Keynote at the 2006 ECOOP Workshop on Aspects, Dependencies, and Interactions (ADI'06), Nantes, France, July 3rd, 2006.

Feature Interactions in Feature-Based Program Synthesis

Don Batory
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

Abstract. A feature is an increment in product functionality. Features can encapsulate different program representations, programs can be synthesized by feature composition, and programs can be declaratively specified using features. We explain in this paper how feature-based program synthesis works, how features interact, how interactions are controlled, and explore the relationship between features and aspects.

1 Introduction

Just as the structure of matter is fundamental to chemistry and physics, so too is the structure of software fundamental to computer science. By structure we mean what is a module? And how are modules composed to build other modules? Sadly, the structure of software is not well understood: software design is an art form, and as long as it remains so, our abilities to automate key tasks in program design, construction, and maintenance will be fundamentally limited.

In the 1950s, Watson and Crick discovered the structure of DNA [33]. Prior to their work, our understanding of cellular processes, the mechanics of cellular diseases and how such diseases could be treated, were quite limited. After their work, science and medicine have made phenomenal advances. The fields of genetics, microbiology, biochemistry, and biotechnology have either flourished or were created as a result of understanding and manipulating the structure of DNA.

We are following a similar historical path with respect to program structure. Among the ultimate goals or "challenge problems" in software development is the synthesis of efficient programs from declarative specifications. (To do this means that we *really* understand program structure.) It requires advances in:

- *generative programming* — letting machines (not people) do the hard work of program development given a design,
- *declarative domain specific languages (DDSLs)* — raising the abstraction of languages beyond Java and C# to specify desired programs declaratively, and
- *automatic programming* — letting machines (not people) do the hard work of designing an efficient program.

Program synthesis offers an appealing future for software engineering: what is well-understood is automated; programs can be automatically customized for performance, capability, or both; and most importantly, program maintenance and development costs are reduced.

Interestingly, these goals have been achieved in other engineering disciplines. For example, pages at the Dell or Gateway web sites allow customers to declaratively specify the features of a personal computer that they want [17][19]. Each page is a DDSL for specifying a Dell or Gateway product. There are other web-based DDSLs: you can customize your own BMW [12], and (of all things) there are pages for customized faucets and sinks [3]! We should be able to do the same for software.

Features make this possible. A *feature* is an increment in product functionality [35]. A target product is specified in terms of its features. Architects use features to reason about product designs. By making features first-class entities in design, specification, and implementation, we have the potential to automate significant tasks of product design and development. A key step is to modularize features. AOP can play a role, as features can use static and dynamic crosscuts.

Feature oriented designs arose from work on *software product lines (SPLs)*, where different programs of a product line or *product family* are differentiated by features. The goal of SPL is the systematic and efficient creation of products. In the next section, tools and ideas for feature-based program synthesis are described. The core technologies are generative programming, DDSLs, automatic programming, and of course, modularization technologies like aspects.

2 GPL: An Example

The *Graph Product Line (GPL)* is a family of related Java packages that implement different combinations of graph algorithms [20]. We synthesize members of GPL by composing features. We use a grammar, where tokens denote features, to define the set of all possible feature compositions. Each composition is represented by a sentence of this grammar, and the set of all sentences is the set of all GPL products that can be built. Product line grammars are usually context sensitive, i.e., selecting a feature may require (or exclude) other features. (We note that such dependencies are a form of feature interactions — more on this later). Figure 1a shows a specification for GPL: above the `%%` marker is a context free grammar and below the marker is a set of contextual constraints (e.g., the strongly connected components algorithm `StrongC` requires `Directed` graphs and a depth-first-search `DFS` algorithm). Tools translate this specification into a DDSL that resembles the Dell and Gateway web sites (Figure 1b) [10].

Users can select the desired features of a graph package using GPL's DDSL. Figure 1b shows the specification of a GPL package that implements vertex numbering (`Number`), strongly connected components (`StrongC`), and cycle checking algorithms (`Cycle`) on a `Weighted`, `Directed` graph using a depth-first search algorithm (`DFS`). (The actual DDSL interface is more sophisticated, as other capabilities are present; Figure 1b is sufficient to convey the idea.) Our DDSL automatically propagates constraints as users select features, so that only correct graph program specifications can be written. Further, there are additional capabilities for debugging product line models (i.e., context sensitive grammars) and proofs are offered to explain automatic selections or deselections [10]. Given a specification, our DDSL generates a file that

Fig 1. GPL DDSL

(a)

