to appear, International Conference on Feature Interactions in Telecommunications and Software Systems (ICFI 2005), Leicester, United Kingdom

Modeling Interactions in Feature Oriented Software Designs[1]

Jia LIU, Don BATORY and Srinivas NEDUNURI
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712, U.S.A.
{jliu,batory,nedunuri}@cs.utexas.edu

Abstract. Feature Oriented Programming (FOP) is a general theory of software development where programs are assembled by composing feature modules. A feature **X** interacts structurally with another feature **Y** by changing **Y**'s source code. We advance FOP by proposing an algebraic theory of structural feature interactions that models feature interactions as derivatives. We use our theory to show how a legacy Java application can be refactored into a feature-based design.

1 Introduction

A *feature* of a program is a functionality or implementation characteristic that is important to a client. The feature set of a program is the set of services that the program provides [13]. A *software product-line* is a family of related programs [13]. Programs — or product-line members — are differentiated by the sets of features they implement. *Feature Oriented Programming (FOP)* is a general theory of software development, and in particular, software product-lines. A product-line model is an *algebra*, where each operation implements a feature. The design of a program is a composition of these operations (features). Composition evaluation synthesizes the target program; composition optimization optimizes the design of that program [5]. Nontrivial systems have been synthesized using FOP, including database systems [3], extensible Java compilers [5], avionics [1], and program verification tools [32].

FOP has direct relevance to main-stream software-development. FOP's focus on features as units of modularity advances contemporary trends in software design. Program features are often expressed as use-cases in UML models and features frequently correspond to product-requirements [15][16]; feature-level modularity helps software product-family optimizations [20]. Further, incremental program evolution and maintenance often involves the operations of adding, removing, or updating features from existing programs [13]. FOP raises these informal activities to 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.

Feature interactions are a key issue in feature-oriented designs. A *feature interaction* occurs when one or more features modify or influence other features [14]. There are many

1. This research is sponsored in part by NSF's Science of Design Project #CCF-0438786.

ways in which features can interact: we focus on a particular form of interactions that are static and structural: *how a feature influences (or changes) the source code of another.* In this paper, we sketch an algebraic theory of structural feature interactions for FOP that has the following characteristics:

- Structural feature interactions are expressed by the concept of a *derivative*: how one feature (or group of features) alters or influences another feature;
- Feature derivatives are general operators in FOP and obey algebraic laws. Derivatives expose a rich algebraic structure of application designs that is suitable for automatic manipulation and automatic synthesis;
- Derivatives enable *refactoring (Java) legacy applications into FOP designs*, thereby giving legacy applications an important measure of *evolvability* by allowing features to be more easily added or removed.

Our work has applicability to domains where features are composed statically to synthesize programs. We believe that many software-intensive domains fall into this category. We begin with a brief introduction on the core ideas of FOP.

2 Feature Oriented Software Design and AHEAD

A feature is an increment in program functionality [37] and often corresponds to a user-oriented functional requirement of an application. The earliest research in product-lines used features both to define requirements incrementally and to specify programs. Not only was declarative program specifications possible using features, it was cheaper to build individual features and to assemble them into desired applications [8][15][16].

The *Graph Product-Line (GPL)* [21] is a simple example (Figure 1a). GPL is a set of programs that implement one or more algorithms on graphs. GPL has 9 features, grouped into 4 categories. A graph can be directed or undirected, its edges are weighted or unweighted, it can have a depth-first or breadth-first search algorithm, and it also has one or more algorithms to detect cycles, calculate shortest paths and find connected regions. Figure 1b lists, among many, two applications that can be built from this product-line: **`GraphApp1`** is a program that implements an unweighted, undirected graph with depth-first search and connected region detection algorithm. **`GraphApp2`** is a program that implements a weighted, directed graph with depth-first search, shortest-path and cycle checking algorithms. We will explain the dot (•) notation shortly.

Algorithm	Search	Weight	Graph Type
Cycle-Checking Shortest-Path Connected-Region	Breadth-First Depth-First	Weighted UnWeighted	Directed Undirected

(a) GPL Features

```
GraphApp1 = Connected-Region • Depth-First • UnWeighted • Undirected
GraphApp2 = Cycle-Checking • Shortest-Path • Depth-First • Weighted • Directed
```

(b) Sample Applications of GPL

Figure 1. A Graph Product-Line (GPL)

Viewing programs in terms of features provides benefits in evolution and maintenance. Both involve requirement changes: some requirements may be modified or become obsolete, while new requirements may be added to a system. Such changes usually translate directly into changes of features — modifying, removing and/or adding them.

2.1 Feature Oriented Programming and AHEAD

The central idea of FOP is to modularize features and build programs from these modules [28][3]. *AHEAD (Algebraic Hierarchical Equations for Application Design)* is a unique formulation of FOP that integrates step-wise development, generative programming, and algebras [2][5]. A fundamental premise of step-wise development is that a complex program can be built from a simple program (called *a base program*) by progressively adding features. This process can be given mathematical precision by appealing to fundamental ideas of generative programming: programs are data and functions (called *transforms*) map programs [7]. AHEAD unifies these ideas to show that models of product-lines have a simple algebraic structure. Namely, base programs are constants, and features that extend programs are functions. Consider the following constants that represent base programs with different features:

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

A *program extension* is a function that takes a program as input and produces a feature-extended program as output, where • denotes function composition.:

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

A multi-featured application is an *equation* that defines how a base program is extended by zero or more features. Different equations define a family of applications, such as:

```
app1 = i•f      // app1 has features i and f
app2 = j•g      // app2 has features j and g
app3 = i•j•f    // app3 has features i, j, f
```

Thus, the features of an application can be determined by inspecting its equation.

2.2 Code Synthesis

Figure 2 depicts a program **P** that is a package of four classes (**class1** — **class4**). **P** is synthesized by composing features **X**, **Y**, and **Z**. Feature **X** encapsulates a fragment of **class1** — **class3**. Feature **Y** extends **class1** — **class3** and introduces **class4**. Feature **Z** extends all four classes. Features encapsulate fragments of classes, and composing features yields packages of fully-formed classes.

Figure 2. P = Z•Y•X

Code synthesis is straightforward: method and class extensions follow common notions of inheritance. Figure 3a shows method **A()** whose body sequentially executes statements **r**, **s**, and **t**. Figure 3b declares an extension of this method whose body says execute **super.A()** followed by statement **w**. The statement **super.A()** invokes the method's previous definition. The composite method is Figure 3c; it is produced by substitution: **super.A()** is replaced with the original body of **A()**.

(a)
```
void A() { r; s; t; }
```
(b)
```
void A() { super.A(); w; }
```
(c)
```
void A() { r; s; t; w; }
```

Figure 3. Method Definition and Extension

Class extensions are similar. Figure 4a shows a class **K** that has three members: methods **A()**, **B()**, and variable **C**. Figure 4b shows an extension of **K** written in an extended-Java language where class extensions are prefaced by the special keyword '**refines**'. 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 4c: 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 [9], mixin-layers [31], and program transformations [7].

