Features, Modularity, and Variation Points

Don Batory
Dept. of Computer Science
University of Texas at Austin
batory@cs.utexas.edu

Peter Höfner
NICTA, Australia
University of New South Wales, Australia
peter.hoefner@nicta.com.au

Bernhard Möller, Andreas Zelend
Universität Augsburg, Germany
{bernhard.moeller,zelend}
@informatik.uni-augsburg.de

Abstract

A *feature interaction algebra (FIA)* is an abstract model of features, feature interactions, and their compositions. A *structured document algebra (SDA)* defines modules with variation points and how such modules compose. We present both FIA and SDA in this paper, and homomorphisms that relate FIA expresssions to SDA expressions. Doing so separates fundamental concepts of *Software Product Lines* (SPLs) that have previously been conflated and misunderstood. Our work also justifies observations and relationships that have been used in prior work on feature-based SPLs.

Categories and Subject Descriptors D.2.10 [*SoftwareEngineering*]: Design; D.2.8 [*Software Engineering*]: Software Architectures

General Terms Theory

Keywords software product lines, features, FOSD

1. Introduction

A *Software Product Line (SPL)* is a family of related programs constructed from a common set of assets. Variations in programs are explained by *features*—increments in program functionality. The assets of an SPL are modules that implement features. These modules are the building blocks of SPL programs.

Today's SPL researchers are exploring two rather different forms of feature-based modularity: alternative-based variation (a.k.a. *classical modularity*) and projectional variation (a.k.a. SYSGEN or *virtual modularity*). Classical modularity is what you would expect: there are physical files that define a feature module and tools that compose modules to produce a desired program. In contrast, virtual modularity is a preprocessor technology called *coloring* [6, 11, 15]. The idea is simple: the code of the `Blue` feature is painted blue; code of the `Green` feature is painted green. Whenever `Blue` is not needed in a product, all blue-colored code is removed or is said to be *projected out*. The tools for virtual modularity are historically-based on text preprocessors; more advanced tools color *abstract syntax trees (ASTs)* [6, 11, 15]. The current debate is which implementation technique is most appropriate for a SPL? No matter the choice, both implement the *same* abstractions—feature modules—in very different ways.

In this paper we show the foundational product-line concepts that *both* modularization technologies must implement. We develop a pair of related algebras that explain the closely related ideas of feature modularities and modularities based on *variation points (VPs)*, both well-known concepts in SPL implementation.

Our *Structured Document Algebra (SDA)* is a simple and effective way to construct documents in a modular way (e.g. text files or ASTs). SDA does *not* deal with features at all: it explains how modules with VPs and their associated fragments can be defined and composed. A second algebra, a *Feature Interaction Algebra (FIA)*, shows how features and their interactions are defined and composed. FIA is an evolution and improvement of our earlier work on a "Coloring Algebra" [6].

SDA and FIA are not the same: they deal with different concepts and are at different levels of abstraction. We explain a fundamental homomorphism that maps FIA expressions to SDA expressions (i.e. how feature compositions are mapped to structured module compositions) to relate these two algebras. Doing so separates fundamental SPL concepts that have previously been conflated and misunderstood. Our work also justifies observations and relationships that have been used in prior work on feature-based SPLs.

The contributions of our paper is a theory, grounded in experience and prior research, that provides:

- modules with variation points and their composition,
- features, feature interactions, and their composition,
- and a mapping (homomorphism) that relates them.

We also explore other properties of SDA that we believe have practical value in future VP-module implementations of SPLs, and generalize prior feature algebras [6, 16–18].

2. Structured Document Algebra

A classical concept in SPL construction is the variation point (VP). A VP is a labeled position in a program or document where contents can differ among programs in an SPL.

We present in this section a formal model of VPs, modules containing VPs, and compositions of such modules as the Structured Document Algebra (SDA). To keep SDA language-independent, we leave the exact nature of fragments open (e.g. text or AST) and view it as a parameter of the algebra.

2.1 Variation Points and Fragments

The basic ingredients of SDA are:

- a set $\mathtt{V}$ of VPs at which fragments may be inserted;
- a set $\mathtt{F}(\mathtt{V})$ of *fragments* which may, among other things, contain VPs from $\mathtt{V}$.

We use a very broad notion of VPs and fragments. Until stated otherwise, what we present below is standard fare for coloring and a VP interpretation of classical modularity.

Consider Figure 1a. It shows a Java file that defines class `A`. Three VPs and their associated fragments, indicated by bold left parentheses, are shown: $\mathtt{vp_a}$, $\mathtt{vp_b}$, and $\mathtt{vp_c}$. $\mathtt{vp_a}$ is a location in a directory at which a file can appear. (Such a VP is called a classpath). To us, this file is a fragment assigned to $\mathtt{vp_a}$. It is not the only fragment/file that could be assigned to $\mathtt{vp_a}$; another possibility is the file of Figure 1b. At most one of these two files/fragments can ever be assigned to $\mathtt{vp_a}$ at a time. This holds for all VPs—at most one fragment can be assigned to a VP at any one time. If there is no assignment, there is no file. We then say that the content of $\mathtt{vp_a}$ is *empty*. Emptiness may hold for all VPs.

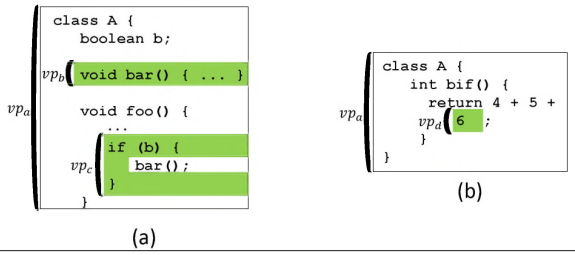

Figure 1. VPs and Fragments.

Now consider VPs $\mathtt{vp_b}$ and $\mathtt{vp_c}$ that are contained in the file-/fragment of Figure 1a. The fragment assigned to $\mathtt{vp_b}$ defines a method `bar`. The fragment at $\mathtt{vp_c}$ is a wrapper of the statement that calls `bar`. Fragments that are internal to a file are of two kinds: those that fill a VP and those that wrap statements at a VP [6, 15].

There is one more possibility: *default fragments*. In general every VP needs a default fragment. All VPs we have seen so far had empty defaults, which is the normal case. But default values need not to be empty; consider $\mathtt{vp_d}$ in Figure 1b. The fragment containing the number `6` fills $\mathtt{vp_d}$. But this VP has a default (not shown) so that if the fragment `6` is removed, the empty fragment cannot be default. Reason: the resulting code would be syntactically incorrect. The default fragment should be a natural number, to make the expressions semantically meaningful. Upon module composition, default fragments can be replaced by non-default fragments, but not vice versa.

Now we depart from standard ideas. It is typical in coloring and the SPL literature that: (1) a VP occurs only once in a SPL program and (2) the set of fragments that can be assigned to that VP are unique to that VP. SDA imposes no such limitations. A VP can appear in multiple places in a document (or documents); when one instance is assigned, they are all assigned the same fragment. Similarly, a single fragment need not be assigned to a unique VP; a fragment can be assigned to multiple distinct VPs. Both of these possibilities should be familiar to readers: aspects in AOP have advice (in the form of fragments) that can be applied to different join points (VPs).

Finally, a word on our above-mentioned "VP interpretation of classical modularity". In classical modularity, a VP corresponds to an interface and a fragment implements that interface. Delaware has shown that a formal (programming language) interface for a VP can be quite sophisticated, and so, too, can the fragment(s) that implement it [9]. And again, a VP has a default fragment (implementation) that can be overridden once by a non-default fragment (implementation).

2.2 SDA Basics

Modules. A *module* is a partial function $\mathtt{m} : \mathtt{V} \rightsquigarrow \mathtt{F(V)}$ such that its domain $\mathtt{dom(m)}$ is finite. VP $\mathtt{v}$ is *assigned* by $\mathtt{m}$ if $\mathtt{v} \in \mathtt{dom(m)}$, otherwise *unassigned* or *external*. Thus the domain $\mathtt{dom(m)}$ of a module is the set of VPs it "knows about" or that it administers.

A module $\mathtt{m}$ can be viewed in a number of ways:

- as a collection of fragments that instantiate the VPs of $\mathtt{dom(m)}$, i.e., a structured document;
- as filling certain VPs with contents (in term rewriting etc., it would be called a *substitution*); and
- as a generalized context-free grammar with $\mathtt{dom(m)}$ as the set of nonterminals and a production $\mathtt{v} \rightarrow \mathtt{m(v)}$ for each $\mathtt{v} \in \mathtt{dom(m)}$.