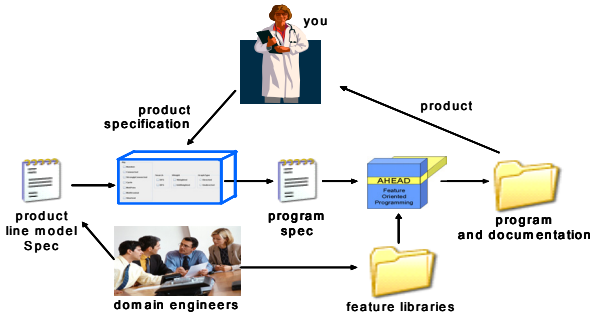
```
GPL : Alg+ Src Wgt Gtp ;
Gtp : Directed | Undirected;
Wgt : Weighted | Unweighted;
Src : BFS | DFS ;
Alg : Number | Connected | StrongC
    | Cycle | MSTPrim | MSTKruskal;
%%
Connected ⇒ Undirected ∧ Src;
StrongC ⇒ Directed ∧ DFS;
Cycle ⇒ Gtp ∧ DFS;
MSTKruskal ∨ MSTPrim ⇒  Undirected ∧ Weighted;
```

(b)

defines the sequence of features that must be composed to synthesize the target application. Other tools perform this composition to yield the source, binaries, and documentation for a GPL application. Other than writing the specification, all steps are done automatically. Although a GPL application is small, the ideas that we describe scale: the tools that we use to synthesize GPL programs were synthesized themselves from features, and collectively these tools are over a hundred times larger (in excess of 200K LOC Java).

Figure 2 shows the big picture of the entire process. Domain engineers understand the variabilities and commonalities of a domain of applications. They create a product line model (i.e., a context sensitive grammar) and a library of implemented features. A user declaratively specifies products using a DDSL (which is synthesized from the product line model) to produce a program specification (i.e., a composition of features). Feature composition tools then synthesize the program and its documentation, and return the result back to the user.

Fig 2. Feature Based Program Synthesis Process



We and others have developed tools to:

- synthesize different program representations (e.g., source, makefiles, grammars) [9][5][18][30],
- automatically analyze feature compositions (e.g., to ensure that all generated programs satisfy particular type safety properties) [11],
- automatically optimize the designs of programs (e.g., choose among different implementations of selected features) [7][36], and
- synthesize tool suites using multi-dimensional separation of concerns (which provides a fundamental way to reduce program specification complexity) [8].

And we have used these tools to synthesize product lines for avionics, fire support simulators, extensible Java compilers, and web applications, among others [9][18]. In the remainder of this paper, we address three questions: (1) How does feature-based program synthesis work? (2) How are features related to aspects? And (3) how do features interact?

3 How Does Feature-Based Program Synthesis Work?

Editing digital photographs is a familiar activity. One takes a photo, uploads it to a PC, and photo editing software is used to, say, crop a photograph and remove red eyes. We generally don't think about this, but photo editors treat photographs as objects. Users can successively invoke methods or functions (e.g., crop, remove red eye) on input photos to produce desired output photos. This is a standard programming language view of applications: the resulting photo is produced by evaluating an expression:

```
newPhoto = redEye(crop(oldPhoto))
```

But what if objects are programs? It is not hard to imagine a tool that allows you to start with a simple program, and by progressively invoking methods (or functions), features are added to this program, both synthesizing its code and documentation. This is exactly how our tools of Section 2 work. Users start with a simple program (e.g., a **`Directed`** graph), add on edge weights (**`Weighted`**), a search algorithm (**`DFS`**), and an assortment of graph algorithms (**`Number`**, **`StronglyC`**, **`Cycle`**). The expression that composes features to produce the target program is called a *feature metaexpression*:

```
program = Cycle(StrongC(Cycle(DFS(Weighted(Directed)))))
```

Metaprogramming treats programs as data and program design and synthesis as a computation. Our tools use features as the basis for a metaprogramming model of program synthesis.

More generally, science is about structure and the manipulation of structure. Software design is about program structure and its manipulation. Our tools manipulate program structure using two operations, which AOP researchers will find familiar [21]:[1]

- adding new structure (i.e., introduction), and
- modifying existing structure (i.e., advising)

In the following sections, we outline two key operations used in feature-based program synthesis, and then explain how these operations relate to feature metaexpressions.

