and extensible software. Along different lines of research, different notions of a feature have been developed. Although these notions have similar goals, no common basis for evaluation, comparison, and integration exists. We present a feature algebra that captures the key ideas of feature orientation and provides a common ground for current and future research in this field, in which also alternative options can be explored.

1 Introduction

Feature-Oriented Software Development (FOSD) is a paradigm that provides formalisms, methods, languages, and tools for building variable, customizable, and extensible software. The main abstraction mechanism of FOSD is the *feature*. A feature reflects a stakeholder's requirement and is an increment in functionality; features are used to distinguish between different variants of a program or software system [1]. *Feature composition* is the process of composing code associated with features consistently.

Research along different lines has been undertaken to realize the vision of FOSD [1,2,3,4,5]. While there are the common notions of a feature and feature composition, present approaches use different techniques, representations, and formalisms. For example, AspectJ[1] and AHEAD[2] can both be used to implement features, but they provide different language constructs: on the one hand pointcuts, advice, and inter-type declarations, and on the other hand collaborations and refinements [5]. A promising way of integrating the separate lines of research is to provide an encompassing abstract framework that captures many of the common ideas like introductions, refinements, or quantification and hides (what we feel are) distracting differences.

We propose a first step toward such a framework for FOSD: a *feature algebra*. Firstly, the feature algebra abstracts from the details of different programming languages and environments used in FOSD. Secondly, alternative design decisions in the algebra, e.g., allowing terminal composition or not, reflect variants and alternatives in concrete

[1] `http://www.eclipse.org/aspectj/`
[2] `http://www.cs.utexas.edu/~schwartz/ATS.html`

programming language mechanisms. Thirdly, the algebra is useful for describing, beside composition, also other operations on features formally and independently of the language, e.g., type checking [6] and interaction analysis [7]. Fourthly, the algebraic description of a software system can be taken as an architectural view. External tools can use the algebra as a basis for feature expression optimization [4, 8].

We introduce a uniform representation of features, outline the properties of the algebra, and explain how the algebra models the key concepts of FOSD.

2 What is a Feature?

Different researchers have been proposing different views of what a feature is or should be. A definition that is common to most (if not all) work on FOSD is: *a feature is a structure that extends and modifies the structure of a given program in order to satisfy a stakeholder's requirement, to implement a design decision, and to offer a configuration option*. This informal definition guides our work on a formal framework of FOSD.

Typically, a series of features is composed to form a final program, which is itself a feature. This way, a feature can be either a complete program which can be executed or a program increment which requires further features to form a complete program.

Mathematically, we describe feature composition by the operator $\bullet$, which is defined over the set F of features. Typically, a program p (which is itself a feature) is composed of a series of simpler features:

$$\bullet : F \times F \rightarrow F \qquad\qquad p = f_n \bullet f_{n-1} \bullet \ldots \bullet f_2 \bullet f_1 \tag{1}$$

The order of features in a composition matters since feature composition is not commutative, and parenthesization does not matter since feature composition is associative, as we will show.

For simplicity, we restrict feature composition such that each single feature can appear only once in a feature expression. Multiple instances of a single feature would be possible but do not add anything new.

3 The Structure of Features

We develop our model of features in several steps and – even though the algebra is language-independent – explain the details of the algebra and its implications by means of Java code. First, a simple form of features, which we call *basic features*, are introduced as trees that describe the collection of elementary components of an artifact, such as classes, fields, or methods in Java (Sec. 3–5.1). In the next step, we introduce *modifications* that act as rewrites on basic features (Sec. 5.2). Finally, *full features* are defined as tuples, called *quarks*, consisting of both, a basic feature and modifications (Sec. 6). Quarks can be composed to describe complex features in a structured way as compositions of sequences of simpler features.

A basic feature consists of one or more source code artifacts, each of which can have an internal structure. We model the structure of a basic feature as a tree, called *feature structure tree* (*FST*), that organizes the feature's structural elements, e.g., classes, fields, or methods, hierarchically. Figure 1 depicts an excerpt of the Java implementation of a feature *Base* and its representation in form of an FST. One can think of an FST as a stripped-down abstract syntax tree; however, it contains only the information that is necessary for the specification of the structure of a basic feature. The nature of this information depends on the degree of granularity at which software artifacts shall be composed, as we discuss below.

```java
package util;
class Calc {
  int e0=0, e1=0, e2=0;
  void enter(int v) { e2=e1; e1=e0; e0=v; }
  void clear() { e0=e1=e2=0; }
  String top() { return String.valueOf(e0); }
}
```

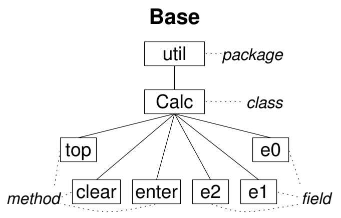

Fig. 1. Implementation and FST of the feature *Base*.

