An Algebra for
Feature-Oriented Software Development

Sven Apel[1], Christian Lengauer[1], Don Batory[2],
Bernhard Möller[3], and Christian Kästner[4]

[1]Department of Informatics and Mathematics, University of Passau
{apel,lengauer}@uni-passau.de

[2]Department of Computer Sciences, University of Texas at Austin
batory@cs.utexas.edu

[3]Institute of Computer Science, University of Augsburg
moeller@informatik.uni-augsburg.de

[4]School of Computer Science, University of Magdeburg
kaestner@iti.cs.uni-magdeburg.de

UNIVERSITÄT
PASSAU

Fakultät für Informatik und Mathematik

An Algebra for
Feature-Oriented Software Development

Sven Apel[1], Christian Lengauer[1], Don Batory[2],
Bernhard Möller[3], and Christian Kästner[4]

[1] Department of Informatics and Mathematics, University of Passau,
{apel,lengauer}@uni-passau.de
[2] Department of Computer Sciences, University of Texas at Austin,
batory@cs.utexas.edu
[3] Institute of Computer Science, University of Augsburg,
moeller@informatik.uni-augsburg.de
[4] 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 vari-
able, customizable, and extensible software. Along different lines of re-
search different ideas of what a feature is have been developed. Although
the existing approaches have similar goals, their representations and for-
malizations have not been integrated so far into a common framework.
We present a feature algebra as a foundation of FOSD. The algebra cap-
tures the key ideas 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 for-
malisms, 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 functional-
ity; features are used to distinguish between different variants of a program or
software system [37].

Research along different lines has been undertaken to realize the vision of
FOSD [37,57,12,22,51,7]. Several concepts, formalisms, languages, and tools have
been developed to support FOSD across the software life cycle. While there is the
common notion of a feature, the present approaches use different representations
and notations.

A promising way to integrate the separate lines of research is to provide
an encompassing abstract framework that captures many of the common ideas
and hides (what we feel are) distracting differences. We propose a first step
toward such a framework for FOSD: a *feature algebra*. We introduce a uniform
representation of features, outline the properties of the algebra, and explain how
the algebra models the key concepts of FOSD. Not surprisingly, the notion of a
feature lies at the heart of the algebra.

2   What is a Feature?

Different researchers have been proposing different views of what a feature is or should be. At the same time, previous work largely leaves the notion of a feature undefined or defined informally. Nevertheless, it pervades the entire software life cycle. Features are present in the analysis, design, implementation, configuration, and maintenance phases of FOSD.

Our formal work on features is guided by the following informal definition: *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 and encapsulate a design decision, and to offer a configuration option.* This definition provides a ground that is common to most (if not all) work on FOSD.

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 demands further features to form a complete program). Our main issues are the structure of and the construction methods for composed features, starting from primitive given ones.

Mathematically, we describe feature composition by the operator $\bullet$, which is defined over the set of features F:[5]

$$\bullet : F \times F \to F \tag{1}$$

Typically, a program p (which is itself a feature) is composed of a series of simpler features:

$$p = f_1 \bullet f_2 \bullet \ldots \bullet f_{n-1} \bullet f_n \tag{2}$$

3   The Structure of Features

It has been observed that the implementation of a feature usually crosscuts several structural elements of a program, e.g., the implementation is scattered across multiple packages, classes, methods and other artifacts [12,51,63,45,46,7,5]. Furthermore, the structural elements of a feature need not be exclusively source code artifacts. A feature may have several representations, e.g., makefiles, design documents, performance profiles, documentation, or deployment descriptors [12]. While our work is not limited to code artifacts, for simplicity, we focus here on the main structural abstractions of object orientation, which are visible in the source code.

The source code for a feature consists of, possibly, several parts, each of which can be modeled by one or multiple instances of what we call a *feature structure tree* (*FST*). An FST organizes the structural elements of a feature hierarchically. The 'part-of' or 'contains' relations of a feature's structural elements become 'child-of' relations of the nodes in the according FST. Figure 1 depicts a excerpt of the implementation of a feature *CalcBase* and its representation in form of an FST. One can think of an FST as an abstract syntax tree that contains

---

[5] We write set names in capital letters and element names in lower case letters.

only the information that is necessary for the specification of the structure of a feature. So, for example, an FST does not contain information about the internal structure of methods.
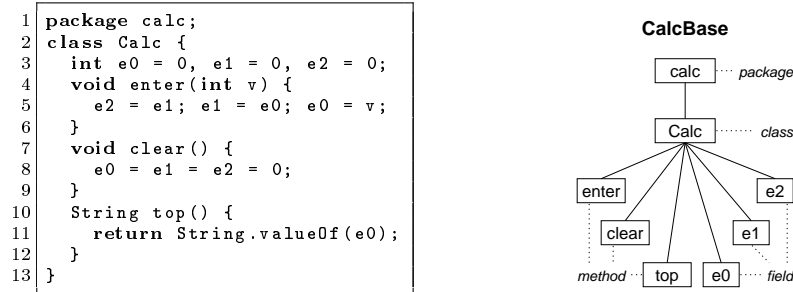
```
1   package calc;
2   class Calc {
3     int e0 = 0, e1 = 0, e2 = 0;
4     void enter(int v) {
5       e2 = e1; e1 = e0; e0 = v;
6     }
7     void clear() {
8       e0 = e1 = e2 = 0;
9     }
10    String top() {
11      return String.valueOf(e0);
12    }
13  }
```

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

An FST with an object-oriented structure contains nodes of different types that represent packages, (inner) classes, (inner) interfaces, fields, and methods. Type information is important during feature composition in order to prevent the composition of incompatible nodes.

The FSTs we consider are unordered trees. That is, the children of a node in an FST do not have a fixed order, much like in object-oriented languages like Java. However, some feature languages may require a fixed order. In future work, we shall discuss the implications of ordered FSTs.

The FST model reflects that typically a feature implementation is scattered across multiple elements of an object-oriented design. In the presence of multiple features their implementations may be tangled inside one element. That is, feature (de)composition is orthogonal to object-oriented (de)composition. It is the process of detaching and attaching elements of an object-oriented design that belong to individual features.

4   Feature Composition

How does the abstract description of a feature composition $f \bullet g$ map to the actual composition at the structural level? That is, how are FSTs composed in order to obtain a new FST? Our answer is: by *tree superimposition*.

4.1   Tree Superimposition

Superimposing trees is not new. Several researchers noted its connection to distributed programming [20,17,38] (i.e., the extension of distributed program structures), object orientation [55,11,63,24] (i.e., the extension and composition of

object-oriented class hierarchies), and component-based systems [16] (i.e., the adaptation of components).

The basic idea is that two trees are composed by composing their nodes, starting from the root and proceeding recursively. Two nodes are composed to form a new node (1) when their parents have been composed already (this is not required for composing root nodes) and (2) when they have the same name[6] and type. If two nodes have been composed, their children are composed as well, if possible. If not, they are added as separate child nodes to the composed parent node. This recurses until all leaves have been reached.

Figure 2 illustrates the process of FST superimposition; Figure 3 depicts the corresponding Java code. Our feature *CalcBase* is composed with a feature *Add*. The result is a new feature, which we call *AddCalc*, that contains the superimposition of the FSTs of *CalcBase* and *Add*. The nodes `calc` and `Calc` are composed with their counterparts and their subtrees are composed in turn.
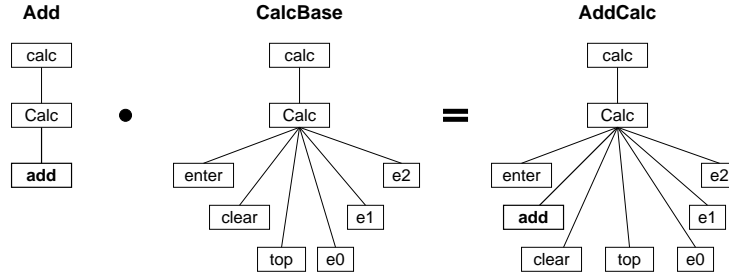

Fig. 2. An example of FST superimposition (*Add* • *CalcBase* = *AddCalc*).

4.2 Terminal and Nonterminal Nodes

An FST is made up of two different kinds of nodes:

Nonterminal nodes are the inner nodes of an FST. The subtree rooted at a nonterminal node reflects the structure of some implementation artifact. Thus, the artifact structure is *transparent* and subject to manipulation by our algebraic operations.

Terminal nodes are the leaves of an FST. Conceptually, a terminal node may also be the root of some structure, but this structure is *opaque* to us and not subject to manipulation by our algebraic operations. It does not appear in the FST.

