Semantic Reasoning about Feature Composition via Multiple Aspect-weavings

Christian Prehofer
Nokia Research Center, Helsinki,
firstname.lastname@nokia.com

Abstract
In this paper, we consider semantic refinement for feature-oriented programming where components are built from features and weavings, which we use to adapt one feature to the context of another one. We address the question of semantic reasoning about multiple weavings. If we know the effect of feature A on X and of feature B on X, what can we conclude about adding both A and B to X? For this, we define conservative weavings which do not modify the state of another feature. We show that composition of several such weavings is however not compositional as it does not preserve semantics. In particular, weavings must consider that other weavings have already been applied. This explains why it is considerably more difficult to reason about multiple aspect weavings. We show criteria on the dependencies between weavings which allow modular, semantics-preserving application of weavings. This is formalized in a calculus for feature composition and also extended to conditional refinements.

ACM Categories & Descriptors

General Terms Design, Languages, Theory, Verification.

Keywords Feature composition, aspect weaving, feature interaction, semantic reasoning and refinement, modular reasoning, feature-oriented programming, aspect-oriented programming.

1. Introduction
Designing software systems by features and aspects has gained considerable attention in the last years. Features are a central concept in software product line research [9] and both aspect-oriented [10] and feature-oriented [3][5][7] programming concepts have been explored.

The motivation of this paper is to consider semantic refinement and composition of features or aspects. We assume that software components are composed from a set of desired features in a plug and play fashion. Features are added by extending the functionality of a component and by adapting the existing functionality by method overwriting or aspect-weaving.

Feature-oriented design aims at the separation of features and the dependencies between them. A main observation is that features have to be adapted in the presence of others. For instance, a base feature like a stack can be extended by a counter or logging features. In both cases, the code of the stack has to be extended by weavings.

We speak of a base feature and its behaviour, which is modified by adding other features. Adding features will add additional classes and variables, but also weavings, which adapt other features. These are as weavings in aspect-oriented programming (AOP) [10].

Most approaches and formalisms for aspect and feature-oriented programming only consider syntactic composition of software in the form of source code. On the other hand, modifying classes by weavings often makes reasoning about the actual semantic behaviour of a piece of software hard [11][12]. We present an approach for semantic refinement and in particular address the following questions:

1. If we add a feature B to a feature A by weaving in code, will feature A still behave in the same, originally intended way? If it does not behave the same, can we specify under what conditions feature A still behaves in the original way? E.g., if feature B is not used, will it still behave the same way?

2. If we know the effect of feature A on X and of feature B on X, what can we conclude about weaving both A and B to X? More formally, if A(X) adds A to X and if we know how both A(X) and B(X) behave, what do we know about A(X) and B(X)?

We formalize these questions and present a new approach to reason modularly about composed components. In practice, many features are quite independent of other features as they do not modify their state. These can be added in a modular fashion without changing other features. To formalize this, we use a concept of conservative weavings [6], which is similar to observer aspects in [12] and to harmless advice in [16].

We show that composition of several “harmless” conservative weavings is however not compositional, i.e. the 2nd question above does not hold in this case. This means that features A and B cannot be composed without looking at the independence of the features A and B wrt X. Even if both A and B do not interact with X, undesired interactions can occur when adding all three features.

The main result of this paper is a set of criteria for features and weavings which allows modular reasoning and refinement about such compositions of several features. While AOP weavings are designed to adapt some specific code, we must also consider adapting code which has already been modified by adding other weavings. Only with this generalization, we can get true compositionality of several features and their weavings. We clarify what interactions can occur and what must be observed when developing feature-oriented software systems. In addition, we extend these results to conditional refinement. This means that the composition properties hold only if some events, e.g., exceptions or error conditions, do not occur.
The results can be applied to AOP as well, with the main difference that the weavings are applied according to feature interactions in a static way. Dynamic application of weavings, which cannot be determined statically, are considerably more complicated and are out of scope of this paper.

In our analysis, we use a semantic view of features. This means that we consider feature interactions as a semantic property regarding the behaviour of a feature. Aspect weaving, in turn, is an implementation technology for resolving interactions by method adaptions. We focus here on the practical aspects of semantic refinement and do not elaborate formal semantics, which can for instance be found in [13][6]. Our reasoning about programs extends concepts which are formally treated in [6].

The paper is organized as follows. In the following subsection, we give a simple programming example which illustrates feature composition. In Section 2, we present a formalization of feature composition based on aspect weavings. Semantic composition of weavings and several features is considered in Section 3. The paper concludes after related work in Section 4.