(a)
```
class K {
  void A(){ x;y;z; }
  void B(){ m;n; }
  int C;
}
```
(b)
```
refines class K {
   void A(){ super.A();w; }
   void B(){ q;super.B(); }
   String D;
}
```
(c)
```
class K {
  void A(){ x;y;z;w; }
  void B(){ q;m;n; }
  int C;
  String D;
}
```

Figure 4. Class Definition and Extension

We scale these ideas as follows: each constant feature encapsulates a set of base classes. Each function feature encapsulates a set of base classes and class extensions. In Figure 5, feature **X** encapsulates the definitions of base classes **class1** — **class3**. Feature **Y** encapsulates extensions of **class1** — **class3** plus base **class4**. When **Y** is composed with **X**, corresponding classes in each feature are composed and the remaining classes (in this case **class4**) are copied. Large systems, in excess of 200K LOC Java, have been synthesized with these ideas [5].

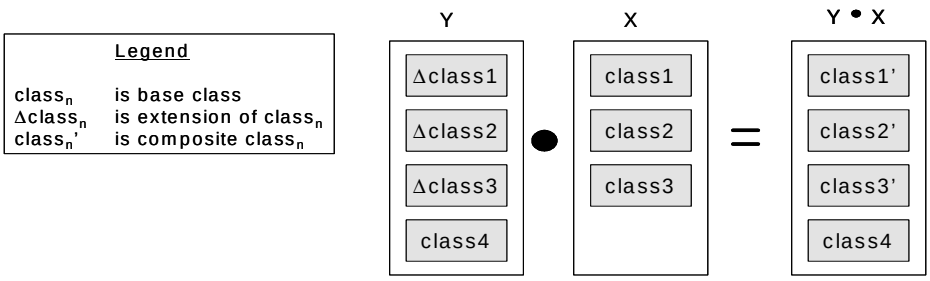


Figure 5. Composition of Features Y and X

3 Structural Feature Interactions

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., [11][17][28][36]): we focus on a particular form of interactions that are static and structural: *how a feature influences (or changes) the source code of another.* In this section we show how structural feature interactions impact program designs with a simple example.

3.1 Stack Product-Line

In 1997, Prehofer presented an FOP model of a stack product-line [28]. The following is a modified version and an AHEAD representation of his example. The product-line consists of three features: **`stack`**, **`counter`**, and **`lock`**. **`stack`** implements basic stack operations such as **`push()`** and **`pop()`**; **`counter`** adds a local counter to track the size of the stack; **`lock`** provides a switch to allow/disallow operations on the stack. Implementations of these features are shown in Figure 6a-c.

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

  void empty() {
    s = "";
  }
  void push( char a ) {
    s = a + s;
  }
  void pop() {
    s = s.substring(1);
  }
  char top() {
    return s.charAt(0);
  }
}
```

(a) stack

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

  int size() { return ctr; }
  void reset() { ctr = 0; }
  void inc() { ctr++; }
  void dec() { ctr--; }

  void empty() {
    reset();
      super.empty();
  }
  void push( char a ) {
    inc();
    super.push( a );
  }
  void pop() {
    dec();
    super.pop();
  }
}
```

(b) counter

```
refines class stackOfChar {
  boolean lck = false;
  void lock() { lck = true; }
  void unlock() {
    lck = false;
  }
  void empty() {
    if ( !lck )
super.empty();
  }
  void push( char a ) {
    if ( !lck )
super.push(a);
  }
  void pop() {
    if ( !lck ) super.pop();
  }
  void inc() {
    if (!lck) super.inc();
  }
  void dec() {
    if (!lck) super.dec();
  }
  void reset() {
    if (!lck) super.reset();
  }
}
```

(c) lock

Figure 6. Feature Modules in Stack Product-Line

This is a common FOP design. There is a kernel feature (**`stack`**) that introduces the underlying class structure of the program and defines the basic stack operations. Each subsequent feature makes an improvement of the base program by adding a coherent set of new functionalities. Note that each new feature builds upon prior features, so that it can make proper extensions to integrate the new functionalities into the program. For example, feature **`lock`** is written with the full knowledge of **`stack`** and **`counter`**, and it updates the stack body and the counter when necessary. To generate programs from this product-line, features are composed in order, for example **`counter•stack`** yields a stack with a counter, and **`lock•counter•stack`** produces a full-featured stack.

Features interact structurally by extending methods defined in other features [28]. Feature **counter** extends methods **push()**, **pop()**, **empty()** defined in feature **stack**. It does so in order to integrate the 'counter' functionality into the stack. Feature **lock** extends methods **push()**, **pop()**, **empty()** defined in feature **stack** and methods **inc()**, **dec()**, **reset()** defined in feature **counter**. Again, these extensions are necessary in order to add the 'lock' functionality to a stack with a counter.

3.2 Feature Optionality Problem

The above design has one problem: suppose we want a stack that is locked with no counter. An intuitive way to compose **stack** with **lock** is by the expression **lock•stack**. A closer examination, however, reveals that the composition does not produce the desired program. Feature **lock** extends methods defined in **stack** (**push()**, **pop()**, **empty()**) *and* methods defined in **counter** (**inc()**, **dec()**, **reset()**). Now that **counter** is absent, **lock** tries to extend non-existent methods; in another words, **lock** interacts with a non-existent feature! This is a general problem — the *Feature Optionality Problem* — in feature-oriented designs: *because we build into a feature its interactions with other features, the feature breaks when one of these other features is not present*. This is an undesirable effect that undermines feature reusability, as feature optionality can no longer be achieved. It is especially noticeable in software product-lines, since product-line members often only use a partial set of all features.

Since the cause of the problem is that different feature interactions are encapsulated in the same module, we can improve our feature design by separating these interactions into different modules [28]. We restructure **counter** into two parts in Figure 7a-b: **counter** is the base feature; $\mathbf{stack}_{\mathbf{counter}}$ encapsulates **counter** interactions with **stack**: these interactions extend methods **push()**, **pop()**, and **empty()**. Similarly we restructure **lock** into three parts as Figure 7c-e shows: **lock** is the base feature; $\mathbf{stack}_{\mathbf{lock}}$ encapsulates the interaction of **lock** with **stack**; and $\mathbf{counter}_{\mathbf{lock}}$ encapsulates the interaction of **lock** with **counter**.