EXAMPLE 2.1. *Figure 2a is a sample file (module) which is structured by the assignment of fragments to its VPs. Its partial function is given in Figure 2b. Here variation points (and their corresponding fragments) are also grouped.* □

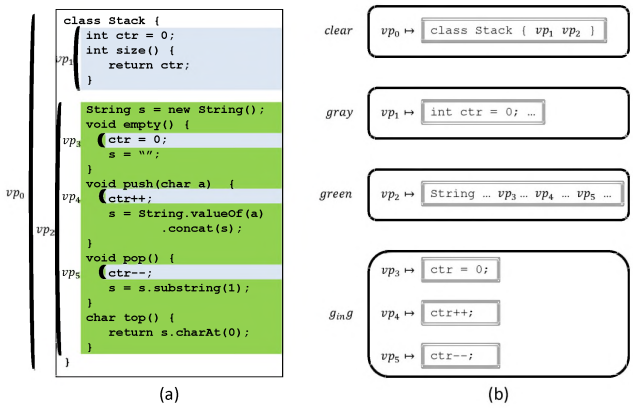

Figure 2. VPs, Fragments, and Modules.

By using partial functions rather than relations, a VP can be filled with at most one fragment (*uniqueness*).

A module should be cycle-free; we will deal with this later. The simplest module is the *empty module* $\mathtt{0}$, i.e., the empty partial map. Since $\mathtt{dom(0)} = \emptyset$, the empty module has no VPs.

We can compare two modules $\mathtt{m}, \mathtt{n}$ by their domains. We write $\mathtt{m} \subseteq_{\mathtt{dom}} \mathtt{n}$ iff $\mathtt{dom(m)} \subseteq \mathtt{dom(n)}$. Other ways to compare modules are discussed shortly.

Module Addition. We want to construct larger modules step by step by assigning more and more fragments to VPs. The central operation for this is module addition ($+$). Addition fuses two modules while maintaining uniqueness (and signaling an error upon a conflict). Desirable properties for $+$ are commutativity and associativity. If the modules to be combined have no VPs in common, the partial functions characterizing the modules can be easily combined. For example, $\mathtt{gray} + \mathtt{green}$ (Figure 2) is the partial function

$$\{\ \mathtt{vp_1} \mapsto \boxed{\mathtt{int\ ctr=\ 0;...}},\ \mathtt{vp_2} \mapsto \boxed{\mathtt{String...}}\ \}$$

To make the handling of conflicts algebraically nicer we put more structure into the set of fragments that could be assigned to a VP. Besides normal or non-default fragments $\mathtt{f}, \mathtt{f_1}, \mathtt{f_2}, \ldots$ we have a default fragment □ and an error $\lightning$.[1] An error occurs when two or more non-default fragments are assigned to the same VP. The arrangement of these elements is the flat lattice of Figure 3a.

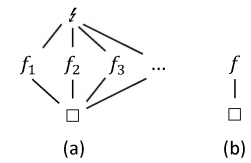

Figure 3. Lattice

Note: Coloring, as currently defined in CIDE and other text coloring tools, is less general. The lattice for them allows

[1] The $\lightning$ fragment has no VPs.

only a default and non-default value, as shown in Figure 3b. SDA deals with a generalization that would be expected for a true modular approach to SPL development.

To prepare a convenient definition of $+$ on modules, we denote the supremum operator in this lattice again by $+$:

$$\square + \mathtt{x} = \mathtt{x} \qquad \lightning + \mathtt{x} = \lightning$$
$$\mathtt{f_i} + \mathtt{f_i} = \mathtt{f_i} \qquad \mathtt{f_i} + \mathtt{f_j} = \lightning \ \ (\mathtt{i} \neq \mathtt{j}),$$

where $\mathtt{x}$ is an arbitrary element, i.e. $\mathtt{x} \in \{\square, \mathtt{f_i}, \lightning\}$. By standard lattice theory this operation is commutative, associative and idempotent. Moreover, it has $\square$ as its neutral element.

The default fragment $\square$ is what makes our definition of modules $\mathtt{m}$ possible: every assigned VP $\mathtt{v} \in \mathtt{dom}(\mathtt{m})$ has at least (even in the lattice sense) the default fragment $\square$ assigned to it.

Addition of modules can now be defined as the lifting of $+$ on fragments to partial functions:

$$(\mathtt{m}+\mathtt{n})(\mathtt{v}) =_{df} \begin{cases} \mathtt{m}(\mathtt{v}) & \text{if } \mathtt{v} \in \mathtt{dom}(\mathtt{m}) - \mathtt{dom}(\mathtt{n}) \\ \mathtt{n}(\mathtt{v}) & \text{if } \mathtt{v} \in \mathtt{dom}(\mathtt{n}) - \mathtt{dom}(\mathtt{m}) \\ \mathtt{m}(\mathtt{v}) + \mathtt{n}(\mathtt{v}) & \text{if } \mathtt{v} \in \mathtt{dom}(\mathtt{m}) \cap \mathtt{dom}(\mathtt{n}) \\ \text{undefined} & \text{if } \mathtt{v} \notin \mathtt{dom}(\mathtt{m}) \cup \mathtt{dom}(\mathtt{n}) \end{cases}$$

If in the third case $\mathtt{m}(\mathtt{v}) \neq \mathtt{n}(\mathtt{v})$ and $\mathtt{m}(\mathtt{v}), \mathtt{n}(\mathtt{v}) \neq \square$ then $(\mathtt{m}+\mathtt{n})(\mathtt{v}) = \lightning$, thus signaling an error.[2]

By the above laws, the set of modules forms a commutative monoid under $+$. Therefore, for a finite family $\{\mathtt{m_i}\}_{\mathtt{i}\in\mathtt{I}}$ the sum $\sum_{\mathtt{i}\in\mathtt{I}} \mathtt{m_i}$ is well-defined. If $\mathtt{I} = \emptyset$ is the empty set of indices we get, as is standard, $\sum_{\mathtt{i}\in\mathtt{I}} \mathtt{m_i} = \mathtt{0}$.

EXAMPLE 2.2. *Figure 2b shows four modules. The* `clear` *module contains a single fragment that is assigned to* $\mathtt{vp_0}$. *The* `gray` *module contains a single fragment that is assigned to* $\mathtt{vp_1}$. *The* `green` *module contains a single fragment that is assigned to* $\mathtt{vp_2}$. *And the* $\mathtt{g_{in}g}$ *(gray in green) module contains fragments that are assigned to* $\mathtt{vp_3}$, $\mathtt{vp_4}$, *and* $\mathtt{vp_5}$. *The module summation* $\mathtt{clear} + \mathtt{gray} + \mathtt{green} + \mathtt{g_{in}g}$ *is the module of Figure 2a.* $\square$

Implementation. A simple example suggests several ways in which SDA modules can be implemented. Figure 4a shows how preprocessors can define three non-default fragments (labeled `BLUE`, `GREEN`, `RED`) and a default for an implicit variation point. Figure 4b shows how this might be rendered in a "coloring" tool where the fragments of a VP are explicitly shown. (There is no need to actually "see" the name of a VP). However, Figure 4b would require significant engineering: a Java compiler would have to understand the preprocessor semantics of coloring (Figure 4a) so as not to alert programmers that the `GREEN` fragments and beyond are unreachable. A more likely possibility—which is consistent with current text coloring tools—would be to fool the compiler that the code of Figure 4c is the definition of the `add` method, where a projection would produce a simpler method with only one assignment to variable `result`.

These ideas are, in effect, standard fare for SPL development, except that the tool support needs to be beautified by coloring and VP recognition. Code fragments or *mini-modules* can indeed be expressed in terms of a classical module system; see [9] for examples.

[2] This definition can be recoded in terms of total functions, which makes it easier to see that the $+$ operation indeed is commutative, associative and idempotent, hence induces a lattice, too. Moreover, it has the empty module $\mathtt{0}$ as its neutral element and satisfies $\mathtt{dom}(\mathtt{m}+\mathtt{n}) = \mathtt{dom}(\mathtt{m}) \cup \mathtt{dom}(\mathtt{n})$.

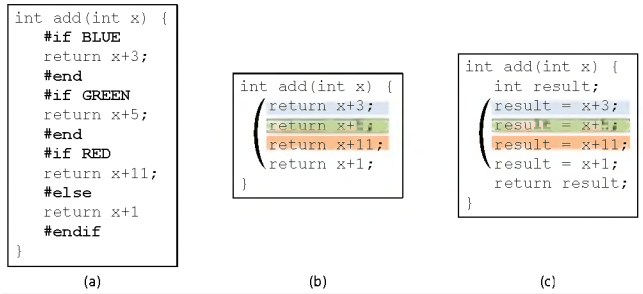

Figure 4. Module Implementations.