For example, the FST we use to represent Java code contains nodes that denote packages, classes, interfaces, fields, and methods, etc. It does not contain information about the internal structure of methods, etc. A different granularity would be to represent only packages and classes but not methods or fields as FST nodes, or to represent statements or expressions as well [9]. However, this decision does not affect our description of the algebra.

Furthermore, a name[3] and type information is attached to each node of an FST. This helps to prevent the composition of incompatible nodes during feature composition, e.g., the composition of two classes with different names, or of a field with a method of the same name.

The rightmost child of a node represents the topmost element in the lexical order of an artifact, e.g., the first member in a class is represented by the rightmost child node. Note that in the chosen granularity for Java the order could be arbitrary, but this is different at a finer granularity (the order of statements matters) and may differ in other languages (the order of XHTML elements matters).

4 Feature Composition

How does the abstract description of a feature composition $g \bullet f$ map to the concrete composition at the structural level? That is, how are FSTs composed in order to obtain a new FST? Our answer is by *FST superimposition* [10, 11, 12, 3].

4.1 Tree Superimposition

The basic idea is that two trees are superimposed by superimposing their subtrees, starting from the root and proceeding recursively. Two nodes are superimposed to form

[3] Depending on the language, a name could be a simple identifier, a signature, etc.

a new node (a) when their parents have been superimposed previously or both are root nodes and (b) when they have the same name and type. If two nodes have been superimposed, the whole process proceeds with their children. If not, they are added as separate child nodes to the superimposed parent node. This recurses until all leaves have been processed.

According to the semantics of FSTs (see Sec. 3), the children are superimposed beginning with the rightmost node preserving the order in the superimposed FST; nodes that have not been superimposed are added to the left.

Figure 2 illustrates the process of FST superimposition; our feature *Base* is superimposed with a feature *Add*. The result is a new feature, which we call *AddBase*. Note that the new method `add` appears to the left in *AddBase*.

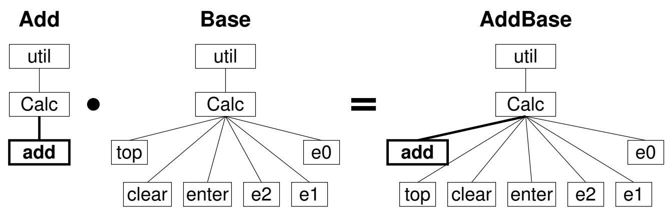

Fig. 2. An example of FST superimposition (*Add* • *Base* = *AddCalc*).

4.2 Terminal and Non-Terminal Nodes

Independently of any particular language, an FST is made up of two different kinds of nodes:

Non-terminal nodes are the inner nodes of an FST. The subtree rooted at a non-terminal node reflects the structure of some implementation artifact of a feature. The artifact structure is regarded as *transparent* (substructures are represented by child nodes) and is subject to the recursive superimposition process. A non-terminal node has only a name and a type, i.e., no superimposition of additional content is necessary.

Terminal nodes are the leaves of an FST. Conceptually, a terminal node may also be the root of some structure, but this structure is regarded as *opaque* in our model (substructures are not represented by child nodes). The content of a terminal is not shown in the FST. A terminal node has a name, a type, and usually some content.

While superimposition of two non-terminals continues the recursive descent in the FSTs, the superimposition of two terminals terminates the recursion and requires a special treatment that may differ for each type of node.

Let us illustrate these concepts for Java. In Java, packages, classes, and interfaces are represented by non-terminals. The implementation artifacts they contain are represented by child nodes, e.g., a package contains several classes and classes contain inner classes, methods, and fields. Two compatible non-terminals are superimposed by superimposing their child nodes, e.g., two packages with equal names are merged into one package that contains the superimposition of the child elements (classes, interfaces, subpackages) of the two original packages. In contrast, Java methods, fields, imports, modifier lists, and `extends`, `implements`, and `throws` clauses are represented by terminals (the leaves of an FST), at which the recursion terminates. For each type of terminal node there needs to be a language-specific rule for superimposing their content.

4.3 Superimposition of Terminals

In order to superimpose terminals, each terminal type has to provide its own rule for superimposition. Here are four examples for Java and similar languages:

- Two methods can be superimposed if it is specified how the method bodies are superimposed (e.g., by overriding and calling the original method by using the keywords `original` [13] or `Super` [3] inside a method body). It is a question of programming style whether to allow or disallow replacement of method bodies (i.e., overriding without calling the original method).
- Two fields are superimposed by replacing one initializing variable declaration with the other or by requiring that at most one of the fields may have an initial value.
- Two `implements`, `extends`, or `throws` clauses are superimposed by concatenating their entries and removing duplicates.
- Two modifier lists are superimposed by a specific algorithm, e.g., `public` replaces `private`, but not vice versa; a modifier list containing `static` superimposed with one not containing `static` is an error; and so on.

