Interference Analysis for AspectJ

Maximilian Störzer, Jens Krinke
Universität Passau
Passau, Germany
{stoerzer, krinke}@fmi.uni-passau.de

March 1, 2003

Abstract

AspectJ is a language implementing aspect-oriented programming on top of Java. Besides modification of program flow and state using *advice*, AspectJ offers language elements to statically modify existing classes by changing their position in the inheritance hierarchy or introducing new members. This can lead to binding interference, i.e. the dynamic lookup of method calls not affected directly by the aspect might change.

This paper presents methods allowing programmers to automatically check the impact of introductions and hierarchy modifications on existing programs.

1 Motivation

Aspect oriented programming (AOP) is a new paradigm in programming, extending traditional programming techniques, first introduced in [5]. Its basic idea is to encapsulate concerns which influence many modules of a given software system, so called *crosscutting concerns*, in a new module called *aspect*.

This encapsulation improves separation of concerns and can avoid invasive changes of a program if crosscutting concerns are affected by system evolution. The functionality defined in the aspect is *woven* into the base system with a so called *aspect weaver*, at compile time, load time, or even run time of the program. Here *AspectJ*—an aspect-oriented language extending Java—is considered. Main features of AspectJ are introduction, modification of class hierarchies and advice. This paper will concentrate on the first two points which are designed to statically change a given system by introducing new members in classes or modifying the structure of an inheritance hierarchy.

AOP is a very powerful technique but includes new risks, too. Changes introduced with AspectJ are not visible *directly* in the source code of the base system. Aspects are a new modularization unit usually stored in separate files. The effect of this code can influence semantics of the whole system. Tool support is necessary to reveal the impact of aspect application. To motivate this necessity, this paper presents problems related to AspectJ language constructs which might be avoided by modifying the AspectJ language itself. However, impact on language design is not in the scope of this paper.

To achieve this support, methods to determine the impact of aspect application have to be developed. As a first step, a method to decide *if* an aspect modifies base system behavior is presented. This analysis will be extended to perform an impact analysis to show *where* system behavior is influenced by an aspect.

Throughout this paper, the simple class hierarchy defined by program 1.1 will be used as an example to demonstrate aspect influence. This hierarchy will be modified using introduction and hierarchy modification and some of the classes will be declared to implement interface `I`.

This paper describes the problem emerging from these transformations, presents an algorithm to detect their effects and suggests how this information can be used to reduce flaws in a software system. Organization is as follows: Each section takes a look at a AspectJ language construct, starting with interface introduction in section 2. Section 3 presents an algorithm to detect binding interference for class introduction, section 4 for hierarchy modification. Section 5 shows how these results can be used for impact analysis. Section 6 presents an example application of this analysis for a given hierarchy. Section 7 briefly summarizes the preliminary implementation and outlines future work. Sections 8 concludes and gives an overview of related work.

Program 1.1 Example Hierarchy

```
class A { void n() {
  print("A.n()"); }}
class B extends A {
  void m() { print("B.m()"); }}
class C extends B {
  public void x() { print("C.x()"); }}
class D extends B {
  public void y() { print("D.y()"); }
  public void x() { print("D.x()"); }}
class E extends C {}
class F extends D {
  void n() { print("F.n()"); }}
class G extends B {
  void n() { print("G.n()"); }}
interface I {
  void x(); void y();
}
```

2 Interface Introduction

Introduction is an AspectJ language construct to add new members to existing classes or interfaces. The purpose of interface introduction is to provide *default implementations* of interface methods which can be used to reduce necessary work for implementation. However, if no multiple inheritance is needed an abstract superclass can often be used instead .

Usage of this feature can result in 'forgotten' implementations which may introduce flaws into a program. The compiler no longer issues an error message if a class implements an interface but does not (re)define all default implementations. To avoid flaws by 'forgotten' redefinitions a compiler warning should be given when a class uses a default method implementation provided by the interface.

A simple analysis of interface introductions can provide the necessary information. Given a class hierarchy and an aspect A, an analysis could be performed in three steps:

1. The set of interfaces for which aspect A provides default implementations has to be determined by scanning A's introductions. Let I_{def} be the set of these interfaces. For $\mathtt{I} \in I_{def}$ let *methods*(I) be the set of methods for which default implementations are given.