1.1 Example Features & Weavings

We discuss the above questions by a simple example modelling stacks, similar to [2][5]. The code below provides base implementations of the individual features using stacks over characters with a simple implementation via strings. For each feature, we use a class with the feature name to define the variables and methods of this feature. We use Java-like notation as in [1][2].

```java
class Stack {
  String s = new String(); // Use Java Strings
  void empty() {s = "";}  
  void push(char a) {s = String.valueOf(a).concat(s);} 
  void pop() {s = s.substring(1);} 
  char top() {return (s.charAt(0));} 
}

class Counter {
  int i = 0;
  void reset() {i = 0; }
  void inc() {i = i+1;}
  void dec() {i = i-1;}
  int size() {return i;}
}

class Lock {
  boolean locked = false;
  void lock() {locked = true;}
  void unlock() {locked = false;}
  boolean is_unlocked() {return !locked;}
}
```

In addition to the base feature implementations, we need to provide the glue to put the features together. These adaptors, also called lifters in [5][7], are weaved into the feature code when features are composed. The adaptors below refine one feature to the context of another feature. We use the construct refine X to Y for adapting the methods of feature X to feature Y. We allow method redefinitions which can implement before, after or around advice in AOP terminology. Compared to AOP, we use super.f() instead of the proceed construct.

```java
refine Stack to Counter{
  void empty() {reset(); super.empty();}
  void push(char a) {inc(); super.push(a);} 
  void pop() {dec(); super.pop();}
}

refine Counter to Lock{
  void reset() {if (is_unlocked()) super.reset();}
  void inc() {if (is_unlocked()) super.inc();}
  void dec() {if (is_unlocked()) super.dec();}
}

refine Stack to Lock{
  void empty() {if (is_unlocked()) {super.empty();}}
  void push(char a) {if (is_unlocked()) {super.push(a);}} 
  void pop() {if (is_unlocked()) {super.pop();}}
}
```

With the above, we can now generate code for customized components by feature combinations such as

```java
c = new Counter ( Stack) 
```

When adding Counter to Stack, the adaptor of Stack to Counter will be weaved into the stack code. This proceeds recursively when combining several features, as illustrated for this specific example in Figure 1. In Figure 1, the Stack feature is adapted by two weavings, first for adding the counter, then for adding the lock feature. In turn, both features are adapted when the lock is added.

```
Figure 1: Composition of Stack Features
```

In this example, we can see that adding the counter to the stack just adds extra functionality and does not modify the original behavior. Hence a stack with counter will behave as any stack without a counter. This is what we call semantic refinement. On the other hand, adding the lock feature can modify the behavior, but only if the lock feature is used. If we know how counter and lock modify the stack, we would like to reason about the behavior of the combination of all three features.

2. Feature Interactions and Composition

In the following, we introduce a notation for features and then present our concepts of semantic refinement.

2.1 Adaptors and Derivatives

In the following, we introduce a more abstract notation for features and adaptors, which we use later to reason about composition of features.

For each feature H, we have a corresponding base implementation module h, denoted lower case, which adds a class with variables and methods. We often identify the feature (module) with this class, as it presents the base functionality of a feature.

An adaptor of a feature X to a feature Y adapts the methods of feature X to the context of feature Y. As in AOP or other object-oriented inheritance concepts, we use before and after advice to adapt a specific method. We can describe this schematically as
refine X to Y { 
  x(p1, ..., pn) { 
    p1', ..., pn' = before_advice(p1, ..., pn); 
    super.x(p1', ..., pn'); 
    after_advice(); } }

Note that the above before advice represents a is not a function call – it is a schema for a block of code which depends on p1 and modifies the parameters of the super.x() call. The before and after advice can use variables and methods of both features, here X and Y. It can also call the original x1 implementation via super.x1, and also with different parameters, but it can do so only once. In object-oriented languages, it may even call super.x1 several times, but we do not consider this as it makes any modular reasoning very hard. The code may have several calls to super.x1 – it is just important that it is called at most once as make explicit in the above form. Also, super.x1 may not be called at all if the before advice terminates the methods, e.g. by a return statement. As in [15], we assume that the before and after advice is not used in recursive or other function calls in super.x(p1', ..., pn').

The above format deviates from common AOP and other concrete languages including the examples above. It is a more functional view which captures the essential method adaptation and is suitable for our analysis later. Implementations can be transformed into this format without changing semantics, as long as the weaving can be determined statically.