Terminal types that do not provide a rule cannot be composed – an error is displayed.

4.4 Discussion

Superimposition of FSTs requires several properties of the language in which the elements of a feature are expressed:

1. The substructure of a feature must be hierarchical, i.e., a general tree.
2. Every structural element of a feature must have a name and type that become the name and type of the node in the FST.
3. An element must not contain two or more direct child elements with the same name and type.
4. Elements that do not have a hierarchical substructure (terminals) must provide superimposition rules, or cannot be superimposed.

These constraints are usually satisfied by contemporary programming languages. But also other (non-code) languages align well with them [3, 14]. Languages that do not satisfy these constraints are not "feature-ready", since they do not provide sufficient structural information. However, it may be possible to make them so by extending them with an overlaying module structure [14].

FST superimposition is associative only if the superimposition of the individual subtrees is associative and, to this end, if merging terminal content is associative. In order to retain associativity, we add a further constraint: superimposition rules of terminals must be associative. This constraint, too, is typically satisfied by contemporary programming languages.

5 Feature Algebra

Our feature algebra models features and their composition on top of FSTs. The elements of an algebraic expression correspond to the elements of an FST. The manipulation of an expression implies a manipulation of one or more FSTs. The changes of an algebraic expression are propagated to the associated feature implementations at code level.

An important design decision is that there is a one-to-one correspondence between an FST and its algebraic expression.[4] That is, the expression is a formal means for reasoning about the FST. Thus, FSTs can be converted, without information loss, to algebraic expressions and vice versa. Our laws for algebraic expressions describe what is allowed and disallowed when manipulating FSTs.

5.1 Introductions

For the purpose of expressing basic features and their composition, we use the notion of an atomic introduction. An *atomic introduction* is a constituent of the implementation of a basic feature that corresponds to a node in the FST, e.g., a method, field, class, or package. When composing two basic features, introductions are the elementary units of difference of one feature composed with another feature. A basic feature is represented by the superimposition of all paths / atomic introductions in its FST. We model the superimposition of FSTs via the operation of *introduction sum*.

Introduction Sum

Introduction sum $\oplus$ is a binary operation defined over the set I of introductions. The result of an introduction sum is again an (non-atomic) introduction. Thus, an FST can be represented in two ways: by the individual (atomic) summands and by a metavariable that represents the sum:

$$\oplus : I \times I \to I \qquad\qquad i_2 \oplus i_1 = i \tag{2}$$

During composition, for each metavariable i, the individual atomic summands $i_2 \oplus i_1$ are preserved. That is, introduction sum retains information about the summands, which is useful for expression manipulation and code generation. Since the nodes of an FST are unique, the atomic summands of a sum of introductions are unique as well, as we will explain shortly.

In order to process algebraic expressions of features, we flatten the hierarchical structure of FSTs. That is, we convert the tree representation of an FST into a set of atomic introductions, one per FST node. But, in order not to lose information about which structural elements contain which other elements, we preserve the paths of the FST nodes.

Specifically, we use a simple prefix notation to identify an atomic introduction, similar to fully qualified names in Java: the name of the FST node is prefixed with the

[4] A one-to-one correspondence for Java was only possible by ordering the children of a node based on their lexical order (see Sec. 3).

name of all its parent nodes, separated by dots.[5] The leftmost prefix contains the name of the feature an introduction belongs to, followed by an '::', although, for brevity, the prefix does not appear in the FST. Our feature *Base* (cf. Fig. 2) is denoted in path notation as follows:

$$\begin{aligned} Prog = {} & Base::util.Calc.top \oplus Base::util.Calc.clear \oplus Base::util.Calc.enter \\ & \oplus Base::util.Calc.e2 \oplus Base::util.Calc.e1 \oplus Base::util.Calc.e0 \\ & \oplus Base::util.Calc \oplus Base::util \end{aligned}$$

The leftmost leaves of an FST become the leftmost summands of its introduction sum. Note that not every sum represents a valid FST. A well-formedness rule is, that for every dot-separated prefix of a summand, there is a summand with the same name, e.g., the prefix $Base::util$ of $Base::util.Calc$ is itself a summand.

Two features are composed by adding their atomic introductions. Since each atomic introduction preserves the path of the corresponding FST node, it is always known from which feature an introduction was added during the manipulation of an algebraic expression, e.g., *Base* in $Base::util.Calc$. Furthermore, we can convert each algebraic expression (containing a sum of introductions with prefixes) straightforwardly back to a tree, either to the original FSTs or to a new composed FST. When converting an introduction sum into a composed FST, it is associated with a new (composed) feature. Two atomic introductions with the same fully qualified name, that belong to different features, are composed via superimposition, as explained informally in Section 4.