With this separation, we can compose the stack program with any combination of features. To build a stack with only a counter, we use **stackcounter•counter•stack**. To build a stack with a lock, we use **stacklock•lock•stack**. For a full-featured stack, we compose every module:

$$\mathbf{counter}_{\mathbf{lock}} \bullet \mathbf{stack}_{\mathbf{lock}} \bullet \mathbf{lock} \bullet \mathbf{stack}_{\mathbf{counter}} \bullet \mathbf{counter} \bullet \mathbf{stack}$$

The value of this idea is clear: interactions and base features separate concerns. *The semantics of interaction modules and base feature modules are different*. However, the idea has its limitations. First, the example is too simple: it deals with a single class. Features generally cross-cut many classes. Second, feature interactions are not limited to two features; multi-feature interactions are possible. Third, 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.

4 Software Derivatives

Base features (e.g. **lock**) encapsulate method and variable definitions of an application or product-line. That is, we generalize the stack 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.*

Interaction module $\mathbf{X_Y}$ expresses the concept of a *derivative*: how feature **X** changes with respect to feature **Y**. Henceforth, we write $\mathbf{X_Y}$ as $\partial\mathbf{X}/\partial\mathbf{Y}$. Like base features, a derivative is a module that encapsulates methods and variables. Additionally and more importantly, a derivative also encapsulates method extensions of its base feature. That is, derivative $\partial\mathbf{X}/\partial\mathbf{Y}$ encapsulates extensions to zero or more methods of **X** made by **Y**. Figure 7 illustrates these ideas. It shows two base features (**counter** and **lock**) and three derivatives — $\mathbf{stack_{counter}}$ is $\partial\mathbf{stack}/\partial\mathbf{counter}$, $\mathbf{stack_{lock}}$ is $\partial\mathbf{stack}/\partial\mathbf{lock}$, and $\mathbf{counter_{lock}}$ is $\partial\mathbf{counter}/\partial\mathbf{lock}$.

This definition of base features and derivatives leads to a large set of commutative identities. For example, base features can be composed in arbitrary order. That is, for two different base features **X** and **Y**:

$$\mathbf{X \bullet Y = Y \bullet X} \qquad \mathbf{(1)}$$

The reason is that base features do not interact and do not share members. Composition reduces to set-union (see [5]), and thus the order in which base features are composed does not matter. Similarly, the composition of derivatives with different *numerators* commute. For different features **X**, **Y**, and **Z**:

$$\partial\mathbf{X}/\partial\mathbf{Y} \bullet \partial\mathbf{Z}/\partial\mathbf{Y} = \partial\mathbf{Z}/\partial\mathbf{Y} \bullet \partial\mathbf{X}/\partial\mathbf{Y} \qquad \mathbf{(2)}$$

```
refines class stackOfChar {
  int ctr = 0;
  int size() { return ctr; }
  void inc() { ctr++; }
  void dec() { ctr--; }
  void reset() { ctr = 0; }
}
```

(a) counter

```
refines class stackOfChar {
  void empty() { reset(); super.empty();}
  void push( char a ) { inc(); super.push( a );}
  void pop() { dec(); super.pop();}
}
```

(b) $\mathbf{stack_{counter}}$

```
refines class stackOfChar {
  boolean lck = false;

  void lock() {
    lck = true;
  }
  void unlock() {
    lck = false;
  }
}
```

(c) lock

```
refines class stackOfChar {
  void push( char a ) {
    if ( !lck )
        super.push( a );
  }
  void pop() {
    if ( !lck )
        super.pop();
  }
  void empty() {
    if ( !lck )
        super.empty();
  }
}
```

(d) $\mathbf{stack_{lock}}$

```
refines class stackOfChar {
  void inc() {
    if ( !lck )
        super.inc();
  }
  void dec() {
    if ( !lck )
        super.dec();
  }
  void reset() {
    if ( !lck )
        super.reset();
  }
}
```

(e) $\mathbf{counter_{lock}}$

Figure 7. Separating Feature Interactions

∂**X**/∂**Y** encapsulates changes of **X** methods by **Y** and ∂**Z**/∂**Y** encapsulates changes of **Z** methods by **Y**. Because the set of **X** and **Z** methods are disjoint, the order in which their derivatives are composed does not matter.

Derivatives with the *same* numerators, however, do not commute — the order of composition is generally important. For different features **X**, **Y**, and **Z**, generally ∂**X**/∂**Y** • ∂**X**/∂**Z** ≠ ∂**X**/∂**Z** • ∂**X**/∂**Y**. A simple example of non-commutativity is depicted in Figure 8, where **Xvar**, **Yvar**, and **Zvar** are variables belonging to features **X**, **Y**, and **Z**, respectively. Composing these derivatives in different orders yields different programs.

(a)∂**X**/∂**Y**

```
void methodX() {
   super.methodX();
   Xvar = Xvar + Yvar;
}
```

(b)∂**X**/∂**Z**

```
void methodX() {
   super.methodX();
   Xvar = Xvar * Zvar;
}
```

Figure 8. Non-Commutable Derivatives

4.1 Forward and Backward Interactions

In our example we see that **stack** is modified by **counter** and **lock**, yielding derivatives as ∂**stack**/∂**counter** and ∂**stack**/∂**lock**. We call these *forward interactions*, because a feature is modified by features that are composed later. Forward interactions occur when a method is extended by subsequent features. If we look at the resulting extended method, it would appear that the method's body references members (e.g., variables, functions) that are defined *in the future* by a feature that is added.

With this same line of reasoning, a feature can be modified by *previously* defined features. We call these *backward interactions*. Backward interactions occur when a feature references a data member or a method defined in previously composed features, which is common in layered program designs. Consider the **undo** feature of Figure 9. It backs up and restores the contents of a counted stack. The composition of three features **stack**, **counter**, and **undo** produces an undoable counted stack.

