

# Feature Interactions and Software Derivatives

Jia Liu, University of Texas at Austin, U.S.A., [jliu@cs.utexas.edu](mailto:jliu@cs.utexas.edu)

## Abstract

*Feature Oriented Programming (FOP)* merges the studies of feature modularity, generative programming, and compositional programming. We advance FOP by proposing the concept of software derivatives, which represent feature interactions. We apply the theory of software derivatives to refactoring legacy Java applications into FOP designs.

## 1 INTRODUCTION

Compositional programming and automated software engineering are important to the future of software development. Our research is *Feature Oriented Programming (FOP)*, which merges the studies of feature modularity, generative programming and compositional programming. FOP is a general theory of generative programming that arose from software product-lines. Programs — or product-line members — are differentiated by the features that they implement, where a *feature* is a unit of functionality that can be shared by many members. It raises the study of features as a fundamental form of software modularity, and shows how feature modules lead to systematic, general, and automatic approaches to software synthesis and evolution. An FOP model of a domain is an *algebra*, where each operator implements a feature. The *design of a program* is an expression, which is a composition of operators (features).

Feature interactions are a key part of feature-oriented designs. A *feature interaction* occurs when one or more features modify or influence another feature. There are many ways in which features can interact (e.g., [3]): we focus on a common form of interactions that are static and structural: *how a feature influences (or changes) the source code of another*. We propose to improve existing work on FOP and develop an algebraic theory of structural feature interactions. Because FOP represents program designs as expressions, the influence of a feature on another feature can be captured by the concept of a *derivative* which is governed by algebraic laws. We begin by explaining the core ideas of FOP, and then derivatives.

## 2 FOP AND AHEAD

*AHEAD (Algebraic Hierarchical Equations for Application Design)* is a realization of FOP based on step-wise refinement, domain algebras, and encapsulation.

A fundamental premise of AHEAD is that programs are constants and refinements are functions that add features to programs. Consider the following constants that represent base programs with different features:

```
f          // program with feature f
g          // program with feature g
```

A *refinement* is a function that takes a program as input and produces a refined or feature-augmented program as output:

```
i • x      // adds feature i to program x
j • x      // adds feature j to program x
```

where  $\bullet$  is function composition. The *design of an application* is a named composition of operators called an *equation*:

```
prog1 = i • f      // program with features i and f
prog2 = j • g      // program with features j and g
prog3 = i • j • g  // program with features i,j,g
```

Thus, the features of an application can be determined by inspecting its equation. An *AHEAD model* or *domain model* is an algebra whose operators are these constants and functions. The set of programs that can be synthesized by composing these operators is the model's *product-line*.

Code synthesis in FOP is straightforward: method and class extensions follow common notions of inheritance. Figure 1a shows a class  $\kappa$  that has three members: methods  $A()$ ,  $B()$ , and variable  $c$ . Figure 1b shows an extension of  $\kappa$  written in an extended-Java language where class extensions are prefaced by the special keyword “*refines*”. We also introduce a new keyword “*super*” to specify references from a refining method to a target method. This particular example encapsulates extensions to methods  $A()$  and  $B()$  and adds a new variable  $d$ . The composition of this base class and extension is Figure 1c: composite methods  $A()$  and  $B()$  are present, plus the remaining members of the base and extension. Although we have illustrated the effects of composition using substitution, there are many other techniques that can realize these ideas, such as mixins [6] and program transformations.

(a)	<pre>class K {   void A(){ x;y; }   void B(){ m;n; }   int C; }</pre>	(b)	<pre>refines class K {   void A(){ Super.A();w; }   void B(){ q;Super.B(); }   String D; }</pre>	(c)	<pre>class K {   void A(){ x;y;w; }   void B(){ q;m;n; }   int C;   String D; }</pre>
-----	---	-----	--	-----	---

Figure 1: Class Definition and Extension

## 3 SOFTWARE DERIVATIVES

### Feature Interactions and Optionality

In his 1997 paper, Prehofer presented an FOP model of a stack product line. We present a modified version of his example. The product line consists of three features: *stack*, *counter*, and *undo*. *stack* implements basic stack operations such as push and pop; *counter* adds a local counter to keep track the size of the stack; *undo* provides an undo function that restores the state of the stack as it was before the last modification. We show the implementations of these features in Figure 2a-c.