3.1 Introduction Sum (+)

New class members, classes, and packages are incrementally added to a program using the operation *introduction sum (+)*. Consider program `P` in Figure 3a which consists of a single class `C` with a single member `b`. We introduce method `foo()` in Figure 3b, and to this add member `i` in Figure 3c, and add class `D` in Figure 3d. We model these transitions algebraically: the original program in Figure 3a is the metaexpression `P=C.b` (i.e., program `P` consists of a single member `b` from class `C`). Introducing method `foo()` adds a new term to `P`'s metaexpression, namely `P=C.b+C.foo`. Introducing member `i` adds yet another term (`P=C.b+C.foo+C.i`), and introducing class `D` adds several more terms, one per member of class `D`: `P=C.b+C.foo+C.i+D.bar+D.cnt`. Evaluating the metaexpression for `P` synthesizes its code of Figure 3d. Here we assume that the order in which terms are added by introduction sum does not matter, e.g., `C.i+C.j=C.j+C.i`.[2]

```
class C {
  String b;
}     (a)
```

```
class C {
  String b;
  void foo()
  {..}
}     (b)
```

```
class C {
  String b;
  void foo(){..}
  int i;
}       (c)
```

```
class C {
  String b;
  void foo(){..}
  int i;
}
class D {
  String bar;
  int cnt(){..}
}       (d)
```

Fig 3. Incremental Development of Program `P`

Introduction sum algebraically models introductions in AOP, but it is more general. Not only can a feature add new members and methods to an existing class, it can also add new tokens and rules to an existing grammar, new headings and bodies to HTML files, new targets and properties of makefiles, as well as add new classes, new packages, new HTML files, etc. Introduction sum is an operation that augments the *structure* of any program artifact, of which code, HTML files, and grammar files are examples.

---

1. There are more operations, but the essence of our approach is described by two operations.
2. There are versions of our theory where introduction sum is not commutative. Also, constructs such as class initialization, throws clauses, extends and implements clauses can be expressed as additional terms.

3.2 Advice Weaving (•)

Our tools are based on a generalization of mixins, a well-understood but not-yet-common OO technology [1][28][32]. A *mixin* is a class whose superclass is specified by a parameter. Mixins extend a parent class with new variables and methods (e.g., introductions), and allow existing methods to be extended (by the mixin overriding a parent method and calling the parent method via `super`). Mixin-like technologies offer a limited pointcut-advice language: around advice with execution pointcuts that capture individual join points. The reason why we have used mixins is simple: a vast majority of features implement object-oriented collaborations, also known as role-based designs [28]. Collaborations primarily rely on *heterogeneous advice* — advice that advises only one join point [16]. (Or more accurately, collaborations advise many join points, each with a unique advice body). In principle, there is no reason why feature-based tools and languages should have limited advice capabilities: *homogenous advice* (advice that advises multiple join points) are needed [16]. Below we will sketch a model that supports a general form of advice that encompasses existing advice mechanisms in AspectJ.

Consider Figure 4a, which shows a program with a class `C` and an after advice which we call `hi`. The weaving of advice `hi` into `C` yields a program that is equivalent to that in Figure 4b. Whether the program of Figure 4b is statically produced, or whether it is emulated by dynamically advising the execution flow of class `C` in Figure 4a is not important; these are two of many possible *implementations* of Figure 4a. Note that we renamed method `setI` to `setI'` in Figure 4b to differentiate it from the original and unadvised `setI` method of Figure 4a.

```
class C {
  int i,j;
  void setI(int x){ i=x; }
  void setJ(int x){ j=x; }
}
after():
  execution(void *.set*(..))
  { print("hi"); }
```
(a)

```
class C {
  int i,j;
  void setI'(int x)
    { i=x; print("hi"); }
  void setJ'(int x)
    { j=x; print("hi"); }
}
```
(b)

Fig 4. Advice Weaving

We model class `C` of Figure 4a by the introduction sum: `C.i+C.j+C.setI+C.setJ`. We model advice as a function that maps programs to programs, and express the program design of Figure 4a as:

```
P = hi•(C.i+C.j+C.setI+C.setJ)                    (1)
```

where • means function application. That is, `hi` advises all members of the program (which happens to be all members of class `C`). At compile time, a compiler inhales `P`'s source, creates metaexpression `(1)`, and evaluates it. The evaluation is performed in several steps. First, advice distributes over introduction sum:

```
P = hi•C.i + hi•C.j + hi•C.setI + hi•C.setJ                    (2)
```

This distributivity property is basic to aspect-oriented programming: an advice advises all parts of a program. Second, we can simplify `(2)` by noticing that `hi` does not advise members `C.i` and `C.j` as `hi` captures none of their join points (or their shadows [22]). This means that `hi•C.i=C.i` and `hi•C.j=C.j`. Simplifying `(2)`:

```
P = C.i + C.j + hi•C.setI + hi•C.setJ                          (3)
```

Third, `hi` advises each `set` method. Let `setI'` and `setJ'` be:

```
void setI'(int x) { i=x; print("hi"); }
void setJ'(int x) { j=x; print("hi"); }                        (4)
```

which represents the advised `SetI` and `SetJ` methods. Using `(4)`, we simplify `(3)` to:

```
P = C.i + C.j + C.setI' + C.setJ'                              (5)
```

which is our algebraic representation of Figure 4b. In short, feature-based synthesis works by creating a *metaexpression* that defines a program, which includes advice weaving and introduction sums, and simplifying the metaexpression so that all advice has been applied, yielding an introduction sum of primitives.