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

  void backup() {
    s_bak = s;
    ctr_bak = ctr;
  }
  void undo() {
    s = s_bak;
    ctr = ctr_bak;
  }
  void empty() {
    backup();
    super.empty();
  }
  void push(char a) {
    backup();
    super.push(a);
  }
  void pop() {
    backup();
    super.pop();
  }
}
```

Figure 9. Feature **undo**

Feature **undo** extends basic stack operations — **push()**, **pop()**, and **empty()** — which produces forward interactions with **stack** just like **counter**. But in method **backup()** and **undo()**, it also references data members defined in previous features: **s** in **stack** and **ctr** in **counter**. These are examples of backward interactions where **undo** can be seen as being "modified" by **stack** and **counter**. We can create derivatives that separate these interactions into individual modules, just as we did earlier. Figure 10a-d shows this separation. Figure 10a is the base feature of **undo** and Figure 10b is the forward interaction between **stack** and **undo**. Figure 10c-d shows **undo**'s backward interactions with **stack** and **counter** respectively.

In the general case, *every feature can influence every other feature in a program*: so not only can a feature influence the features that were composed previously, *it also can*

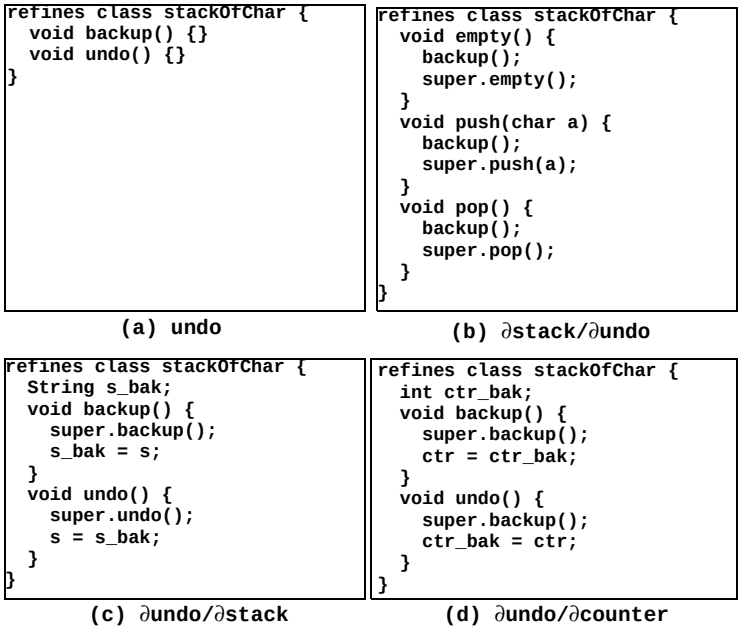


Figure 10. Derivatives of Bi-Directional Interactions

exert an influence on features that will be subsequently composed. Both kinds of interactions are common in applications we have seen.

4.2 Higher-Order Derivatives

As in differential calculus, derivatives are operators. We believe the same applies here. ∂/∂**X** denotes an operator on programs that denotes how feature **X** changes a program. If **P** is a program, the change is denoted ∂**P**/∂**X**. This leads to a pair of interesting ideas and generalizations.

First, we now have a general way to express interactions among multiple features. For example, the interaction of **X** with the interaction of **Y** with **Z** is a *second order derivative*:

$$(\partial/\partial \mathbf{X})(\partial \mathbf{Z}/\partial \mathbf{Y}) = \partial^2 \mathbf{Z}/\partial \mathbf{X}\partial \mathbf{Y} \qquad (3)$$

Such derivatives have a simple interpretation: ∂^2**Z**/∂**X**∂**Y** is a module that encapsulates the changes made to feature **Z** by the *combined* features **X** and **Y**. Such a module typically encapsulates (before, after, and around) extensions of **Z** methods, such as:

```
void methodZ() {          // example of “after advice”
  super.methodZ();
  code that references members of by X, Y, and/or 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. That is, ∂^n**A**/(∂$\mathbf{B}_1$...∂$\mathbf{B}_n$) defines a module that extends methods introduced by **A** and references members in **A**, $\mathbf{B}_1$...$\mathbf{B}_n$.

Second, our observations on a wide set of programs suggest that differentiation distributes over composition. That is, the derivative of a composition equals the composition of its derivatives:

$$(\partial/\partial X)(\ R \bullet S\ ) = (\partial R/\partial X) \bullet (\partial S/\partial X) \qquad (4)$$

Intuitively, **(4)** states the fact that the impact of a feature (**X**) to a composite of two modules (**R** and **S**) can be seen as the composition of this feature's impact to each of the two modules ($\partial R/\partial X$ and $\partial S/\partial X$). A simple case is when **R** and **S** are base features or derivatives of different base features, meaning they define or extend different set of class members. In this case their composite **R•S** is the union of the definitions/extensions they each encapsulate, and any modification that **X** makes to them can be classified as either **X**'s modification to **R** ($\partial R/\partial X$), or **X**'s modification to s ($\partial S/\partial X$). It is less obvious when **R** and **S** define or extend the same class members. Feature **X** further modifies these class members by extending them after the composition of **R** and **S**. We believe these modifications made by **X** can be factored into two parts, $\partial R/\partial X$ and $\partial S/\partial X$, where **(4)** holds.

4.3 Abstract and Concrete Models

In earlier discussions we implicitly used two different feature models: an abstract model and a concrete model. An abstract model contains a set of abstract features where feature interactions are *implicit*. This represents a user's view of the system. An abstract feature **X** (names of abstract features are underlined) has the exactly the same methods of its concrete base feature **X**, but the implementation of these methods is undefined. For the stack example, the abstract model **A** has four features:

$$A = \{\ \underline{stack},\ \underline{counter},\ \underline{lock},\ \underline{undo}\ \}$$

Applications of a product-line are produced by composing abstract features. Among the members of **A**'s product-line are:

```
stack                 // stack
counter • stack       // stack with a counter
lock • stack          // stack with a lock
lock • counter • stack // stack with both a counter and a lock
undo • stack          // undoable stack
undo • counter • stack // undoable stack with a counter
```

On the other hand, a concrete model makes feature interactions *explicit* — it is the set of all base features and all feature interaction modules that represent the implementation view of a product-line. For the stack example, a concrete model **C** contains 10 non-empty modules — 4 base features and 6 derivatives:

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

Although product-line members are specified by compositions of abstract features, they must be built by composing concrete modules — base features and derivatives. Thus, we need a way to translate abstract expressions into concrete expressions.

Let **X** and **Y** be abstract features and **X•Y** their composition. Let **X**, **Y**, ∂**Y**/∂**X**, and ∂**X**/∂**Y** denote the corresponding concrete base features and their interactions. The relationship that maps an abstract feature expression to a concrete feature expression is:

$$\mathbf{X} \bullet \mathbf{Y} = \partial\mathbf{X}/\partial\mathbf{Y} \bullet \partial\mathbf{Y}/\partial\mathbf{X} \bullet \mathbf{X} \bullet \mathbf{Y} \quad \textbf{(5)}$$

The semantics of **(5)** is straightforward: to compose abstract features **X** and **Y**, we compose their base features (**X** and **Y**) and their mutual interactions (∂**X**/∂**Y** and ∂**Y**/∂**X**). **(5)** elevates this relationship to a general identity. Here is why **(5)** 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:

undo • **counter** • **stack**
= ∂^2**undo**/∂**counter**∂**stack** • ∂**undo**/∂**counter** • ∂**undo**/∂**stack** • **undo**
• ∂^2**counter**/∂**undo**∂**stack** • ∂**counter**/∂**undo** • ∂**counter**/∂**stack**
• **counter** • ∂^2**stack**/∂**undo**∂**counter** • ∂**stack**/∂**undo** • ∂**stack**/∂**counter**
• **stack**

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. These same tools will lookup each term in the database: if a term is not present (such as ∂^2**stack**/∂**undo**∂**counter** above), it is inferred to be empty. (An *empty* module is equivalent to the identity function). If it is present, the module is retrieved. All non-empty modules are then composed in the order dictated by the concrete expression, and the target application is then synthesized:

(∂**undo**/∂**counter** • ∂**undo**/∂**stack** • **undo)** • **counter**
• **(**∂**stack**/∂**undo** • ∂**stack**/∂**counter** • **stack)**

In theory, a composition of *n* abstract features may map to a composition of $O(n*2^n)$ concrete features. We have observed that a *vast* majority of derivatives are empty, so that a composition of *n* abstract features maps to a composition of something like $O(n^2)$ concrete features or less. Future experimentation will better reveal the nature of this expansion.