Packages, classes, and interfaces are represented by nonterminals. The implementation artifacts they contain are represented by child nodes, e.g., a package contains a class and a class contains an inner class and a method. Two compatible

---

[6] Mapped to specific languages that implement features, a name could be a string, an identifier, a signature, etc.
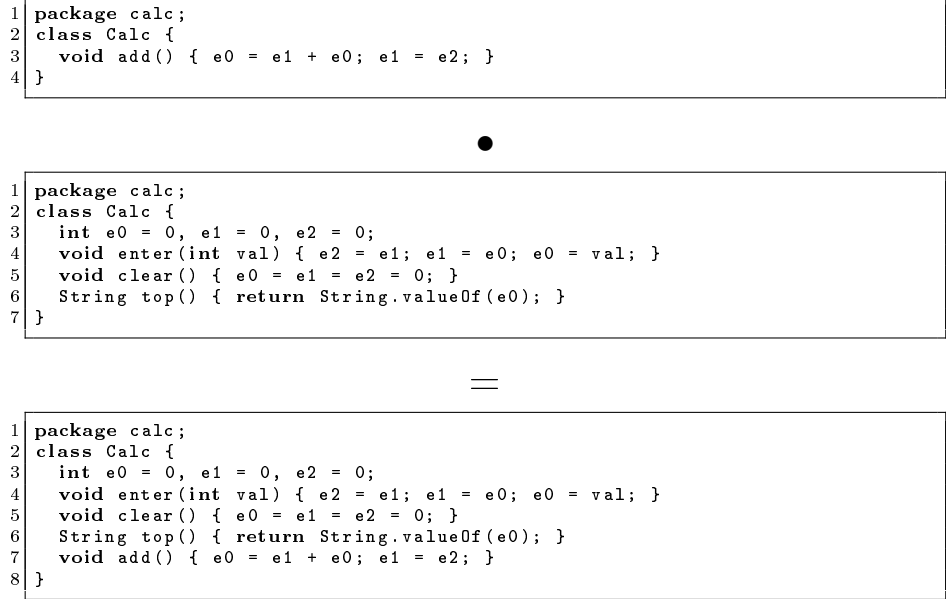
```
1 package calc;
2 class Calc {
3   void add() { e0 = e1 + e0; e1 = e2; }
4 }
```

●

```
1 package calc;
2 class Calc {
3   int e0 = 0, e1 = 0, e2 = 0;
4   void enter(int val) { e2 = e1; e1 = e0; e0 = val; }
5   void clear() { e0 = e1 = e2 = 0; }
6   String top() { return String.valueOf(e0); }
7 }
```

$=$

```
1 package calc;
2 class Calc {
3   int e0 = 0, e1 = 0, e2 = 0;
4   void enter(int val) { e2 = e1; e1 = e0; e0 = val; }
5   void clear() { e0 = e1 = e2 = 0; }
6   String top() { return String.valueOf(e0); }
7   void add() { e0 = e1 + e0; e1 = e2; }
8 }
```

Fig. 3. Java code for the compostion $Add \bullet CalcBase = AddCalc$.

nonterminals are composed by composing their child nodes, e.g., two packages with equal names are merged into one package that contains the composition of the child elements of the original packages.

Methods and fields are represented by terminals, in which the recursion terminates. Their inner structure is not considered in the algebra, e.g., the fact that a method contains a sequence of statements or a field contains a value or an expression. The composition of terminals requires a special treatment and there is a choice of whether and how to compose them:

Option 1: Two terminal nodes with the same name and type *cannot* be composed. If this occurs, it is considered an error.

Option 2: Two terminal nodes with the same name and type *can* be composed in some circumstances; each type has to provide its own rule for composition (see Sec. 4.3).

The first option ignores the common practice of overriding methods and fields in object-oriented programming. But this option has the strength that in some circumstances[7] the order of composing features is not relevant to the behavior of the composed program [55], i.e., feature composition is commutative.

The second option incorporates the notion of field and method overriding. It is in line with many approaches of feature composition [33,49,52,14,19]. However, with this option, the composition order affects the resulting program behavior.

_____
[7] Unordered FSTs are a prerequisite for the commutativity of feature composition.

We choose the second option to benefit from the advantages of overriding. Thus, in our feature algebra, composition is not commutative.

4.3 Composition of Terminals

In order to compose terminals, each terminal type has to provide its own rule for composition. Here are some examples:

- Two methods can be composed if it is specified how the method bodies are composed (e.g., by calling `original`, `super`, or `inner` from inside a method body).
- Two fields can be composed by replacing one with the value of the other or by requiring that one has a value assigned (e.g., `int i=0;`) and the other has not (e.g., `int i;`).
- Clauses attached to classes and methods (e.g., `implements`, `extends`, or `synchronized`) can be composed in the obvious ways: the arguments of two `implements` or `extends` clauses[8] are concatenated; the composition of two `synchronized` clauses results again in a `synchronized` clause.

4.4 Discussion

Superimposition of FSTs imposes several constraints on the programming language in which the artifacts of a feature are expressed:

1. Every element of an artifact must provide a name that becomes the name of the node in the FST.
2. An element must not contain two or more direct child elements with the same name.
3. Elements that do not have a hierarchical structure (terminals) must provide composition rules.

These constraints are usually satisfied by object-oriented languages. But also other representations of features align well with them [12,68,9]. Languages that do not satisfy these constraints do not provide enough structural information necessary for feature composition, i.e., they are not *feature-ready*.

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, i.e., the changes are propagated to the associated feature implementations at the code level.

Of course, it is reasonable not to propagate algebraic manipulations individually but, instead, to perform a series of algebraic manipulations and then propagate the result to the code level.

---

[8] If a target language does no support multiple inheritance an `extends` clause either replaces the other or their composition is not supported.

5.1 Introduction

For the purpose of expressing features and their composition, we use the notion of an *introduction*. An introduction is a constituent of the implementation of a feature, e.g., a method, a field, a class, or even an entire package. When composing two features, introductions are the elementary units of difference (a.k.a. increments) of one feature composed with another feature. Any path in an FST from the root to a node corresponds to an introduction. Thus, a feature is represented by the superimposition of all paths in its FST. We model the superimposition of paths and trees via the operation of *introduction sum*. As we will see later on, introduction sum is not the only operation used when composing features.

Introduction Sum

Introduction sum $\oplus$ is a binary operation defined over the set of introductions I:

$$\oplus : I \times I \to I \tag{3}$$

The result of an introduction sum is again an introduction. Thus, an FST can be represented in two ways: (1) by the individual summands and (2) by a metavariable that represents the sum:

$$i_0 \oplus \ldots \oplus i_n = i_{0\oplus\ldots\oplus n} \tag{4}$$

During composition, for each metavariable $i_{0\oplus\ldots\oplus n}$, the individual summands $i_0 \oplus \ldots \oplus i_n$ are preserved. That is, introduction sum retains information about the summands.

For example, our feature *Calc* can be represented by one FST or multiple superimposed FSTs (cf. Fig. 2). Thus, we can represent it via the metavariable *CalcBase* or as a sum of multiple introductions; ignoring the package declarations, we can write:

$$CalcBase = Calc \oplus enter \oplus clear \oplus top \oplus e0 \oplus e1 \oplus e2$$

In order to process algebraic expressions of features, we flatten the hierarchical structure of FSTs. But, in order not to lose information about which structural elements contain which other elements, we represent introductions in a prefix notation. Our example in prefix notation is denoted as follows:

$$CalcBase = Calc$$
$$\oplus Calc.enter \oplus Calc.clear \oplus Calc.top$$
$$\oplus Calc.e0 \oplus Calc.e1 \oplus Calc.e2$$

Note that, for brevity, we leave implicit that class `Calc` belongs to package `calc`.

Each introduction encodes the entire path from the root of the FST to the introduced node with a sequence of dot-separated identifiers. This enables us to maintain the structural information necessary to model hierarchical feature composition.

Finally, two features f_1 and f_2 are composed by adding their introductions:

$$\underbrace{i_0 \oplus \ldots \oplus i_n}_{f_1} \oplus \underbrace{j_0 \oplus \ldots \oplus j_m}_{f_2} \qquad (5)$$

During the manipulation of an algebraic expression it is always known from which feature an introduction was introduced. We display the information to which feature an introduction belongs via underbraces, however, we do this only if necessary for understanding an example.

Semantics of Introduction

As explained before, introduction sum is a superimposition of FSTs. Now let us examine the semantics of the composition of two nodes in more detail. We distinguish two cases, which occur typically in FOSD [12,57,68,69,44,45,4]:

Subtree merging. Composing two nonterminal nodes with the same name and type creates a new node with the same name and type. If a node does not have a counterpart (a node with the same parent and the same name), it is just added to the parent node, which has been composed already. The entire process proceeds recursively from the root to all leaves (Fig. 2).

Wrapping. Composing two terminal nodes with the same name and type can be viewed as installing a wrapper. One node envelops the other. Wrappers abstract from different type-specific composition rules (cf. Sec. 4.3). Figures 4 and 5 depict how, during the composition of the two features *Count* and *CalcBase*, two Java methods are composed by one method (`enter`) wrapping the other (`enter_wrappee`). The (non-standard Java) keyword `original`[9] provides a means to specify (without knowledge of their source code) how method bodies are merged. The wrapping order is prescribed by the composition order. Applying two wrappers to one node in different orders leads to a different program behavior.
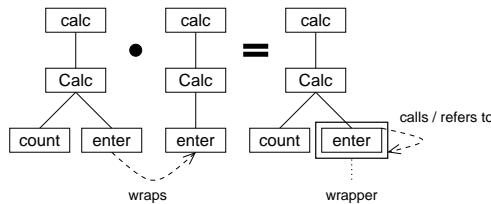

Fig. 4. Installing wrapper around a node (FST representation).

