Feature Interactions, Products, and Composition

Don Batory
University of Texas at Austin
Austin, TX 78712 USA
batory@cs.utexas.edu

Peter Höfner
University of Augsburg,
Germany
NICTA, Australia
peter.hoefner@nicta.com.au

Jongwook Kim
University of Texas at Austin
Austin, TX 78712 USA
jongwook@cs.utexas.edu

ABSTRACT

The relationship between feature modules and feature interactions is not well-understood. To explain classic examples of feature interaction, we show that features are not only composed sequentially, but also by cross-product and interaction operations that heretofore were implicit in the literature. Using the *Colored IDE (CIDE)* tool as our starting point, we (a) present a formal model of these operations, (b) show how it connects and explains previously unrelated results in *Feature Oriented Software Development (FOSD)*, and (c) describe a tool, based on our formalism, that demonstrates how changes in composed documents can be back-propagated to their original feature module definitions, thereby improving FOSD tooling.

Categories and Subject Descriptors D.2.10 [Software Design]

General Terms design, theory

Keywords FOSD, CIDE, back-propagation, feature interactions, feature products

1. INTRODUCTION

Feature Oriented Software Development (FOSD) is the study of modularizing features (increments in program functionality), feature composition, and the use of features to synthesize programs of *software product lines (SPLs)* [2].

In FOSD, a *feature module* encapsulates changes that are made to a program in order to add a new capability or functionality. Such modules (often interpreted as transformations) are composed sequentially: if `f` and `g` are feature modules, their composition $\mathtt{f} \cdot \mathtt{g}$ represents the combined set of changes made by `f` and `g`. Not all compositions of features – called *expressions* – are meaningful: *feature models* define all legal expressions [8, 20]. Each expression, when evaluated, synthesizes a distinct program in an SPL. Each program in an SPL has an expression [4, 9, 34].

Features are the building blocks of programs. But what are the building blocks of features? In this paper, we present an algebra that shows modules called *colors* to be their building blocks. Just as programs are compositions of features, features are compositions of colors. Our research extends a long line of prior work [3, 6, 23, 24, 26, 27, 30, 34].

The novelty and significance of our work is recognizing additional operations on features that are implicit in the literature, but never before made explicit. Besides sequential ($\cdot$) composition, there is also cross-product ($\times$) and interaction (#) composition. In brief, when architects want both features `f` and `g`, they are asking for their cross-product, $\mathtt{f} \times \mathtt{g}$, which is governed by the following axiom:

$$\mathtt{f} \times \mathtt{g} = (\mathtt{f}\#\mathtt{g}) \cdot \mathtt{g} \cdot \mathtt{f} \qquad (1)$$

That is, architects want not only the composition of feature modules `f` and `g`, *but also a module* (`f#g`) *that modifies and/or integrates* `f` *and* `g` *so that they work correctly together*. Modules `f`, `g`, and `f#g` are colors.

We explore feature products and feature interactions in this paper, spelling out the implications of (1). We review classic examples of feature interactions and introduce a formal model (coloring algebra) that was inspired by Kästner's Color IDE [21] and that defines the sequential, cross-product, and interaction composition operations to be consistent with these examples. Our algebra unifies previously unrelated results in FOSD and reveals how changes in composed documents can be back-propagated to their original feature module definitions, thereby improving FOSD tooling. Lastly, we present a tool to create product lines of MS Word documents that supports this back-propagation idea.

2. MOTIVATING EXAMPLES

We start with examples that lead us to postulate (1). Although taken from different domains, readers will recognize their underlying similarity.

2.1 Fire-and-Flood Control

A classic example of feature interactions is fire-and-flood control [19]. Adding fire control to a building requires fire sensors to be placed on every ceiling. When a sensor detects a fire, sprinklers are activated. Adding flood control is similar: water sensors are placed on every floor. When standing water is detected, the water main is turned off.

Constructing a building with either flood control or fire control is straightforward. Problems arise when both features are present: suppose a fire is detected at time `i`. Fire control activates sprinklers at time $\mathtt{i} + 1$, standing water is

detected by flood control at time `i + 2`, the water main is turned off at time `i + 3`, and the building burns down. *The solution is to modify the fire and flood modules so that they work together correctly.* From an architect's perspective, we want the cross-product of `fire` and `flood`:

$$\texttt{fire} \times \texttt{flood} = (\texttt{fire\#flood}) \cdot \texttt{fire} \cdot \texttt{flood}$$

Namely `fire` and `flood` are composed, followed by module (`flood#fire`) that modifies the `floor` and `fire` modules to correctly coordinate their behavior, typically by giving one feature priority over another [18].

> **Note:** A common definition of *module* is a unit of code that may be linked with other modules but otherwise remains *unmodified* [1, 16]. In this paper, we assume (color) modules could be typical modules or more generally be code patches [33].

2.2 Call Waiting and Call Forwarding

Another classic example is the call waiting `CW` and call fowarding `CF` features in telephony [6, 18, 10]. `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 must decide whether the call should be forwarded or the user should be notified that another call has arrived. The resolution is defined by module `CW#CF`. Without a resolution, the phone system may behave erroneously.

Call waiting and call forwarding is similar to fire-and-flood control in that # defines a priority. In a phone system with both call waiting and call forwarding, we want the product `CW` × `CF` to include an appropriate resolution `CW#CF`.

> **Note:** The Feature Interaction community uses the term "feature interaction" to mean a change in behavior when features are composed [10]. Formal analyses are used to detect such interactions. "Resolution" is a term indicating the changes needed to get the desired behavior; these changes are color modules called *interaction* modules. Henceforth, the name of a module `A#B` indicates the *interaction* of features `A` and `B`, and the module contents is the *resolution* of their interaction.

2.3 CIDE

Colored IDE (CIDE) is an advance in FOSD tooling that reduces the granularity of feature modules [22]. A source document is painted in different colors, one color per feature. All source that is painted "red" belongs to the `Red` feature, all source that is painted "green" belongs to the `Green` feature, etc. Red that appears inside green indicates an interaction – how the `Red` feature changes the source of the `Green` feature. Symmetrically, green that appears inside red indicates how the `Green` feature changes the source of the `Red` feature. We assume that colors only nest and otherwise do not overlap.

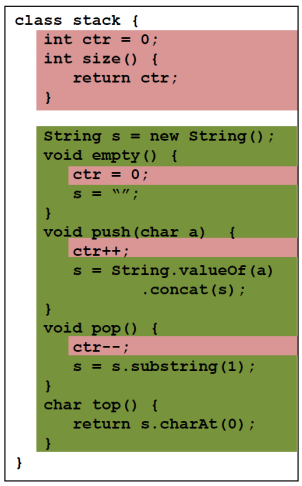

Figure 1: The Counted Stack (`Counter` × `Stack` × `Base`)

Consider a counted character stack [26], where characters are pushed and popped from a String and the number of characters on the stack are counted (Figure 1). There are three features: `Base`, `Stack`, and `Counter`. The `Base` feature is clear and represents an empty stack class. The `Stack` feature is green and contains the standard push, pop, empty, and top methods, along with a String that encodes the character stack. The `Counter` feature is red and contains an integer counter and size method. `Stack` and `Counter` interactions are red inside green, which reset, increment, and decrement the counter variable.

CIDE has preprocessor semantics, where the code of a feature `F` is effectively surrounded by `#ifdef F` − `#endif` statements. (CIDE differs from traditional preprocessors as it uses ASTs, rather than text, and is integrated with feature models [22]). Programs in CIDE can be "developed" incrementally by exposing features one at a time; this is how cross-products of features are simulated. Initially, the `Base` feature exposes only an empty stack class (Figure 2a). When the `Stack` feature is added, green and clear code is made visible (Figure 2b). And when the `Counter` feature is added, all code is exposed (Figure 1).