While treating advice as a function that maps programs to programs is not standard AOP fare [23], our use of algebra does not prevent a standard AOP interpretation. For example, it is perfectly acceptable to interpret `hi•C.setJ` to mean the execution of the `C.setJ` method as advised by `hi`, or the metaexpression `a3•a2•a1•m0` to mean the execution of method `m0` as advised by advice `a1`, `a2`, and `a3` in this order.

Before we proceed, note that the body of an advice can be advised by other advice. We argue, like others [25][4], that an advice body is really a form of introduction, and should be treated as such. Further, advice should be specified separately from its body. While we may write a piece of advice using AspectJ syntax:

```
after(): execution( void *.set*(..)) { print("hi"); }
```

Internally, we treat this as an implicit method introduction and *pure advice*, advice whose body invokes a method and itself has no join points [21]:

```
static void hi() { print("hi"); } // implicit introduction

after(): execution (void *.set*()) hi();   // pure advice
```

See [25][4] for possible implementations of these ideas.

3.3 Metaexpressions

Readers may have noticed our use of two different kinds of metaexpressions. Feature metaexpressions define a program in terms of a composition of features. The metaexpressions that we used in the last two sections involving the operations (+,•) are different: we call them *module expressions*. Module expressions give us a simple way to explain how features are implemented using introductions and advice.

In principle, a base program `B` is simply an introduction sum of terms, where a term is a method or variable. A feature is a function that maps programs to programs. A general form of a feature function `F` is:

```
F(x) = i + a•x                                                    (6)
```

That is, `F` applies advice (`a`) to its input program (`x`), and introduces new terms (`i`).[3] This gives us a simple and precise way to define features and program structure by composing features.

As an example, we can synthesize program `P` of Figure 4 from the following features. Suppose the base program `I` implements class `C` with only variable `i` and the `setI` method. We model `I` as:

```
I = C.i + C.setI                                                  (7)
```

Now, suppose feature `J` introduces variable `j` and the `setJ` method. We model `J` as:

```
J(x) = C.j + C.setJ + x                                           (8)
```

where `J` does not advise its input. Finally, let feature `HI` use `hi` to advise its input:

```
HI(x) = hi•x
```

Given the feature metaexpression `HI(J(I))`, we can expand it to our earlier module definition `(1)`:

```
P      =   HI(J(I))
       =   hi•(C.j + C.setJ + (C.i + C.SetI))   // substitution
       =   hi•(C.i + C.j + C.setI + C.setJ)     // reorder sum
```

The justification for the last step follows from our earlier statement that the order in which terms are added by introduction sum does not matter. By defining features as module expressions, we can directly translate a program's feature metaexpression into module expression. By simplifying the module expression, we can then synthesize a program's code.

---

3. Advice `a` can be a composition of multiple pieces of advice. See [21] for further details.

Incidentally, our discussion of metaprogramming focusses on the static creation of programs given a static composition of features. Feature orientation is much more general: features can be added and removed from programs dynamically. A program's structure — whether or not it changes dynamically — is still expressed in terms of feature metaexpressions and module expressions.

3.4 Bounded Quantification

Another difference between feature orientation and aspect orientation is the use of bounded quantification: advice is not applied at the *end* of a program's construction in feature-based development, but rather to the *current* state of a program's development. That is, when a feature is added to a program, its advice is applied immediately and its advice does *not* "come alive" again to weave subsequent introductions. This is different than AOP and is historically consistent with years of prior work on metaprogramming [27], program transformations [34], and model driven design [31].

As an example of bounded quantification, again let's revisit program `P` of Figure 4a. Suppose feature `K` introduces variable `k` and method `setK`:

```
K(x) = C.setK + C.k + x                                    (9)
```

If we apply `K` to `P`, the module expression for the resulting program (shown in Figure 5) is:

```
K(P) = C.k + C.setK + hi•(C.i + C.j + C.setI + C.setJ)  (10)
```

Note that the `hi` advice is *not* applied to subsequent introductions. Feature-based program synthesis uses *bounded* quantification, i.e., advice is applied after each feature is added to a program, and once woven, the weaving of that advice is finished. In contrast, AOP uses *unbounded* quantification, i.e., advice is always applied at the end of a program's construction and any subsequent introductions are woven by this advice. For example, if only unbounded advice were used, the module expression for the resulting program would be:

```
class C {
  int i,j;
  void setI'(int x)
  { i=x; print("hi"); }
  void setJ'(int x)
  { j=x; print("hi"); }
  int k;
  void setK(int x)
  { k=x; }
}
```

Fig 5. Introduction after Weaving

```
P = hi•(C.k + C.setK + C.i + C.j + C.setI + C.setJ)     (11)
```

which is not the same as `(10)`. That is, `(10)` and `(11)` are different programs. From a mathematical viewpoint, unbounded quantification is a special case of bounded quantification (i.e., all advice is applied at the end of a program's construction).