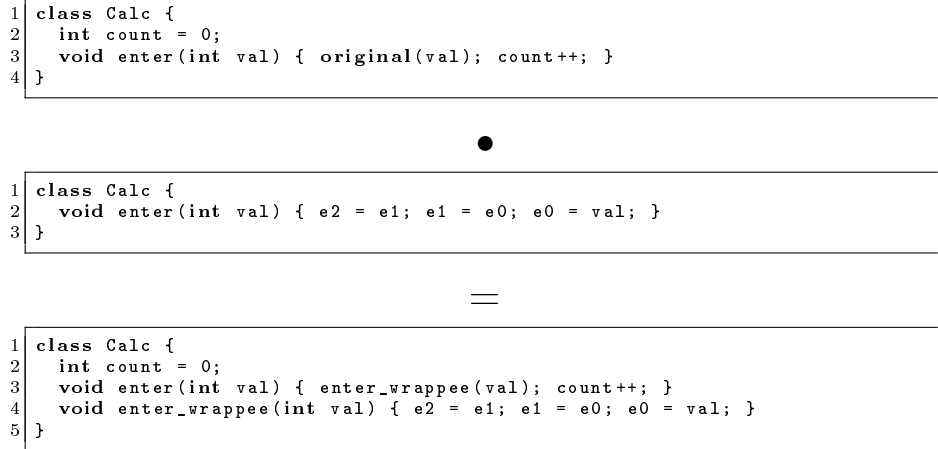[9] In the composed variant `original` is replaced by a call to the wrapper.

8

```
1  class Calc {
2    int count = 0;
3    void enter(int val) { original(val); count++; }
4  }
```

●

```
1  class Calc {
2    void enter(int val) { e2 = e1; e1 = e0; e0 = val; }
3  }
```

$=$

```
1  class Calc {
2    int count = 0;
3    void enter(int val) { enter_wrappee(val); count++; }
4    void enter_wrappee(int val) { e2 = e1; e1 = e0; e0 = val; }
5  }
```

Fig. 5. Installing wrapper around a node (Java code).

Duplicates vs. wrappers. During composition the introductions with the same name (path) and type are composed as described above. An important precondition is that these introductions must stem from different features, which is typically the case, e.g.:

$$\underbrace{Calc.enter}_{Count} \oplus \underbrace{Calc.enter}_{Calc}$$

But, in the case that there are two introductions with the same name and type which stem from the same feature (a.k.a. *duplicates*), the introduction added subsequently is removed during compilation:

$$\underbrace{Calc.enter}_{Calc} \oplus \ldots \oplus \underbrace{Calc.enter}_{Calc} = \underbrace{Calc.enter}_{Calc}$$

The rationale of this is to avoid several problems that occur when composing features multiple times and to avoid code duplication in the course of algebraic optimization (cf. Sec. 5.2).

Algebraic Properties

Introduction sum $\oplus$ over the set of introductions I forms a *non-commutative idempotent monoid* $(I, \oplus, \xi)$:[10]

Closure: Adding two introductions creates again an introduction because superimposing two FSTs creates again an FST.

---

[10] All standard definitions of algebraic structures and properties are based on Hebisch and Weinert [31].

Associativity: $(i \oplus j) \oplus k = i \oplus (j \oplus k)$

Introduction sum is associative because FST superimposition is associative. This applies for terminal and nonterminal nodes.

Identity: $\xi \oplus i = i \oplus \xi = i$

ξ is the empty introduction, i.e., an FST without nodes.

Non-commutativity: $i \oplus j \neq j \oplus i$

Since we consider composition of terminal nodes with the same name and type, FST superimposition and, consequently, introduction sum is not commutative. We consider the right operand to be first, the left is added. If we forbade composing terminal nodes (method and field overriding), we could attain commutativity (cf. Sec. 4).

Idempotence: $i \oplus j \oplus i = j \oplus i$

Only the right-most occurrence of an introduction i is effective in a sum, because it has been applied first. That is, duplicates of i have no effect, as motivated in Section 5.1. 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 composition techniques have been proposed in the literature [65,66,50,48,40,70,1]. An approach frequently discussed has its roots in *metaprogramming* and *metaobject protocols* [41]. The idea is that, when expressing the change a feature causes to another feature, we specify the points at which the two features are supposed to be composed. These ideas have been explored in depth in work on *subject-oriented programming* [30], *multi-dimensional separation of concerns* [66], and *aspect-oriented programming* [40,48]. According to this composition model, we define declaratively where two features are composed and how. The process of determining where two features are composed is called *quantification* [26] or *code querying* [29]. In the remainder, we distinguish between two approaches of composition: *composition by superimposition* and *composition by quantification*. 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 FST model, a modification is a tree walk that determines the nodes which are being modified and that applies the necessary modifications to these nodes (Fig. 6). The advantage of composition by quantification is that querying an FST can return more than one node at a time. This allows us to modify a whole set of nodes without reiterating the modification again and again.

Nevertheless, composition by superimposition and composition by quantification are siblings. Quantification enables us to express a feature more generically than superimposition. But once it is known which points have to be extended,
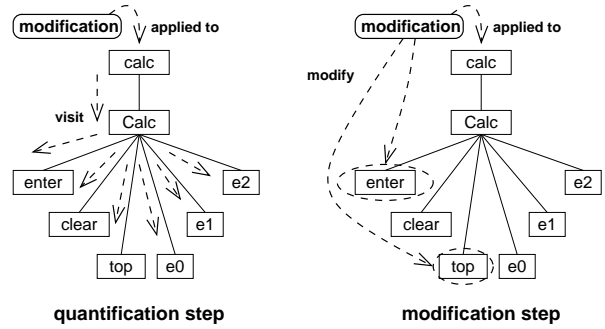
Fig. 6. Modifications are FST walks.

they become equivalent. Figure 7 illustrates their duality. We observed this duality before, but at the level of two concrete programming techniques [7,5]. Our FST model and the algebra make it explicit at a more abstract level.[11]
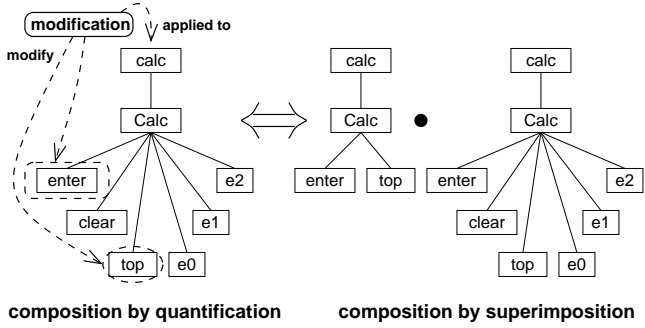

Fig. 7. The duality of composition by quantification and composition by superimposition.

Semantics of Modification

In order for a modification to take effect, we must determine (1) where it affects other features and (2) what changes are to be applied. Specifically, a modification m is made up of a query q that selects a subset of the paths of an introduction

---

[11] Note that the duality does not imply that both approaches are equivalent in every case. Which approach is superior depends on the composition model inside a programming language or environment and on the implementation problem [7,4]. It has been observed that there are indeed problems that demand either one or the other [51,5].

sum and a definition of change c that makes the desired changes:

$$m = (q, c) \tag{6}$$

Query. A query is represented by an FST in which the node names may contain wildcards. For example, the query $q_{calc.Calc.*}$ with the search expression '*calc.Calc.**' applied to our example would return all introductions that are members of the class `Calc`. Figure 8 depicts its FST representation.
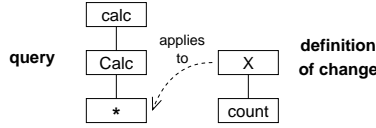

Fig. 8. A query (left side) and a definition of change (right side) represented as FSTs.

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

$$q(i) = \begin{cases} i, & \text{when } i \text{ is matched by } q \\ \xi, & \text{when } i \text{ is not matched by } q \end{cases} \tag{7}$$

A query applied to an introduction sum queries each summand:

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

Definition of change. An introduction i selected by a query is modified according to the modification's definition of change c, which is an introduction that is added to i; the result is $c \oplus i$. Thus, c represents an FST that contains the changes. Much like a query, c contains some generic portion that is necessary to apply the change to different nodes in the FST. For example, '$\chi.count$' adds a field to an arbitrary node (see Fig. 8). We use χ (and also γ) as a metavariable that is substituted with the node to be modified.