Alternatively, we could expose or compose features in a different order: after `Base`, we could expose `Counter` (Figure 2c), which shows only the `Counter` introductions. Exposing `Stack` reveals the remaining code (Figure 1). Each of these "progressions" corresponds to a particular cross-product of features, as indicated in the subtitles of Figures 1 and 2. We return to this example later.

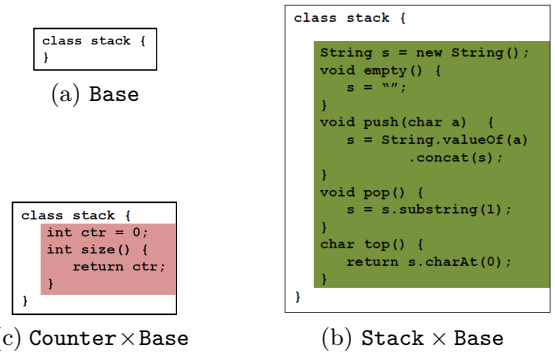

Figure 2: Stepwise Developments of Counted Stack

CIDE offers a visually simple way to recognize n-way (or n^{th}-order) interactions by the nesting of n colors. So an interaction module `f#g#h` would be the set of all fragments that are nested 3-deep using any permutation of colors/features `f`, `g`, and `h`. In practice, 2-way interactions are common. 3-way interactions arise occasionally. 4-way or higher-order interactions seem rare. In any case, a formal model must be able to express arbitrary-order interactions.

2.4 Interaction of Language Features

The Feature Interaction community focusses on finding semantic interactions of features [10]. We do not dispute the importance of semantic interactions, but we note that interaction modules also arise in semantic documents. Here is a recent example [13].

Programming languages evolve through the addition of

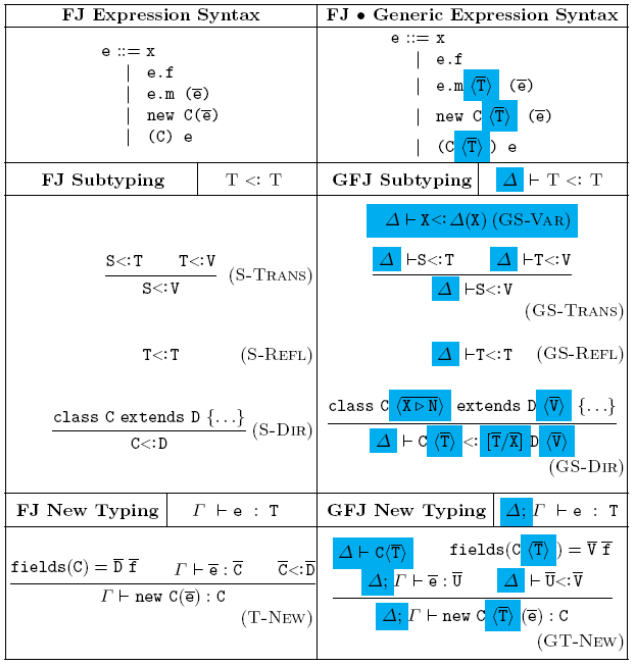

Figure 3: Selected FJ Definitions with GFJ Changes

features, which may include new control structures, abstractions, or typing constructs. Each feature changes the syntax and semantics of a language.

Consider adding `Generics` to the calculus of *Featherweight Java (FJ)* to produce the calculus of *Generic Featherweight Java (GFJ)*. The required changes are woven throughout the syntax and semantics of FJ. The left-hand column of Figure 3 presents a subset of the syntax of FJ, the rules which formalize the subtyping relation that establish the inheritance hierarchy, and the typing rule that ensures expressions for object creation are well-formed. The corresponding definitions for GFJ = `Generics` × `FJ` appear in the right-hand column where CIDE-shading indicates differences. As in CIDE, when the `Generics` feature is removed, the right-hand column simplifies to the left-hand column. These highlighted changes are the introductions and fragments of definitions that belong to the `Generics#FJ` color.

The same holds for proofs of type soundness, the guarantee that the desired run-time behavior of a language, typically preservation and progress, is enforced. That is, proofs of type soundness for FJ are altered when the `Generics` feature is added: new proof cases are added and existing lemmas may be altered. The changes to the FJ proofs are also contained in the `Generics#FJ` color.

2.5 Recap

The sequential, cross-product, and interaction composition of features are pervasive in FOSD. A formal model is needed to define their properties precisely. Doing so axiomatizes the concepts in CIDE and our motivating examples.

3. A COLORING ALGEBRA

We now develop an algebra for coloring where all colors are treated identically. Our model of FOSD is rooted in the way CIDE expresses features and their compositions. For exposition reasons, however, we motivate two types of colors: base and interaction.

A *base color* represents an individual feature whose module is a collection of one or more documents. When a base is added to a program, its documents are added; when the base is removed, its documents are removed. Base colors are denoted by individual letters (`R`, `S`, `T`). A dot-composition of base colors is the disjoint union of their documents.

A document can have any number of labeled *variation points (VPs)*, i.e. points at which a document fragment can be inserted. An *interaction color* is a #-expression (e.g. `R#S`, `S#T`, `R#S#T`) whose module consists of zero or more documents and document fragments that are to be inserted or *installed* at VPs. It is possible for some fragments to remain uninstalled after composition, as they may be installed later when another color module adds the required VPs. Think of a base color as an interaction color without document fragments.

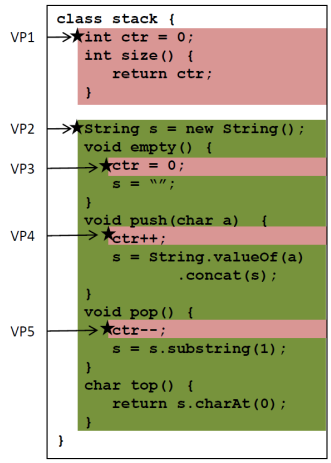

Figure 4: The Counted Stack With Variation Points

Recall the counted stack. Figure 4 shows its five variation points indicated by ⋆. Each VP is associated with precisely one fragment. `Base` is a single document (an empty stack class) with two variation points `VP1` and `VP2`. The `Counter#Base` module (red inside clear) contains the fragment that is installed at `VP1`. The `Stack#Base` module (green inside clear) contains the fragment that is installed at `VP2`. This fragment has three variation points `VP3`, `VP4`, and `VP5`. The `Counter#Stack#Base` module contains the three fragments that are installed at these points.

Figure 4 exhibits a key property: VPs and fragments are *always* in 1-to-1 correspondence [7]. It is not possible for multiple fragments to be installed at the same variation point. (VPs could be placed next to each other to give the appearance that multiple fragments are installed at the same VP).

The next sections give our axiomatization of CIDE. We start with dot(·)-composition. We use `1` to denote the *empty color* or *empty module*. `1` contains no documents or fragments.