2.3 SDA Utilities

Cycle-Freeness. For a fragment $\mathtt{f} \in \mathtt{F}(\mathtt{V})$ denote by $\mathtt{VP}(\mathtt{f})$ the set of VPs that are in $\mathtt{f}$. We define a *direct dependence relation* $\mathtt{dep_m} \subseteq \mathtt{V} \times \mathtt{V}$ within a module $\mathtt{m}$:

$$\mathtt{v}\ \mathtt{dep_m}\ \mathtt{w} \ \Leftrightarrow_{df} \ \mathtt{v} \in \mathtt{dom}(\mathtt{m}) \wedge \mathtt{w} \in \mathtt{VP}(\mathtt{m}(\mathtt{v}))$$

This means that VP $\mathtt{w}$ occurs in the fragment assigned to VP $\mathtt{v}$ by $\mathtt{m}$. For example, in Figure 2, $\mathtt{vp_2}\ \mathtt{dep_{green}}\ \mathtt{vp_3}$. A module $\mathtt{m}$ (*which could be formed by the summation of other modules*) is *acyclic* if no VP depends directly or indirectly on itself, i.e., no VP $\mathtt{v}$ satisfies $\mathtt{v}\ \mathtt{dep_m^+}\ \mathtt{v}$, where $\mathtt{dep_m^+}$ is the transitive closure of $\mathtt{dep_m}$. Acyclicity states the obvious property that ASTs should be trees and not cyclic graphs, and similarly text-based documents with VPs should not have cycles either.

There are several ways in which `dep` can be implemented. One way is to examine every module $\mathtt{m}$ and create a graph $\mathtt{G}$ where vertex $\mathtt{v}$ points to $\mathtt{w}$ iff $\mathtt{v}\ \mathtt{dep_m}\ \mathtt{w}$. Once $\mathtt{G}$ has been assembled, a cycle checking algorithm can be applied.

Another variation that arises in SPL implementations is that modules can be composed in a predetermined order [4, 5]. This admits the possibility of assigning labels to VPs such that if module `m1` is always composed before `m2`, then all labels of VPs in `m1` must have a lower value than those in `m2`. In this way, testing for cycles can be quite efficient. Periodically, however, VPs may need to be relabeled when certain code refactorings are performed.

Henceforth we only consider cycle-free modules.

Assembling Fragments. We now describe how to assemble a structured document into a single fragment (while "forgetting" the structure). To define this formally we use an auxiliary function $\mathtt{single_fill}(\mathtt{f},\mathtt{m})$. It takes a fragment $\mathtt{f}$ and a module $\mathtt{m}$ and yields the fragment that results from $\mathtt{f}$ by replacing, in parallel, all occurrences of every $\mathtt{w} \in \mathtt{VP}(\mathtt{f})$ by the corresponding fragment $\mathtt{m}(\mathtt{w})$ (if any). The precise definition of `single_fill` depends on the special type of fragments considered; as stated in the introduction we want to keep that parametric. For an acyclic module $\mathtt{m}$ and $\mathtt{v} \in \mathtt{dom}(\mathtt{m})$, the fragment $\mathtt{frag}(\mathtt{v},\mathtt{m})$ can be computed by iterating the `single_fill` function. By acyclicity of $\mathtt{m}$ this always terminates. To cope with the case of unassigned VPs we assume that every VP is also a fragment, i.e., that $\mathtt{V} \subseteq \mathtt{F}(\mathtt{V})$, and simply can be left unchanged by the assembly function. A corresponding program looks as follows:

```
fragment frag (vp v, module m){
  fragment f = v;
  while (VP(f) ∩ dom(m) != ∅)
    f = fillin(f,m);
  return f;    }
```

Once again, there are many ways in which to implement the above. Normally the target of `frag` is the VP of an entire file. By assigning a unique VP (such as $\mathtt{vp_0}$) for file fragments, the result

of $\texttt{frag}(\texttt{vp}_0, \texttt{m})$ for a (possibly composed) module $\texttt{m}$ is the text of the entire file. A fast way to do this is to hash fragments on the VPs to which they can be assigned. From $\texttt{vp}_0$ its fragment can be found quickly, and so too can each of its VPs and their assigned fragments, recursively. An error $\lightning$ is issued when two non-default fragments are assigned to the same VP.

Module Equivalence. The left-hand side of Figure 5 shows a fragment assigned to VP $\texttt{v}$. The right-hand side shows a composition of fragments whose content is equivalent to the left-hand-side, but has more VPs (e.g. $\texttt{g}$ and $\texttt{h}$). By "forgetting" these extra VPs on the right-hand-side we can define equivalence.

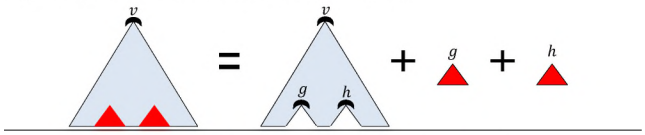

Figure 5. Equivalence of Two Fragments.

The function $\texttt{frag}$ does this. Consider modules $\texttt{m}_1, \texttt{m}_2$ and VPs $\texttt{v}_1, \texttt{v}_2$ with $\texttt{v}_i \in \texttt{dom}(\texttt{m}_i)$. Then we call the pairs $(\texttt{v}_1, \texttt{m}_1)$ and $(\texttt{v}_2, \texttt{m}_2)$ *equivalent* if the same fragment can be assembled from them, formally:

$$(\texttt{v}_1, \texttt{m}_1) \equiv (\texttt{v}_2, \texttt{m}_2) \Leftrightarrow_{df} \texttt{frag}(\texttt{v}_1, \texttt{m}_1) = \texttt{frag}(\texttt{v}_2, \texttt{m}_2)\ .$$

EXAMPLE 2.3. *Let module* $\texttt{m}_1$ *assign VP* $\texttt{v}$ *the fragment on the left-hand side of Figure 5. Let module* $\texttt{m}_2$ *assign VPs* $\texttt{v}, \texttt{h}, \texttt{g}$ *the fragments on the right of Figure 5. Then* $(\texttt{v}, \texttt{m}_1) \equiv (\texttt{v}, \texttt{m}_2)$. □

Two modules $\texttt{m}_1, \texttt{m}_2$ are *equivalent* if they fill the same variation points in the same way. Formally:

$$\texttt{m}_1 \equiv \texttt{m}_2 \Leftrightarrow_{df} \texttt{dom}(\texttt{m}_1) = \texttt{dom}(\texttt{m}_2) \wedge \\ \forall \texttt{v} \in \texttt{dom}(\texttt{m}_1) : (\texttt{v}, \texttt{m}_1) \equiv (\texttt{v}, \texttt{m}_2)\ .$$

If modules consist of AST fragments, it is a simple matter of starting at both AST roots and comparing whether each AST has the same sequence of nodes. If the modules consist of text fragments, the corresponding calls to $\texttt{frag}$ will produce two Strings. In this case the equality involved in the definition of $\equiv$ should be equality up to white space.

More. The essence of SDA was just presented. More results on SDA are presented later in Section 5.

2.4 Extended Example

The expression problem is a classical example of a product line [20]. Figure 6 shows three modules $\texttt{base}, \texttt{print}, \texttt{eval}$. Module $\texttt{base}$ represents the shell of a program that can encode the sum and product of integers as operator trees. Module $\texttt{print}$ enables operator trees to be printed and module $\texttt{eval}$ enables operator trees to be evaluated.

Figure 7a shows the module sum $\texttt{base} + \texttt{print}$, a program that can create and print sums and products of integers. Note that VPs $\{\texttt{vp}_2, \texttt{vp}_4, \texttt{vp}_6, \texttt{vp}_8\}$ have empty default fragments. VPs $\{\texttt{vp}_1, \texttt{vp}_3, \texttt{vp}_5, \texttt{vp}_7\}$ have been assigned their non-default fragments (and whose VP names are not shown). Figure 7b shows the module sum $\texttt{base} + \texttt{print} + \texttt{eval}$, a program that can create, print, and evaluate sums and products of integers.

As an aside, how the semantics of these modules is determined—*something that SDA does not provide*—is the subject of the next section (Feature Interaction Algebra) and the following section (Homomorphisms).