2. The set of classes implementing an interface $\mathtt{I} \in I_{def}$ has to be identified. Let $\mathcal{C}_{I_{def}}$ be the set of these classes.

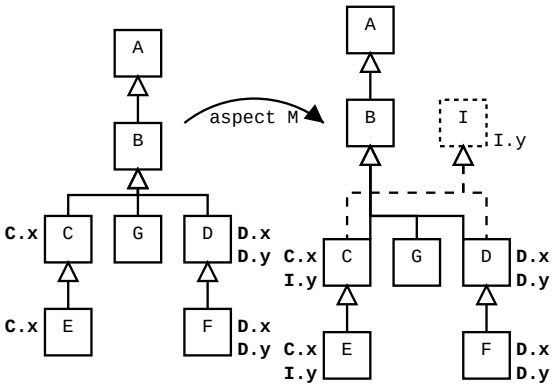

Figure 1: Using default implementations.

3. The set of classes $\mathcal{C}_{di}$ which do not provide an implementation of all interface methods (i.e. which use the default implementations) has to be determined. Let *methods*(C) be the set of all methods defined in Class C. Then

$$\mathcal{C}_{di} = \{\mathtt{C} \in \mathcal{C}_{I_{def}} \mid \exists \mathtt{I} \in I_{def} : \mathtt{C} \text{ implements } \mathtt{I} \\ \wedge \mathit{methods}(\mathtt{I}) - \mathit{methods}(\mathtt{C}) \neq \emptyset\}$$

It is sufficient to check weather all methods in *methods*(I) are implemented as other missing methods are detected by the java compiler. Note that any subclass of an affected class is influenced as well, unless it implements the necessary method and thus overrides the default implementation.

The programmer must examine affected classes to check whether the default implementation given by the interface is appropriate.

As an example consider aspect M given by program 2.1, which declares that classes C and D implement interface I and introduces a default implementation of method y to the interface.

Program 2.1 Adding interface implementation.

```
aspect M {
  declare parents: C implements I;
  declare parents: D implements I;

  public void I.y() { print("I.y()"); }
}
```

Figure 1 presents the effects of this modifications. Note that classes C and E—maybe unexpectedly—use the im-

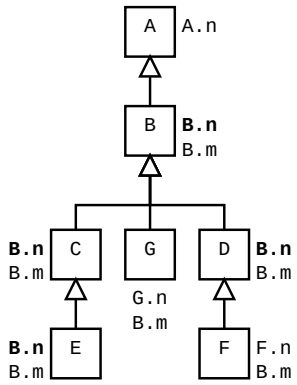

Figure 2: Example hierarchy, effects of introduction.

plementation given by `I.y`. This fact is reported by the proposed analysis.

3 Noninterference Criterion for AspectJ Introduction

In contrast to interface introduction, class introduction is more complex as program semantics may change without modifying any class directly. These effects are described in the following.

3.1 Impact of Class Introduction

Introducing members to classes can result in changes of dynamic lookup if the introduced method redefines a method of a superclass, called *dynamic interference* in [10]. However, as the term dynamic is misleading, the term *binding interference* is preferred. Consider the example hierarchy defined by program 1.1 and aspect `N` to be applied:

```
aspect N {
  void B.n() { print("B.n()"); }
}
```

This aspect introduces a method `n` to class `B`, which is already defined in superclass `A` of `B`. Any (virtual) call e.g. from class `C` now results in call of `B.n()` and not in `A.n()` as before. So, the semantics of a call to `n` has possibly changed for any object of class `B` or any subclass thereof without direct modification of these classes. Figure 2 indicates the changed lookups in bold.

The presented considerations abstract from Java access specifiers: All methods are considered `public`. Addition of access specifiers reduces the set of inherited methods (some might not be visible in the subclass), thus reducing binding interference.

If the introduced method `B.n()` redefines `A.n()` with respect to behavioral sub-typing [6], a (unknown) client of a subclass of `B` may still work as expected. However, neither Java nor AspectJ guarantees this kind of method redefinition. The described problem is a special case of the *fragile base class problem* [7]—subclasses change behavior because of changes in the superclass. Although tracking down bugs introduced by changing a base class is difficult, the problem is even worse with aspect languages as modifications of the base class are not visible if the code is viewed in isolation (i.e. without the applied aspect). To track bugs emerging from dynamic interference, impact analysis of aspect application should reveal method calls whose *dynamic lookup has changed*.