3.1 Dot Composition

Let `R`, `S`, and `T` be colors. The dot-composition `R·S` is the operation that (a) forms the disjoint union of their documents and (b) installs their document fragments, if possible. `R·S` represents a composite color. Three axioms of · are:

$$\textit{Identity}: \quad \mathtt{R}\cdot\mathtt{1} = \mathtt{R} \tag{2}$$

$$\textit{Commutativity}: \quad \mathtt{R}\cdot\mathtt{S} = \mathtt{S}\cdot\mathtt{R} \tag{3}$$

$$\textit{Associativity}: \quad \mathtt{R}\cdot(\mathtt{S}\cdot\mathtt{T}) = (\mathtt{R}\cdot\mathtt{S})\cdot\mathtt{T} \tag{4}$$

Axioms (2) and (4) are standard for FOSD. Unlike tradi-

tional models (such as AHEAD [9], FeatureHouse [4], and DOP [11]), feature composition in CIDE is commutative (3) – the order in which a set of colors are dot-composed (a.k.a. "turned on") does not matter. The reason is that CIDE has no overriding (the ability to delete or replace non-empty code fragments). Moreover, as we will see, commutativity follows from another axiom we define later.

3.2 Interaction Composition

R#S is the color that defines how R and S interact—it is the set of changes that are needed to make R and S work together correctly. Three axioms of # are:

$$\textit{No Interaction}: \quad \mathtt{R\#1} = \mathtt{1} \tag{5}$$

$$\textit{Commutativity}: \quad \mathtt{R\#S} = \mathtt{S\#R} \tag{6}$$

$$\textit{Associativity}: \quad \mathtt{R\#(S\#T)} = \mathtt{(R\#S)\#T} \tag{7}$$

(5) states the elementary fact that that 1 cannot be changed and that it does not change other colors. As with sequential (dot) composition, the order in which features are #-composed in CIDE does not matter. An interaction module R#S#T represents the set of changes for all permutations of colors R, S, and T. This justifies (6) and (7).

For the remainder of the paper we assume that interaction composition (#) binds stronger than dot-composition (·). These operations are related by a distributivity law [26]:

$$\textit{Distributivity I}: \quad \mathtt{R\#(S \cdot T)} = \mathtt{(R\#S) \cdot (R\#T)} \tag{8}$$

That is, the interaction of R with S·T is the dot-composition of interactions R#S and R#T. By commutativity of # (6), we can immediately derive the second distributivity law:

$$\textit{Distributivity II}: \quad \mathtt{(S \cdot T)\#R} = \mathtt{(S\#R) \cdot (T\#R)} \tag{9}$$

Intuitively, these distributivity laws state that the interaction between a color R and a composed color S · T can be described by the composition of the interaction of R with S and T separately. From a practical point of view, defining interaction on base colors is sufficient.

3.3 Product Composition

R × S is the color that represents the product of R and S: as discussed earlier, it is the dot-composition of R and S with their interaction resolution R#S:

$$\textit{Product}: \quad \mathtt{R \times S} = \mathtt{R\#S \cdot R \cdot S} \tag{10}$$

The following theorems of × can be proven given the previous axioms (see Appendix):

$$\textit{Identity}: \quad \mathtt{R \times 1} = \mathtt{R} \tag{11}$$

$$\textit{Commutativity}: \quad \mathtt{R \times S} = \mathtt{S \times R} \tag{12}$$

$$\textit{Associativity}: \quad \mathtt{(R \times S) \times T} = \mathtt{R \times (S \times T)} \tag{13}$$

The meaning of these theorems should be self-evident.

3.4 Involution Axioms of CIDE

So far, our axioms are standard and the algebraic structures are well-known. Now we consider two basic behaviors of CIDE. In doing so, we are confronted by fundamental questions that lurk in a dark corner of classical feature modeling:

- What are the semantics of replicated features? What is R · R and R × R and R#R?
- Can a feature interact with itself? What is R#R?
- Do features have inverses?

As we will see, their answers depend on each other.

Consider the first question on replicated features. In classical feature models, a feature is either selected or it is not; feature replication *never* occurs. So color expressions like R#R, R · R, and R × R never arise. But CIDE raises the question of feature replication in an unusual way by nesting colors, forcing us to address replication.

Consider R#R. Red inside red is indistinguishable from red. If R denotes red, this reads as:

$$\mathtt{(R\#R) \cdot R} = \mathtt{R}$$

Let B be blue. A more complex example is red-inside-blue-inside-red, which is indistinguishable from blue-inside-red:

$$\mathtt{(R\#B\#R) \cdot (B\#R) \cdot R} = \mathtt{(B\#R) \cdot R}$$

In creating our algebra, we faced the following design decision: should we admit an infinite number of non-empty terms to which we can ascribe no useful meaning or distinction (R#R, R#R#R, R#S#R, R#R#R#R, ...)? Or do we eliminate them for a simpler explanation? We chose the latter, and assert that a feature does not interact with itself:

$$\textit{\#-Involution}: \quad \mathtt{R\#R} = \mathtt{1} \tag{14}$$

Given (14), the equalities of the above examples follow.

Now consider the meaning of R · R and R × R. We can proceed in two ways; both are equivalent. The first recognizes a surprising property of CIDE. Its colors are invertable as CIDE fragments never override or delete other fragments. In other words, either the documents of a color are present or they are not. And a color's fragments are either installed or they are not. So colors have a binary behavior: If a color R is to be dot composed with some program T, R checks to see if it is already installed. If it was, R removes its documents and fragments. Otherwise, R installs its documents and fragments as usual. Involution captures this two state existence elegantly:

$$\textit{Dot-Involution}: \quad \mathtt{R \cdot R} = \mathtt{1} \tag{15}$$

That is, *all* colors (base and interaction) are the inverses of themselves. This axiom does not hold for other FOSD models [4, 9, 11], but it does hold for CIDE. It immediately follows that:

$$\textit{Involution}: \quad \mathtt{R \times R} = \mathtt{1} \tag{16}$$

That is, features toggle from on to off, and a feature is a ×-involution of itself.

> **Note:** Note that the meaning of sequential composition (dot) is now stronger than we informally described in the Introduction. Before, R·S meant the combined set of changes of R and S. But back then, R and S were implicitly distinct modules with disjoint contents. Our strengthening of (dot) now covers a corner case previously and implicitly omitted.

Here is the second way: Features in CIDE (or any FOSD model) are either selected or they are not. A feature toggles between on and off, which is expressed by (16). Given this, (15) is immediately derived.

Although (15) is surprising, it comes with useful benefit that has long been absent and needed in FOSD. Namely, the ability to solve dot-equations for unknowns. We explore the utility of this capability in Section 4.3.

There are also other, deeper reasons for (15). We know of only three possible ways to deal with replicated features: (a) disallow them, (b) assume involution, or (c) assume idempotence. We ruled out (a) above and presented the consequences of (b). The remaining alternative is idempotence [5]: there is only one copy of a feature (i.e. $\mathtt{R} \cdot \mathtt{R} = \mathtt{R}$). As mentioned earlier, the algebra should provide inverses (either to solve dot-equations for unknowns or for "turning off" features). The problem here is profound: the existence of inverse and idempotence implies that the universe consists of only one color: $\mathtt{R} = \mathtt{1} \cdot \mathtt{R} = (\mathtt{R}^{-1} \cdot \mathtt{R}) \cdot \mathtt{R} = \mathtt{R}^{-1} \cdot (\mathtt{R} \cdot \mathtt{R}) = \mathtt{R}^{-1} \cdot \mathtt{R} = \mathtt{1}$. This is yet another reason for asserting (15).

3.5 Axiom Consistency and Irredundancy

Our axioms are consistent, i.e. they do not contradict themselves and there exist models – color expression instances and deductions – that satisfy these axioms. We have sketched some models in this section and used Mace4 [29] to generate models with a finite number of elements.