5 Feature-Oriented Refactoring of Legacy Applications

How can derivatives be used to solve a practical problem? We consider such an application in this section, along with some preliminary experimental results.

A common maintenance request is to add and remove features from an existing application. If the application has an FOP design, this is a simple task. Features are added and removed, and the application is rebuilt. If non-FOP designs are used, this task is much more expensive because the source for individual features is scattered throughout the code base, and engineers must manually find and edit this code. An interesting problem is to refactor legacy applications which do not have FOP designs into an equivalent form where they do have FOP designs. By doing so, FOP's ease of feature extensibility can be exploited.

To refactor legacy applications, we will use a relationship between FOP designs and OO frameworks [4]. An OO framework is a set of abstract classes that encapsulate generic

code. Some methods are abstract — called *hot-spot* methods — whose bodies have yet to be defined. An OO framework is instantiated when a concrete class is created for every abstract class. These concrete classes define the bodies of hot-spot methods. From an FOP perspective, the resulting program is **C•F**, meaning **F** is the (possibly composite) feature that defines a set of abstract classes of the framework, and **C** is the (possibly composite) feature that defines an extension to **F** that completes the program.

Legacy applications can be refactored into an FOP design in two phases. First, a user defines an abstract FOP expression from which the legacy program is to be synthesized. This expression states what abstract features are present in the program and what order these features are composed. Our model of derivatives tells us that features partition the set of data members and methods of an application. The first step in a Legacy-to-FOP refactoring is the process of assigning data members and methods of the application to distinct features. Doing so partitions the application so that each member and method is "tagged" with the name of a feature that introduced the method. Tool support can simplify this tagging process [30].

The next step is to refactor the bodies of methods into base components and derivatives. For each feature, its data member definitions and method definitions should be put into its base feature. If a method does not reference any member defined outside the feature, its body goes to the concrete base feature. The only non-obvious cases are methods that reference data members and/or methods defined in other features. These are hot-spot methods.

Hot-spot methods are easy to detect. Suppose method **M** belongs to feature $\underline{\mathbf{F}}_i$. If **M** references data members or methods of another feature $\underline{\mathbf{F}}_j$, then **M** is a hot-spot method. That is, **M** must be partitioned into a core method $\mathbf{M}_i$ (which references only $\underline{\mathbf{F}}_i$ members and will be part of the its base feature) and a method extension $\mathbf{M}_j$ (which references $\underline{\mathbf{F}}_j$ members and will be part of the derivative $\partial\mathbf{F}_i/\partial\mathbf{F}_j$). If $j>i$, meaning that $\underline{\mathbf{F}}_j$ was composed after $\underline{\mathbf{F}}_i$, then $\partial\mathbf{F}_i/\partial\mathbf{F}_j$ is a forward interaction; otherwise it is a backward interaction. Figure 11 illustrates this concept. Of course, we expect the original methods may not be easily partitioned without rewriting, as first observed by Murphy [24]. As in [24], our initial approach will ask users to partition methods manually. As our understanding of this process matures, we expect to add to our tools an infrastructure that analyzes and manipulates control flow and data flow graphs to suggest an automatic means of method partitioning.

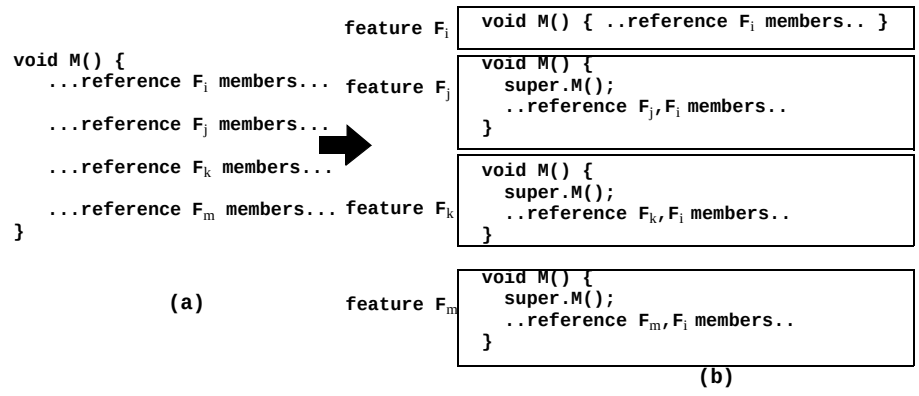


Figure 11. Factoring Hot-Spot Methods

In summary, we begin with a legacy application **P** (Figure 12a). We then assign each class member to a feature (Figure 12b) which vertically partitions an application. Each partition contains only the class members assigned to the corresponding feature. Next, we classify methods into derivatives and allow users to refactor methods so that each vertical partition is decomposed into a base feature and derivatives (Figure 12c). We then use our algebra to reshuffle the resulting FOP expression into "horizontal" layers which yields an FOP design (Figure 12d) that conforms to Figure 2. That is, each layer defines an implementation of an abstract feature the user specified in the beginning of the refactoring process. *Without derivatives, we could not do these last critical steps*. We believe that the resultant FOP design will make it easier to add new features to **P**, remove existing features from **P** or modify existing features, thus making **P** easier to evolve.

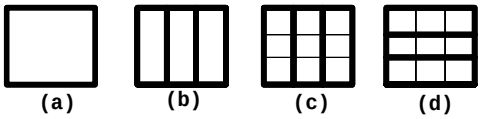


Figure 12. Refactoring Process

5.1 An Example — Java Notepad Application

In this section we use a notepad application[2] as an example to demonstrate our feature-oriented refactoring process. This is a simple application written in Java that supports common file operations (open, save, etc.) and edit operations (copy, paste, etc.) as Figure 13 illustrates. The original program consists of a single source file **`Notepad.java`** which is about 24K bytes in size and contains 1K lines of code.

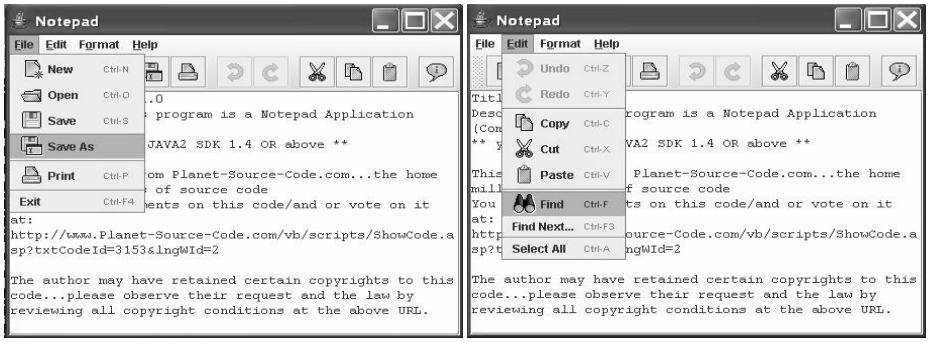


Figure 13. `Notepad.java`

To refactor this program, our first step identifies the set of features in the application and defines a composition order. We identified 13 features and grouped them into 6 categories as Table 1 shows.