3.2 Detecting Semantical Changes

To detect semantical changes in the hierarchy, the interference criterion of [10]—informally stating that all virtual calls evaluate to the same target as before—is applied to aspects by reducing introduction to hierarchy composition. As a result, the correctness proof of the criterion can be applied to aspect introduction as well.

In contrast to Hyper/J, AspectJ is much more restrictive in the possible static modifications of the class hierarchy. Modification of system behavior is mainly achieved by using advice. However, introduction can be viewed as a hierarchy composition. Let a hierarchy $\mathcal{H}$ be defined as in [10]:

Definition 3.1 (Class Hierarchy) *A class hierarchy $\mathcal{H}$ is a set of classes and an inheritance relation: $\mathcal{H} = (\mathcal{C}, \leq)$. A class $\mathtt{C} \in \mathcal{H}$ has a name and contains a set of members. According to this definition, members$(\mathtt{C})$ does not contain inherited members that are declared in super-classes of* `C`.

To indicate the members of class `C` defined in hierarchy $\mathcal{H}$ we write *members*$_{\mathcal{H}}(\mathtt{C})$; $\mathtt{C}_{\mathcal{H}}$ references definition of class `C` in hierarchy $\mathcal{H}$.

Any AspectJ introduction can be viewed as a hierarchy composition by defining a new hierarchy induced by an aspect `A`.

Definition 3.2 (Hierarchy induced by Introduction) *Let $\mathcal{H} = (\mathcal{C}, \leq)$ be a hierarchy an aspect* `A` *is applied to. Let I be the set of introduction statements of this aspect. Elements of I have the form $(\mathtt{C}, \mathtt{m})$. $\mathtt{C} \in \mathcal{C}$ indicates the class where the new member* `m` *should be introduced to. Then:*

1. $\forall C \in \mathcal{H}$ *create a new empty class named* C*, add it to* $\mathcal{C}'$
2. $\forall(C,m) \in I$ *add member* m *to the corresponding class* $C \in \mathcal{C}'$ *created in (1)*
3. $(\leq') = (\leq)$ *(same inheritance relations as in* $\mathcal{H}$*)*

The hierarchy induced by I is $\mathcal{H}' = (\mathcal{C}', \leq')$.

Informally, the resulting hierarchy contains no members from the base hierarchy but any introduced member and mirrors the inheritance relations. Empty classes are possible.

As name clashes or *static interference* are considered an error by the AspectJ compiler *ajc*

$$\forall C \in \mathcal{C}' : \forall m \in \mathit{members}_{\mathcal{H}'}(C) : \\ C \in \mathcal{C} \wedge m \notin \mathit{members}_{\mathcal{H}}(C)$$

always holds for syntactically correct AspectJ programs. AspectJ[1] does not allow overriding introductions. So only *basic compositions*, i.e. compositions without priority rules to choose from a set of possible method implementations, have to be considered.

The hierarchy induced by an aspect needs not to be syntactically correct as methods introduced by the aspect might reference methods not present in $\mathcal{H}'$ but only in $\mathcal{H}$. All these dangling references are bound after combination of the resulting hierarchies if the original AspectJ-program was correct.

The hierarchy $\mathcal{H}'$ induced by the introductions of an aspect A will now be composed with the hierarchy of the base system $\mathcal{H}$ by using a hierarchy composition operator $\oplus_s$. When working with arbitrary hierarchies, the inheritance relations of both hierarchies can be contradictory, e.g. if $(B,C) \in \leq_1$ and $(C,B) \in \leq_2$.

This is immpossible if a hierarchy induced by an aspect should be combined with the base hierarchy, as the resulting inheritance relation is always conflict free (here, they are identical), no collapsing of cycles is necessary and the general combination operator of [10] can simplified as follows:

Definition 3.3 (Simplified Hierarchy Composition) *Let* $\mathcal{H}_1 = (\mathcal{C}_1, \leq_1)$*,* $\mathcal{H}_2 = (\mathcal{C}_2, \leq_2)$ *be two class hierarchies with conflict free inheritance relations* $\leq_1$*,* $\leq_2$ *and no static interference. Then* $\mathcal{H}_1 \oplus_s \mathcal{H}_2 = (\mathcal{C}, \leq)$ *is defined as follows*[2]*:*

1. $\mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2$
2. $(\leq) = (\leq_1 \cup \leq_2)$
3. $\forall C \in \mathcal{C}_{\mathcal{H}} : \mathit{members}(C_{\mathcal{H}}) = \mathit{members}(C_{\mathcal{H}_1}) \cup \mathit{members}(C_{\mathcal{H}_2})$ [3]

It is easy to see that the effect of composing $\mathcal{H}$ and $\mathcal{H}'$ using operator $\oplus_s$ has the same effects as the introductions of AspectJ: Both operations simply add the introduced members to the respective classes of the resulting hierarchy.