We have also used Prover9 [29], an automated theorem prover, to prove that axioms (3) and (5) can be entailed from the other axioms, leaving (2), (4), (6)–(10) irredundant. This can be shown by removing one axiom from the set and adding its formula as goal. An example is given in the Appendix.

From our experience, defining a consistent set of axioms was not easy; a slight change in one may expose truly unexpected contradictions that only tools like Prover9/Mace4 could find. With this confidence, we can compute the interactions of interactions – the interaction of R#S and T#U is R#S#T#U – and the product of interactions, etc., should they ever be needed.

3.6 Open Problems

There are four axioms/theorems of CIDE that are not shared by other FOSD approaches, namely dot-commutativity (3), ×-commutativity (12), dot-involution (15), and ×-involution (16). We conjecture that there is a subset of coloring axioms that can be used to model feature interations in non-CIDE approaches, but this task is beyond the scope of this paper. Further, exploring the role of inverses in non-CIDE approaches, perhaps combining results from this paper, would also be useful. We know, for example, that transformations (features) that override (delete, replace), rather than extend, are assumed not to have inverses. But retaining the history of a derivation, as does the `unmixin` tool of [9], does permit inverses to be computed. Again, we leave this exploration for future work.

4. OBSERVATIONS AND IMPLICATIONS

4.1 Altering Module Composition Order

Kästner and Apel observed that the order in which feature modules are composed can be changed, *but this requires an alteration of the contents of their modules* [3, 24]. It was conjectured that for all composable feature modules F and G, there exists modules $\mathtt{F}'$ and $\mathtt{G}'$ such that:

$$\mathtt{F} \cdot \mathtt{G} = \mathtt{G}' \cdot \mathtt{F}' \tag{17}$$

where the informal meanings of F and $\mathtt{F}'$ (G and $\mathtt{G}'$) are essentially the same. That is both F and $\mathtt{F}'$ add the capabilities of feature F, but do so in different ways so that (17) holds. How F maps to $\mathtt{F}'$ (and G to $\mathtt{G}'$) is not fully understood.

The essence of the solution was first suggested in [3, 24] and is captured elegantly by (1). Suppose f and g are features to compose and let F, $\mathtt{F}'$, G, and $\mathtt{G}'$ be their modules. We want the cross-product $\mathtt{g} \times \mathtt{f}$, where we start with module F and then dot-compose module G:

$$\begin{aligned} \mathtt{g} \times \mathtt{f} &= \mathtt{G} \cdot \mathtt{F} \\ (\mathtt{g}\#\mathtt{f} \cdot \mathtt{g}) \cdot \mathtt{f} &= \mathtt{G} \cdot \mathtt{F} \end{aligned}$$

Module $\mathtt{F} = \mathtt{f}$ consists of a single color and module $\mathtt{G} = \mathtt{g}\#\mathtt{f} \cdot \mathtt{g}$ is composite. Should we reverse the product order of f and g, we have definitions for modules $\mathtt{F}'$ and $\mathtt{G}'$:

$$\begin{aligned} \mathtt{f} \times \mathtt{g} &= \mathtt{F}' \cdot \mathtt{G}' \\ (\mathtt{f}\#\mathtt{g} \cdot \mathtt{f}) \cdot \mathtt{g} &= \mathtt{F}' \cdot \mathtt{G}' \end{aligned}$$

Module $\mathtt{G}' = \mathtt{g}$ and module $\mathtt{F}' = \mathtt{f}\#\mathtt{g} \cdot \mathtt{f}$. It is the color $\mathtt{f}\#\mathtt{g}$ that "migrates" from G to $\mathtt{F}'$ that explains why feature module contents must change when their composition order changes. (Note: if $\mathtt{f}\#\mathtt{g} = \mathtt{1}$, modules F and G are commutative). Permuting the order in which feature modules are composed is a matter of migrating interaction modules.

> **Note:** Readers can see in Figures 1-2 an example. For the composition `Counter` × `Stack` × `Base`, the Stack module is `Stack#Base` and the Counter is `Counter#Stack#Base` · `Counter#Base`. For the composition `Stack` × `Counter` × `Base`, the Counter module is `Counter#Base` and the Stack module is `Counter#Stack#Base` · `Stack#Base`.

4.2 Cross-Product Expression Evaluation

To select a target program in an SPL, architects select the set of features that they want, such as $\{\mathtt{F}, \mathtt{G}, \mathtt{H}\}$. A tool based on the coloring algebra forms the cross-product of these features, yielding a dot-product of colors using the axioms above:

$$\mathtt{F} \times \mathtt{G} \times \mathtt{H} = \mathtt{F}\#\mathtt{G}\#\mathtt{H} \cdot \mathtt{F}\#\mathtt{G} \cdot \mathtt{F}\#\mathtt{H} \cdot \mathtt{G}\#\mathtt{H} \cdot \mathtt{F} \cdot \mathtt{G} \cdot \mathtt{H} \tag{18}$$

Color modules are then retrieved from a repository and composed, thereby synthesizing the target program.

Note that a cross-product of n features produces a dot-expression of $(2^{\mathtt{n}} - \mathtt{1})$ colors. Experience shows that a *vast* majority of colors equal 1; rough indications are that $\mathtt{O}(\mathtt{n}) - \mathtt{O}(\mathtt{n}^2)$ colors are non-identities [21, 26]. Owing to color naming conventions, expression expansion need not be exponential in complexity, but linear in the size of a document. Here's how: Only non-identity colors are stored in a repository. Instead of expanding $\mathtt{F} \times \mathtt{G} \times \mathtt{H}$, a tool searches the repository for color modules whose names reference only F, G, and H, and composes them. (Module T#H would not be retrieved as $\mathtt{T} \notin \{\mathtt{F}, \mathtt{G}, \mathtt{H}\}$, but F#H would be). This is a linear operation in the total number of colors in the entire document. Stated differently, this operation is as fast as searching the entire document once.

The reason why a coloring algebra produces an exponential number of terms is that it must account for *all* possible interactions of features, although in any practical setting, a vanishingly small percentage of colors are non-empty.

4.3 New Tool Technologies

The following situation can arise: The source of a program in an SPL is produced and given to a client. The client modifies the program (possibly to fix bugs, improve performance, etc.). Now the changes to this program must be back-propagated to the original feature modules to make the changes permanent. If the client has access to the source

of the entire SPL, this can be done. But suppose the client does not, and herein lies a difficulty.

Our algebra suggests a solution. A client requests program $\mathtt{P}$, which corresponds to the following composite color $\mathtt{P} = \mathtt{T}_1 \cdot \ldots \cdot \mathtt{T}_n$. The client manually modifies $\mathtt{P}$ to produce program $\mathtt{Q} = \mathtt{T}_0 \cdot \mathtt{T}'_1 \cdot \ldots \cdot \mathtt{T}_n$. (The client still sees VPs although their fragments may have been removed. He could add new VPs and fragments, and could change or delete any fragment or VP present in the program). When the client submits the updated program $\mathtt{Q}$, a tool would know the original value of $\mathtt{P}$ and could solve for the changes $\Delta\mathtt{P}$ that were made to it:

$$\begin{array}{rcll} \Delta\mathtt{P} \cdot \mathtt{P} & = & \mathtt{Q} & \text{// given} \\ \Delta\mathtt{P} \cdot \mathtt{P} \cdot \mathtt{P} & = & \mathtt{Q} \cdot \mathtt{P} & \text{// } \cdot\mathtt{P} \text{ to both sides} \\ \Delta\mathtt{P} & = & \mathtt{Q} \cdot \mathtt{P} & \text{// (15)} \\ \Delta\mathtt{P} & = & \mathtt{T}_0 \cdot \mathtt{T}'_1 \cdot \ldots \cdot \mathtt{T}_n \cdot \mathtt{T}_1 \cdot \ldots \cdot \mathtt{T}_n & \text{// substitution} \\ \Delta\mathtt{P} & = & \mathtt{T}_0 \cdot \mathtt{T}'_1 \cdot \mathtt{T}_1 & \text{// (3) and (15)} \end{array}$$

Solving for $\Delta\mathtt{P}$ is what diff-based tools do: they ignore unchanged colors and reveal colors that were changed, added or deleted. $\Delta\mathtt{P}$ shows that color $\mathtt{T}_0$ was added and $\mathtt{T}_1$ was changed, where $\mathtt{T}'_1 \cdot \mathtt{T}_1$ is difference between the updated and original $\mathtt{T}_1$ color.