Note that a modification cannot delete nodes. That is, if we have a modification that matches certain nodes in an FST, we cannot apply another modification (accidentally) so that the first modification affects a smaller or zero number of nodes:

$$\forall q, i, c : q(i) = i \implies q(c \oplus i) = q(c) \oplus i \tag{9}$$

A modification can make the following changes to a target introduction:

Introduce a new child node ($\chi.n$): The metavariable χ inside a definition of change is replaced with the target introduction. For example, applying ($q_{calc.*}, \chi.count$) to the introduction *calc.Calc* evaluates to *calc.Calc.count* $\oplus$ *calc.Calc*.

Install a wrapper $(\omega(\chi))$: A wrapper $\omega(\chi)$ inside a definition of change is replaced with a new introduction that equals the target introduction in name and type. For example, applying $(q_{calc.Calc.*}, \omega(\chi))$ that belongs to one feature to an introduction $calc.Calc.enter$ that belongs to another feature evaluates to $calc.Calc.enter \oplus calc.Calc.enter$.

The changes a feature can make via modifications are similar to the ones possible via introduction sum, but expressed differently (cf. Fig. 7).

Modification Application and Composition

For simplicity, we usually hide the steps of querying and applying the changes. We introduce an operator for *modification application* defined over the set of modifications M and introductions I:

$$\odot : M \times I \to I \tag{10}$$

A modification applied to an introduction returns either the introduction again or the introduction that has been changed:

$$m \odot i = (q, c) \odot i = \begin{cases} c \oplus i, & q(i) = i \wedge i \neq \xi \\ i, & q(i) = \xi \end{cases} \tag{11}$$

A consequence of this definition is that a modification cannot extend the empty introduction, i.e., the empty program. This is a notable difference between introduction sum and modification application. Using introduction sum we can extend empty programs and using modification application we cannot. While this fact is just a result of our definition, it reflects what contemporary languages that support quantification are doing (see Sec. 8).

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

$$m \odot (i_1 \oplus i_2 \oplus \ldots \oplus i_n) = (m \odot i_1) \oplus (m \odot i_2) \oplus \ldots \oplus (m \odot i_n) \tag{12}$$

Assuming that $m = (q, c)$ affects all summands $i_1, \ldots, i_n$, we can write:

$$(q, c) \odot (i_1 \oplus i_2 \oplus \ldots \oplus i_n) = (c \oplus i_1) \oplus (c \oplus i_2) \oplus \ldots \oplus (c \oplus i_n) \tag{13}$$

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. With a pair of modifications as argument, it is *modification composition*, defined as follows:

$$\odot : M \times M \to M \tag{14}$$

The semantics of modification composition is that the left operand is applied to an introduction and then the right operand to the result:

$$(m_1 \odot m_2) \odot i = m_1 \odot (m_2 \odot i) \tag{15}$$

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

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

$$(m_1 \odot m_2 \odot \ldots \odot m_n) \odot i = m_1 \odot (m_2 \odot (\ldots \odot (m_n \odot i) \ldots)) \qquad (16)$$

Assuming that i is modified by $m_1 = (q_1, c_1), \ldots, m_n = (q_n, c_n)$, we can write:

$$((q_1, c_1) \odot (q_2, c_2) \odot \ldots \odot (q_n, c_n)) \odot i = c_1 \oplus (c_2 \oplus (\ldots (c_{n-1} \oplus (c_n \oplus i)) \ldots)) \qquad (17)$$

Note that applying a modification may add new introductions that can be changed subsequently by other modifications. But, as prescribed by (9), it is not possible to change an introduction sum such that some introductions are removed and the modifications applied subsequently cannot affect them anymore. So, (17) is correct since q_i returns always the same result regardless of whether there have been changes $(c_{i+1} \ldots c_n)$ applied already.

Modification Sum

Finally, we overload the operator $\oplus$ for adding modifications, which we call *modification sum*:

$$\oplus : M \times M \to M \qquad (18)$$

The semantics of modification sum is defined as the composition of queries of two modifications and the composition of the changes they prescribe. Since both queries and definitions of change are expressed with FSTs, they can be composed via superimposition:

$$m_1 \oplus m_2 = (q_1 \oplus q_2, c_1 \oplus c_2) \qquad (19)$$

Applying a sum of two modifications to an introduction is defined as follows (assuming $q_1(i) = i$ and $q_2(i) = i$):

$$(q_1 \oplus q_2, c_1 \oplus c_2) \odot i = (c_1 \oplus c_2) \oplus i \qquad (20)$$

Suppose a modification sum $m_1 \oplus m_2$, in which m_1 adds a field `count` to every class of package `calc` and m_2 adds a method `size` to the class `Calc` in `calc`:

$$m_1 = (q_{calc.*}, \chi.count) \text{ and } m_2 = (q_{calc.Calc}, \gamma.size)$$

Note that the queries $q_{calc.*}$ and $q_{calc.Calc}$ may return overlapping subtrees of an input FST. For example, applied to $calc.Calc \oplus calc.GUI$ they return:

$$q_1(calc.Calc \oplus calc.GUI) = calc.Calc \oplus calc.GUI$$
$$q_2(calc.Calc \oplus calc.GUI) = calc.Calc$$

14

By annotating the selected introductions it is guaranteed that the changes of m_1 ($\chi.count$) and m_2 ($\gamma.count$) apply to the appropriate nodes, i.e., *calc.Calc* is marked to be changed by m_1 and m_2 and *calc.GUI* is marked to be changed by m_1.

Modification application distributes from the right over modification sum:

$$(m_1 \oplus m_2 \oplus \ldots \oplus m_n) \odot i = (m_1 \odot i) \oplus (m_2 \odot i) \oplus \ldots \oplus (m_n \odot i) \qquad (21)$$

Assuming that i is modified by $m_1 = (q_1, c_1), \ldots, m_n = (q_n, c_n)$, we can write:

$$(q_1 \oplus q_2 \oplus \ldots \oplus q_n, c_1 \oplus c_2 \oplus \ldots \oplus c_n) \odot i = (c_1 \oplus c_2 \oplus \ldots \oplus c_n) \oplus i \qquad (22)$$

The distributivity law (21) induces the following equality:

$$(c_1 \oplus c_2 \oplus \ldots \oplus c_n) \oplus i = (c_1 \oplus i) \oplus (c_2 \oplus i) \oplus \ldots \oplus (c_n \oplus i) \qquad (23)$$

This equality holds because modifications apply two kinds of changes that respect the distributivity law (21):

Subtree merging. Adding two sets of child nodes by means of two distinct modifications to an introduction equals adding the superimposed set of child nodes to the introduction. Figure 9 shows two modifications, each of which introduce new children, and their sum that introduces the superimposed set of children.
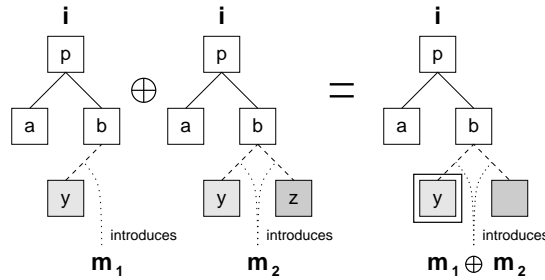

Fig. 9. Applying subtrees separately equals applying their superimposition.

Expressing this example in our algebra, we annotate the introductions in order to keep track to which feature they belong:

$$\underbrace{(q_{p.b}, \chi.y)}_{f_1} \odot \underbrace{(p.a \oplus p.b)}_{f_3}) \oplus (\underbrace{(q_{p.b}, \gamma.y \oplus \gamma.z)}_{f_2} \odot \underbrace{(p.a \oplus p.b)}_{f_3}) =$$

$$(\underbrace{p.b.y}_{f_1} \oplus \underbrace{p.a}_{f_3} \oplus \underbrace{p.b}_{f_3}) \oplus (\underbrace{p.b.y}_{f_2} \oplus \underbrace{p.b.z}_{f_2} \oplus \underbrace{p.a}_{f_3} \oplus \underbrace{p.b}_{f_3})$$

The duplicates (equal introductions that stem from the same feature) are removed (cf. Sec. 5.1) resulting in:

$$\underbrace{p.b.y}_{f_1} \oplus \underbrace{p.b.y}_{f_2} \oplus \underbrace{p.b.z}_{f_2} \oplus \underbrace{p.a}_{f_3} \oplus \underbrace{p.b}_{f_3}$$

Wrapping. Applying two wrappers to one introduction results in an enveloped wrapper. Figure 10 shows two modifications that install three wrappers, and the sum of the modifications that installs two wrappers and one enveloped wrapper.
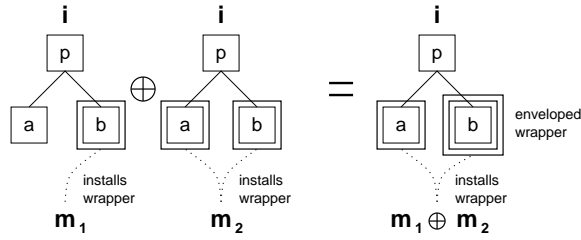

Fig. 10. Installing wrappers separately equals installing enveloped wrappers.