For example, the introduction sum that represents the non-terminal superimposition of Figure 2 is as follows:

$$Add::util.Calc.add \oplus \ldots \ \oplus \ Base::util.Calc.top \oplus Base::util.Calc.clear \oplus \ldots$$

It follows that the above sum represents a composed FST consisting of a package `util` with a class `Calc` that contains four methods (including `add`) and three fields.

For example, the superimposition of the two methods `enter` is represented in the corresponding introduction sum as:

$$Count::util.Calc.enter \oplus \ldots \ \oplus \ Base::util.Calc.enter \oplus \ldots$$

This sum represents a composed FST (only an excerpt is shown) consisting of a package `util` with a class `Calc` that contains three methods and three fields, and the bodies of the two `enter` methods are merged (similarly for `clear`).

Algebraic Properties

Introduction sum $\oplus$ over the set I of introductions forms a *non-commutative idempotent monoid* $(I, \oplus, \xi)$:[6]

[5] To be specific, the fully qualified name of an atomic introduction must also include the type of each path element. For lack of space and because there are no ambiguities in our examples, we omit the type information here.

[6] All standard definitions of algebraic structures and properties are according to Hebisch and Weinert [15].

Associativity: $(k \oplus j) \oplus i = k \oplus (j \oplus i)$ — Introduction sum is associative because FST superimposition is associative. This applies for terminals and non-terminals.

Identity: $\xi \oplus i = i \oplus \xi = i$ — ξ is the empty introduction, i.e., an FST without nodes.

Non-commutativity: Since we consider superimposition of terminals, introduction sum is not generally commutative. We consider the right operand to be introduced first, the left one is added to it.

Idempotence: $i \oplus j \oplus i = j \oplus i$ — Only the rightmost occurrence of an introduction i is effective in a sum, because it has been introduced first. That is, duplicates of i have no effect, as stressed at the end of Section 2. We refer to this rule as *distant idempotence*. For $j = \xi$, *direct idempotence* ($i \oplus i = i$) follows.

5.2 Modification

Beside superimposition also other techniques for feature composition have been proposed, most notably *composition by quantification* [16,5]. The idea is that, when expressing the changes that a feature causes to another feature, we specify the points at which the two features are supposed to be composed. This idea has been explored in depth in work on *subject-oriented programming* [17] and *aspect-oriented programming* [18]. The process of determining where two features are to be composed is called *quantification* [19]. In the remainder, we distinguish between two approaches of composition: *composition by superimposition* and *composition by quantification*. Our definition of feature composition ($\bullet$) incorporates both (see Sec. 6).

In order to model composition by quantification, we introduce the notion of a modification. A *modification* consists of two parts:

1. A specification of the nodes in the FST at which a feature affects another feature during composition.
2. A specification of how features are composed at these nodes.

In the context of our model, a modification is performed as an FST walk that determines the nodes which are being modified and applies the necessary changes to these nodes. The advantage of composition by quantification is that the specification of where a program is extended is declarative. Querying an FST can return more than one node at a time. This allows us to specify the modification of a whole set of nodes at once without having to reiterate it for every set member.

Note that composition by superimposition and composition by quantification are siblings. Quantification enables us to address parts of a feature more generically than superimposition. But, once it is known which points have to be changed, the two kinds of composition become equivalent. We have observed their conceptual duality before, but at the level of two concrete programming techniques [5]. The feature algebra makes it explicit at a more abstract level.

Semantics of Modification

A modification m consists of a query q that selects a subset of the atomic introductions of an introduction sum and a definition of change c that will be used to effect the desired changes:

$$m = (q, c) \tag{3}$$

Query. A simple query can be represented by an FST in which the node names may contain wildcards.[7] For example, the query q with the search expression '$util.Calc.*$' applied to our example would return the sum of all introductions that are members of the class `Calc`. This motivates the following definition.

Formally, a query applied to an atomic introduction returns either the same introduction or the empty introduction:

$$q(i) = \begin{cases} i, & \text{when } i \text{ is matched by } q \\ \xi, & \text{when } i \text{ is not matched by } q \end{cases} \tag{4}$$

A query applied to an introduction sum queries each summand:

$$q(i_n \oplus \ldots \oplus i_2 \oplus i_1) = q(i_n) \oplus \ldots \oplus q(i_2) \oplus q(i_1) \tag{5}$$

Definition of change. An introduction i selected by a query is modified according to the modification's definition of change c; c is a rewrite that is able to apply two kinds of changes: (a) it can add a new child to a given non-terminal and (b) it can alter the content of a terminal; the application of c distributes over introduction sum:

$$c(i_n \oplus \ldots \oplus i_2 \oplus i_1) = c(i_n) \oplus \ldots \oplus c(i_2) \oplus c(i_1) \qquad c(i_j) = \tau_c(i_c, i_j) \oplus i_j \tag{6}$$

The *atomic* introduction i_c represents a new child or the change applied to a terminal. It has to be provided by the programmer in the form of a generic piece of code or some other kind of specification. The function τ_c takes the generic definition of change i_c and the atomic introduction i_j to be changed and generates the final non-generic definition of change. That is, τ_c eliminates the genericity of i_c by substituting missing parts with details of the program to which c is applied.