This is a design commonly seen in FOP. There is a kernel feature (`stack`) that introduces the underlying class structure of the program and defines the most basic operations. Each new feature extends the base program by adding a coherent set of new functionalities. Note that every new feature is built upon existing features, so that it can make proper refinements to integrate the new functionalities into the program. For example, feature `undo` is written with the full knowledge of `stack` and `counter`, and for every stack operation it inserts the backup of the stack body and the counter. To generate programs from this product-line, features are composed in order, e.g. `counter*stack` yields a stack with counter, and `undo*counter*stack` produces a full-featured stack.

However, a problem exists in this design. Suppose we want a stack with undo operation but without a counter.

An intuitive way is to compose `stack` with `undo` by the composition `undo*stack`. A closer examination, however, would reveal that this composition does not produce the stack we want. The `backup()` method defined in `undo` backs up not only the stack body but also the counter. Now that we do not have `counter` in our composition, it would reference a non-existent data member; in other words, `undo` interacts with a non-existent feature! This reflects a general problem in feature oriented designs: *because we encapsulate into a feature its interactions with other features, the feature breaks when one of its interacting features is not present in a system*. This is an undesirable effect that undermines feature reusability, as feature optionality is made more difficult to achieve. It is especially harmful in software product-line designs, since product-line members often only use a partial set of all features.

```
class stackOfChar {
    String s = new String();

    void push( char a ) {
        s = a + s;
    }
    void pop() {
        s = s.substring(1);
    }
}
```

(a) `stack`

```
refines class
stackOfChar {
    int i = 0;

    int size() { return i;
    }
    void push() {
        Super.push();
        i++;
    }
    void pop() {
        Super.pop();
        i--;
    }
}
```

(b) `counter`

```
refines class
stackOfChar {
    String s_bak;
    int i_bak;

    void backup() {
        s_bak = s;
        i_bak = i;
    }
    void undo() {
        s = s_bak;
        i = i_bak;
    }
    void push() {
        backup();
        Super.push();
    }
    void pop() {
        backup();
        Super.pop();
    }
}
```

(c) `undo`

Figure 2: Feature Modules in Stack Product-Line

As the cause of the problem is that different feature interactions are encapsulated in the same feature module, we can improve the feature design by separating these interactions into different modules as illustrated in Figure 3. We restructure

<pre>refines class stackOfChar {   void backup() {}   void undo() {}   void push() {     backup();     super.push();   }   void pop() {     backup();     super.pop();   } }</pre>	<pre>refines class stackOfChar {   String s_bak;    void backup() {     super.backup();     s_bak = s;   }   void undo() {     super.undo();     s = s_bak;   } }</pre>	<pre>refines class stackOfChar {   int i_bak;    void backup() {     super.backup();     i = i_bak;   }   void undo() {     super.backup();     i_bak = i;   } }</pre>
(a) <code>undo</code>	(b) <code>undo<sub>stack</sub></code>	(c) <code>undo<sub>counter</sub></code>

Figure 3: Separating Feature Interactions

`undo` into three parts: `undo` is the base feature; `undostack` encapsulates its interactions with `stack`; and `undocounter` encapsulates its interactions with `counter`. With this separation, we can compose the stack program with any combination of features we desire. To build a stack with undo functions, we use the composition `undostack•undo•stack`. For a full-featured stack, the composition is `undocounter•undostack•undo•counter•stack`. The general composition rule is that if a feature is present, the corresponding interaction module(s) must also be present. For example, as far as feature `undo` is concerned, whenever feature `counter` is present, `undocounter` must be added to a composition.