This raises an interesting question as to why aspects have unbounded quantification semantics. I believe the reason is historical: aspects originated from work on metaclasses [13], where an interpreter is *the* program, and user programs are simply data

that the interpreter executes. In this context, one thinks exclusively about extending the interpreter with new features. In the Figure 6a, we see two methods of an interpreter (`load()` – which loads the text of a program and `methcall()` – which executes a method call), and the text of a program where three method invocations (`m()`, `n()`, and `p()`) are shown. This interpreter constitutes a base program.

In Figure 6b, we see a refinement of the interpreter: after each method call, the string "`hi`" is printed. As the interpreter executes the program text, the advice of each method invocation inserted into the program's control flow (shown by a bubble call-out). In Figure 6c, we see another two refinements of the interpreter, but this time the `load()` method is advised: feature 2 introduces method `y()` to the program text, and feature 3 introduces method `x()` to the program text. So when the interpreter is run, the original program text is loaded, methods `y()` and `x()` are introduced into the text. (`y()` invokes method `q()`, and `x()` invokes method `r()`). Then the program is interpreted, method calls are advised on *both* original program text and on *all* introductions. This is exactly the phenomena of unbounded quantification: one can think of advice and introductions in AspectJ as refinements of *interpreters*, and indirectly refinements of program text.

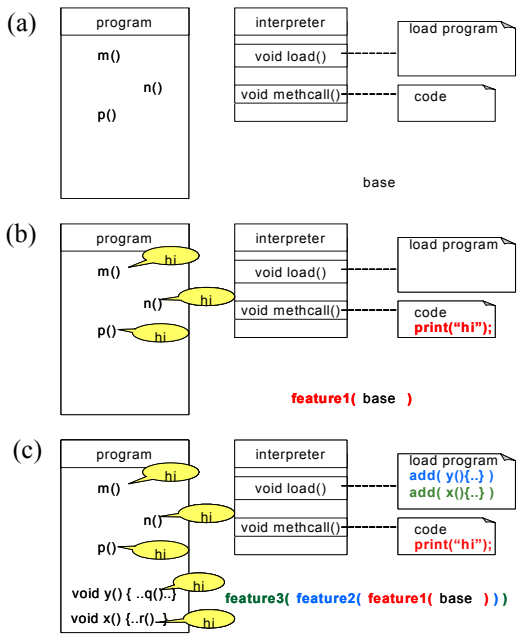


Fig 6. Refining Interpreters

What makes the distinction of bounded vs. unbounded quantification difficult to appreciate is that bounded quantification can *sometimes* be emulated using unbounded quantification by (a) eliminating wild cards and enumerating all targeted join points and/or (b) using "`declare precedence`" constructs. But not always. First, enumeration works in the context of a *single* program where there is a fixed set of features that are composed prior to a given feature. This is uncommon in product lines and program synthesis; it is the norm that there are *many* different sets of features that are composed prior to any given feature. Second, enumerating wild cards is impractical if the number of join points is large.

In principle, features likely need both kinds of quantification: bounded advice is the norm in product lines and incremental development while unbounded advice is used to implement program-wide constraints, like invariants [29][26]. To support feature-based development properly using aspects, several forms of language assistance are

needed. First, features require a much broader notion of introduction than that offered by AspectJ, e.g., we want to encapsulate new classes, new grammar files, etc. within a feature, as well as encapsulating the introduction of new members to existing classes. Second, pieces of advice need to be tagged with a "**bounded**" or "**unbounded**" modifier, to indicate when a piece of advice is local to a particular step in a program's development or when it is a program invariant.

4 Feature Interactions in Program Synthesis

We and others have observed that structuring a program in terms of features helps designers understand and control feature interactions [35].[4] We have recognized two different kinds of interactions. The first is called a *reference interaction* — when a feature **F1** references members introduced by feature **F2**.[5] This is a standard call-graph concept, which requires that feature **F2** be composed before **F1** in a feature metaexpression, i.e., **F1(..F2(..))**. That is, a member must be defined before it can be referenced. The second kind is called a *structural interaction* — when a feature **F1** refines (advises, modifies) members that were introduced by **F2**. Again, **F2** must be composed before **F1** in a feature metaexpression — a member must be defined before it is refined (advised, modified). In both kinds of interactions, no "**declare precedence**" clauses are used. The concept of "**precedence**" is the order in which features are composed.