For example, suppose a feature *Count* applies two modifications m_1 and m_2 to the introductions of *Base*, with c_1 adding a new field and c_2 altering the method `enter`:

$$\begin{aligned} c_1(Base::util.Calc) &= \tau_{c_1}(count, util.Calc) \oplus Base::util.Calc \\ &= Count::util.Calc.count \oplus Base::util.Calc \\ c_2(Base::util.Calc.enter) &= \tau_{c_2}(enter, util.Calc.enter) \oplus Base::util.Calc.enter \\ &= Count::util.Calc.enter \oplus Base::util.Calc.enter \end{aligned}$$

Of course, applying c_1 and c_2 to a different feature (say *Base*$_2$) results in a different program. Since change is expressed as an introduction sum, a modification cannot delete nodes. The changes a feature can make via modifications are similar to the ones possible via introduction sum, but expressed differently.

Modification Application and Composition

For simplicity, we usually hide the steps of querying and applying the changes. We define an operator *modification application* ($\odot$) over the set M of modifications and

[7] In practice, queries with regular expressions or queries over types might be useful.

the set I of introductions. A modification applied to an introduction returns either the introduction again or the introduction that has been changed:

$$\odot : M \times I \rightarrow I \qquad m \odot i = (q, c) \odot i = \begin{cases} c(i), \ q(i) = i \wedge i \neq \xi \\ \ i, \quad q(i) = \xi \end{cases} \tag{7}$$

A consequence of this definition is that a modification cannot extend the empty introduction, i.e., the empty program. This is different from introduction sum which we can use to extend empty programs. While this fact is just a result of our definition, it reflects what contemporary languages that support quantification are doing, e.g., AspectJ's advice and inter-type declarations cannot extend the empty program.

A modification is applied to a sum of introductions by applying it to each introduction in turn and summing the results:

$$m \odot (i_n \oplus \ldots \oplus i_2 \oplus i_1) = (m \odot i_n) \oplus \ldots \oplus (m \odot i_2) \oplus (m \odot i_1) \tag{8}$$

The successive application of changes of a modification to an introduction sum implies the left distributivity of $\odot$ over $\oplus$.

Furthermore, the operator $\odot$ is overloaded.[8] With two modifications as arguments, it denotes the operation *modification composition*. The semantics of modification composition is that the right operand is applied to an introduction, and then the left operand to the result:

$$\odot : M \times M \rightarrow M \qquad (m_2 \odot m_1) \odot i = m_2 \odot (m_1 \odot i) \tag{9}$$

Here, the leftmost of the four occurrences of $\odot$ is modification composition, all others are modification application.

A fully precise definition of modification composition requires an elaborate construction for combining the queries involved. Due to lack of space, we refer the reader to a technical report [20, pp. 14ff].

Using modification composition, a series of modifications can be applied to an introduction step by step:

$$(m_n \odot \ldots \odot m_2 \odot m_1) \odot i = m_n \odot (\ldots \odot (m_2 \odot (m_1 \odot i)) \ldots) \tag{10}$$

Note that the application of a modification may add new introductions that can be changed subsequently by other modifications. But, as prescribed by Equation 6, it is not possible to change an introduction sum such that some introductions are removed and the modifications applied subsequently cannot affect them anymore. This design decision is justified by the design of current languages that support feature composition, e.g., AspectJ's aspects or AHEAD's refinements [3] cannot remove members or classes.

Algebraic Properties

We define two modifications m_1 and m_2 as *equivalent* if they act identically on all introductions, i.e., if $m_1 \odot i = m_2 \odot i$ for all i. In the following, we write M also for

[8] We reuse the symbol $\odot$ because introduction sum and modification application and composition become all integrated into one algebraic structure with identical operator symbols for application and composition (see Sec. 5.3).

the set of equivalence classes of modifications and $\odot$ for the corresponding induced operation on them. This induces a *non-commutative non-idempotent monoid* $(M, \odot, \zeta)$:

Associativity: $(o \odot n) \odot m = o \odot (n \odot m)$ — Modification composition is associative by the definition of modification application.

Identity: $\zeta \odot m = m \odot \zeta = m$ — ζ is the equivalence class of empty modifications. ζ does not change a given introduction.

Non-commutativity: Modification composition is not commutative because introduction sum is not commutative.

Non-idempotence: Although the changes made by a modification reduce to introduction sum (cf. Eq. 6), and introduction sum is distantly idempotent, the consecutive application of several modifications is *not* idempotent. The reason is that a modification m can add an introduction that is selected and changed by itself when applied repeatedly.