2. This application was written by S. Al-Thubaiti and can be downloaded at `http://www.pscode.com/vb/scripts/ShowCode.asp?txtCodeId=3153&lngWId=2`

Table 1. Feature Set of Notepad

Group	Base	File	Edit	Print	Format	Help
Features	Base	New Open Save SaveAs	Cut Copy Paste Find	Print	Format LineWarp	Help

Feature **Base** provides a framework for the notepad. There are 4 file features, 4 edit features, and some additional features such as **Print**, **Format**, **LineWarp**, and **Help**. These features are composed in an order (there are many such orders) that is consistent with step-wise development:

```
Notepad =  Help
         • LineWarp • Format
         • Print
         • Find • Paste • Copy • Cut
         • SaveAs • Save • Open • New
         • Base
```

During refactoring, we noticed that there are no interactions between features from different groups other than **Base**. This is because other than the adding menu items and buttons to the base notepad frame, each feature group implements separate functions of the application that neither depend nor influence each other. For example, file operations and edit operations are independent of each other and do not change each other's code at all. However, inside each feature group there are quite a number of interactions. To name a few, the presence of feature **Save** requires prompting to save the current file before opening a new file with feature **Open**; feature **Save** and feature **SaveAs** share common code for disk operations. The following list shows the base features and interactions among the base frame and file operation group:

$$\{\ \mathbf{Base},\ \mathbf{New},\ \mathbf{Open},\ \mathbf{Save},\ \mathbf{SaveAs},\ \partial\mathbf{Base}/\partial\mathbf{Open},$$
$$\partial^2\mathbf{Base}/\partial\mathbf{Open}\partial\mathbf{New},\ \partial\mathbf{Base}/\partial\mathbf{Save},\ \partial\mathbf{Base}/\partial\mathbf{SaveAs},$$
$$\partial^2\mathbf{Base}/\partial\mathbf{SaveAs}\partial\mathbf{Open},\ \partial^2\mathbf{Base}/\partial\mathbf{Save}\partial\mathbf{Open},\ \partial\mathbf{New}/\partial\mathbf{Base},$$
$$\partial^2\mathbf{New}/\partial\mathbf{Open}\partial\mathbf{Base},\ \partial\mathbf{Open}/\partial\mathbf{Base},\ \partial\mathbf{Open}/\partial\mathbf{Save},$$
$$\partial\mathbf{Save}/\partial\mathbf{Base},\ \partial\mathbf{Save}/\partial\mathbf{Open},\ \partial\mathbf{Save}/\partial\mathbf{SaveAs}\ \}$$

For the most part, methods of the original application were easily factorable as suggested in Figure 11, although doing so manually consumes time. As expected, we did have to restructure some methods — by reordering statements for example — so that they could be composed from feature modules. These restructurings were not difficult to do and did not alter the application's behavior. We did notice that we could have used several different ways to restructure a method. Choosing the "best" requires some foresight on what new features might be added subsequently to the application. This is typical of issues in FOP product-line designs and is not new or surprising.

Once we refactored the application into base features and derivatives, we were able to build different versions by omitting abstract features. Suppose a user wants a notepad **N** with the basic file functions — only **New**, **Open** and **SaveAs**, which is the abstract composition

$$N = \underline{SaveAs} \bullet \underline{Open} \bullet \underline{New} \bullet \underline{Base}$$

To build **N**, we need to map this abstract expression to a concrete expression. Applying the mapping rule **(5)** and eliminating empty terms, we get:

$$N = (\partial^2 Base/\partial SaveAs \partial Open \bullet \partial Base/\partial SaveAs \bullet SaveAs) \quad \text{// implements } \underline{SaveAs}$$
$$\bullet\ (\partial Open/\partial Base \bullet \partial^2 New/\partial Open \partial Base \bullet \partial^2 Base/\partial Open \partial New$$
$$\bullet\ \partial Base/\partial Open \bullet Open) \quad \text{// implements } \underline{Open}$$
$$\bullet\ (\partial New/\partial Base \bullet New) \quad \text{// implements } \underline{New}$$
$$\bullet\ Base \quad \text{// implements } \underline{Base}$$

Note that we have rearranged terms with commutativity law **(2)** so that it shows the layered structure with each layer implementing an abstract feature under this configuration. When an abstract feature is composed to the program, derivatives are introduced only if their corresponding feature interactions appear. Evaluating this concrete feature expression yields the version of notepad illustrated in Figure 14.

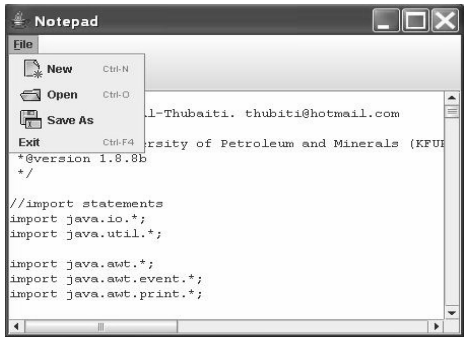
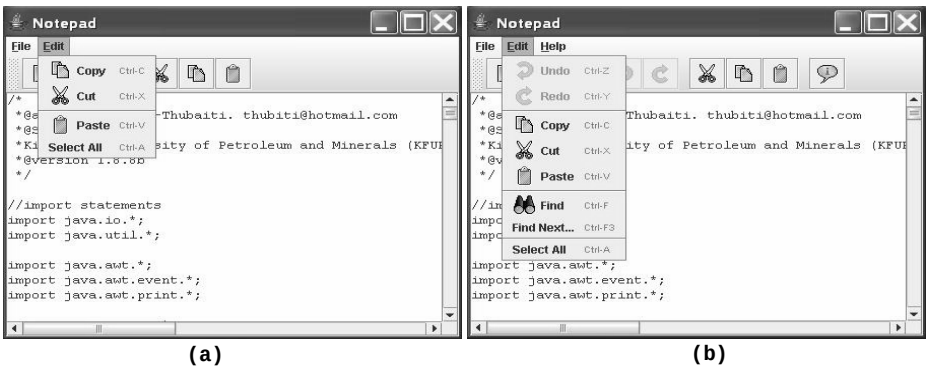


Figure 14. Notepad with Basic File Operations

Given an FOP design, it is easy to synthesize other versions of notepad. Figure 15a is a version with both basic file operations and edit operations. Figure 15b enhances it with additional edit operations and a help menu. Although much of our work for this example was done manually, it demonstrated the basic soundness of our approach. Our next steps are to develop tools to aid this process.