Now let's look at the feature metaexpression **F4(F3(F2(F1)))**. Let $\mathbf{i}_i$ denote the module expression of the introductions for feature $\mathbf{F}_i$, and let $\mathbf{a}_i$ denote $\mathbf{F}_i$'s advice. Figure 7a graphically depicts this feature metaexpression and exposes all possible interactions. The top-most box in a column represents the introductions of a feature, and boxes vertically below indicate the advice that is being applied by subsequent features. Figure 7b shows the corresponding module expression, where each summand represents the module expression of a column, e.g., **a4•a3•a2•i1** is the module expression for the right-most column.

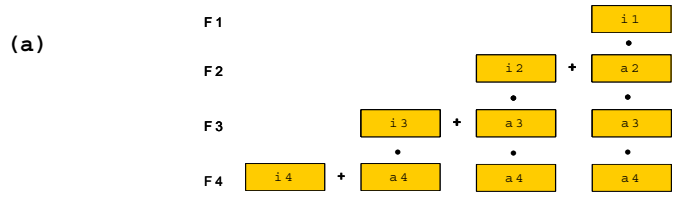


Fig 7. Feature Compositions

4. The literature on feature interactions is vast. See [14][24] for related work.
5. In Section 2, we noted that selecting a feature **F2** may require another feature **F1**. Such constraints are often due to reference interactions.

Each advice box in Figure 7a *in principle* can advise prior introductions, and doing so would constitute a "structural interaction" between features (i.e., the advice of **`F4`** modifies the introductions of **`F1`**). Of course, a feature can reference any member that was defined prior to that feature's addition to a program. However, it would seem that the number of *structural interactions* grows as the square of the number of features (i.e., every subsequent feature can advise all prior features). Our theory admits this possibility. But in practice, our experience is different.

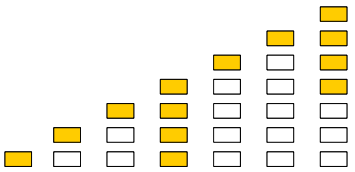


Fig 8. Feature Interactions

We have observed that the contents of most features is predominantly (95%) introductions and very few pieces of advice are used in feature-based program synthesis [6]. So, although composing features might potentially have many structural interactions as Figure 8 suggests, most interactions (i.e., the white boxes in Figure 8) are vacuous (empty). That is, features advise the introductions of a few specific features, and rarely advise introductions from all features. Of course, different case studies may reveal different statistics, but the pattern of using a few and directed pieces of advice has been observed by others even when the full range of advice in aspect languages is available [4]. The reason is that as collaborations grow in size, the dominance of introductions becomes overwhelming and the number of structural interactions tends to shrink because 'dynamic' rather than 'structural' refinements are preferred. For example, Eclipse plug-ins are features (i.e., increments in Eclipse functionality). Such plug-ins are *all* introductions, and have *no* structural interactions with the Eclipse code base. (More accurately, the introductions that plug-ins make do not modify existing classes, but instead are new classes, introductions that AspectJ cannot express). In contrast, as features shrink in size, the need for structural interactions seems to increase.

5 Topics of the ADI 2006 Workshop

Several topics were proposed for discussion at the ADI 2006 Workshop. In this section, we briefly explain how some of these topics are addressed in the context of feature-based program synthesis.

Mutual Exclusion and Dependencies among Features. It is very common that two features cannot both be present in a program or that using one feature implies the use of another. We capture these dependencies in a product line model (e.g., a context-sensitive grammar like that in Figure 1a). For example, the mutual exclusion of features **`A`** and **`B`** could be expressed in the grammar itself:

```
C : A | B ;          // choose only 1
```

Or as a contextual constraint:

```
A ∨ B ⇒ (¬A∧B) ∨ (A∧¬B)    // at most one of A,B is true
```

Known Interactions and Domain-Specific Languages (DSLs). Security features are known to impact performance. Should DSLs be used to express this interaction? The way we handle this is to recognize that programs and features have multiple representations. Suppose a program has two representations: code (to implement it), and a performance model (to estimate the program's performance). When a feature is composed with the program, both program representations may be updated. In the case of a security feature, both the program code and performance model are updated. The updated model allows us to assess the performance impact of using that security feature. The same applies to non-security features. We note that this is a standard paradigm not only for feature-based program synthesis, but also its predecessor: relational query optimization.[6]

Ordering of Aspects. Is the "`precedence`" concept sufficient? To us, aspects tell us how the existing structure of a program changes when a feature is added. The order in which features are composed determines the application order of aspect (advice).

Order of Advice Execution. The existing rules in AspectJ for ordering advice within an aspect are complex, and to specify a particular ordering can be difficult. We advocate simpler rules: the order in which advice is listed in an aspect is the order in which it is applied [21]. This makes advice execution order easy to specify and understand.