5.3 Introductions and Modifications in Concert

In order to describe feature composition, our algebra integrates our two algebraic structures $(I, \oplus, \xi)$ and $(M, \odot, \zeta)$ by means of the operation of modification application.

$(I, \oplus, \xi)$ induces a non-commutative idempotent monoid and $(M, \odot, \zeta)$ induces a non-commutative non-idempotent monoid. A notable property of $(I, \oplus, \xi)$ is that it is a *semimodule over the monoid* $(M, \odot, \zeta)$ since the distributive and associative laws (8) and (9) hold. In fact, the operation of modification application induces the semimodule on top of the individual operations introduction sum and modification composition. A semimodule over a monoid is related to a vector space but weaker (modification application plays the role of the scalar product) [15]. In a vector space, there would be an operation of *modification sum* that adds modifications similarly to introduction sum. In prior work, we have explored and integrated modification sum into the feature algebra [20] but, due to lack of space, we omit its description here. Moreover, the additive and multiplicative operations in vector spaces are commutative and there are inverse elements with respect to addition and multiplication. Nevertheless, the semimodule property guarantees a pleasant and useful flexibility of feature composition, which is manifested in the associativity and distributivity laws.

6 The Quark Model

So far, we have introduced two sets (I and M) and three operations ($\oplus : I \times I \rightarrow I$, $\odot : M \times M \rightarrow M$, and $\odot : M \times I \rightarrow I$) for feature composition. Now we integrate them in a compact and concise notation. This way, we allow *full features* that involve both introductions and modifications. Furthermore, we need to distinguish between local and global modifications. For this purpose, we introduce the *quark model*.[9]

[9] The idea and name of the quark model are due to Don Batory. Subsequently, the model was developed further in cooperation with us [20]. The term ‘quark’ was chosen as an analogy to the physical particles in quantum chromodynamics. Originally, quarks have been considered to be fundamental, but newer theories, e.g., preon or string theory, predict a further substructure.

A *quark* is a triple that represents a full feature, which consists of a composition g of *global* modifications, a sum i of introductions, and a further composition l of *local* modifications:

$$f = \langle g, i, l \rangle = \langle g_j \odot \ldots \odot g_1, i_k \oplus \ldots \oplus i_1, l_m \odot \ldots \odot l_1 \rangle \tag{11}$$

Here, i is the introduction sum of feature f and represents an FST; l and g contain the modifications that the feature f can make. A basic feature is represented in the quark model as a triple $\langle \zeta, i, \zeta \rangle$ where ζ is the empty modification. The application of quark q to introduction i is defined as the composition $q \bullet \langle \zeta, i, \zeta \rangle$.

When two quarks are composed, a new quark is constructed following certain composition rules. The new introduction part of the quark is constructed using modification application and introduction sum, while the new modification parts result by modification composition. We distinguish between two kinds of modifications because there are two options of using modifications when composing quarks: (a) *Local modifications* (l) can affect only already present introductions of features. (b) *Global modifications* (g) can affect also introductions that are just being constructed during the composition. For quarks that represent basic features (g and l are empty) both definitions (a) and (b) yield $\langle \zeta, i_2, \zeta \rangle \bullet \langle \zeta, i_1, \zeta \rangle = \langle \zeta, i_2 \oplus i_1, \zeta \rangle$, which in retrospect justifies our use of $\bullet$ also for FST superimposition in Section 4.

The difference between local and global modifications requires a special treatment of composition of full quarks. When composing a sequence of quarks, we can apply the local modifications immediately. We cannot apply the global modifications immediately. We have to wait until all introductions and local modifications in a series of quarks have been composed; only then we can apply all global modifications. So, we generalize the binary operator $\bullet$ to an n-ary one:

$$\begin{aligned} f_n \bullet \ldots \bullet f_2 \bullet f_1 &= \langle g_n, i_n, l_n \rangle \bullet \ldots \bullet \langle g_2, i_2, l_2 \rangle \bullet \langle g_1, i_1, l_1 \rangle \\ &= \langle g_n \odot \ldots \odot g_1, (g_n \odot \ldots \odot g_1) \odot \\ &\quad (i_n \oplus (l_n \odot (\ldots (i_2 \oplus (l_2 \odot i_1))))), l_n \odot \ldots \odot l_1 \rangle \end{aligned} \tag{12}$$

This does not mean that the associativity properties of introduction sum and modification composition are useless. Associativity is necessary to make the application of local modifications to sums of introductions work smoothly.

7 Related Work

Lopez-Herrejon, Batory, and Lengauer model features as functions and feature composition as function composition [21, 7]. They distinguish between introductions and advice, which correspond roughly to our introductions and modifications. However, in their work there is no semantic model that defines precisely what introductions and advice are. In our feature algebra, we define introductions in terms of FSTs and modifications in terms of tree walks. This enables us to bridge the gap between algebra and implementation.