Assuming ω and υ are the wrappers to be installed, our example is expressed algebraically as follows:

$$\underbrace{\big((q_{p.b}, \omega(\chi)\big) \odot}_{f_1} \underbrace{(p.a \oplus p.b)\big)}_{f_3} \oplus \underbrace{\big((q_{p.*}, \upsilon(\gamma)\big) \odot}_{f_2} \underbrace{(p.a \oplus p.b)\big)}_{f_3} =$$
$$\big(\underbrace{p.b}_{f_1} \oplus \underbrace{p.a}_{f_3} \oplus \underbrace{p.b}_{f_3} \big) \oplus \big(\underbrace{p.a}_{f_2} \oplus \underbrace{p.b}_{f_2} \oplus \underbrace{p.a}_{f_3} \oplus \underbrace{p.b}_{f_3} \big)$$

The duplicates (equal introductions that stem from the same feature) are removed (cf. Sec. 5.1) resulting in:

$$\underbrace{p.b}_{f_1} \oplus \underbrace{p.a}_{f_2} \oplus \underbrace{p.b}_{f_2} \oplus \underbrace{p.a}_{f_3} \oplus \underbrace{p.b}_{f_3}$$

Modification Sum vs. Modification Application

The reader may have noted that modification sum and modification application are quite similar. Applying the sum of two modifications to an introduction results in the same program as applying the modifications consecutively. This becomes apparent when comparing the result of modification sum and modification application, i.e., the right sides of the equations (23) and (17):

$$(c_1 \oplus i) \oplus (c_2 \oplus i) \oplus \ldots \oplus (c_n \oplus i) = c_1 \oplus (c_2 \oplus (\ldots (c_{n-1} \oplus (c_n \oplus i)) \ldots)) \quad (24)$$

In Figure 11 we illustrate, using the FST model, how the two algebraic operations lead to the same result. An FST is modified by two modifications m and n that add new children. With modification sum the two modifications m and n are applied separately to the introduction i and then the two modified introductions are added subsequently (left side). With modification application, we first apply the modification n and then we apply to the resulting introduction the modification m (right side). Figure 12 illustrates how the FST example of Figure 11 is translated to our algebra.
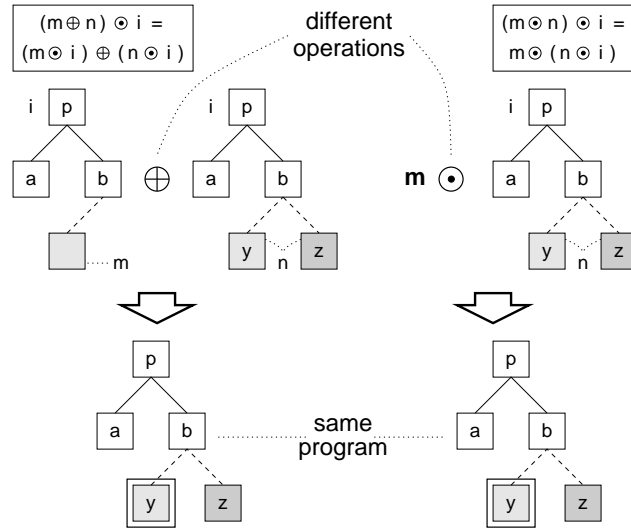

Fig. 11. Modification sum (left side) versus modification composition (right side).

Algebraic Properties

Modification sum. Modification sum $\oplus$ over the set of modifications M induces a _non-commutative idempotent monoid_ $(M, \oplus, \zeta)$:

Closure: Adding two modifications creates again a modification, as explained previously.

Associativity: $(m \oplus n) \oplus o = m \oplus (n \oplus o)$

Modification sum is associative because the superimposition of the changes that modifications apply is associative.

Identity: $\zeta \oplus m = m \oplus \zeta = m$

ζ is the empty modification whose query $\boldsymbol{\xi}$ (bold) returns always the empty introduction ξ and whose definition of change is the empty introduction: $\zeta = (\boldsymbol{\xi}, \xi)$.

Modification sum:

$$\underbrace{\left((q_{p.b} \oplus q_{p.b}, \chi.y \oplus \gamma.y \oplus \gamma.z)\right.}_{f_1 \bullet f_2} \odot \underbrace{(p.a \oplus p.b)}_{f_3})$$

$$= (\underbrace{(q_{p.b}, \chi.y)}_{f_1} \odot \underbrace{(p.a \oplus p.b)}_{f_3}) \oplus (\underbrace{(q_{p.b}, \gamma.y \oplus \gamma.z)}_{f_2} \odot \underbrace{(p.a \oplus p.b)}_{f_3})$$

$$= (\underbrace{p.b.y}_{f_1} \oplus \underbrace{p.a}_{f_3} \oplus \underbrace{p.b}_{f_3}) \oplus (\underbrace{p.b.y}_{f_2} \oplus \underbrace{p.b.z}_{f_2} \oplus \underbrace{p.a}_{f_3} \oplus \underbrace{p.b}_{f_3})$$

$$= \underbrace{p.b.y}_{f_1} \oplus \underbrace{p.b.y}_{f_2} \oplus \underbrace{p.b.z}_{f_2} \oplus \underbrace{p.a}_{f_3} \oplus \underbrace{p.b}_{f_3}$$

Modification composition:

$$(\underbrace{(q_{p.b}, \chi.y)}_{f_1} \odot \underbrace{(q_{p.b}, \gamma.y \oplus \gamma.z))}_{f_2} \odot \underbrace{(p.a \oplus p.b)}_{f_3})$$

$$= \underbrace{(q_{p.b}, \chi.y)}_{f_1} \odot (\underbrace{(q_{p.b}, \gamma.y \oplus \gamma.z)}_{f_2} \odot \underbrace{(p.a \oplus p.b)}_{f_3})$$

$$= \underbrace{(q_{p.b}, \chi.y)}_{f_1} \odot (\underbrace{p.b.y}_{f_2} \oplus \underbrace{p.b.z}_{f_2} \oplus \underbrace{p.a}_{f_3} \oplus \underbrace{p.b}_{f_3})$$

$$= \underbrace{p.b.y}_{f_1} \oplus \underbrace{p.b.y}_{f_2} \oplus \underbrace{p.b.z}_{f_2} \oplus \underbrace{p.a}_{f_3} \oplus \underbrace{p.b}_{f_3}$$

Fig. 12. Modification sum versus modification composition.

Non-commutativity: $m \oplus n \neq n \oplus m$

Modification sum is not commutative because installing wrappers and composing subtrees is not commutative.

Idempotence: $m \oplus n \oplus m = n \oplus m$

As with introductions, only the right-most occurrence of a modification m is effective in a sum (distant idempotence). For $n = \zeta$ direct idempotence ($m \oplus m = m$) follows.

Modification application. Modification application $\odot$ over the set of modifications M induces a *non-commutative monoid* $(M, \odot, \zeta)$:

Closure: Applying two modifications consecutively is similar to function composition.

Associativity: $(m \odot n) \odot o = m \odot (n \odot o)$

Modification application is associative because applying modifications consecutively is associative.

Identity: $\zeta \oplus m = m \oplus \zeta = m$

ζ is the empty modification whose query $\boldsymbol{\xi}$ returns always the empty introduction ξ and whose definition of change is the empty introduction: $\zeta = (\boldsymbol{\xi}, \xi)$.

Non-commutativity: $m \odot n \neq n \odot m$

Modification application is not commutative because installing wrappers and composing subtrees is not commutative.

5.3 Introductions and Modifications in Concert

In order to describe feature composition, our algebra has to integrate both composition models: composition by superimposition and composition by quantification. Consequently, we have to integrate and relate our three algebraic structures $(I, \oplus, \xi)$, $(M, \oplus, \zeta)$, and $(M, \odot, \zeta)$.

Binoid

The set of modifications together with the operations $\oplus$ and $\odot$ form a *non-commutative binoid* [36] $(M, \oplus, \odot, \zeta)$ because $(M, \oplus, \zeta)$ induces a non-commutative idempotent monoid and $(M, \odot, \zeta)$ induces a non-commutative monoid (cf. Sec. 5.2). A binoid is a weak form of a semiring, in which the neutral elements of both monoids are equal. Furthermore, the neutral element ζ of modification sum is not a *right* or *left annihilator* for modifications, which is in contrast to *full semirings* [42], i.e., $m \odot \zeta \neq \zeta$ and $\zeta \odot m \neq \zeta$.

Semimodule

A notable property of the non-commutative idempotent monoid $(I, \oplus, \xi)$ is that it is a *semimodule over the binoid* $(M, \oplus, \odot, \zeta)$ since the following distributivity laws hold [31], as we have explained:

$$\forall m \in M \,:\, \forall i, j \in I \,:\, m \odot (i \oplus j) = (m \odot i) \oplus (m \odot j) \tag{25}$$

$$\forall m, n \in M \: : \: \forall i \in I \: : \: (m \oplus n) \odot i = (m \odot i) \oplus (n \odot i) \qquad (26)$$

$$\forall m, n \in M \: : \: \forall i \in I \: : \: (m \odot n) \odot i = m \odot (n \odot i) \qquad (27)$$

A semimodule over a binoid is related to a vector space but weaker [31]. The additive and multiplicative operations in vector spaces are commutative and there are inverse elements with respect to addition and annihilation. Nevertheless, the fact that our feature algebra is a semimodule over a binoid 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 (introductions and modifications) and two operators ($\oplus$ and $\odot$) for feature composition. Now we integrate them in a compact and concise notation. We allow complex features that involve introductions and local and global modifications. For this purpose, we introduce the *quark model*.