Formal Representations and Analyses. We use a simple algebra to express program structure and model structural interactions (advising). We also use elementary laws (e.g., advice distributes over introduction sum) to reason about program designs, and do so in a way that is independent of their representation. That is, our metaexpressions could represent Java code, or grammars, or HTML. This is consistent with prior work on program generation [27], and is what would be expected of a structural model for programs, i.e., the meaning of particular terms is uninterpreted and all interpretations (i.e., program representations) follow the same rules.

When to Recognize Interactions. Program synthesis is the inverse of refactoring. Given a feature metaexpression `A(B(C))`, we can synthesize a program `P`. To give a legacy program `P` a feature-based design requires us to map the structure of `P` to a feature metaexpression `A(B(C))`. What is interesting here is that we can understand interactions of features in two different ways. We can recognize feature interactions early in product line design or we can discover interactions late (after a program has been implemented) during a refactoring process. In either case, we create a product line model (i.e., context sensitive grammar) to define legal uses of a feature and explicitly state each feature's dependencies (in terms of the presence, absence, and ordering of other features).

6. A retrieval program is specified declaratively as an SQL statement. A parser maps this statement to a relational algebra expression, i.e., a feature metaexpression. An optimizer rewrites this expression, and derives a performance model for the expression by composing the performance model of each operation in the expression. In general, a relational operation encapsulates two different representations: code to implement it and a performance model [27].

Interaction Detection. We classify interactions among features `F1` and `F2` in two ways: does `F2` reference an introduction of `F1`? And does `F2` advise an introduction of `F1`? There is considerable available compiler support for detecting reference interactions. In contrast, tool support for structural interactions is lacking. Here's a motivating example. Consider legacy class `C` in Figure 9a. We want to decompose this class into a composition of features `F2` and `F1`, shown in Figure 9b. To know whether a decomposition is correct, we have to prove `C=F2(F1)`. That is, we have to prove that the code of `C` is equivalent to the code that is produced by `F2(F1)`. The notion of equivalence we use is called *source equivalence*: differences in white space are ignored, and the order in which members are listed in a class can vary (as it does not matter to the Java compiler)[7]. Tool support to verify a refactoring of a legacy application into features is something that is currently lacking. Hopefully the idea is clear.

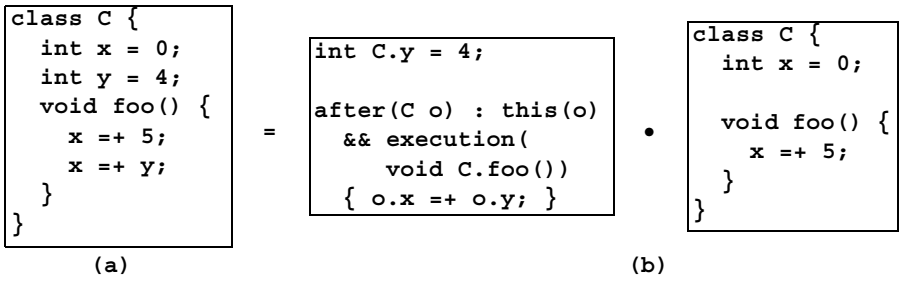


Fig 9. Equalities in Legacy Refactoring

It is worth noting that others in the AOP community have taken a similar stand: Cole and Borba have presented aspect refactorings as identities much in the same way that we describe here [15][2]. We anticipate more such work in the future, as guarantees for structural and behavioral equivalence between unfactored and feature-refactored code will become increasingly important.

6 Conclusions

Science has benefited when structure is imposed on problems. Structure gives us a greater understanding and insight into the workings of Nature. It is our belief that understanding the structure of software will be the key to automating many of tasks that we have to perform manually today and a simple mathematical representation of program structure will be essential.

The history of imperative programming languages and software design has been a progression of increasingly more abstract concepts of modular structures: from unstructured programs of the 1960s, to structured programs of the 1970s, to object-oriented programs of the 1980s, to component-based programs of the 1990s, and now to features and aspects.

7. Aspect compilers generally do not transform source code, and a different notion of program equivalence may be needed.

Feature orientation imposes a simple structure on the universe of programs and program representations. Features are the semantic building blocks of programs, and program structure is (largely) defined and manipulated through the use of two simple operations, introduction sum (+) and advice (•), on primitive program elements. We have argued that giving program structure a mathematical description simplifies program design, controls artificial complexity, helps understand feature interactions, permits reasoning about programs algebraically, supports program analyses, and automates rote tasks of software development. More importantly, it allows us to think in broader terms: we don't have to restrict our ideas about program design only in terms of code; we can now think in more abstract ways that can be expressed using many different program representations, and our tools will be based on sound and simple mathematical concepts.