### 2.2 Notation for Feature Composition

We use an algebraic formalism developed in [1][2][4] for the syntactic description composition of weavings and features in AOP and FOP. Note that this is not an algebraic semantics, it is used to formalize the feature composition. The semantic view will be introduced in Section 3 based on this notation.

For adaptors we use the notation of derivatives as in [1][2]. An adaptor of h to a feature G is denoted as δh / δG, and refines the class module h to feature G. The advantage of this notation is that we can nicely calculate with derivatives as in the above example [1][2], which extends the concepts of [5][7].

We define the sum of two modules h + f as the aggregation of two classes, which is the syntactic combination of disjoint pieces of code.

The second main operation is *, denoting function composition or weaving of two derivatives or a derivative on a module. For instance, p * h weaves an adaptor p to the module h.

We define [H] as the implementation module, or code, of a feature H or a feature expression if H = J(G). With this notation, we can specify several equations and then derive feature composition rules as in [5].

A basic property is distributivity

\[ d * (a + b) = d * a + d * b \]

Notice that + is symmetric while * is not. Next, consider how derivatives apply to modules. We have

\[ \delta b / \delta H * a = a \]

for two different base modules of two features.

By abstracting the module to be modified, we can define \( \delta / \delta F \) as the derivative of F for any other module, which can be applied to a module m denoted as \( \delta m / \delta F \) or \( \delta / \delta F \) m.

Derivatives distribute over sum + and weaving * in the following way [1]:

\[
\delta / \delta F (a + b) = \delta a / \delta F + \delta b / \delta F \\
\delta / \delta F (a * b) = \delta a / \delta F * \delta b / \delta F 
\]

The first equation expresses that computing the differential of \( a + b \) can be done by composing two differentials.

Based on the above equations, we can define the composition of two features with the differential as follows:

\[
\begin{align*}
[H(J)] &= j + \delta h / \delta J * h + (\delta^2 b / \delta J \delta H) * \delta b / \delta H * b 
\end{align*}
\]

Here f can be primitive, i.e. a feature or a feature expression. The differential notation can also be used to describe interactions between several features [7], which cannot be expressed by two-feature interactions. We denote this by a second-order differential

\[
\delta^2 x / \delta H \delta G
\]

which is the differential of x wrt both features H and G. Such a second-order derivative is typically needed if the adaptation of x to the features H and G cannot be expressed by two features at a time, i.e. \( \delta x / \delta G \) and \( \delta x / \delta H \) are not sufficient. This is also called a true 3 feature interaction [7], which cannot be reduced to binary interactions between two features. For an example, we refer to Section 3.2.

Based on the above, we can derive a rule for the composition of several features [2]

\[
[J(H(B))] = j + \delta h / \delta J * h + (\delta^2 b / \delta J \delta H) * \delta b / \delta H * b 
\] (1)

This means that module b is first modified by two derivatives of J and H, and then by the combined derivative of J and H, called second-order derivative. This formally shows that for nested feature expressions, composition must take into account the interactions between several features. We can now illustrate this equation in Figure 2, which formalizes the example in Figure 1:

\[
\begin{array}{c|c|c}
\text{j} & + & \\
\hline
\delta h / \delta J & * & h \\
\hline
(\delta^2 b / \delta J \delta H) * \delta b / \delta H & * & h
\end{array}
\]

**Figure 2: Feature Composition with Derivatives**

### 3. Semantic Refinement and Weavings

While the above formalizes the syntactic construction of features, we now develop formulae for semantic properties. The main new idea is that we use semantic equality instead of syntactic equality (on source code) as done above. This equality is used in following sections to formalize semantic properties of feature composition.

We consider the behavior of a single component which is composed from several features. For a specific instance of a component with feature B, we define semantic equality over the state of feature B of two expressions as follows:

\[
expression_1 =_B expression_2
\]

holds if the state of B (variables) is identical after execution if it has been before execution of both expressions. Note that we leave the actual component implicit, as we focus on refining a single
component. More precisely, for all initial variable assignments, 
\( \text{expression}_1 =_B \text{expression}_2 \) holds if for any variable \( x \) defined in feature \( B \), \( x \) is identical after both executions, which we denote as

\[ \text{Run } \{ \text{expression}_1 \} =_x \text{Run } \{ \text{expression}_2 \} . \]

We can apply this notation for two modules which define the same methods, e.g., \( H(B) =_B B \) holds if the equality holds for all methods \( h(p_1, \ldots, p_n) \) defined in \( B \) for any parameters \( p_i \).