3. Feature Interaction Algebra

Let us now look at features, their composition, and interactions in an abstract way. We define a *Feature Interaction Algebra (FIA)*

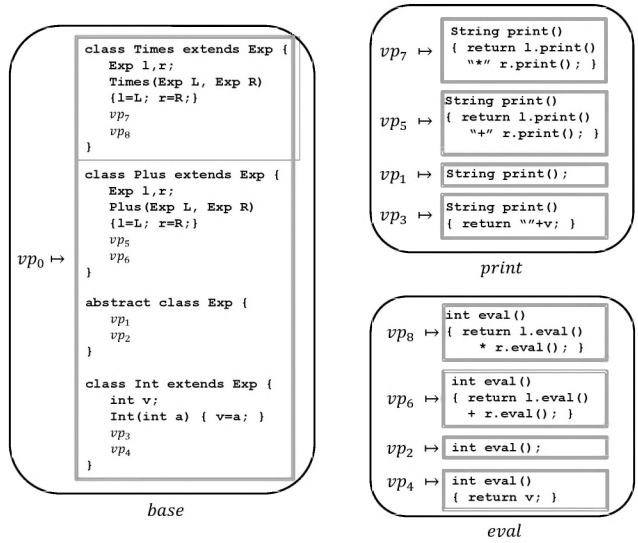

Figure 6. Three Modules.

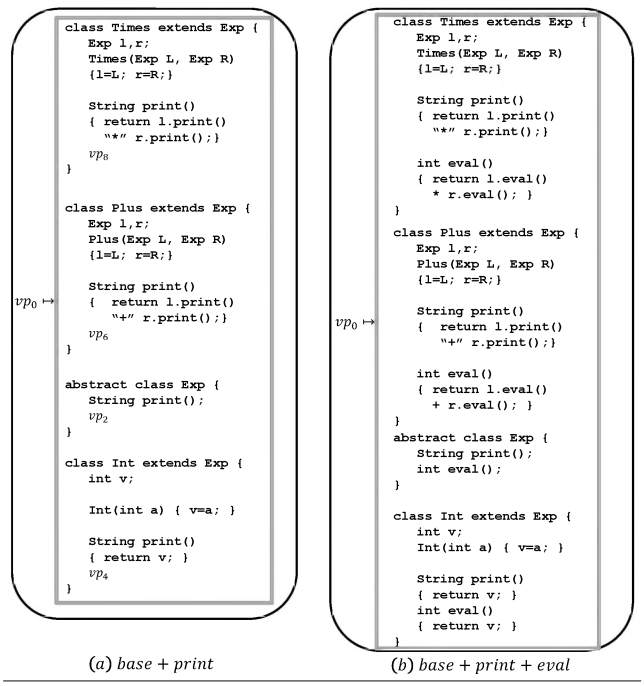

Figure 7. Different Module Summations.

that acts at the level of specifications of feature-based systems. To keep FIA language-independent, we leave the implementation of features open and view it as a parameter of the algebra. In Section 4, we define projections from FIA to SDA to show how FIA expressions can be implemented.

3.1 The Axioms of FIA

We first deal with the operator $+$ for feature composition. The minimal set of axioms for feature composition is:

$$\texttt{A} + \texttt{B} = \texttt{B} + \texttt{A} \qquad (\texttt{A} + \texttt{B}) + \texttt{C} = \texttt{A} + (\texttt{B} + \texttt{C}) \\ \texttt{A} + \texttt{0} = \texttt{0}$$

These are commutativity, associativity, and a neutral element $\texttt{0}$. Algebraically this means that FIA is based on an abstract set $\texttt{F}$ of features and has a binary operator $+$ and a distinguished element

$0 \in F$ such that $(F, +, 0)$ forms a commutative monoid. 0 is the empty or null feature.

Features interact. The core idea is that features A and B have well-defined behaviors in isolation, but behave differently when they are together. In such cases, a mediating feature $A\#B$ is needed to make A and B work correctly together. We say that features A and B *interact* if $A\#B \neq 0$; otherwise they do not interact. When features interact, we say $A\#B$ denotes their *resolution*.

In FIA, feature interaction $\#$ is again a binary operator on F for which we stipulate the following axioms:

$$A \# B = B \# A \qquad (A \# B) \# C = A \#(B \# C)$$
$$A \# 0 = 0 \qquad A \#(B + C) = (A \# B) + (A \# C)$$

The first pair of axioms (commutativity and associativity) state that the order of interaction does not matter—the resolution of the interaction of A and B (or B and A) is the same. The third axiom expresses that the empty feature 0 cannot interact with any feature A, so that the "repair" feature $A \# 0$ is empty as well. The last axiom (distributivity) expresses that we view feature interaction as modular and the entire approach *compositional*: feature interaction of composed modules can be determined from the feature interaction of their components.

EXAMPLE 3.1. *A classic example from telephony are the call waiting* CW *and call forwarding* CF *features.* CF *enables a customer to specify a secondary phone number to which additional calls are forwarded when a phone is busy.* CW *allows one call to be suspended while another call is answered. If both features are present and a call comes in while another is active, the phone system has to decide whether the call should be forwarded or the user should be notified that another call has arrived. The resolution is provided by* CW#CF. *Without a resolution, the phone system may behave or terminate erroneously.* □

When architects select two (or more) features, they want their interaction resolutions to be included. For this purpose, we define the *full product* of two features as:

$$A \times B = (A \# B) + A + B\,.$$

That is, when architects want features A and B, they also want both features to work correctly together, which requires the addition of their interaction resolution $A\#B$.

The above axioms form the base of our abstract view. We call the structure $(F, +, \#, 0)$ a *Feature Interaction Algebra (FIA)*. The axioms of FIA are a subset of the axioms of a commutative semiring (using $+$ as addition and $\#$ as multiplication): the part missing is a multiplicative unit ($A \# 1 = A$).[3] As a consequence the axioms are consistent and there are well known instances. Adding a unit would result in the product family algebra presented in [12].

3.2 Variants of the Axiomatization

Choosing axioms to set up a theory is crucial. On the one hand, the axioms have to reflect the natural behavior of SPLs; on the other hand they should neither contradict each other nor lead to unwanted and undesirable behavior.

We have explored all of the possible axiom combinations for feature interaction algebras and can report that very few are useful. That is, there are very few additions to FIA that will not lead to contradictions. In this section, we report our findings on the space of legal possibilities.

We discovered that the only sensible variations to FIA are the axioms for self-addition ($A + A$) and self-interaction ($A \# A$). For each, either the operation is idempotent ($A + A = A$ or $A\#A = A$), or is an involution ($A + A = 0$ or $A\#A = 0$), or we can say nothing (as we do now).

In [6], we showed that adding the axioms $A + A = 0$ and $A \# A = 0$ to FIA axioms leads to a consistent algebra. However, the analysis of that set of axioms in [13] showed that they seem to admit only very tricky and not very intuitive mathematical models, contrary to the axiom set in the present paper. We have since discovered that adding axioms $A + A = 0$ and $A \# A = A$ (self-interaction is idempotent) to FIA are also consistent. The benefit of involution axioms is that they allow feature removal and the ability to solve equations for unknowns (given $A+D = A+B+C$ we could infer $D = B + C$). As we will introduce a subtraction operation in SDA later on, we do not need one for FIA. Moreover, it seems conceptually more appealing to keep subtraction and addition separate.

However, we were surprised by the limitations imposed by the idempotence of sum ($A + A = A$), to which we now turn.

3.3 Properties of FIA with Axiom $A + A = A$

Throughout this section we assume the idempotence axiom $A+A = A$ in addition to the FIA axioms. First, it is well-known that then $+$ induces an upper semilattice w.r.t. the *natural order* or *subsumption order* defined by:

$$A \leq B \Leftrightarrow_{df} A + B = B\,.$$

This means that all features of (possibly composite) A are already contained in B. As a partial order, $\leq$ is reflexive, transitive and antisymmetric. Moreover, 0 is the least element and addition $+$ and interaction $\#$ are monotonically increasing w.r.t. to $\leq$:

$$A \leq B \Rightarrow (A + C) \leq (B + C)$$
$$A \leq B \Rightarrow (A \# C) \leq (B \# C)$$

Let us briefly summarize some facts about FIA.

LEMMA 3.2. *Assume an FIA* $(F, +, \#, 0)$. *Then*

1. $A + B \leq A \times B$.
2. $(A + B) \times C = (A \times C) + (B \times C)$.

Proof. Part 1 is immediate from the definitions of $\times$ and $\leq$, while Part 2 follows by distributivity of $\#$ over $+$, commutativity, and $+$-idempotence. □

One might be tempted to add axioms for feature interaction. For example, one might want to enforce that self-interaction does not have any effect, i.e. $A \# A = 0$. However, this has unintended consequences, as we will now show.

LEMMA 3.3. *If* $\#$ *satisfies the axiom of involution, i.e.,* $A \# A = 0$, *then* $\#$ *is constant and* $\times$ *and* $+$ *coincide.*

Proof. Since $A, B \leq A + B$ and $\#$ is monotonic, we have, together with involution of $\#$:

$$A \# B \leq (A + B) \#(A + B) = 0\,.$$

As mentioned, 0 is the least element w.r.t. $\leq$, so that by antisymmetry of $\leq$ we infer $A \# B = 0$. The second claim follows from the definition of $\times$. □

More generally, we cannot set self-interaction to an arbitrary fixed value.