Differencing leads to the possibility of a shredding tool, that takes a program with variation points as input and shreds it into colors. Only those colors that are new or that have changed must be updated in the color repository. In the next section, we describe a tool based on these ideas.

5. THE PAAN TOOL

Office Open XML is an open standard of XML schemas adopted by Microsoft Office for its default file format. It specifies a compressed, XML-based encoding of Microsoft Office 2007 and 2010 documents, where different XML formats are used for Word, Visio, Excel, and InfoPath [17]. This allows non-Microsoft tools to extract and manipulate Office documents. By changing a `.docx` file to `.zip` and unzipping, the contents of a Word document (consisting of multiple XML files and directories) becomes visible.

We created a tool, called *Paan* — Korean for 'version', that enables us to explore a new implementation of CIDE concepts, but using the coloring algebra as its inspiration [25]. Specifically Paan works with MS Word documents, where it relies on the Custom XML Markup facility of MS Word to define nested regions of color and variation points. A markup *tag* is used to assign a feature name to a region (a.k.a. fragment) of a Word document. A fragment is identified by enclosing start and end tags. In Figure 5a, a pair of tags named blue surrounds a "Hello World" fragment; its XML representation is shown in Figure 5b.

```
<?xml version="1.0"?>
<w:document xmlns:wne="…">
   <w:body>
      <w:customXml w:uri="mySchema" w:element="blue">
         <w:p>
            <w:r>
               <w:t>Hello World</w:t>
            </w:r>
         </w:p>
      </w:customXml>
   </w:body>
</w:document>
```

(b)

(a) blue ⟨ Hello World ⟩ blue

Figure 5: MS Word Custom Markup Tags and its XML

In CIDE, colors are nested like preprocessor `#ifdef-#endif` declarations. An inner color appears only if all of its enclosing colors (features) have been selected. Paan works the same way. In Figure 6a, red tags wrap vowels. Being surrounded by a blue tag, vowels appear only when both the blue and red features are selected.

The removal of unwanted features from a colored document is called *projection*. For implementation reasons, when a feature is projected (removed), a variation point is marked by an additional tag named `_reserved_`. Figure 6b shows the removal of the blue feature from Figure 6a; Figure 6c shows the removal of the red feature.

Paan differs from CIDE in several ways. One, obviously, is the colorability of Word documents. More importantly, Paan was designed for the following scenario. Imagine documents where some features encompass proprietary or sensitive data that can only be exposed to certain communities. The full Word document, in such cases, could not be distributed. Instead, *only document projections are distributed*. Further, each community could *edit* their projected documents. Paan can automatically back-propagate these changes into the Paan repository, similar to the diffing concepts described in Section 4.3. The novelty of Paan is (a) it demonstrates how this scenario can be addressed, (b) it strictly follows the laws (axioms, theorems, ...) of the coloring algebra (Section 5.3), and (c) works with MS Word documents.

5.1 Back-Propagation of Changes

Let $\mathtt{W}$ be a colored Word document and let $\mathtt{W}_\mathtt{p}$ be a projection of $\mathtt{W}$, where $\mathtt{p}$ is the set of features that have been retained. A user can now modify $\mathtt{W}_\mathtt{p}$ at will, adding new VPs (that are instantiated with their text fragments), modifying visible fragments, and deleting existing VPs (including VPs whose text has been projected).

To back-propagate the changes in $\mathtt{W}_\mathtt{p}$ to $\mathtt{W}$, Paan does the following. First, it maintains a copy of $\mathtt{W}$ in its repository that existed prior to projection. It then traverses $\mathtt{W}_\mathtt{p}$ to locate VPs whose fragments were projected (removed). For each such VP `i`, it finds fragment `i` in $\mathtt{W}$ and restores that fragment in $\mathtt{W}_\mathtt{p}$. At the end, fragments of $\mathtt{W}$ that were removed to produce $\mathtt{W}_\mathtt{p}$ have been restored. Paan then discards the original copy $\mathtt{W}$ and replaces it with $\mathtt{W}_\mathtt{p}$. The projection-back-propagate cycle continues.

> **Note:** The restoration of projected VPs can be accomplished in linear time: a single pass through $\mathtt{W}$ to find all (VP,fragment) pairs and a single pass through $\mathtt{W}_\mathtt{p}$ to restore projected VPs.

Paan's back-propagation implements Section 4.3: A Paan repository can consist of multiple Word documents and directories. If a Word document has not been changed (which Paan knows by examining a Word document's revision number and comparing it to the revision number in the repos-

(a) blue ⟨ H red ⟨ e ⟩ ll red ⟨ o ⟩ W red ⟨ o ⟩ rld ⟩ blue

(b) blue ⟨ _reserved_ ⟩ _reserved_ blue

(c) blue ⟨ H red ⟨ _reserved_ ⟩ ll red ⟨ _reserved_ ⟩ W red ⟨ _reserved_ ⟩ rld ⟩ blue

Figure 6: Nesting and Projection of Tags

```
(a)  void m() { counter++; }
(b)  void m() { print("before" + counter);
                Super.m();
                print("after" + counter);
              }
(c)  void around() : execution(void C.m())
              { print("before" + counter);
                proceed();
                print("after" + counter);
              }
(d)  void m() { print("before" + counter);
                counter++;
                print("after" + counter);
              }
(e)  void m(){ RED print("before" + counter);
               BASE counter++; )
               print("after" + counter); )
              }
```

Figure 7: Wrappers

itory), Paan does not update the repository's copy. When updating individual Word documents, Paan simply assumes that all fragments in W_p have been modified, and proceeds to update its repository copy on this conservative (and functionally equivalent) basis.

5.2 Wrappers

Paan also differs from CIDE in that it natively supports wrappers. A *wrapper* is a fragment that surrounds another fragment. Wrappers occur in FOSD languages as method extensions and in AOP as around advice of execution pointcuts of individual methods. Figure 7a shows a base method `m()` of class `C`. Figure 7b shows an extension of `m()` in AHEAD syntax that wraps `m()`. Figure 7c shows the identical extension of `m()` in AspectJ syntax. Figure 7d is the result of this extension. Figure 7e is how the extension and base is colored in Paan. Wrapper tags (`BASE` and `RED`) are in uppercase to distinguish them from non-wrappers (`base`) which are in lower-case.

An interesting property of wrappers is that they have exactly the opposite semantics of color nesting in CIDE. Let B be a base fragment and W be a wrapper of B. If B and W are also the names of their respective features, B belongs to the `B` module and wrapper W belongs to the interaction module `W#B`. Unlike CIDE, where an interaction module `T#B` that modifies `B` is fully inside `B`, wrapping reverses the roles where the wrapped module `B` is fully inside `W#B`.