We can also extend the notation for derivatives, if we consider the method redefinitions as methods where the \( \text{super}.x() \) call is considered as the null operation. E.g., \( \delta b/\delta f =_B \delta b/\delta H \). Also, the notation naturally extends to equality on several features such as \( H(B) =_B H \).

This equality is interesting to analyze the effect of changing a module by weaving code. We use this to formalize semantic refinement, which means that newly added code does not modify the behavior of a feature on its own state. It does not capture external effects of newly added code. This is suitable for our case, as the woven code intentionally modifies other, external state or adds extra output events to resolve feature interactions.

### 3.1 Conservative Weavings

With the above notation, we can now introduce an important class of weavings, which add additional behaviour but do not modify the original features. These are called conservative, as originally introduced in a functional language model in [6]. The concept formalizes the notion of observer in [12] and is similar to harmless advice in [16].

A weaving \( D \) is called conservative wrt a feature \( F \) if

\[ D * F =_F F \]

This means that for all methods \( f(p_1, \ldots, p_n) \) of \( F \), the following equation holds:

\[ D * f(p_1, \ldots, p_n) =_F f(p_1, \ldots, p_n) . \]

With conservative weavings, the effect on the internal state of a feature by the original implementation is unchanged. As they preserve the original behaviour, they establish semantic refinement by definition.

Conservative weavings are an important case of an adaptation of a method which just adds extra functionality, without changing the original behavior. Many typical examples fall into this class, such as the counter feature above, logging, debugging code or I/O operations. The adaptation of the stack to the lock feature above is however not a conservative weaving. In case the lock is active, no changes are permitted, which modifies the original behaviour. We will cover such cases in the section below on conditional refinement.

There are simple, syntactic criteria for conservative weavings of a method \( f \) wrt a feature \( X \): The weavings must both
- call \( \text{super}.f() \) exactly once with original parameters
- not modify variables of \( X \) directly or indirectly

We think that this class of weavings covers many practical cases of weavings which can be added without a deeper (semantic) analysis of the resulting code in each specific case. This is needed in the general case of aspect weavings, as weavings can modify the behavior [11]. For this reason, we take this as the basis for reasoning about the composition of several features.

The notion of equality does capture changes in state but not the external behavior such as any output events. From a theoretical point of view, these can also be recorded as internal state and then compared, but this is not a way for the practical analysis of programs. For such input/output oriented systems, refinement based on a stream semantic of external behavior of reactive systems is suitable and similar refinements are discussed in [8].

### 3.2 Semantic Composition of Weavings

In the following, we look at the composition of several derivatives. We would like to infer when \( H(G(F)) \) is a conservative extension of \( X \), which is equivalent to a semantics preserving refinement. We have to consider the composition of several weavings, which will turn out to be considerably more complex than the above case.

We first show that the notion of conservative extension is not sufficient in this case. Consider for instance the following simplified example:

```java
class X {
    int a = 0;
    void m() { a++; }
}
class G {
    int g = 0;
}
class F {
    int back;
}
refines X to G {
    void m() { g++; super.m(); }
}
refines X to F uses G {
    void m() {
        back = g;
        super.m();
        if (g != back) {a++;}}
}
```

We have here two adaptors, which can be woven into the class \( X \) to adapt to features \( G \) and \( F \), respectively. Here, \( \text{refines } X \) to \( F \) uses \( G \) makes it explicit that the adaptor refers to variables of \( G \). In this example, the variable \( g \) is read, but not modified by this adaptor.

First observe that the two refinements of \( X \) to \( G \) and \( F \) are both conservative over \( X \). Also, \( G \) and \( F \) are independent and the weaving from \( X \) to \( F \) does not modify variables of \( G \). However, their composition, \( \delta X / \delta F \) is not conservative wrt \( X \), as the last adaptor checks if both features \( X \) and \( G \) are present and only then modifies variable \( a \). While the above example is simplified to show the effect more clearly, such a dependency can also occur in many others ways, e.g., via some other variable which is external to \( F \), \( G \) and \( X \).

This shows that feature interactions and aspect weavings cannot be considered just in isolation, as their dependencies have to be considered as well. This holds for both normal and second-order derivatives.

The main goal now is to establish a conservative extension of a combination of two derivatives, i.e. the following equation:

\[ \delta x / \delta F \delta x / \delta G X =_X X \]

First, we assume that both derivatives are conservative, i.e. \( \delta x / \delta F \) \( X =_X X \) and \( \delta x / \delta G \) \( X =_X X \). For the product of two conservative derivatives to be conservative, we have to show that the two derivatives are independent. This is formalized next.