LEMMA 3.4. *Assume* $A \# A = R$ *for all* A. *Then* $A \# B = R$ *for all* A, B, *i.e.,* $\#$ *is a constant operator and hence* $A \times B = A + B + R$.

The proof is similar to that of the previous lemma.

Another immediate idea how to characterize self-interaction would be the idempotence axiom $A \# A = A$. This could be interpreted as saying that a module cannot interact with itself and hence

[3] It might be interesting to add a unit 1—the neutral element w.r.t. $\#$. However its interpretation in the context of features is unclear.

is its own repair. However, this leads to the same effect as the involution axiom.

LEMMA 3.5. *If # is idempotent then $\times$ and $+$ coincide.*

Proof. As in the Lemma we derive $\mathtt{A}\,\#\,\mathtt{B} \leq (\mathtt{A}+\mathtt{B})\,\#(\mathtt{A}+\mathtt{B})$. But since # is idempotent, we have $(\mathtt{A}+\mathtt{B})\,\#(\mathtt{A}+\mathtt{B}) = \mathtt{A}+\mathtt{B}$. By definition of $\leq$, $\mathtt{A}\,\#\,\mathtt{B} \leq \mathtt{A}+\mathtt{B}$ is equivalent to $\mathtt{A}\,\#\,\mathtt{B}+\mathtt{A}+\mathtt{B} = \mathtt{A}+\mathtt{B}$ and hence $\mathtt{A}\times\mathtt{B} = \mathtt{A}+\mathtt{B}$. □

In short, in presence of idempotence of $+$ we can neither assume involution nor idempotence of feature self-interaction without making $\times$ and # collapse into the same operator, rendering $\times$ useless.

As a consequence of the above lemmas, we have to leave self-interaction unspecified. For some features A we may have $\mathtt{A}\,\#\,\mathtt{A} = \mathtt{0}$. But there is also a reasonable interpretation of the case $\mathtt{A}\,\#\,\mathtt{A} \neq \mathtt{0}$: this may be viewed as an indication that A is somehow defective and needs the repair $\mathtt{A}\,\#\,\mathtt{A}$.

3.4 Recap

Our FIA axioms form a minimal set for a useful theory. Problems arise when feature replications occur—that is, how to define the semantics of expressions like $\mathtt{A}+\mathtt{A}$, $\mathtt{A}\,\#\,\mathtt{A}$, and consequently $\mathtt{A}\times\mathtt{A}$? This raises an interesting issue: in classical feature models, features are never replicated; hence expressions like $\mathtt{A}+\mathtt{A}$, $\mathtt{A}\,\#\,\mathtt{A}$, and $\mathtt{A}\times\mathtt{A}$ never arise.

It is still an on-going debate whether features can indeed be replicated (c.f. [3]). Settling this issue is the subject of future work. What we can say at this point is that our FIA is consistent with classical feature modeling.

4. Homomorphisms

FIA and SDA are distinct algebras that deal with different concepts and are at different levels of abstraction. FIA deals with the terms (features and feature interactions) that are the semantic building blocks of SPLs. SDA deals with the syntax of modules with VPs and their composition. In this section, we define the relationship between SDA and FIA (i.e. syntax and semantics).

Our vision of this relationship is displayed in Figure 8. A user selects features to specify a desired member of an SPL. The cross-product of selected features is taken to produce an FIA expression of the target program. This expression is then mapped to an SDA module expression. Evaluating the SDA module expression constructs the program. Here we assume classical feature models—models that do not have attributes or permit replication of features; later we discuss how these restrictions can be removed.

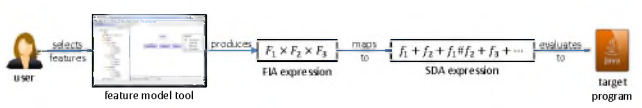

Figure 8. Feature Model Tools, FIA, and SDA.

The key to Figure 8 is the homomorphism μ : FIA $\rightarrow$ SDA that maps an FIA expression to an SDA expression. The simplest μ is the definition of coloring.

4.1 The Coloring Homomorphism

Coloring, first, is a preprocessor technology. Every feature of a product line is assigned a distinct color. A program P (which represents the entire code base of an SPL) is colored: all code belonging to the BLUE feature is painted blue; all code belonging to the RED feature is painted red. Every fragment of code in P is painted by at least one color. Coloring also is a projection technology: if a code fragment is painted multiple colors (e.g. BLUE $\wedge$ RED), it appears *only* when all of its colors (BLUE and RED) are selected.

Consider the Venn diagram of Figure 9. The entire codebase of a program P is represented by the area within the rings for colors RED, BLUE, and GREEN. Every partition in this diagram represents the contents (code fragments) of a unique SDA module. There are seven SDA modules total: $\mathtt{r}, \mathtt{g}, \mathtt{b}, \mathtt{r}\#\mathtt{g}, \mathtt{g}\#\mathtt{b}, \mathtt{r}\#\mathtt{b}, \mathtt{r}\#\mathtt{b}\#\mathtt{g}$. The sum of these modules yields P:

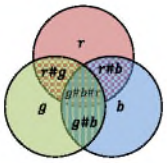

Figure 9. Venn Diagram

$$\mathtt{P} = \mathtt{r}+\mathtt{g}+\mathtt{b}+\mathtt{r}\#\mathtt{g}+\mathtt{g}\#\mathtt{b}+\mathtt{r}\#\mathtt{b}+\mathtt{r}\#\mathtt{b}\#\mathtt{g}$$

Note that the token "#" in the module names on the right-hand side is *not* the operator # of feature interaction on modules, but simply a character in a name. The authors are still debating whether "#" on the right-hand side should be named differently. We explore this point further in Section 4.2.

A characteristic of coloring is that each term of a feature expression (i.e. features and feature interactions) maps directly to a distinct SDA module. For Figure 9:

$$\begin{array}{rcl}
\mu(\mathtt{RED}) & = & \mathtt{r} \\
\mu(\mathtt{GREEN}) & = & \mathtt{g} \\
\mu(\mathtt{BLUE}) & = & \mathtt{b} \\
\mu(\mathtt{RED}\,\#\,\mathtt{GREEN}) & = & \mathtt{r}\#\mathtt{g} \\
\mu(\mathtt{RED}\,\#\,\mathtt{BLUE}) & = & \mathtt{r}\#\mathtt{b} \\
\mu(\mathtt{BLUE}\,\#\,\mathtt{GREEN}) & = & \mathtt{b}\#\mathtt{g} \\
\mu(\mathtt{RED}\,\#\,\mathtt{BLUE}\#\mathtt{GREEN}) & = & \mathtt{r}\#\mathtt{b}\#\mathtt{g}
\end{array}$$

Here is the general mapping: Let FS be the set of features and FIS be the set of feature interactions. *Coloring is the homomorphism that maps sums features and feature interactions to sums of SDA modules*:

$$\mu(\mathtt{A}+\mathtt{B}) = \mu(\mathtt{A})+\mu(\mathtt{B}) \quad // \text{ for all } \mathtt{A},\mathtt{B} \in (\mathtt{FS}\cup\mathtt{FIS})$$

4.2 Interaction Homomorphism

There is one other way to relate FIA to SDA—the *interaction homomorphism*:

$$\mu(\mathtt{A}\,\#\,\mathtt{B}) = \mu(\mathtt{A})\ \#_\mu\ \mu(\mathtt{B}) \quad // \text{ for all } \mathtt{A},\mathtt{B} \in (\mathtt{FS}\cup\mathtt{FIS})$$

That is, given modules $\mu(\mathtt{A})$ and $\mu(\mathtt{B})$, one can compute (using a new SDA operation $\#_\mu$) the module of their interaction $\mu(\mathtt{A}\,\#\,\mathtt{B})$.

In general, an algorithm for $\#_\mu$ is undecidable. There is not enough information within $\mu(\mathtt{A})$ and $\mu(\mathtt{B})$ to know what changes must be contained in $\mu(\mathtt{A}\,\#\,\mathtt{B})$ to lead to the desired program. Research on feature interactions can *detect* when $\mu(\mathtt{A}\,\#\,\mathtt{B})$ is non-empty (meaning that A and B interact), but such analyses cannot always compute the resolution (contents of $\mu(\mathtt{A}\,\#\,\mathtt{B})$) [10]. Global information about the program is needed.