The value of this idea is clear: interactions and base features separate concerns. That is, *the semantics of interaction modules and base feature modules are fundamentally different*. However, the idea has its limitations. First, feature interactions are generally not limited between two features; it is possible to have multi-feature interactions. Second, there is no algebra or architectural model to express these ideas or to generate the correct compositions of base features and interaction modules from higher-level specifications. We show how to remove these limitations in the next section by presenting a series of ideas that generalize this solution.

### Software Derivatives: Encapsulating Feature Interactions

Base features (e.g. `undo`) partition the set of methods and variables of an application or product-line. That is, we generalize Prehofer's example so that a base feature encapsulates *any* number of methods and variables belonging to any number of classes, not just a single class. Further, we require that *no two base features define the same method or variable*. Each method and variable is either private or public. If private, a member represents an implementation detail that is not exposed to other features (and hence can be ignored in our analyses). If public, a member is visible to other features and is subject to modification in feature compositions.

Interaction module  $x_y$  expresses the concept of a *derivative*: how feature  $x$  changes with respect to feature  $y$ . Henceforth, we write  $x_y$  as  $\partial x/\partial y$ . A derivative or feature interaction is a module that encapsulates any number of methods and variables. Unlike base features, a derivative can *only* extend public methods of a base feature: it cannot introduce new public methods or variables. (It can introduce private members, but these are invisible to other features). Thus, a derivative  $\partial x/\partial y$  encapsulates extensions to one or more public methods of  $x$  made by  $y$ .

Recognizing derivatives are operators, we now have a general way to express interactions among multiple features, which can be seen as interactions of interactions. For example, the interaction of  $x$  with  $\partial z/\partial y$  is a *second order derivative*:

$$(\partial/\partial x) (\partial z/\partial y) = \partial^2 z/\partial x \partial y \tag{1}$$

Such derivatives have a simple interpretation:  $\partial^2 z/\partial x \partial y$  is a module that encapsulates the changes made to feature  $z$  by the *combined* features  $x$  and  $y$  (i.e.,  $x \bullet y$ ). Such a module encapsulates extensions of public  $z$  methods, such as:

```
void methodZ() {code references members introduced by X, Y, Z}
```

where `methodZ` is a method introduced by  $z$ , and its body references members introduced by features  $x$ ,  $y$ , and/or  $z$ . *Nth*-order derivatives have a similar interpretation:  $\partial^n a / (\partial b_n \dots \partial b_1)$  defines a module that extends methods introduced by  $a$  and references members in  $a, b_1 \dots b_n$ . A derivative is *empty* if it equals to the identity function  $\text{id}$ .

## Mappings between Abstract and Concrete Feature Models

In Section we implicitly used two different feature models: an abstract model and a concrete model. The concrete model was explicit — it was the set of all base features and all feature interaction modules. For the stack example, a concrete model  $c$  contains five non-empty operators — three base features and two derivatives:

```
C = { stack, counter, undo,  $\partial \text{undo}/\partial \text{stack}$ ,  $\partial \text{undo}/\partial \text{counter}$  }
```

where operators of the concrete model are composed by  $\bullet$ .

We also used an implicit abstract model — a set of abstract features where feature interactions are implicit. An abstract feature  $\underline{x}$  has the exactly the same methods of its concrete base feature  $x$ , but the implementation of these methods is to be defined. For the stack example, the abstract model  $a$  has three features:

```
A = { stack, counter, undo }
```

We use  $*$  to compose abstract features. (We will see shortly that  $*$  is different from  $\bullet$ ). The product-line of model  $a$  allows at least the following compositions:

```
stack // stack  
counter*stack // stack with counter  
undo*stack // undoable stack  
undo*counter*stack // undoable stack with counter
```

Implicit in the model is the mapping of an abstract expression — a  $\ast$  composition of abstract features — to a concrete expression — a  $\bullet$  composition of concrete features. Let  $\underline{x}$  and  $\underline{y}$  be abstract features and  $\underline{x}\ast\underline{y}$  their composition. Let  $x$ ,  $y$ , and  $\partial y/\partial x$  denote the corresponding concrete base features and their interaction. The relationship to map an abstract feature expression to a concrete feature expression is:

$$\underline{x} \ast \underline{y} = \partial x/\partial y \bullet x \bullet y \tag{2}$$