Acknowledgements. I thank the workshop organizers for inviting me to speak at ADI 2006. I also thank Ruzanna Chitchyan, Sven Apel, and Chris Lengauer for their helpful advice. This research is sponsored by NSF's Science of Design Project #CCF-0438786.

7 References

[1] AHEAD Tool Suite. **`www.cs.utexas.edu/users/schwartz/ATS.html`**

[2] V. Alves, P. Matos Jr., L. Cole, P. Borba, G. Ramalho. "Extracting and Evolving Mobile Games Product Lines", *SPLC 2005*.

[3] American Standard. **`www.americanstandard-us.com/planDesign/`**

[4] S. Apel, T. Leich, and G. Saake. "Aspectual Mixin Layers: Aspects and Features in Concert", *ICSE 2006*.

[5] S. Apel, T. Leich, M. Rosenmüller, and G. Saake. "FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming", *GPCE 2005*.

[6] S. Apel and D. Batory. "When to Use Features and Aspects? A Case Study", *GPCE 2006*.

[7] D. Batory, G. Chen, E. Robertson, and T. Wang. "Design Wizards and Visual Programming Environments for GenVoca Generators", *IEEE TSE*, May 2000.

[8] D. Batory, J. Liu, and J.N. Sarvela. "Refinements and Multi-Dimensional Separation of Concerns", *ACM SIGSOFT 2003*.

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

[10] D. Batory. "Feature Models, Grammars, and Propositional Formulas", *SPLC 2005*.

[11] D. Batory and S. Thaker. "Towards Safe Composition of Product Lines", submitted 2006.

[12] BMW. **`www.bmwusa.com`**

[13] S. Chiba, "Program Transformation with Reflection and Aspect-Oriented Programming", in *Generative and Transformational Techniques in Software Engineering*, LNCS 4143, 2006.

[14] M. Calder, M. Kolberg, E.H. Magill, and S. Reiff-Marganiec, "Feature Interaction: A critical Review and Considered Forecast". *Computer Networks* 41(1), 2003.

[15] L. Cole and P. Borba. "Deriving Refactorings for AspectJ", *AOSD 2006*.

[16] A. Colyer, A. Rashid, and G. Blair. "On the Separation of Concerns in Program Families". Computing Department, Lancaster University, UK, TR COMP-001-2004.

[17] Dell web site, **`www.dell.com`**

[18] O. Diaz, S. Trujillo, and F.I. Anfurrutia. “Supporting Production Strategies as Refinements of the Production Process”, *SPLC 2005*.

[19] Gateway web site, **`www.gateway.com`**

[20] R.E. Lopez-Herrejon and D. Batory. “A Standard Problem for Evaluating Product-Line Methodologies”, *Generative and Component Software Engineering (GCSE)* 2001.

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

[22] E. Hilsdale and J. Hugunin. “Advice Weaving in AspectJ”, *AOSD 2004*.

[23] G. Kiczales, “It’s Not Metaprogramming”, *Dr. Dobbs Portal*, October 11, 2004. `www.ddj.com/dept/architect/184415220`

[24] C. Prehofer. “Feature-Oriented Programming: A New Way of Object Composition”, *Concurrency and Computation*, vol. 13, 2001.

[25] H. Rajan and K.J. Sullivan. “Classpects: Unifying Aspect- and Object-Oriented Programming”, *ICSE 2005*.

[26] T. Rho, and G. Kniesel. “Uniform Genericity for Aspect Languages”, Technical Report IAI-TR-2004-4, Computer Science Department III, University of Bonn. ISSN 0944-8535. December 2004.

[27] P. Selinger, M.M. Astrahan, D.D. Chamberlin, R.A. Lorie, and T.G. Price. “Access Path Selection in a Relational Database System”, *ACM SIGMOD 1979*.

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

[29] D. Smith. “A Generative Approach to Aspect-Oriented Programming”, *GPCE 2004*.

[30] R.E.K. Stirewalt and L.K. Dillon. “A Component-Based Approach to Building Formal Analysis Tools”, *ICSE 2001*.

[31] J. Sztipanovits and G. Karsai. “Generative Programming for Embedded Systems”, *GPCE 2002*.

[32] M. VanHilst and D. Notkin. “Using Role Components to Implement Collaboration-Based Designs”. *OOPSLA 1996*.

[33] J. Watson. *The Double Helix: A Personal Account of the Discovery of the Structure of DNA*, Penguin/Putnam, 1968.

[34] E. Visser. “Stratego: A Language for Program Transformation Based on Rewriting Strategies”, *Lecture Notes in Computer Science, Vol. #2051*, 2001.

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

[36] C. Zhang, G. Gao, and H.-A. Jacobsen. “Towards Just-in-time Middleware Architectures”, *AOSD 2005*.