Quark composition with local modification application:

$$
\begin{aligned}
f_1 \bullet f_2 \bullet \ldots \bullet f_n &= \langle \zeta, i_1, m_1 \rangle \bullet \langle \zeta, i_2, m_2 \rangle \bullet \ldots \bullet \langle \zeta, i_n, m_n \rangle \\
&= \langle \zeta, i_1 \oplus (m_1 \odot (i_2 \oplus (m_2 \odot (\ldots (i_{n-1} \oplus (m_{n-1} \odot i_n)))))), \\
&\quad m_1 \odot m_2 \odot \ldots \odot m_n \rangle
\end{aligned}
\qquad (28)
$$

Quark composition with global modification application:

$$
\begin{aligned}
f_1 \bullet f_2 \bullet \ldots \bullet f_n &= \langle g_1, i_1, \zeta \rangle \bullet \langle g_2, i_2, \zeta \rangle \bullet \ldots \bullet \langle g_n, i_n, \zeta \rangle \\
&= \langle g_1 \odot g_2 \odot \ldots \odot g_n, (g_1 \odot g_2 \odot \ldots \odot g_n) \odot (i_1 \oplus i_2 \oplus i_n), \zeta \rangle \,(29)
\end{aligned}
$$

Quark composition with both local and global modification application:

$$
\begin{aligned}
f_1 \bullet f_2 \bullet \ldots \bullet f_n &= \langle g_1, i_1, m_1 \rangle \bullet \langle g_2, i_2, m_2 \rangle \bullet \ldots \bullet \langle g_n, i_n, m_n \rangle \\
&= \langle g_1 \odot \ldots \odot g_n, (g_1 \odot \ldots \odot g_n) \odot (i_1 \oplus (m_1 \odot (i_2 \oplus (m_2 \odot (\ldots \\
&\quad (i_{n-1} \oplus (m_{n-1} \odot i_n))))))), m_1 \odot \ldots \odot m_n \rangle
\end{aligned}
\qquad (30)
$$

Fig. 13. Composition of n quarks with local and global modification application.

A *quark* represents a feature. It is a triple, which consists of a sum g of modifications, a sum i of introductions, and a further sum m of modifications:

$$f = \langle g, i, m \rangle \tag{31}$$

i is the introduction sum of feature f representing an FST; m and g contain the modifications that feature f makes. We can always bring quarks into the following form:

$$f = \langle g_1 \oplus \ldots \oplus g_j, i_1 \oplus \ldots \oplus i_k, m_1 \oplus \ldots \oplus m_l \rangle \tag{32}$$

We distinguish between two modification sums because there are two options of applying modifications when composing quarks:

Local modification application: The modifications of m are applied locally:

$$f_1 \bullet f_2 = \langle \zeta, i_1, m_1 \rangle \bullet \langle \zeta, i_2, m_2 \rangle = \langle \zeta, i_1 \oplus (m_1 \odot i_2), m_1 \odot m_2 \rangle \tag{33}$$

Local modifications can affect only introductions of features that have already been composed. In our example, m_1 affects only i_2. For a composition of n features $f_1 \bullet f_2 \bullet \ldots \bullet f_n$, a modification m_i of feature f_i can affect only the introductions of a feature f_j for all $i < j$. The modifications m_1 and m_2 are composed like functions.

Global modification application: The modifications of g are applied globally:

$$f_1 \bullet f_2 = \langle g_1, i_1, \zeta \rangle \bullet \langle g_2, i_2, \zeta \rangle = \langle g_1 \odot g_2, (g_1 \odot g_2) \odot (i_1 \oplus i_2), \zeta \rangle \tag{34}$$

Global modifications can affect any introduction added in a series of features. In our example, both, g_1 and g_2 may affect i_1 and i_2. For a composition of n features $f_1 \bullet f_2 \bullet \ldots \bullet f_n$, a modification m_i of feature f_i can affect the introductions of a feature f_j for any pair (i, j). The modifications g_1 and g_2 are composed like functions.

The difference between local and global modification application demands a special treatment of quark composition. When composing a series of quarks, we can apply the local modifications immediately. Equation (33) implies that the local modifications affect only the features that have been composed already.

But equation (34) implies that we cannot apply the global modification immediately. We have to wait until all introductions and local modifications in a series of quarks have been composed and then we can apply all global modifications subsequently. Figure 13 depicts quark composition based on local and global modification for the general case.

Both variants of feature composition have been discussed before in the context of software product lines and aspect-oriented programming [45,47,8]. Since there is no agreement on which variant is superior we include both in our quark model.

7   Related Work

7.1   Authors' Previous Work

Batory et al. were among the first to note the potential of algebra for reasoning about feature composition [12]. They model features as functions and feature composition as function composition.

In follow-up work Lopez-Herrejon, Batory, and Lengauer refined the algebraic model by distinguishing between introductions and advice [46,47], which correspond roughly to our introductions and modifications. However, there is no semantic model that defines what introductions and advice precisely 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.

In a different line of research, Liu, Batory, and Lengauer developed an algebra to describe feature interactions [44]. They use derivatives to represent feature interactions. Derivatives are hidden features that must be added in the presence of other features in order to guarantee a proper feature interaction. This notion of a derivative is at a more abstract level, at which a program is modeled simply by a sequence of features. We could also make derivatives explicit in our feature algebra. This would not incur any further algebraic operators.

Apel and Liu proposed a simple algebraic model for understanding aspects as functions [8]. They derived several properties of aspects regarding their interchangeability during aspect composition. The model does not provide a view of the internal structure of aspects and its effects on composition.

Möller et al. developed an algebra to express software and hardware variabilities in the form of features [32]. The algebra focuses on the analysis phase of FOSD, in which sets of features define program families and feature selections define programs. In their work a feature refers to a variation point. The structure of features and their implementation is not considered. Our feature algebra is a link between the analysis level and the implementation level.

7.2   Work of other Researchers

Object, component, and aspect calculi. There are some calculi that support feature-like structures and composition by superimposition [33,25,21,28,27,53,34]. Deep is closest to our algebra. It is a formal object calculus that provides type-safe support for features, composition by superimposition, and virtual classes. Deep is tailored to Java-like languages and emphasizes the type system. Instead, our feature algebra allows to reason about feature composition on a more abstract level. We emphasize the structure of features and their static composition, independently of a particular language or execution semantics.

The notion of a feature is close to that of a component. Bosch noted the possibility of superimposing the internal structures of components for adaptation purposes [16]. However, many contemporary component calculi focus on concurrency and process-theoretic issues as well as on connector and composition

languages [1,58,76,62,73]. We use superimposition and quantification to control composition. The selection of a set of features is equivalent to a specification in a composition language; modifications are equivalent to connectors. Our FST model emphasizes the static structure of features, not considering process, concurrency, and orchestration issues. The static view enables us to model not only code artifacts, but any kind of artifact that provides a sufficient structure, like makefiles, grammar specifications, or documentation (see below).

In the field of aspect-oriented programming several approaches have been proposed that model and formalize quantification mechanisms [48,35,43,71,74,2,23]. However, their focus is on the control flow, typing issues, and operational semantics. Our feature algebra provides a static view of quantification, which is useful for feature composition that involves the introduction of new structures via quantification.

Languages and tools. Several languages support features and their composition by superimposition, e.g., *Scala* [54], *Jiazzi* [49], *Classbox/J* [14], *FeatureC++* [6], and *Jak* [12]. Our algebra formalizes features and feature composition on top of FSTs. It is a theoretical backbone that underlies and unifies all these languages and tools. It reveals the properties a language or tool must have in order to be feature-ready.

Several aspect-oriented programming languages provide mechanisms to quantify over the structure and the computation of a program, e.g., *AspectJ* [39], *AspectC++* [64], *Eos* [59]. Our algebra models quantification as a tree walk and reveals the duality between quantification and superimposition. However, it does not model quantification over the program control flow and the application of behavioral changes.

Integrating superimposition and quantification. Several researchers realized the synergetic potential of superimposition and quantification [66,51,7]. CaesarJ, Hyper/J, FeatureC++ are languages that provide full support for both composition by superimposition and composition by quantification. The feature algebra allows us to study their relationship and their integration, independently of a specific language.

Non-source code artifacts. As mentioned previously, features are implemented not only by source code. Several tools support the feature composition of non-source code artifacts [12,3,18,68]. Our algebra is general enough to describe a feature containing these non-code artifacts since all their representations can be mapped to the FST model.

8   Conclusions

We conclude with the main insights that the algebra has given us and with some perspectives for future work.