(a) **(b)**

Figure 15. Different Customizations of Notepad

6 Related Work

6.1 Feature-Related Models

FOP models have a long history, originating in collaboration-based designs and their implementations as mixins and mixin-layers [9]. Features encapsulate *cross-cuts*, a concept that was popularized by *Aspect-Oriented Programming (AOP)* [18]. There are three basic differences between AOP and FOP. First, FOP is based on a step-wise development methodology that is innate to the familiar component-based software development approaches; AOP is based on a different methodology [22]. Second, the starting points for FOP and AOP differ: product-lines are the consequence of pre-planned designs (so features are designed to be composable); this is not a part of the standard AOP paradigm. Third, the novelty and power of AOP is in quantification. Quantification is the specification of where advice is to be inserted (or the locations at which extensions are applied). The use of quantification in AHEAD is no different than that used in traditional OO frameworks [4][23]. There is work on aspect "interactions", but it deals more with the ordering in which aspects are composed, rather than the changes one aspect makes to another [12].

Multi-Dimensional Separation of Concerns (MDSoC) is another program extension/composition technology [33], where Hyper/J [25] is the premier tool. Features correspond to hyperslices, and feature compositions correspond to compositions of hyperslices. The primary difference between FOP and Hyper/J is the algebraic foundation of FOP and its relationship to step-wise program development. Hyper/J has no algebraic model of program synthesis/composition, and no emphasis on feature/concern interactions.

Plath and Ryan proposed a feature integration model for specifying and composing features [27]. Features are written in a model checking language, where each feature can introduce new modules and/or change modules defined in other features. These feature specifications can be used as inputs to a model checker to verify certain temporal logic properties of the system. While our model deals with code artifacts, the work on feature integration models features highly abstractly.

6.2 Feature Interactions

There is an enormous literature on feature interactions in telecommunications (e.g., [10][11][17][36]). Despite progress, it is found that feature interactions still are intrinsically difficult [10][11]. Generally prior work addresses different topics of feature interactions, as most emphasize the dynamic or run-time impact features have on each other (e.g. [36]), rather than the structural interactions that we examine. There are a few examples of static interaction models but they follow Prehofer's initial work [28][29].

It has been argued that prior research on structural feature interactions in telecommunications (e.g., [26][28]) has not worked [14]. A typical argument goes like this: Features are defined by *finite state machines (FSMs).* A FSM defines a base telecommunications system. Feature A adds states to this FSM and transitions to/from the base to these states. Feature B adds its own set of states and transitions to/from the base. Events that are specific to B can arise during the execution of A-added states, and vice versa. To properly handle these events, transitions between A- and B-added states are needed. This clearly doesn't scale, as feature interactions grow exponentially.

There is a simple explanation for this: *not all domains should compose features statically.* An example is an FOP design created for Motorola's product-line of hand-held

radios [6]. Features were modularized as message and signal transducers and each feature was realized as a state machine. Features were composed dynamically at *run-time*, where the order in which features were composed changed as external events were processed. Creating a static FSM to describe feature interactions exposes the problems outlined in the previous paragraph. This and earlier studies [1] have shown that when features are composed dynamically, features transform *messages/signals*. When features are composed statically, they transform *source code*.

The architecture invented for Motorola is similar to Jackson and Zave's *Distributed Feature Composition (DFC)* [14][38]. That is, features are defined as message and signal transducers and the order in which features are composed changes at run-time as external events are processed. Many of the advantages cited for DFC are *exactly* the same as those we have observed for AHEAD, and consequently we believe AHEAD and DFC are closely related. For example, even when features are composed dynamically, software derivatives can arise. Zave gives the following example [38]: a mid-call-move (**MCM**) and call-waiting (**CW**) feature can be composed in different orders to yield different behaviors:

> "Because of the structure imposed by the DFC architecture, the composition is guaranteed to be well-defined. On the other hand, the requirements analyst might feel that a subscriber should be prevented from executing a mid-call move when he has someone on hold. To achieve this behavior, it will be necessary to compromise the modularity slightly. For example, the **CW** program might be modified to absorb move commands when someone is on hold, so that they never reach an **MCM** box."

This "compromise in modularity" is changing ∂**CW**/∂**MCM** from an identity function to an extension of **CW** that absorbs mid-call move events.

6.3 Feature Refactoring and Program Slicing

Feature refactoring focuses on identifying and extracting features from a program and is a relatively new area of research. Prior work known to us concentrates on identifying code of a feature (for example, having tools display how feature code is distributed throughout a program) and factoring the code into a single module or aspect. Unlike our work, there is (1) no underlying algebraic model of composition (i.e., to know what to do with the feature after it has been identified and modularized), (2) no precise guidelines to determine what can go into a feature and what should not, (3) no mechanism by which identified features can be added to other programs (as is done in product-lines and generative programming), and (4) no model or notion of feature interaction. Despite these differences, the essential idea of labeling methods to identify feature contents is similar to the approach that we are taking.

Robillard and Murphy [30] utilize a feature exploration and analysis tool called FEAT to find concerns in software applications. After a user has specified a concern and identified one class that belongs to it as a seed, FEAT analyzes the program and constructs a concern graph by tracing class member references and declarations relationships starting from the seed. It allows users to traverse the graph and examine class members encountered and decide whether to label them as part of the concern. Another technique to identify features by running test cases has been proposed in [19]. Test cases are classified by the features they belong, and features can be identified by analyzing code blocks that are impacted by each group of test cases.

Program slicing [34][35] is a technique to isolate code in a program that is dependent on a particular code fragment, so that programmers can better understand and debug the program. It also supports various kinds of code refactorings, such as function extraction and function in-lining. A major difference between program slices and derivatives is that slices are identified by data flow and control flow dependence, while derivatives represent changes made to features' source code. Derivative-based refactorings may be performed by a simple static code analysis. Slice construction process, especially data dependence analysis, is very difficult by just looking at the source code; it is usually done by generating execution traces that track control and data flows.

7 Conclusions

A feature is an increment in program functionality [37]. Features reflect a user's view of a software application, where features generally correspond to end-user requirements. Feature Oriented Programming is a general theory of software development, and in particular, software product-lines where each member program is built from a composition of features. FOP studies 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. FOP has direct relevance to main-stream software-development. Its focus on features as a unit of modularity advances contemporary trends in software design, and supports feature-based software specification, optimization and evolution. Nontrivial systems of different domains have been synthesized using FOP.