For a set of variables \( V = v_1, \ldots, v_n \), we define the random \( V \)-extension of a method \( x() \) as \( x() \) followed by an assignment \( v_j = r_j \), where \( r_1, \ldots, r_n \) are random values of suitable type. This means that the body of \( x() \) is extended like in an after advice by an
We need to apply the above result first to the derivatives of the conservative extension of a feature. Note that we do not formalize further which variables are modified by D'. This typically includes the state of the features concerned but also possible side effects. These are often difficult to capture, but are often the reason for conflicts.

This definition captures the intuition that a derivative may not just modify the base feature, but can also modify a feature which has already been extended by another feature and hence has been adapted by some derivatives. For instance, the two derivatives in the example in Section 3.2 are not independent.

We can now establish the main result when the composition of weavings is conservative.

**Theorem:** A composition D * D' of two derivatives is a conservative extension of a feature X if the following holds:

- D and D' are conservative over X
- D is independent of derivative D' over X

**Proof:** Consider the application of the two derivatives to a method x_i() of X in a schematic way:

```latex
x_i(p_1, ..., p_n) {
    p_1', ..., p_n' = before_advice_D(p_1, ..., p_n);
    p_1'', ..., p_n'' = before_advice_D'(p_1', ..., p_n');
    super.x(p_1', ..., p_n');
    after_advice_D();
    after_advice_D();
}
```

We need to show that after this code execution, the variables of X are the same when just x_i() is applied. The first before_advice_D(p_1, ..., p_n) can modify variables which affect D', including the parameters p_i. Also, D can modify the state of X to X_0 (even though it is conservative over X).

Now, consider the inner code of the x_i(...) call with the D' before and D' after advice. As D' is conservative over X for any initial variable settings (here X_0), the variables of X will be same as x_i(...) executed with state X_0. Also, the variables affected by D' may be modified. This inner code is hence equivalent to a random V-extension of x_i(...), on the variables V modified by D', as D' is conservative over X. Note that we do not allow recursive calls of x_i(...) which are modified by D, hence D advice does not affect this inner code. As we can show the above for all methods x_i() of X and since D is independent of derivative D' over X, the results follows. □

### 3.3 Semantic Composition of Features

We now discuss the semantic composition of several features as shown in equation (1). Following equation (1), we need to show

\[ j + \delta h/\delta J \ast h + (\delta^2 h/\delta J H) \ast \delta h/\delta J \ast \delta h/\delta H \ast b = y \]

Assuming that j and \delta h/\delta J \ast h do not affect b, we can reduce this to

\[ (\delta^2 h/\delta J H) \ast \delta h/\delta J \ast \delta h/\delta H \ast h = y \]

We need to apply the above result first to the derivatives \delta h/\delta J and \delta h/\delta H, then it can be applied to the derivatives \((\delta^2 h/\delta J H)\) and \(\delta h/\delta J \ast \delta h/\delta H\).

In practice, we think that second-order derivatives are not frequent and only a single application of the theorem is needed. In the same way we can also now proceed to the combination of more than three features.

### 3.4 Conditional Refinement

The above results are valid for a large class of features which do not modify the behavior of the inner features at all. There is however also a large class of features which only modify the behavior in some exceptional cases. Hence, if these cases do not occur, the behavior is as specified originally.

Typical examples are as follows:

- The lock feature above can lock the functionality of the stack feature. If unlocked, the behavior of the stack feature is unchanged.
- An error handling feature for the stack feature can avoid run time exceptions when a pop operation is attempted on an empty stack.

For such cases, we can formalize conditional refinement and establish the results as far, with the difference that we need to maintain global constraints. These constraints may for instance state that the lock is always unlocked.

These conditions can be formalized and added for the above formal treatment in a canonical way as shown in [6][7]. The conditions of all equations considered in the prerequisites must be globally considered and must be conjointed.

For instance, we can formalize that the lock feature is a conditional, conservative extension as follows:

\[ \text{locked} = \text{false} \rightarrow \delta \text{Lock Stack} = \text{Stack} \]

Combining the results of this and the last section, we can now show that the two features are a conservative, conditional extension of the stack feature, i.e.

\[ \text{locked} = \text{false} \rightarrow \text{Lock}(\text{Counter(Stack)}) = \text{Stack}. \]

### 4. Related Work

There has been a recent discussion on modularity of reasoning in aspect-oriented programming [11][12][15]. In particular, [11] argues that the full code of aspect weavings must be available to reason modularly about code. The main focus is reasoning informally about how local code changes affect the overall program. It is shown that AOP in some cases simplifies the reasoning, as code is modularized in aspects. This differs from our approach where we want to reason about the change which features (or aspects) do have on a component.