23

8.1 Insights

Composition models. We have presented a model of FOSD in which a feature is represented as a tree structure (FST). Feature composition is expressed by both tree superimposition and tree walks. This reflects the current developments in programming languages and composition models. For example, *collaboration-based design* [61,72,63], *higher-order hierarchies* [24], *subject-oriented programming* [30], and *feature-oriented programming* [57] favor superimposition as composition model; *metaobject protocols* [41], *aspect-oriented programming* [40,26,48], and *program transformation systems* [13] are based on tree walks and composition by quantification; some approaches integrate both [51,66,7]. Our algebra describes precisely what the properties of the composition models are and how they can be integrated. This is not obvious from previous work, which is based on specific instantiations and implementations of the composition models.

Duality. The feature algebra makes the duality between composition by superimposition and composition by quantification explicit. Previous work was not able to condense this result convincingly [4]. Interestingly, we found in our algebra a subtle difference between them, namely that modifications are not able to extend the empty program. While this is foremost a result of our definition, it reflects the current situation in languages that support quantification. For example, AspectJ's advice mechanism cannot extend an empty program. Whether this is a general property of quantification or just coincidence remains to be answered.

Quarks. On the basis of introductions and modifications we introduced the quark model in order to express how algebraic expressions map to feature implementations. It allows us to choose between local and global modification application. This variability is also motivated by the presence of different lines of research. Local modification application follows the paradigm of *incremental software development* [75,56,60,12]. Global modification quantification is motivated by work on metaobject protocols and aspect-oriented programming. Again, the feature algebra describes precisely the differences between both approaches and, based on some restrictions we impose, it provides a way to integrate them.

Algebraic properties. Possibly, the most remarkable result is that our feature algebra forms a semimodule over a binoid, 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 indeed 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.

Design decisions. Although our algebra is quite flexible, we also made several restrictive decisions. For example, introduction sum is idempotent and modifications are allowed only to add children and install wrappers. An advantage of

our 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 composition of terminal nodes and presume sets of child nodes (instead of ordered lists) we achieve the commutativity of feature composition. We decided otherwise in order to increase the expressive power of introduction sum by including overriding. Or, forbidding modifications to remove nodes from an FST allows us to make some assumptions about the result of a composition, e.g., that features do not delete nodes that other features depend on. Exploring further the implications of our and alternative decisions in real software projects is a promising avenue of further work.

8.2 Perspectives

Higher-order modifications. A notable result is the inclusion of modification sum as an operation. While, formally and conceptually, modification sum is straightforward and integrating it in our feature algebra was easy, we are not aware of a formalism or programming system that supports this operation explicitly. One reason for including modification sum that we not discussed so far is a possible generalization of our algebra.

A fundamental asymmetry in our algebra is the distinction between introductions and modifications. Since a modification applies to an introduction sum it is on a meta level. In the literature, the addition of further levels on top of modifications has been proposed, e.g. [67,15,9]. In our algebra, this would require the inclusion of modifications that modify other modifications, i.e., *higher-order modifications* that modify modifications of a lower order.

This generalization works only with an operation like modification sum. Higher-order modifications quantify over sums of lower-order modifications, much like modifications quantify over sums of introductions. Our introductions can be viewed as 0-order modifications, modifications that apply to introductions are 1-order, modifications that apply to 1-order modifications are 2-order, and so on ad infinitum. We assume, in practice, orders higher than 2 or 3 will rarely be needed.

The big picture. Finally, the big picture of our endeavor is as follows. The feature algebra serves as the formal foundation for the vision of automatic feature-based program synthesis [10,22]. Treating programs as values of metaprograms that manipulate them requires a formal theory that describes what is allowed and what not. For example, a program transformation that simply deletes an input program is certainly not useful in program synthesis. Metaprograms that apply arbitrary changes are even dangerous since they can introduce subtle errors.

The algebra will be at the heart of a new direction of research on program synthesis and generative programming, which is called *architectural metaprogramming* [10]. It applies metaprogramming techniques at the level of the software architecture. The algebra provides a formalism to express the necessary abstraction from the implementation level. In fact, the feature algebra is a means to reason about and manipulate software architecture. Metaprograms operate on

feature algebra expressions to synthesize programs at the architectural level. At every step, the algebra maintains the connection between the architectural and the implementation level. It guarantees that the operations transform the structures from one to another consistent state.

Acknowledgments

References

1. F. Achermann and O. Nierstrasz. A Calculus for Reasoning About Software Composition. *Theoretical Computer Science*, 331(2–3):367–396, 2005.
2. J. Aldrich. Open Modules: Modular Reasoning about Advice. In *Proc. of European Conf. on Object-Oriented Programming*, volume 3586 of *LNCS*, pages 144–168. Springer, 2005.
3. V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, and C. Lucena. Refactoring Product Lines. In *Proc. of Intl. Conf. on Generative Programming and Component Engineering*, pages 201–210. ACM Press, 2006.
4. S. Apel. *The Role of Features and Aspects in Software Development*. PhD thesis, School of Computer Science, University of Magdeburg, 2007.
5. S. Apel and D. Batory. When to Use Features and Aspects? A Case Study. In *Proc. of Intl. Conf. on Generative Programming and Component Engineering*, pages 59–68. ACM Press, 2006.
6. S. Apel, T. Leich, M. Rosenmüller, and G. Saake. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *Proc. of Intl. Conf. on Generative Programming and Component Engineering*, volume 3676 of *LNCS*, pages 125–140. Springer, 2005.
7. S. Apel, T. Leich, and G. Saake. Aspectual Mixin Layers: Aspects and Features in Concert. In *Proc. of Intl. Conf. on Software Engineering*, pages 122–131. ACM Press, 2006.
8. S. Apel and J. Liu. On the Notion of Functional Aspects in Aspect-Oriented Refactoring. In *Proc. of the ECOOP Workshop on Aspects, Dependencies, and Interactions*, pages 1–9. Computing Department, Lancaster University, 2006.
9. Sven Apel, Christian Kästner, Thomas Leich, and Gunter Saake. Aspect Refinement - Unifying AOP and Stepwise Refinement. *Journal of Object Technology – Special Issue: Proc. of Intl. TOOLS EUROPE'07 Conf. - Objects, Models, Components, Patterns*, 2007.
10. D. Batory. From Implementation to Theory in Program Synthesis, 2007. Keynote at the Intl. Symposium on Principles of Programming Languages.
11. D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Trans. on Software Engineering and Methodology*, 1(4):355–398, 1992.
12. D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Trans. on Software Engineering*, 30(6):355–371, 2004.
13. I. D. Baxter. Design Maintenance Systems. *Comm. of the ACM*, 35(4):73–89, 1992.
14. A. Bergel, S. Ducasse, and O. Nierstrasz. Classbox/J: Controlling the Scope of Change in Java. In *Proc. of Intl. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 177–189. ACM Press, 2005.

15. E. Bodden, F. Forster, and F. Steimann. Avoiding Infinite Recursion with Stratified Aspects. In *Proc. of Intl. Net.ObjectDays Conf.*, pages 49–64. Gesellschaft für Informatik, 2006.

16. J. Bosch. Super-Imposition: A Component Adaptation Technique. *Information and Software Technology*, 41(5):257–273, 1999.

17. L. Bouge; and N. Francez. A Compositional Approach to Superimposition. In *Proc. of Intl. Symposium on Principles of Programming Languages*, pages 240–249. ACM Press, 1988.

18. M. Bravenboer and E. Visser. Concrete Syntax for Objects: Domain-Specific Language Embedding and Assimilation Without Restrictions. In *Proc. of Intl. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 365–383. ACM Press, 2004.

19. R. Cardone and C. Lin. Comparing Frameworks and Layered Refinement. In *Proc. of Intl. Conf. on Software Engineering*, pages 285–294. IEEE Computer Society, 2001.

20. M. Chandy and J. Misra. An Example of Stepwise Refinement of Distributed Programs: Quiescence Detection. *ACM Trans. on Programming Languages and Systems*, 8(3):326–343, 1986.

21. D. Clarke, S. Drossopoulou, J. Noble, and T. Wrigstad. Tribe: A Simple Virtual Class Calculus. In *Proc. of Intl. Conf. on Aspect-Oriented Software Development*. ACM Press, 2007.

22. K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

23. D. S. Dantas and D. Walker. Harmless Advice. In *Proc. of Intl. Symposium on Principles of Programming Languages*, pages 383–396. ACM Press, 2006.

24. E. Ernst. Higher-Order Hierarchies. In *Proc. of European Conf. on Object-Oriented Programming*, volume 2743 of *LNCS*, pages 303–329. Springer, 2003.

25. E. Ernst, K. Ostermann, and W. R. Cook. A Virtual Class Calculus. In *Proc. of Intl. Symposium on Principles of Programming Languages*, pages 270–282. ACM Press, 2006.

26. R. E. Filman and D. P. Friedman. Aspect-Oriented Programming Is Quantification and Obliviousness. In *Aspect-Oriented Software Development*, pages 21–35. Addison-Wesley, 2005.