Coloring is no exception. It is impossible to compute $\mu(\mathtt{A}\,\#\,\mathtt{B})$ from $\mu(\mathtt{A})$ and $\mu(\mathtt{B})$. But coloring does the next best thing, the topic of the next section.

4.3 Virtual Modularity

VPs are implicit in coloring. At every point in a document where coloring changes, an implicit VP is created. Figure 10a shows an AST where the coloring of the fragment at $\mathtt{vp}_\alpha$ changes to BLUE at $\mathtt{vp}_\beta$. Figure 10b shows how this might be rendered by a colored text editor. Figure 10c shows an explicit encoding using a preprocessor.

Because one colors the entire code base of a product line, it is possible to compute the contents of SDA modules and their VPs. *This is the essence of virtual modularity.* Let F denote a feature and let f denote its SDA module. Again let $\mathcal{F}$ denote the set of all code fragments (ASTs) that have the F color, and $\bar{\mathcal{F}}$ denote the set of all code fragments (ASTs) that do *not* have the F color.

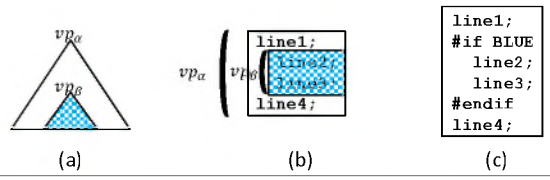

Figure 10. Coloring and VPs.

The contents of module $\mathtt{f_i\#f_j}$, where # is *not an operation but simply a character in a composite name*, are computed by the formula:

$$\mathtt{f_i\#f_j} \;=\; \mathcal{F}_\mathtt{i} \cap \mathcal{F}_\mathtt{j} \cap \bigcap_{\mathtt{r}\neq\mathtt{i,j}} \bar{\mathcal{F}}_\mathtt{r}$$

That is, the code fragments of module $\mathtt{f_i\#f_j}$ are the intersection of the ASTs of $\mathcal{F}_\mathtt{i}$ and $\mathcal{F}_\mathtt{j}$ and the removal of all ASTs that belong to $\mathcal{F}_\mathtt{r}$ where $\mathtt{r} \notin \{\mathtt{i},\mathtt{j}\}$. This formula generalizes in the obvious way to compute modules for individual features ($\mu(\mathtt{F}) = \mathtt{f}$) as well as modules for n-way interactions ($\mu(\mathtt{F_1\#\ldots\#F_n}) = \mathtt{f_1\#\ldots\#f_n}$).

> Note this does not contradict what we said in Section 4.1: to compute $\mathtt{f_i\#f_j}$ one needs much more than modules $\mathtt{f_i}$ and $\mathtt{f_j}$—one needs knowledge of the coloring of the entire program P to determine the contents of module $\mathtt{f_i\#f_j}$.

4.4 Other Homomorphisms

FIA defines the key terms (features and feature interactions) that are the semantic building blocks of SPLs. If colored modules are not used as an implementation, other homomorphisms are needed to map FIA terms to concrete representations. Here are some recent or well-known results with non-coloring or non-SDA implementations of features:

- Apel et al [1] showed how different program representations can be encoded as syntax-trees and feature composition maps to syntax-tree composition. Given the grammar of a language λ and rules for composing λ syntax-trees, FeatureHouse generates a tool that implements the λ homomorphism:

  $$\lambda(\mathtt{A}+\mathtt{B}) \;=\; \lambda(\mathtt{A}) \;+_\lambda\; \lambda(\mathtt{B})$$

  That is, a FeatureHouse-generated tool parses the λ modules for features A and B and composes them with the syntax-tree composition operation $+_\lambda$.

- Siegmund et al. [21, 22] showed how to compute a performance estimate π for a given workload for any program in an SPL. Procedures were given to estimate the performance delta that features and feature interactions contribute to a program. Assuming performance deltas of features are arithmetically added, their work relied on the π homomorphism:

  $$\pi(\mathtt{A}+\mathtt{B}) \;=\; \pi(\mathtt{A}) + \pi(\mathtt{B})$$

  Surprisingly accurate predictions were reported using this simple approach.

- The most sophisticated use to date of homomorphisms is by Delaware et al. [9], who showed how proofs of correctness of a program could be synthesized from its FIA expression. The SPL contained dialects of Featherweight Java. An integral part of any type system are the meta-theoretic proofs that show type soundness—the guarantee that the type system statically enforces the desired run-time behavior of a language, typically preservation and progress.[4] Four different representations of each feature—syntax, typing rules for preservation, evaluation rules for progress, and the proofs—were encoded as distinct modules in the Coq proof assistant [7]. Two homomorphisms were used: δ composed syntax, typing rule, and evaluation rule modules; ψ composed proof modules. Both δ and ψ were implemented as Coq libraries:

  $$\begin{aligned}\delta(\mathtt{A}+\mathtt{B}) &= \delta(\mathtt{A}) \;+_\delta\; \delta(\mathtt{B})\\ \psi(\mathtt{A}+\mathtt{B}) &= \psi(\mathtt{A}) \;+_\psi\; \psi(\mathtt{B})\end{aligned}$$

  Each distinct Coq module for feature syntax, feature typing rules, etc. is certified once by Coq (this is the expensive part) and reused as-is. Coq mechanically verifies the correctness of a composite proof by a simple interface check.

4.5 Removing Restrictions

Classical feature models do not allow features to have attributes. Attributes can be layered on top of FIA and SDA algebras by permitting code fragments to be parameterized macros. Once a module m is composed, the values of feature attributes can be substituted. This would give our approach the power of classical preprocessors [14].

As we said in Section 3.4, classical feature models also do not permit multiple instances of features. Whether features can indeed be replicated is still a subject of debate [3]. SDA *does* permit replicated code fragments, as noted in Section 2.1. But replicated modules simply do not occur, as module sum is idempotent ($\mathtt{m}+\mathtt{m} = \mathtt{m}$). It is possible that this debate can be resolved using run-time instances of a code module to create replicas. Settling this issue is yet another subject of future work.

5. SDA Extras

SDA has a wealth of useful capabilities beyond addition. We show some potentials and relationships to prior work.

5.1 Other Operations on Modules

Deletion and Subtraction. There are two ways of defining "inverses" to addition.

Variant I: We define the operation of *deletion* to shrink the domain of a partial map. For a module m and a set $\mathtt{U} \subseteq \mathtt{V}$ of VPs we define the module $\mathtt{m} \ominus \mathtt{U}$ by:

$$(\mathtt{m} \ominus \mathtt{U})(\mathtt{v}) \;=_{df}\; \begin{cases} \mathtt{m}(\mathtt{v}) & \text{if } \mathtt{v} \in (\mathtt{dom}(\mathtt{m}) - \mathtt{U}) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Deletion satisfies the following laws, which are shown by straightforward calculation:

$$\begin{aligned}
\mathtt{dom}(\mathtt{m} \ominus \mathtt{U}) &= \mathtt{dom}(\mathtt{m}) - \mathtt{U}\\
\emptyset \ominus \mathtt{U} &= \emptyset\\
(\mathtt{m}+\mathtt{n}) \ominus \mathtt{U} &= (\mathtt{m} \ominus \mathtt{U}) + (\mathtt{n} \ominus \mathtt{U})\\
\mathtt{m} \ominus (\mathtt{U} \cup \mathtt{W}) &= (\mathtt{m} \ominus \mathtt{U}) \ominus \mathtt{W}\\
\mathtt{m} \ominus \emptyset &= \mathtt{m}\\
\mathtt{m} \ominus \mathtt{dom}(\mathtt{m}) &= \emptyset\\
\mathtt{m} \ominus \mathtt{U} &\subseteq \mathtt{m}\\
\mathtt{dom}(\mathtt{m}) \subseteq \mathtt{U} &\Leftrightarrow (\mathtt{m} \ominus \mathtt{U}) = \emptyset
\end{aligned}$$

A major drawback of this operation is its asymmetric functionality, i.e. $\ominus$ has arguments of different types.

[4] *Preservation* says if expression e of type T evaluates to a value v then v also has type T. *Progress* says expression evaluation does not get "stuck", i.e. there are no expressions that cannot be evaluated.