Related work has considered the problem of encapsulation for aspect-oriented programming, which means that implementations can be changed independently of the additional code which is weaved in later. The work in [15] on TinyAspects is concerned with the impact of an aspect weaving on the code and the encapsulation principle. This means that code of a module can be changed internally without affecting the external interfaces. It shows that normal AOP does not have this property and introduces a restricted language which has full encapsulation, but limits method modifications as in our case. In our work, the focus is on criteria for independence of aspects and features which enables the composition of more than two weavings.

The work in [12] addressed the modularity problem of specifications in aspect-oriented programming. It introduces observers and assistants, which are two classes of aspects. Observers
are similar to our conservative weaving, but are not formalized in
[12]. A similar concept of harmless advice was formalized in [16]
by using a lambda calculus with a dedicated type system to
formalize that aspects do not change the code they are applied to.
The concepts developed do not focus on the interference between
aspect weavings. The notion of non-interference considered in [16]
may be useful to formalize the concepts of our research.

Our work uses a sophisticated formalism to describe features and
their dependencies presented in [1] [2], which however does not
consider semantic properties. This work generalized the original
concept presented in terms of features and lifters in [5] [7], which
introduced feature-oriented programming. A main difference to [5]
[7] is the explicit model of second-order derivatives or three-feature
interactions [7], which were excluded purposely in [5]. In [6], also
specification concepts were presented considering conservative
extensions, but not focusing on the semantic interaction of several
refinements.

Another line of related work is on semantic subtyping [14], which
does however not consider the flexible and modular weavings,
albeit the semantic preservation concepts are similar.

5. Conclusions

In this paper, we have looked at the semantic interaction of several
aspect weavings. Most of the work on semantic aspect weavings
considers adding a weaving to some code. This is not sufficient for
composition of multiple weavings. We have shown that the intuitive
notion of conservative weavings, which do not modify the state of
other features, is not modular. This shows that reasoning about
several weavings is considerably more complicated than just
looking at one weaving. This is due to possible semantic
interactions between the weavings. We must also consider that the
code has been modified by other weavings before a weaving is
added. Only if a weaving preserves semantics with this assumption,
we can compose several weavings.

We have formalized the semantic composition and refinement of
several features by multiple aspect weavings with an algebraic
calculus for (syntactic) feature composition. We establish semantic
refinement of several conservative weavings under some specific
non-interference conditions. This gives practical criteria for
composing weavings and illustrates where unforeseen interactions
between features may occur. We also extend this work to
conditional semantic refinement. Our setting is restricted to static
weavings and also limits weavings to externally called functions. On
the other hand, we address realistic languages and develop simple,
practical criteria for composition.

Acknowledgements. The author is grateful for the careful and
detailed comments from the reviewers and George Heineman which
significantly helped to improve the presentation of this paper.

6. References

Refactoring of Legacy Applications", International
Conference on Software Engineering 2006, Shanghai, China.

Feature Oriented Designs", International Conference on
Feature Interactions (ICFI), June 2005.

Tool Suite", International Conference on Software

Disciplined Approach to Aspect Composition", ACM
SIGPLAN 2006 Symposium on Partial Evaluation and
Program Manipulation (PEPM '06), Charleston, SC, January
2006.

Objects, ECOOP '97 -- Object-Oriented Programming,

A Feature-Oriented Approach, Habilitation Thesis, Technical
University of Munich, 1999.

Object Composition, Concurrency and Computation. Vol. 13,

[8] C. Prehofer, Plug-and-Play Composition of Features and
Feature Interactions with Statechart Diagrams, Journal on

and evolving a product-line approach, ACM Press/Addison-
Wesley, New York, NY, 2000


modular reasoning, ICSE'05

Reasoning, and the Behavioral Subtyping Analogy". SPLAT
2003.


[14] G. Castagna, A. Frisch, A gentle introduction to semantic
subtyping, International Conference on Principles and
Practice of Declarative Programming, Lisbon, Portugal 2005

[15] J. Aldrich, Open Modules: Modular Reasoning about Advice,
European Conference on Object-Oriented Programming,
2005

SIGPLAN Notices, 2006