Feature interactions are important in feature-oriented designs. Features interact with each other structurally by influencing or changing each other's source code in controlled and well-defined ways. The need for modularizing feature interactions becomes obvious when some feature **X** modifies the source code of optional feature **Y**. Changes to **Y** can't be modularized with **X**, because **Y** is not always present. Such changes must be stored in a separate module. We outlined a theory based on software derivatives in which structural feature interactions are distinguished from base features and are encapsulated separately. Our theory treats feature interactions as general operators that obey algebraic laws. This makes it possible to reason algebraically about features and their interactions in software design, and allows tools to be developed to aid in the process of synthesizing applications from base features and their interactions.

This paper describes only the first few steps and experiments that we have undertaken to develop this theory and to use it to solve an interesting and challenging practical problem. The problem is to refactor legacy OO applications into a feature-based design, where common maintenance operations of feature addition, modification, and removal should be easier to express. The next steps in our research are to develop the theory more formally, to build tools that will aid in the process of legacy-to-FOP refactorings, and to apply this theory and tools to the refactoring of large legacy applications.

References

[1] D. Batory, L. Coglianese, M. Goodwill, and S. Shafer, "Creating Reference Architectures: An Example from Avionics", In *Proc. Symposium on Software Reusability*, Seattle Washington, April 1995.

[2] D. Batory, J. Liu, J.N. Sarvela, “Refinements and Multi-Dimensional Separation of Concerns”, *ACM SIGSOFT 2003 (ESEC/FSE2003)*.

[3] D. Batory and S. O’Malley, “The Design and Implementation of Hierarchical Software Systems with Reusable Components”, *ACM TOSEM*, October 1992.

[4] D. Batory, R. Cardone, and Y. Smaragdakis, “Object-Oriented Frameworks and Product-Lines”. *1st Software Product-Line Conference*, Denver, Colorado, August 1999.

[5] D. Batory, J.N. Sarvela, and A. Rauschmayer, “Scaling Step-Wise Refinement”, *IEEE Transactions on Software Engineering*, June 2004.

[6] D. Batory, B. Tremain, S. Nayar, “A Layered Architecture for Ergonomics Software”, internal Motorola design document, 1997.

[7] I. D. Baxter, “Design Maintenance Systems”, *CACM*, Vol. 55, No. 4 (1992) 73-89.

[8] D. Beuche, H. Papajewski, W. Schröder-Preikschat, “Variability management with feature models”, *Science of Computer Programming*, Volume 53, Issue 3, Pages 333-352, December 2004.

[9] G. Bracha and W. Cook, “Mixin-based inheritance”. In *Proc. of OOPSLA*, 1990.

[10] M. Calder and E. Magill, editors. *Feature Interactions in Telecommunications and Software Systems VI*, IOS Press, Amsterdam, 2000.

[11] E.J. Cameron, N.D. Griffeth, Y.-J. Lin, M.E. Nilson, et al, “A Feature Interaction Benchmark for IN and Beyond”, *Feature Interactions in Telecommunications Systems*, IOS Press, pp. 1-23, 1994.

[12] R. Douence, P. Fradet, and M. Sudholt, “A Framework for the Detection and Resolution of Aspect Interactions”, *Generative Programming and Component Engineering (GPCE) 2002*.

[13] M. Griss, “Implementing Product-Line Features by Composing Component Aspects”, *First International Software Product-Line Conference (SPLC)*, Denver, August 2000.

[14] M. Jackson and P. Zave, “Distributed Feature Composition: A Virtual Architecture For Telecommunications Services”, *IEEE Transactions on Software Engineering*, October 1998

[15] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, “Feature-Oriented Domain Analysis (FODA) Feasibility Study”. Technical Report CMU/SEI90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[16] K. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh, “FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architectures”. Annals of Software Engineering, 5:143--168, 1998.

[17] D.O. Keck and P.J. Kuehn, “The Feature and Service Interaction Problem in Telecommunications Systems: A Survey”, *IEEE Transactions on Software Engineering*, October 1998.

[18] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, “Aspect-oriented programming”. In *Proc. of ECOOP*, 1997.

[19] D. Licata, C. Harris, and S. Krishnamurthi, “The Feature Signatures of Evolving Programs”, *IEEE International Symposium on Automated Software Engineering* (short paper), 2003.

[20] J. Liu and D. Batory, "Automatic Remodularization and Optimized Synthesis of Product-Families", *Generative Programming and Component Engineering (GPCE)*, October 2004.

[21] R. E. Lopez-Herrejon and D. Batory, “A Standard Problem for Evaluating Product-Line Methodologies”, In *Proc. of GCSE*, 2001.

[22] R. Lopez-Herrejon and D. Batory, “Improving Incremental Development in Aspectj by Bounding Quantification”, *Software Engineering Properties and Languages for Aspect Technologies (SPLAT)*, March 2005.

[23] M. Mezini and K. Ostermann, “Variability Management with Feature-Oriented Programming and Aspects”, *SIGSOFT 2004*.

[24] G. C. Murphy, A. Lai, R.J. Walker, and M. P. Robillard, “Separating Features in Source Code: An Exploratory Study”. In *Proc. of ICSE*, 2001.

[25] H. Ossher and P. Tarr, “Hyper/J: Multi-Dimensional Separation of Concerns for Java”. In *Proc. of ICSE*, 2000.

[26] J. Pang and L. Blair, “Separating Interaction Concerns from Distributed Feature Components”, *Electronic Notes in Theoretical Computer Science*, 2003.

[27] M. Plath and M. D. Ryan, “Feature Integration using a Feature Construct”, *Science of Computer Programming*, 41(1), 53-84.

[28] C. Prehofer, “Feature-Oriented Programming: A Fresh Look at Objects”. *ECOOP*, 1997.

[29] C. Prehofer, "Feature-Oriented Programming: A New Way of Object Composition," Concurrency and Computation, vol. 13, 2001.

[30] M. P. Robillard and G. C. Murphy, “Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies”. In *Proc. of ICSE*, May 2002.

[31] Y. Smaragdakis and D. Batory, “Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs”, *ACM TOSEM*, April 2002.

[32] R. E. K. Stirewalt and L. K. Dillon, “A Component-Based Approach to Building Formal Analysis Tools”, *International Conference on Software Engineering*, 2001, 57-70.

[33] P. Tarr, H. Ossher, W. Harrison and S.M. Sutton, “N Degrees of Separation: Multi-Dimensional Separation of Concerns”. In *Proc. of ICSE*, 1999.

[34] F. Tip, “A Survey of Program Slicing Techniques”, *Journal of Programming Languages*, 3(3), September 1995.

[35] M. Weiser, “Program Slicing”, *IEEE Transactions on Software Engineering*, 10(4), 1984.

[36] P. Zave, “Address Translation in Telecommunication Features”, *ACM Transactions on Software Engineering and Methodology*, January 2004.

[37] P. Zave, “FAQ Sheet on Feature Interactions”, `http://www.research.att.com/~pamela/faq.html`

[38] P. Zave, “Distributed Feature Composition: Middleware for Connection Services”, available from `http://www.research.att.com/projects/dfc`.