Möller et al. have developed an algebra for expressing software and hardware variabilities in the form of features [22]. This has recently been extended [23] to express a limited form of feature interaction. However, their algebra does not consider the structure and implementation of features.

There are some calculi that support feature-like structures and composition by superimposition [24, 25, 26, 27, 28, 29]. These calculi are typically tailored to Java-like languages and emphasize the type system. Instead, our feature algebra enables reasoning about feature composition on a more abstract level. We emphasize the structure of features and their static composition, independently of a particular language.

Several languages support features and their composition by superimposition [30, 31, 13, 32, 3]. Our algebra is a theoretical backbone that underlies and unifies all these languages. It reveals the properties a language must have in order to be feature-ready. Several languages exploit the synergistic potential of superimposition and quantification [16,17,32,5]. The feature algebra allows us to study their relationship and integration, independently of a specific language.

Features are implemented not only by source code. Some tools support the feature-based composition of non-source code artifacts [3, 33, 14]. Our algebra is general enough to describe a feature containing these non-code artifacts since all their representations can be mapped to FSTs.

Finally, we have implemented a tool, called *FSTComposer*, that implements feature composition as described by our algebra [34]. With it, we have been able to demonstrate that such different languages as Java, XML, or Bali can be treated uniformly and the composition scales to medium-sized software projects. The integration of a new language requires only marginal effort because most information can be inferred from the language's grammar.

8 Conclusions & Perspectives

We have presented a model of FOSD in which features are represented as FSTs and feature composition is expressed by tree superimposition and tree walks. This reflects the state of the art in programming languages and composition models that favor superimposition and quantification. Our algebra describes precisely what their properties are and how such concepts from FOSD languages like aspects, collaborations, or refinements can be integrated. Though some of these approaches were integrated before in concrete languages, e.g., in FeatureC++ [32], aspectual feature modules [5], or Caesar [16], the algebra integrates these approaches for the first time formally and exposes fundamental concepts like the distinction of local vs. global modifications that prompted controversial discussions in earlier work, e.g., [21].

Our feature algebra forms a semimodule over a monoid, which is a weaker form of a vector space. The flexibility of this algebraic structure suggests that our decisions regarding the semantics of introductions and modifications and their operations are not arbitrary. With the presented configuration of our algebra, we achieve a high flexibility in feature composition, which is manifested in the associativity and distributivity laws.

Although our algebra is quite flexible, we also made several restrictive decisions. For example, introduction sum is idempotent and modifications are only allowed to add children and to compose content of terminals. An advantage of an algebraic approach is that we can evaluate the effects of our and alternative decisions directly by examining the properties of the resulting algebra. For example, if we forbid superimposition of terminals we can achieve commutativity of feature composition. Although this design decision might appear trivial, it is not obvious from contemporary programming languages but rather appears to be a byproduct of integrating other language constructs. With our formalization, such consequences become obvious and are helpful for carefully balancing expressiveness and composition flexibility when designing a new language. In our algebra, we decided to abandon commutativity in order to increase the expressive power of introduction sum by including overriding. Likewise, disallowing modifications to remove nodes from an FST guarantees that the targets of a feature remain present in a composition. Exploring the implications of our and alternative decisions is a promising avenue of further work.

Finally, with the feature algebra, we provide a framework for feature composition that is independent of a concrete language. Based on this framework, we have built the language-independent composition tool FSTComposer. Uniformity in feature composition has been a long-standing goal of FOSD [3] but, until now, feature composition tools for new languages were usually developed ad-hoc. In future work, we will also use the algebra for reasoning about types [6] and for interaction analysis [7] independently of concrete language mechanisms, e.g., of AspectJ or AHEAD.

Acknowledgments

We thank Don Batory, Tony Hoare, and Peter Höfner for helpful comments.

References

1. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, SEI, CMU (1990)
2. Prehofer, C.: Feature-Oriented Programming: A Fresh Look at Objects. In: Proc. Europ. Conf. Object-Oriented Progr. (1997)
3. Batory, D., Sarvela, J., Rauschmayer, A.: Scaling Step-Wise Refinement. IEEE Trans. Softw. Eng. **30**(6) (2004)
4. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley (2000)
5. Apel, S., Leich, T., Saake, G.: Aspectual Feature Modules. IEEE Trans. Softw. Eng. **34**(2) (2008)
6. Thaker, S., Batory, D., Kitchin, D., Cook, W.: Safe Composition of Product Lines. In: Proc. Int'l. Conf. Generative Program. and Component Eng. (2007)
7. Liu, J., Batory, D., Lengauer, C.: Feature-Oriented Refactoring of Legacy Applications. In: Proc. Int'l. Conf. Softw. Eng. (2006)
8. Batory, D.: From Implementation to Theory in Product Synthesis (Keynote). In: Proc. Int'l. Symp. Principles of Program. Lang. (2007)
9. Kästner, C., Apel, S., Kuhlemann, M.: Granularity in Software Product Lines. Proc. Int'l. Conf. Softw. Eng. (2008)
10. Chandy, M., Misra, J.: An Example of Stepwise Refinement of Distributed Programs: Quiescence Detection. ACM Trans. Program. Lang. Syst. **8**(3) (1986)
11. Ossher, H., Harrison, W.: Combination of Inheritance Hierarchies. In: Proc. Int'l. Conf. Object-Oriented Progr., Syst., Lang., and App. (1992)