Variant II: *Subtraction* is an operation with symmetric functionality. For modules $\mathtt{m}$ and $\mathtt{n}$ we define module $\mathtt{m} - \mathtt{n}$ as:

$$\mathtt{m} - \mathtt{n} =_{df} \mathtt{m} \ominus \mathtt{dom}(\mathtt{n})$$

This spells out to:

$$(\mathtt{m} - \mathtt{n})(\mathtt{v}) =_{df} \begin{cases} \mathtt{m}(\mathtt{v}) & \text{if } \mathtt{v} \in (\mathtt{dom}(\mathtt{m}) - \mathtt{dom}(\mathtt{n})) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Note that $\mathtt{m} - \mathtt{n}$ is *not* the set-theoretic difference of $\mathtt{m}$ and $\mathtt{n}$ considered as sets of argument-value pairs: let $\mathtt{f}_1, \mathtt{f}_2$ be different fragments and $\mathtt{u} \in \mathtt{V}$ be a VP. Set $\mathtt{m}_\mathtt{i} = \{(\mathtt{u}, \mathtt{f}_\mathtt{i})\}$, i.e.,

$$\mathtt{m}_\mathtt{i}(\mathtt{v}) =_{df} \begin{cases} \mathtt{f}_\mathtt{i} & \text{if } \mathtt{v} = \mathtt{u} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Then the set theoretic difference of $\mathtt{m}_1$ and $\mathtt{m}_2$ is $\mathtt{m}_1$. In contrast, $\mathtt{m}_1 - \mathtt{m}_2 = 0$ since $\mathtt{dom}(\mathtt{m}_1) = \mathtt{dom}(\mathtt{m}_2) = \{\mathtt{u}\}$.

Subtraction satisfies the following laws:

$$\begin{array}{rcl} \mathtt{dom}(\mathtt{m} - \mathtt{n}) & = & \mathtt{dom}(\mathtt{m}) - \mathtt{dom}(\mathtt{n}) \\ \emptyset - \mathtt{n} & = & \emptyset \\ (\mathtt{m} + \mathtt{n}) - \mathtt{p} & = & (\mathtt{m} - \mathtt{p}) + (\mathtt{n} - \mathtt{p}) \\ \mathtt{m} - (\mathtt{n} + \mathtt{p}) & = & (\mathtt{m} - \mathtt{n}) - \mathtt{p} \\ \mathtt{m} - \emptyset & = & \mathtt{m} \\ \mathtt{m} - \mathtt{m} & = & \emptyset \\ \mathtt{m} - \mathtt{n} & \subseteq & \mathtt{m} \\ \mathtt{m} \subseteq \mathtt{n} & \Rightarrow & \mathtt{m} - \mathtt{n} = \emptyset \end{array}$$

Note that the last law is only an implication, while the corresponding one for set $(-)$ is an equivalence. For the reverse direction we only have $\mathtt{m} - \mathtt{n} = \emptyset \Rightarrow \mathtt{dom}(\mathtt{m}) \subseteq \mathtt{dom}(\mathtt{n})$.

Overriding. Ideally, modules that are composed have disjoint domains. And by using subtraction or deletion, modules can be customized. Still, object-oriented programmers are used to the notion of *overriding* or *replacing* definitions, an operation that can be defined in terms of subtraction and deletion. Module $\mathtt{m}$ overrides $\mathtt{n}$, written $\mathtt{m}\ \mathtt{onto}\ \mathtt{n}$:

$$\mathtt{m}\ \mathtt{onto}\ \mathtt{n} = \mathtt{m} + (\mathtt{n} \ominus \mathtt{dom}(\mathtt{m})) = \mathtt{m} + (\mathtt{n} - \mathtt{m})$$

This replaces all assignments in $\mathtt{n}$ for which $\mathtt{m}$ provides a new value. It may destroy acyclicity. $\mathtt{onto}$ is associative and idempotent with neutral element $\emptyset$, but not commutative.

EXAMPLE 5.1. *Figure 11 shows two modules* $\mathtt{n}$ *and* $\mathtt{m}$ *with non-default fragments for* $\mathtt{vp}_1$. $\mathtt{m}\ \mathtt{onto}\ \mathtt{n}$ *replaces* $\mathtt{n}$*'s fragment at* $\mathtt{vp}_1$ *with* $\mathtt{m}$*'s fragment.* □

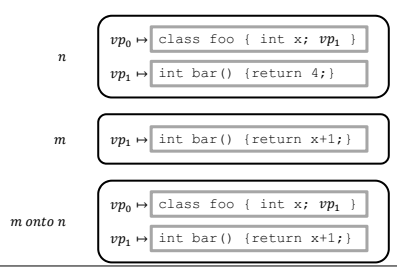

Figure 11. Onto Example.

5.2 The GenVoca Homomorphism

GenVoca is a model of SPLs where features are program transformations and feature composition is function composition. Features can add details to programs as well as override (replace) existing details. Let $m(\mathtt{x})$ denote the program transformation for feature $\mathtt{M}$ and let $\mathtt{m}$ be its SDA module. $m(\mathtt{x})$ is defined as:

$$m(\mathtt{x}) = \mathtt{m}\ \mathtt{onto}\ \mathtt{x}$$

If $\mathtt{M}$ is a base feature, m simplifies to:

$$m() = \mathtt{m}$$

GenVoca features were composed in a fixed order (see [4] for details). Further, every feature and feature interaction was encoded as a program transformation. Although FIA did not exist when GenVoca was created, an FIA explanation of GenVoca is simple: the cross-product of selected features was taken in a particular order and the resulting FIA expression was expanded in a fixed way to produce a sum of features and feature interactions.[5] Each of these terms was then mapped to a function that implemented that term. Again let FS be the set of features and FIS be the set of feature interactions. *GenVoca is the* γ *homomorphism that maps the sum of features and feature interactions to compositions of program transformations, where* $\cdot$ *is function composition:*

$$\begin{array}{lll} \gamma(\mathtt{A} + \mathtt{B}) & = \gamma(\mathtt{A}) \cdot \gamma(\mathtt{B}) & // \text{ for all } \mathtt{A}, \mathtt{B} \in (\mathtt{FS} \cup \mathtt{FIS}) \\ & = a \cdot b & \end{array}$$

Delta-Oriented Programming appears to be a variation of this homomorphism [20]. Delta modules can add new elements and delete existing elements. This requires additional information (a partial order) in which delta modules are composed, rather than a fixed order which γ homomorphism assumes. More on this in Section 6.

5.3 Solving Model Equations

As discussed in [6], it is useful to be able to solve module equations. Subtraction and its relatives enable us to do so. Suppose that $\mathtt{m}$ and $\mathtt{n}$ are modules such that $\mathtt{m} \leq \mathtt{n}$. Then the equation $\mathtt{m} + \mathtt{x} = \mathtt{n}$ has $\mathtt{x} = \mathtt{n} - \mathtt{m}$ as a solution. Moreover, this is the unique solution that is domain-disjoint from $\mathtt{m}$.[6]

EXAMPLE 5.2. *Consider the composition:*

$$\mathtt{a} = \mathtt{b} + \mathtt{c} + \mathtt{d} + \mathtt{e}$$

of modules $\mathtt{a} \ldots \mathtt{e}$ *where the domains of these modules are disjoint. Then the equation* $\mathtt{a} = \mathtt{b} + \mathtt{x} + \mathtt{d} + \mathtt{e}$ *has the unique solution* $\mathtt{x} = \mathtt{c}$ *domain-disjoint from* $\mathtt{b} + \mathtt{d} + \mathtt{e}$. □

Note that the condition $\mathtt{dom}(\mathtt{m}) \subseteq \mathtt{dom}(\mathtt{n})$ is necessary for $\mathtt{m} + \mathtt{x} = \mathtt{n}$ to be solvable, because we need to have $\mathtt{dom}(\mathtt{m} + \mathtt{x}) = \mathtt{dom}(\mathtt{m}) \cup \mathtt{dom}(\mathtt{x}) = \mathtt{dom}(\mathtt{n})$ which implies $\mathtt{dom}(\mathtt{m}) \subseteq \mathtt{dom}(\mathtt{n})$. The above assumption $\mathtt{m} \leq \mathtt{n}$ implies that necessary condition. In fact, solvability of $\mathtt{m} + \mathtt{x} = \mathtt{n}$ conversely implies $\mathtt{m} \leq \mathtt{n}$ since:

$$\mathtt{m} + \mathtt{n} = \mathtt{m} + (\mathtt{m} + \mathtt{x}) = (\mathtt{m} + \mathtt{m}) + \mathtt{x} = \mathtt{m} + \mathtt{x} = \mathtt{n}\,.$$

In short: $\mathtt{m} + \mathtt{x} = \mathtt{n}$ is solvable iff $\mathtt{m} \leq \mathtt{n}$.