> **Note:** Wrappers and non-wrappers are both colors, and their distinction is irrelevant to the coloring algebra. Stated differently, wrappers and non-wrappers are just different ways of implementing colors.

To understand how wrappers are projected, we need to generalize the CIDE projection algorithm. Associated with every color module is a propositional formula whose terms are non-negated feature variables. As a first approximation, module `F` has the formula (`F`) and the interaction $F_1\#...\#F_n$ represents the conjunction $(F_1 \wedge ... \wedge F_n)$.

Projection of wrappers is accomplished in the following way: Paan traverses the Word document `W` in its repository in its entirety. Let `p` denote the set of features that were selected (meaning that their fragments are to remain after projection). The traversal of `W` encounters a sequence of

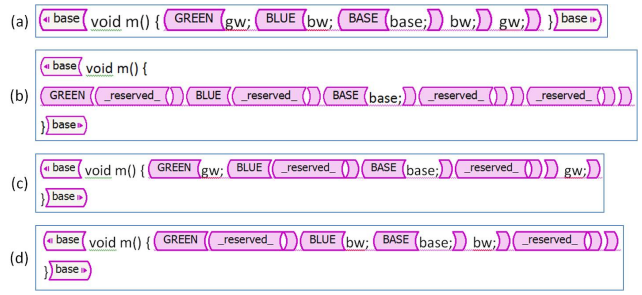

Figure 8: Projecting Wrappers

fragments. Let `T` be a fragment and `T(x)` be its propositional formula. If `T(p)` is true, `T` is present in W_p. Otherwise, `T` is not included, and the traversal of `T` to the next fragment inside `T` continues. This is different than a document without wrappers, as once a fragment is eliminated, there is no need to search inside the fragment further.

To illustrate, Figure 8a shows a `BASE` fragment wrapped by a `BLUE` and `GREEN` fragment. Figure 8b shows only the `BASE` feature. Figure 8c shows only the `BASE` and `GREEN` features, and Figure 8d only the `BASE` and `BLUE` features.

Paan enables arbitrary higher-order wrappers by allowing users to define the predicate (and hence the interaction module) of a wrapper, so that all interactions permitted by the coloring algebra can be expressed.

In summary, wrappers slightly increase the complexity of the projection algorithm. Interestingly, the back-propagation algorithm for transferring edits of color modules back to the repository is uneffected.

5.3 Paan's Support of the Coloring Algebra

Paan supports the coloring algebra several visible ways. First is the non-involution axioms of # and dot that define the nesting or wrapping hierarchical coloring structure that is imposed on Word documents. Second is its enforcement of #-involution, which requires special support. Figure 9a shows an edited document containing interaction `R#B#R` (red-inside-blue-inside-red). When this change is back-propagated, Paan moves the contents of `R#B#R` into `B#R` (Figure 9b), therefore enforcing (14). Doing so shows that Paan makes use of commutativity and involution to simplify interaction colors. The same applies to wrappers.

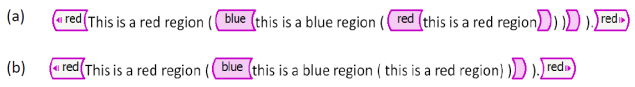

Figure 9: Paan's Support for Axiom (14)

The third way is allowing users to select features in arbitrary orders, upholding the cross-product and associativity axioms, and cross-product involution by asserting that a feature is either selected or deselected. The dot- involution axiom is essential to back-propagation.

Having said the above, it is possible that a tool like Paan could have been developed without a coloring algebra. The basic ideas of back-propagation are conceptually straightforward. But there are certain design decisions – such as involution – whose implications are not obvious, nor are they obviously consistent. Our formalization provides a confi-

dence in this approach that an implementation could never provide or guarantee.

6. EXPERIENCE AND LESSONS LEARNED

Paan was evaluated on several SPLs [25]. One experiment converted an AHEAD SPL with nine Java classes into nine MS Word files, one per class. This SPL had 25 features; each of the MS Word files were colored accordingly. Three separate projections (configurations) were tested; the resulting Word files were converted into text files and then into Java files for compilation and subsequent execution (to verify that the Paan projections were correct). Another experiment converted HTML documentation for another AHEAD SPL, which included graphics, into a single MS Word file, from which different projections (documentation for specific SPL members) were produced. Back-propagation was tested by manually editing the above projected documents. Although more sophisticated and thorough testing was possible, manual comparisons were sufficient for our goals.

Word documents must conform to an Office Open XML schema. A straight-forward implementation of projection (leaving customized XML nodes indicating a projected VP) can invalidate schema conformity. So too can the restoration of a fragment at a VP during back-propagation invalidate schema conformity, if not done carefully.

An example is that `<paragraph>` structures in Office Open XML cannot be nested. Figure 10a shows "`Hello world`" enclosed in a `blue` region. Figure 10b shows a projection of `blue`. A string "`abc`" is appended before the projected VP (Figure 10c). We expect that back-propagation restores the "`Hello world`" fragment at the VP to produce Figure 10d. Unfortunately, the resulting Word file invalidates schema conformity. The reason is that the "`abc`" text is a `<paragraph>` that includes the VP. The fragment at the VP is also a `<paragraph>`, leading to nested `<paragraph>`s, which is an illegal structure.

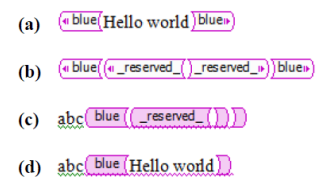

Figure 10: Back-Propagation Error

Our solution was to recognize the errors that resulted in projection and/or back-propagation, and to apply local transformations that repaired the structure. From this we learned our most important lesson: *Coloring is a functionality that should be an integral part of a tool's design: it should not be an after-thought, or be implemented as an after-thought, as we have done. The semantics of marking and coloring should be aligned from the beginning, thereby simplifying tool development.*

7. RELATED WORK

Coloring can be traced to [12] where elements of UML models could be tagged with feature predicates. Given a set of selected features, an element would be removed from a model if its predicate is false. Modularizing elements that share the same predicate is the essence of coloring.

The coloring algebra is a descendant of [24, 26, 27]. *Derivatives* were the first identified building blocks of feature modules.[1] Unfortunately, the mathematics of derivatives was incomplete as compositions of derivatives were not associative. This made it impossible to algebraically calculate the results of feature splitting (replacing `T` with $\mathtt{R} \times \mathtt{S}$ if `T` is split into features `R` and `S`) and feature merging (replacing $\mathtt{R} \times \mathtt{S}$ with `T`). CIDE 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 [5, 28], focus on the internal structure of color modules, rather than feature interactions. [5] is the first algebra (to our knowledge) that dealt with feature replication. Their solution uses *distance idempotence* (a form of idempotence where adjacency of identical features is not required). Feature composition was not commutative and feature modules (called feature structure trees) have no inverses.

AHEAD [9] and FeatureHouse [4] are compositional approaches to FOSD where only dot-composition is supported. There is no notion of interaction modules, cross-product, and #-product operations. The distinctions of cross-products and #-products could be layered onto AHEAD and FeatureHouse as all expressions with cross-products can be reduced to dot-products of modules (where each color module can be encoded as an AHEAD or FeatureHouse module). Back-propagation of edits in composed modules is supported by the AHEAD tool called '`unmixin`'. However, there was no formalization of back-propagation in AHEAD.

Delta-oriented Programming (DOP) [11, 34] is another compositional approach to FOSD which has a module structure similar to that of colors. DOP does permit features (or colors) that can override (replace, delete) existing modules, so a restriction of our algebra (as discussed in Section 3.6) may be needed to describe it. DOP does not support cross-products or interaction operations (although it could, just like AHEAD and FeatureHouse). The main advance of our work is the axiomatization of coloring.