12. Bosch, J.: Super-Imposition: A Component Adaptation Technique. Information and Software Technology **41**(5) (1999)
13. Bergel, A., Ducasse, S., Nierstrasz, O.: Classbox/J: Controlling the Scope of Change in Java. In: Proc. Int'l. Conf. Object-Oriented Progr., Syst., Lang., and App. (2005)
14. Anfurrutia, F., Díaz, O., Trujillo, S.: On Refining XML Artifacts. In: Proc. Int'l. Conf. Web Eng. (2007)
15. Hebisch, U., Weinert, H.: Semirings. World Scientific (1998)
16. Mezini, M., Ostermann, K.: Variability Management with Feature-Oriented Programming and Aspects. In: Proc. Int'l. Symp. Foundations of Softw. Eng. (2004)
17. Tarr, P., Ossher, H., Harrison, W., Sutton, Jr., S.: N Degrees of Separation: Multi-Dimensional Separation of Concerns. In: Proc. Int'l. Conf. Softw. Eng. (1999)
18. Masuhara, H., Kiczales, G.: Modeling Crosscutting in Aspect-Oriented Mechanisms. In: Proc. Europ. Conf. Object-Oriented Progr. (2003)
19. Filman, R., Friedman, D.: Aspect-Oriented Programming Is Quantification and Obliviousness. In: Aspect-Oriented Software Development. Addison-Wesley (2005)
20. Apel, S., Lengauer, C., Batory, D., Möller, B., Kästner, C.: An Algebra for Feature-Oriented Software Development. Technical Report MIP-0706, Department of Informatics and Mathematics, University of Passau (2007)
21. Lopez-Herrejon, R., Batory, D., Lengauer, C.: A Disciplined Approach to Aspect Composition. In: Proc. Int'l. Symp. Partial Evaluation and Semantics-Based Program Manipulation. (2006)
22. Höfner, P., Khedri, R., Möller, B.: Feature Algebra. In: Proc. Int'l. Symp. Formal Methods. (2006)
23. Höfner, P., Khedri, R., Möller, B.: Algebraic View Reconciliation. Technical Report 2007-13, Institute of Computer Science, University of Augsburg (2007)
24. Apel, S., Kästner, C., Lengauer, C.: An Overview of Feature Featherweight Java. Technical Report MIP-0802, Department of Informatics and Mathematics, University of Passau (2008)
25. Hutchins, D.: Eliminating Distinctions of Class: Using Prototypes to Model Virtual Classes. In: Proc. Int'l. Conf. Object-Oriented Progr., Syst., Lang., and App. (2006)
26. Ernst, E., Ostermann, K., Cook, W.: A Virtual Class Calculus. In: Proc. Int'l. Symp. Principles of Program. Lang. (2006)
27. Flatt, M., Krishnamurthi, S., Felleisen, M.: Classes and Mixins. In: Proc. Int'l. Symp. Principles of Program. Lang. (1998)
28. Findler, R., Flatt, M.: Modular Object-Oriented Programming with Units and Mixins. In: Proc. Int'l. Conf. Functional Program. (1998)
29. Odersky, M., Cremet, V., Röckl, C., Zenger, M.: A Nominal Theory of Objects with Dependent Types. In: Proc. Europ. Conf. Object-Oriented Progr. (2003)
30. Odersky, M., Zenger, M.: Scalable Component Abstractions. In: Proc. Int'l. Conf. Object-Oriented Progr., Syst., Lang., and App. (2005)
31. McDirmid, S., Flatt, M., Hsieh, W.: Jiazzi: New-Age Components for Old-Fashioned Java. In: Proc. Int'l. Conf. Object-Oriented Progr., Syst., Lang., and App. (2001)
32. Apel, S., Leich, T., Rosenmüller, M., Saake, G.: FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In: Proc. Int'l. Conf. Generative Program. and Component Eng. (2005)
33. Alves, V., Gheyi, R., Massoni, T., Kulesza, U., Borba, P., Lucena, C.: Refactoring Product Lines. In: Proc. Int'l. Conf. Generative Program. and Component Eng. (2006)
34. Apel, S., Lengauer, C.: Superimposition: A Language-Independent Approach to Software Composition. In: Proc. Int'l. Symp. Softw. Comp. (2008)