27. R.B. Findler and M. Flatt. Modular Object-Oriented Programming with Units and Mixins. In *Proc. of Intl. Conf. on Functional Programming*, pages 94–104. ACM Press, 1998.

28. M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and Mixins. In *Proc. of Intl. Symposium on Principles of Programming Languages*, pages 171–183. ACM Press, 1998.

29. E. Hajiyev, M. Verbaere, and O. de Moor. codeQuest: Scalable Source Code Queries with Datalog. In *Proc. of European Conf. on Object-Oriented Programming*, volume 4067 of *LNCS*, pages 2–27. Springer, 2006.

30. W. Harrison and H. Ossher. Subject-Oriented Programming: A Critique of Pure Objects. In *Proc. of Intl. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 411–428. ACM Press, 1993.

31. U. Hebisch and H. J. Weinert. *Semirings*. World Scientific, 1998.

32. P. Höfner, R. Khedri, and B. Möller. Feature Algebra. In *Proc. of Intl. Symposium on Formal Methods*, volume 4085 of *LNCS*, pages 300–315. Springer, 2006.

33. D. Hutchins. Eliminating Distinctions of Class: Using Prototypes to Model Virtual Classes. In *Proc. of Intl. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 1–19. ACM Press, 2006.

34. A. Igarashi, C. Saito, and M. Viroli. Lightweight Family Polymorphism. In *Proc. of Asian Symposium on Programming Languages and Systems*, volume 3780 of *LNCS*, pages 161–177. Springer, 2005.

35. R. Jagadeesan, A. Jeffrey, and J. Riely. A Calculus of Untyped Aspect-Oriented Programs. In *Proc. of European Conf. on Object-Oriented Programming*, volume 2743 of *LNCS*, pages 54–73. Springer, 2003.

36. W.B. Vasantha Kandasamy. *Bialgebraic Structures and Smarandache Bialgebraic Structures*. American Research Press, 2003.

37. K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

38. S. Katz. A Superimposition Control Construct for Distributed Systems. *ACM Trans. on Programming Languages and Systems*, 15(2):337–356, 1993.

39. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *Proc. of European Conf. on Object-Oriented Programming*, volume 2072 of *LNCS*, pages 327–353. Springer, 2001.

40. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proc. of European Conf. on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer, 1997.

41. G. Kiczales and J. Des Rivieres. *The Art of the Metaobject Protocol*. MIT Press, 1991.

42. W. Kuich and A. Salomaa. *Semirings, Automata and Languages*, volume 5 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1985.

43. J. Ligatti, D. Walker, and S. Zdancewic. A Type-Theoretic Interpretation of Pointcuts and Advice. *Science of Computer Programming*, 63(3):240–266, 2006.

44. J. Liu, D. Batory, and C. Lengauer. Feature-Oriented Refactoring of Legacy Applications. In *Proc. of Intl. Conf. on Software Engineering*, pages 112–121. ACM Press, 2006.

45. R. Lopez-Herrejon. *Understanding Feature Modularity*. PhD thesis, Department of Computer Sciences, The University of Texas at Austin, 2006.

46. R. Lopez-Herrejon, D. Batory, and W. R. Cook. Evaluating Support for Features in Advanced Modularization Technologies. In *Proc. of European Conf. on Object-Oriented Programming*, volume 3586 of *LNCS*, pages 169–194. Springer, 2005.

47. R. Lopez-Herrejon, D. Batory, and C. Lengauer. A Disciplined Approach to Aspect Composition. In *Proc. of Intl. Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 68–77. ACM Press, 2006.

48. H. Masuhara and G. Kiczales. Modeling Crosscutting in Aspect-Oriented Mechanisms. In *Proc. of European Conf. on Object-Oriented Programming*, volume 2743 of *LNCS*, pages 2–28. Springer, 2003.

49. S. McDirmid, M. Flatt, and W. C. Hsieh. Jiazzi: New-Age Components for Old-Fashioned Java. In *Proc. of Intl. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 211–222. ACM Press, 2001.

50. M. Mezini and K. Ostermann. Integrating Independent Components with On-Demand Remodularization. In *Proc. of Intl. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 52–67. ACM Press, 2002.

51. M. Mezini and K. Ostermann. Variability Management with Feature-Oriented Programming and Aspects. In *Proc. of Intl. Symposium on Foundations of Software Engineering*, pages 127–136. ACM Press, 2004.

52. N. Nystrom, S. Chong, and A. C. Myers. Scalable Extensibility via Nested Inheritance. In *Proc. of Intl. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 99–115. ACM Press, 2004.

53. M. Odersky, V. Cremet, C. Röckl, and M. Zenger. A Nominal Theory of Objects with Dependent Types. In *Proc. of European Conf. on Object-Oriented Programming*, volume 2743 of *LNCS*. Springer, 2003.

54. M. Odersky and M. Zenger. Scalable Component Abstractions. In *Proc. of Intl. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 41–57. ACM Press, 2005.

55. H. Ossher and W. Harrison. Combination of Inheritance Hierarchies. In *Proc. of Intl. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 25–40. ACM Press, 1992.

56. D. L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Comm. of the ACM*, 15(12):1053–1058, 1972.

57. C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proc. of European Conf. on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 419–443. Springer, 1997.

58. R. Pucella. Towards a Formalization for COM Part I: The Primitive Calculus. In *Proc. of Intl. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 331–342. ACM Press, 2002.

59. H. Rajan and K. J. Sullivan. Classpects: Unifying Aspect- and Object-Oriented Language Design. In *Proc. of Intl. Conf. on Software Engineering*, pages 59–68. ACM Press, 2005.

60. V. Rajlich. Changing the Paradigm of Software Engineering. *Comm. of the ACM*, 49(8):67–70, 2006.

61. T. Reenskaug, E. Andersen, A. Berre, A. Hurlen, A. Landmark, O. Lehne, E. Nordhagen, E. Ness-Ulseth, G. Oftedal, A. Skaar, and P. Stenslet. OORASS: Seamless Support for the Creation and Maintenance of Object-Oriented Systems. *Journal of Object-Oriented Programming*, 5(6):27–41, 1992.

62. J. C. Seco and L. Caires. A Basic Model of Typed Components. In *Proc. of European Conf. on Object-Oriented Programming*, volume 1850 of *LNCS*, pages 108–128. Springer, 2000.

63. Y. Smaragdakis and D. Batory. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. *ACM Trans. on Software Engineering and Methodology*, 11(2):215–255, 2002.

64. O. Spinczyk, D. Lohmann, and M. Urban. AspectC++: An AOP Extension for C++. *Software Developer's Journal*, pages 68–74, 2005.

65. C. Szyperski, D. Gruntz, and S. Murer. *Component Software. Beyond Object-Oriented Programming*. Addision-Wesley, 2002.

66. P. Tarr, H. Ossher, W. Harrison, and Jr. S. M. Sutton. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proc. of Intl. Conf. on Software Engineering*, pages 107–119. IEEE Computer Society, 1999.

67. S. Trujillo, M. Azanza, and O. Diaz. Generative Metaprogramming. In *Proc. of Intl. Conf. on Generative Programming and Component Engineering*, 2007.

68. S. Trujillo, D. Batory, and O. Diaz. Feature Refactoring a Multi-Representation Program into a Product Line. In *Proc. of Intl. Conf. on Generative Programming and Component Engineering*, pages 191–200. ACM Press, 2006.

69. S. Trujillo, D. Batory, and O. Díaz. Feature Oriented Model Driven Development: A Case Study for Portlets. In *Proc. of Intl. Conf. on Software Engineering*, pages 44–53. IEEE Computer Society, 2007.

70. E. Truyen, B. Vanhaute, W. Joosen, P. Verbaeten, and B. Nørregaard Jørgensen. Dynamic and Selective Combination of Extensions in Component-Based Applications. In *Proc. of Intl. Conf. on Software Engineering*, pages 233–242. IEEE Computer Society, 2001.

71. D. Tucker and S. Krishnamurthi. Pointcuts and Advice in Higher-Order Languages. In *Proc. of Intl. Conf. on Aspect-Oriented Software Development*, pages 158–167. ACM Press, 2003.

72. M. VanHilst and D. Notkin. Using Role Components in Implement Collaboration-based Designs. In *Proc. of Intl. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 359–369. ACM Press, 1996.

73. J.-Y. Vion-Dury, L. Bellissard, and V. Marangozov. A Component Calculus for Modeling the Olan Configuration Language. In *Proc. of Intl. Conf. on Coordination Languages and Models*, volume 1282 of *LNCS*, pages 392–409. Springer, 1997.

74. D. Walker, S. Zdancewic, and J. Ligatti. A Theory of Aspects. *SIGPLAN Notices*, 38(9):127–139, 2003.

75. N. Wirth. Program Development by Stepwise Refinement. *Comm. of the ACM*, 14(4):221–227, 1971.

76. M. Zenger. Type-Safe Prototype-Based Component Evolution. In *Proc. of European Conf. on Object-Oriented Programming*, volume 2374 of *LNCS*, pages 470–497. Springer, 2002.