Aspect-oriented Programming (AOP) is related to colors in the following way. Base colors contain only introductions. AOP pointcuts designate join-points, which are implicit variation points. AOP advice designates code fragments to be inserted at join-points (or rather join-point shadows). The primary distinction between AOP and FOSD (and our work) is that aspects do not have simple composition semantics (e.g. they cannot always be expressed as a composition of simpler aspects [28]). Consequently the mathematics behind aspects is complicated.

Others have created similar tools to Paan although none support back-propagation. Rabiser et al. describe a tool that adopts DocBook for variability modeling [32]. Although the tool does not support wrappers, it uses generative techniques that are more powerful than coloring to produce customized documents for SPL members. Pure::Systems uses tagging (like Paan) to create a product line of Word documents [31]. Other than available web videos, little more is known about this tool.

[1] The essential idea is this: $\mathtt{F}|\mathtt{G}$ denotes the changes made by `F` to `G`. Symmetrically, $\mathtt{G}|\mathtt{F}$ denotes the changes made by `G` to `F`. Our interaction module $\mathtt{F}\#\mathtt{G}$ equals the dot composition of these two derivatives ($\mathtt{F}\#\mathtt{G} = \mathtt{F}|\mathtt{G} \cdot \mathtt{G}|\mathtt{F}$). And in general, an `n`-th order interaction $\mathtt{A}_1\# \ldots \#\mathtt{A}_n$ corresponds to a dot composition of all `n`! permutations of these features as derivatives ($\mathtt{A}_1|\ldots|\mathtt{A}_n \cdot \mathtt{A}_n|\ldots|\mathtt{A}_1 \cdot \ldots$). In short, derivatives are the building blocks of colors.

Finally, there is a connection between back-propagation and maintaining the consistency of pairs of models [14, 15], one of which is derived from the other and the derived one is updated. Our work is a special case of this more general problem.

8. CONCLUSIONS

The relationship between features and feature interactions has long been a subject of interest. FOSD brings a twist in its focus on feature modularity and the composition of feature modules to build programs of product lines. Interactions occur when the behavior of a feature changes in the presence of another feature. Interaction modules contain the changes (a.k.a. resolution) to existing modules so that their features work correctly together.

We reviewed classical examples of feature interactions and presented a formalism (the coloring algebra) that (a) is based on Kästner's CIDE, (b) faithfully captures these examples and (c) provides a general framework in which to understand feature modules and feature interactions. The essence of our approach is recognizing the product ($\times$) and interaction (#) operations besides sequential (dot) composition. Our algebra spells out the properties of all these operations and their interrelationships. By doing so, we have shown how prior and unconnected results in FOSD can be unified. Further, the algebra suggested a more general way to support coloring, so that documents of a product line that were synthesized could be modified, and that these modifications could subsequently be back-propagated to their original feature-based product line representations. We presented a tool, Paan, that (a) implemented our algebra, (b) demonstrated the feasibility of back-propagation as we described it, and (c) worked with MS Word documents, all of which we believe are novel.

Our work advances the state of the art in understanding the integration of feature composition and feature interactions, and improved tooling for FOSD.

Acknowledgments. We thank J. Atlee and K. Czarnecki for their comments on a presentation of this work. We are grateful to the GPCE referees for their helpful comments. Batory and Kim are supported by the NSF's Science of Design Project CCF 0724979. Höfner was supported by the DFG grant #MO 690/7.

9. REFERENCES

[1] A. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools Second Edition*. pearson Education, 2006.

[2] S. Apel and C. Kästner. An overview of feature-oriented software development. *Journal of Object Technology*, July-August 2009.

[3] S. Apel, C. Kästner, and D. Batory. Aspectual feature modules. *ACM TSE*, 2008.

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

[5] S. Apel, C. Lengauer, B. Möller, and C. Kästner. An algebraic foundation for automatic feature-based program synthesis. *Sci. of Comp. Programming*, 2010.

[6] S. Apel, W. Scholz, C. Lengauer, and C. Kästner. Detecting dependences and interactions in feature-oriented design. In *ISSRE*, 2010.

[7] P. Bassett. Frame-based software engineering. *IEEE Software*, 4(4), 1987.

[8] D. Batory. Feature Models, Grammars, and Propositional Formulas. In *SPLC*, Sept. 2005.

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

[10] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec. Feature interaction: A critical review and considered forecast. In *Computer Networks*, 2002.

[11] D. Clarke, M. Helvensteijn, and I. Schaefer. Abstract delta modeling. In *GPCE*, 2010.

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

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

[14] Z. Diskin. Algebraic models for bidirectional model synchronization. In *MoDELS*, 2008.

[15] J. N. Foster, M. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. In *POPL*, 2005.

[16] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, 2002.

[17] E. International. Office open xml file formats, 2nd edition. `http://www.ecma-international.org/publications/standards/Ecma-376.htm`, 2008.

[18] M. Jackson and P. Zave. Distributed feature composition: A virtual architecture for telecommunications services. *IEEE TSE*, Oct 1998.

[19] K. Kang. Private Correspondence, Oct. 2003.

[20] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-oriented domain analysis (foda) feasibility study. CMU/SEI-90-TR-021, 1990.

[21] C. Kästner. *Virtual Separation of Concerns: Toward Preprocessors 2.0*. PhD thesis, University of Magdeburg, 2010.

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

[23] C. Kästner and et al. On the impact of the optional feature problem: Analysis and case studies. In *SPLC*, 2009.

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

[25] J. Kim. Paan: A Tool for Back-Propagating Changes to Projected Documents. M.Sc. Thesis, The University of Texas at Austin, 2011.

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

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

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

[29] W. McCune. Prover9 and mace4. `http://www.cs.unm.edu/~mccune/prover9/`, 2010.

[30] C. Prehofer. Feature Oriented Programming: A Fresh Look at Objects. In *ECOOP*, 1997.

[31] Automatic generation of word document variants.

http://www.pure-systems.com/flash/pv-wordintegration/flash.html, 2010.
[32] R. Rabiser and et al. A Flexible Approach for Generating Product-Specific Documents in Product Lines. In *SPLC*, 2010.
[33] D. Roundy. Darcs: Distributed version management in haskell. In *Workshop on Haskell*, 2005.
[34] I. Schaefer, L. Bettini, F. Damiani, and N. Tanzarella. Delta-oriented programming of software product lines. In *SPLC*, 2010.

APPENDIX

A. DEFERRED PROOFS AND PROPERTIES

This appendix gives more details about properties of our algebra. Specifically, we list proofs we have skipped.