That is, composing abstract features  $\underline{x}$  and  $\underline{y}$  is realized by composing their concrete base features  $x$  and  $y$  plus the changes  $y$  makes to  $x$ . An example of this identity was seen earlier: we saw that a “undoable stack”  $\underline{\text{undo}}\ast\underline{\text{stack}}$  maps to  $\partial \text{undo}/\partial \text{stack} \bullet \text{undo} \bullet \text{stack}$  — that is, the composition of the base  $\text{undo}$  and  $\text{stack}$  features with the interaction of  $\text{undo}$  with  $\text{stack}$ . (2) elevates this relationship to a general identity.

Here is why (2) is useful: it specifies how a composition of abstract features can be automatically translated to a composition of concrete features, which would otherwise be *much* larger and *much* more difficult to write:

$$\underline{\text{undo}}\ast\underline{\text{counter}}\ast\underline{\text{stack}} = \partial^2 \text{undo}/\partial \text{counter}\partial \text{stack} \bullet \partial \text{undo}/\partial \text{counter} \bullet \partial \text{undo}/\partial \text{stack} \bullet \text{undo} \\ \bullet \partial \text{counter}/\partial \text{stack} \bullet \text{counter} \bullet \text{stack}$$

Some derivatives in this expression may be empty (such as  $\partial^2 \text{undo}/\partial \text{counter}\partial \text{stack}$ ); we only need to compose the non-empty ones. We can build a database or code repository of non-empty base features and derivatives. Tools will translate abstract expressions into their corresponding concrete expressions, then look up and retrieve the terms present in the composition from the database. The retrieved modules are composed in the order dictated by the concrete expression, and the target application is synthesized.

## 4 APPLICATION

A common software maintenance request is to add and remove features from an existing application. If non-FOP designs are used, this task can be exorbitantly expensive. A challenge problem is to refactor legacy applications which do not have FOP designs into an equivalent form where they do have FOP designs, and thus can take advantage of FOP’s feature extensibility. Our theory outlines a general solution to the feature refactoring problem.

Our software derivative model supports the refactoring of legacy applications into an FOP design. Based on our theory on FOP, we envision a tool that helps a user to partition a legacy application into abstract features by identifying data members and methods that each feature introduces. Once this is done, the code for abstract features can be automatically refactored into base features and derivatives. *With an FOP design of a legacy application extracted, the evolution of this application — by adding, removing and/or replacing features — should be simplified.* This work has broad applicability to domains where features are composed statically to synthesize programs. We believe that most software-intensive domains fall into this category.

## References

- [1] D. Batory, “The Road to Utopia: A Future for Generative Programming”. Keynote presentation at *Dagstuhl for Domain-Specific Program Generation*, March 23-28, 2003.
- [2] J. Liu and D. Batory, “Automatic Remodularization and Optimized Synthesis of Product-Families”. *To appear in GPCE’04*.
- [3] E. J. Cameron, N. D. Griffeth, Y.-J. Lin, M. E. Nilson, W. K. Schure, and H. Velthuijsen, “A Feature Interaction Benchmark for IN and Beyond”. *Feature Interactions in Telecommunications Systems*, IOS Press, 1994.
- [4] C. Prehofer, “Feature-Oriented Programming: A Fresh Look at Objects”. *ECOOP*, 1997.
- [5] D. Batory, J.N. Sarvela, and A. Rauschmayer, “Scaling Step-Wise Refinement”. *IEEE Transactions on Software Engineering*, June 2004.
- [6] M. Van Hilst and D. Notkin, “Using Role Components to Implement Collaboration-Based Designs”. *OOPSLA*, 1996.
- [7] I. D. Baxter, “Design Maintenance Systems”. *CACM*, Vol. 55, No. 4 (1992).