Next, we have a brief look at equations involving overriding. Since $\mathtt{dom}(\mathtt{m}\ \mathtt{onto}\ \mathtt{n}) = \mathtt{dom}(\mathtt{n}\ \mathtt{onto}\ \mathtt{m}) = \mathtt{dom}(\mathtt{m}) \cup \mathtt{dom}(\mathtt{n})$, again $\mathtt{dom}(\mathtt{m}) \subseteq \mathtt{dom}(\mathtt{n})$ is necessary for $\mathtt{m}\ \mathtt{onto}\ \mathtt{x} = \mathtt{n}$ and $\mathtt{x}\ \mathtt{onto}\ \mathtt{m} = \mathtt{n}$ to be solvable. By the definition of $\mathtt{onto}$, solvability of $\mathtt{m}\ \mathtt{onto}\ \mathtt{x} = \mathtt{n}$ implies the stronger necessary condition $\mathtt{m} \leq \mathtt{n}$. In this case again $\mathtt{x} = \mathtt{n} - \mathtt{m}$ is the unique solution domain-disjoint from $\mathtt{m}$. A closer inspection shows that the same is the case for the equation $\mathtt{x}\ \mathtt{onto}\ \mathtt{m} = \mathtt{n}$. This means that it is sufficient to restrict interest to the solution of equations involving $+$.

[5] By "fixed way" we mean that the FIA $\times$ and $+$ operators are not commutative read as associating to the right.

[6] Another solution is $\mathtt{x} = \mathtt{n}$, since $\mathtt{m} \leq \mathtt{n}$ means $\mathtt{m} + \mathtt{n} = \mathtt{n}$. Such solutions are uninteresting.

6. Related Work

Our work is a direct outgrowth of the Coloring Algebra [6] and differs in several important ways:

- we separate features from their implementations (i.e. the distinction of FIA and SDA),
- we use of homomorphisms to map FIA expressions to (SDA) implementations,
- SDA presents a more general model of module composition via variation points, and
- we explored different and consistent sets of axioms to define feature algebras, of which [6] and our FIA are among the few reasonable possibilities.

The computation of SDA modules from coloring can be traced to [8] where elements of UML models could be tagged with feature predicates. Given a set of selected features, an element is removed from a model if its predicate is false. Modularizing elements that share the same predicate is the essence of coloring and SDA modularization.

Our work is a descendant of [16–18]. *Derivatives* were the first identified building blocks of feature modules. Unfortunately, the mathematics of derivatives was incomplete as composition of derivatives was not associative. This made it impossible to algebraically calculate the results of feature splitting (replacing `T` with `R` $\times$ `S` if `T` is split into features `R` and `S`) and feature merging (replacing `R` $\times$ `S` with `T`). CIDE [15] showed a simple way to visualize features and their interactions, resulting in the coloring algebra, which does support splitting and merging.

Other algebras for feature-based composition, such as [2, 19], focus on the internal structure of color modules, rather than feature interactions. [2] is the first algebra (to our knowledge) that dealt with feature replication. It uses *distance idempotence* (a form of idempotence where adjacency of identical features is not required). Feature composition is not commutative and feature modules (called feature structure trees) have no inverses.

The *Choice Calculus (CC)* [23] offers an interesting and alternative approach to our work. Among the goals of CC are to integrate classical and virtual modularity, but to do so in the context of a formal programming language. Large-scale fragments can be placed in modules of their own, while small-scale fragments (suitable for annotations) can be embedded into other modules. As in coloring and `ifdef` preprocessing, variation points are implicit. The key difference between our work and CC is that the issues of classical and virtual modularity are not limited to a fixed set of programming languages. The ability to map an FIA expression to different module implementations (modular units of makefiles, HTML pages, performance models) other than traditional programming languages is basic to feature-oriented development. CC may be one of many good implementation targets for mapping FIA expressions.

Delta Oriented Programming (DOP) is another interesting language-based approach within our field of work. Delta modules are qualified to be composed into a product when the corresponding `where` clause is satisfied. Such a clause is a propositional formula over features, namely the conjunction feature formulas that arise in coloring (and the coloring homomorphism of Section 4.1). Adding feature negation and disjunction seems more general. Disjunction allows a single module to be reused in different contexts (rather than requiring a module to be replicated for each context). Negation seems to offer a more general way for defining alternatives. Understanding this connection is yet another subject for future work. Delta modules also have `after` clauses, which specify a partial ordering in which to compose them. We suspect that the GenVoca homomorphism of Section 5.2 encodes this partial ordering implicitly.

7. Conclusions and Outlook

Feature-oriented design and development is based on the composition and manipulation of structures. We want its tools and concepts to be based on formal models and rock-solid foundations. In this paper, we have contributed important steps toward this goal.

FIA acts at the level of specifications to express features, feature interactions, and their compositions. In contrast, SDA is a general model of modules with VPs, and how such modules can be added and subtracted. FIA deals with the 'semantics' of features and SDA deals more with the 'syntax' of modules. (Stated differently, FIA deals with the 'problem space' and SDA deals with the 'solution space'.) Projections of FIA to SDA via homomorphisms define the relationships between these two universes, which in prior work were conflated and not well-understood.

We pointed out that existing tools implement some of our ideas (and, to be frank, inspired our work). We also indicated how existing tools can be (or need to be) generalized to implement all the possibilities we have explored.

We further showed that there are only a few basic permutations in which FIAs can be constructed. We have chosen the simplest, which is consistent with classical feature modeling. But the next step is to explore and clarify the role of multiple instances of features in formal algebras of feature-development. Our work in this paper shows that replicated features can lead to contradictions in algebraic models. Generalizing algebraic models to permit (or emulate) multiple feature instances is an important open problem in feature-based development.

Acknowledgements We gratefully acknowledge support for this work by NSF grants CCF 0724979 and OCI-1148125. Andreas Zelend provided many helpful comments, notably concerning the proofs in Section 3.3.

References

[1] S. Apel, C. Kästner, and C. Lengauer. Featurehouse: Language-independent, automated software composition. In *ICSE*, 2009.

[2] S. Apel, C. Lengauer, B. Möller, and C. Kästner. An algebraic foundation for automatic feature-based program synthesis. *Science of Computer Programming*, pages 1022–1047, 2010.

[3] K. Bak, K. Czarnecki, and A. Wasowski. Feature and meta-models in clafer: Mixed, specialized, and coupled. In *SLE*, 2010.

[4] D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM TOSEM*, 1992.

[5] D. Batory, J. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE TSE*, June 2004.

[6] D. Batory, P. Höfner, and J. Kim. Feature Interactions, Products, and Composition. In *GPCE*, 2011.

[7] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer Verlag, 2004.

[8] K. Czarnecki and M. Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In *GPCE*, 2005.

[9] B. Delaware, W. Cook, and D. Batory. Theorem proving for product lines. In *OOPSLA/SPLASH*, 2011.

[10] J. D. Hay and J. M. Atlee. Composing features and resolving interactions. In *SIGSOFT*, 2000.

[11] F. Heidenreich. Towards systematic ensuring well-formedness of software product lines. In *FOSD*, 2009.

[12] P. Höfner, R. Khedri, and B. Möller. Feature algebra. In *Formal Methods*, 2006.

[13] P. Höfner, B. Möller, and A. Zelend. Foundations of coloring algebra with consequences for feature-oriented programming. In W. Kahl and T. G. Griffin, editors, *Relational and Algebraic Methods in Computer*

Science - 13th International Conference, RAMiCS 2012, Cambridge, UK, September 17-20, 2012. Proceedings, volume 7560 of *Lecture Notes in Computer Science*, pages 33–49. Springer, 2012.

[14] S. Jarzabek. *Effective Software Maintenance and Evolution: Reuse-based Approach*. CRC Press Taylor and Francis, 2007.

[15] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In *ICSE*, 2008.

[16] C. H. P. Kim, C. Kästner, and D. Batory. On the modularity of feature interactions. In *GPCE*, 2008.

[17] J. Liu, D. Batory, and S. Nedunuri. Modeling interactions in feature oriented designs. In *ICFI*, 2005.

[18] J. Liu, D. Batory, and C. Lengauer. Feature Oriented Refactoring of Legacy Applications. In *ICSE*, 2006.

[19] R. Lopez-Herrejon, D. Batory, and C. Lengauer. A Disciplined Approach to Aspect Composition. In *PEPM*, 2006.

[20] I. Schaefer, L. Bettini, F. Damiani, and N. Tanzarella. Delta-oriented programming of software product lines. In *SPLC*, 2010.

[21] N. Siegmund, M. Rosenmüller, C. Kästner, P. G. Giarrusso, S. Apel, and S. S. Kolesnikov. Scalable prediction of non-functional properties in software product lines. In *SPLC*, 2011.

[22] N. Siegmund, S. S. Kolesnikov, C. Kästner, S. Apel, D. S. Batory, M. Rosenmüller, and G. Saake. Predicting performance via automated feature-interaction detection. In *ICSE*, 2012.

[23] E. Walkingshaw and M. Erwig. A calculus for modeling and implementing variation. In *GPCE*, 2012.