LEMMA A.1. *The identity tile* $\mathtt{1}$ *is a left unit:*

$$\mathtt{1}\cdot\mathtt{R} = \mathtt{R}$$

PROOF. The claim follows immediately from (right) identity (2), involution (15), associativity (4) and involution again:

$$\mathtt{R} = \mathtt{R}\cdot\mathtt{1} = \mathtt{R}\cdot(\mathtt{R}\cdot\mathtt{R}) = (\mathtt{R}\cdot\mathtt{R})\cdot\mathtt{R} = \mathtt{1}\cdot\mathtt{R}$$

□

LEMMA A.2. *Dot-composition is commutative:*

$$\mathtt{R}\cdot\mathtt{S} = \mathtt{S}\cdot\mathtt{R}$$

PROOF. By involution (15), we get:

$$(\mathtt{S}\cdot\mathtt{R})\cdot(\mathtt{S}\cdot\mathtt{R}) = \mathtt{1}$$

From this we can show the claim:

$$\begin{aligned}\mathtt{R}\cdot\mathtt{S} &\overset{(2)}{=} (\mathtt{R}\cdot\mathtt{S})\cdot\mathtt{1} \overset{(15)}{=} (\mathtt{R}\cdot\mathtt{S})\cdot(\mathtt{S}\cdot\mathtt{R})\cdot(\mathtt{S}\cdot\mathtt{R})\\ &\overset{(4)}{=} (\mathtt{R}\cdot(\mathtt{S}\cdot\mathtt{S})\cdot\mathtt{R})\cdot(\mathtt{S}\cdot\mathtt{R}) \overset{(15)}{=} (\mathtt{R}\cdot\mathtt{1}\cdot\mathtt{R})\cdot(\mathtt{S}\cdot\mathtt{R})\\ &\overset{(2)}{=} (\mathtt{R}\cdot\mathtt{R})\cdot(\mathtt{S}\cdot\mathtt{R}) \overset{(15)}{=} \mathtt{1}\cdot(\mathtt{S}\cdot\mathtt{R}) \overset{(A.1)}{=} \mathtt{S}\cdot\mathtt{R}\end{aligned}$$

□

LEMMA A.3. *The identity tile* $\mathtt{1}$ *does not interact with any tile, i.e.,* $\mathtt{R}\#\mathtt{1} = \mathtt{1}$.

PROOF. The claim follows from involution (15), distributivity (8) and involution again:

$$\mathtt{R}\#\mathtt{1} = \mathtt{R}\#(\mathtt{S}\cdot\mathtt{S}) = \mathtt{R}\#\mathtt{S}\cdot\mathtt{R}\#\mathtt{S} = \mathtt{1}$$

□

LEMMA A.4. *Product composition satisfies the following laws:*

(a) $\mathtt{R}\times\mathtt{1} = \mathtt{R}$
(b) $\mathtt{R}\times\mathtt{S} = \mathtt{S}\times\mathtt{R}$
(c) $(\mathtt{R}\times\mathtt{S})\times\mathtt{T} = \mathtt{R}\times(\mathtt{S}\times\mathtt{T})$
(d) $\mathtt{R}\times\mathtt{R} = \mathtt{1}$

PROOF.

(a) Neutrality of $\mathtt{1}$ follows from the definition of product (10), Lemma A.3, and identity (2) and Lemma A.1:

$$\mathtt{R}\times\mathtt{1} = \mathtt{R}\#\mathtt{1}\cdot\mathtt{R}\cdot\mathtt{1} = \mathtt{1}\cdot\mathtt{R}\cdot\mathtt{1} = \mathtt{R}$$

The remaining claims (Parts (b)–(d)) follow from the corresponding properties of dot- and interaction-composition. we only give references to the corresponding axioms and theorems.

(b) $\mathtt{R}\times\mathtt{S} \overset{(10)}{=} \mathtt{R}\#\mathtt{S}\cdot\mathtt{R}\cdot\mathtt{S} \overset{(3)}{=} \mathtt{S}\#\mathtt{R}\cdot\mathtt{S}\cdot\mathtt{R} \overset{(6)}{=} \mathtt{S}\times\mathtt{R}$

(c)
$$\begin{aligned}(\mathtt{R}\times\mathtt{S})\times\mathtt{T} &\overset{(10)}{=} (\mathtt{R}\#\mathtt{S}\cdot\mathtt{R}\cdot\mathtt{S})\times\mathtt{T}\\ &\overset{(10)}{=} (\mathtt{R}\#\mathtt{S}\cdot\mathtt{R}\cdot\mathtt{S})\#\mathtt{T}\cdot(\mathtt{R}\#\mathtt{S}\cdot\mathtt{R}\cdot\mathtt{S})\cdot\mathtt{T}\\ &\overset{(9),(4)}{=} \mathtt{R}\#\mathtt{S}\#\mathtt{T}\cdot\mathtt{R}\#\mathtt{T}\cdot\mathtt{S}\#\mathtt{T}\cdot\mathtt{R}\#\mathtt{S}\cdot\mathtt{R}\cdot\mathtt{S}\cdot\mathtt{T}\\ &\overset{(3)}{=} \mathtt{R}\#\mathtt{S}\#\mathtt{T}\cdot\mathtt{R}\#\mathtt{S}\cdot\mathtt{R}\#\mathtt{T}\cdot\mathtt{R}\cdot\mathtt{S}\#\mathtt{T}\cdot\mathtt{S}\cdot\mathtt{T}\\ &\overset{(9),(4)}{=} \mathtt{R}\#(\mathtt{S}\#\mathtt{T}\cdot\mathtt{S}\cdot\mathtt{T})\cdot\mathtt{R}\cdot(\mathtt{S}\#\mathtt{T}\cdot\mathtt{S}\cdot\mathtt{T})\\ &\overset{(10)}{=} \mathtt{R}\times(\mathtt{S}\#\mathtt{T}\cdot\mathtt{S}\cdot\mathtt{T})\\ &\overset{(10)}{=} \mathtt{R}\times(\mathtt{S}\times\mathtt{T})\end{aligned}$$

(d) $\mathtt{R}\times\mathtt{R} \overset{(10)}{=} \mathtt{R}\#\mathtt{R}\cdot\mathtt{R}\cdot\mathtt{R} \overset{(14)}{=} \mathtt{1}\cdot\mathtt{R}\cdot\mathtt{R} \overset{(15)}{=} \mathtt{1}\cdot\mathtt{1} \overset{(2)}{=} \mathtt{1}$

□

B. PROVER9 AND MACE4

Below we list an input template for the paramodulation-based theorem prover Prover9 [29]. It encodes the axioms of the presented algebra in an intuitive way, i.e., it accepts operators in infix, prefix and postfix notation; hence it is easy to use. Moreover, a quantification of the variables involved is often not necessary.

The same input file is accepted by the model generation tool Mace4—which complements Prover9. It can be used to detect non-theorems. All theorems of this paper can be proved fully automatically, Prover9 needs less than a second to prove each of them.

```
% LANGUAGE SPECIFICATION
 op(500, infix, ";").    % dot-composition (·)
 op(490, infix, "+").    % interaction-composition (#)
 op(500, infix, "X").    % product-composition (×)

% AXIOMS
 formulas(sos).
   % dot-composition
     x;1=x.
     x;(y;z) = (x;y);z.
     x;x =1.
   % interaction-composition
     x+y = y+x.
     x+(y+z) = (x+y)+z.
     x+x = 1.
     x+(y;z) = (x+y);(x+z).
   % product-composition
     x X y=(x+y);(x;y).
 end_of_list.

 % CONJECTURE
 formulas(goals).
   % lemma to be proved, e.g.,
   x X y = y X x.
 end_of_list.
```

C. IRREDUNDANCY EXAMPLE

We illustrate irredundancy by an example. Assume that we want to show that identity (2) does not follow from the other axioms. For that we feed Mace4 with axioms (4), (15), (6), (7), (14), (8), (10) and set (2) as goal. The tool immediately returns a model that satisfies these axioms, but violates (2) (where $\mathtt{R}$ below denotes an arbitrary tile):

$\#$	R	1
R	1	1
1	1	1

$\cdot$	R	1
R	1	1
1	1	1

$\times$	R	1
R	1	1
1	1	1

This model shows that $\mathtt{R}\cdot\mathtt{1} = \mathtt{1}$ and hence identity (2) does not hold and cannot be inferred from the other axioms.