Following the analysis of [10], it is now possible to apply the stated noninterference criterion for AspectJ introduction as well, which informally states that all used virtual calls must evaluate to the same method as before.

3.3 Finding Changed Lookups

To test the interference criterion it has to be checked, whether the dynamic lookup for any possible call has changed. The analysis described below only needs the hierarchy and signature information as input; method bodies are *not* analyzed. This approach guarantees that the hierarchy preserves its behavior if *no* binding interference occurs at all.

For impact analysis, this information is insufficient as the set of changed lookups calculated by the subsequent analysis demands that behavior of *any* affected class together with its subclasses has to be considered as beeing changed. The reason is that methods defined in a class in $\mathcal{H}$ might transitively use a call with a changed lookup in their implementation.

To reduce the set of affected classes, a simple code scanning of an affected method for calls with changed lookup might be enough—methods only using unchanged calls in their implementation as well as calls evaluating to unaffected classes are guaranteed to work as before if only these methods are called. The call graph is an appropriate data structure to calculate all this information.

Note that newly introduced methods may very well change the state of objects, thus altering system behavior. Anyhow, introduced methods are never called by the original system as the system would not have been syntactically correct otherwise—the method did not exist in the original system[4].

[1] Referenced Version is 1.0.6.

[2] In this paper $\oplus$ will refer to $\oplus_s$.

[3] Here, $\mathit{members}(C_{\mathcal{H}_j})$ indicates the set of members defined in class C in hierarchy $\mathcal{H}_j$. If $C \notin \mathcal{H}_j$, then $\mathit{members}(C_{\mathcal{H}_j}) = \emptyset$.

[4] Keep in mind, that advice is not considered here—advice code might call newly introduced methods.

The information necessary to check the interference criterion as well as for impact analysis is the set of changed lookups. In [10], calculation of changed lookups is more precise as only calls actually appearing in the hierarchy are examined (using points-to analysis). The method proposed here calculates any possible change in lookup due to aspect application. The loss of precision might be negligible as the set of changed lookups is much smaller (explicit introduction instead of arbitrary hierarchy combination). As an additional advantage, our algrithm is independent of a specific client, because all statically possible calls are examined.

This set of method calls can easily be calculated by a modified version of breadth first search, given by algorithm 3.1. Recall that a class hierarchy in Java (as well as in AspectJ) always defines a tree. Therefore, the inheritance relation $\leq$ always contains `java.lang.Object` as maximal element. For the algorithm let $\mathtt{C} \in \mathcal{C}$ be a class. Then $Ints(\mathtt{C})$ is the set of all methods introduced in class `C`. For the root object, define $\Delta lookup(father(root)) = \emptyset$.

Algorithm 3.1 Calculation of Changed Lookups

algorithm get-binding-interference
input: hierarchy $\mathcal{H} = (\mathcal{C}, \leq), \forall \mathtt{C} \in \mathcal{C} : Ints(\mathtt{C})$
output: $\forall \mathtt{C} \in \mathcal{C} : \Delta lookup(\mathtt{C})$

queue = $\{\max(\leq)\}$
while queue $\neq \emptyset$ do
  $\mathtt{C} = remove(queue)$
  $\Delta lookup(\mathtt{C}) = (\Delta lookup(father(\mathtt{C}))$
    $-members(\mathtt{C})) \cup Ints(\mathtt{C})$
  $\forall \mathtt{D} : \mathtt{D} \leq \mathtt{C}$ do: add `D` to queue

The changes in lookup are used as input for a subsequent impact analysis (refer to section 5). However, changes in lookup are not only due to introduction but can have a different reason: hierarchy modification. Its effects are examined in the next section.

4 Noninterference Criterion for Hierarchy Modification

Besides introduction, AspectJ allows structure modification of inheritance hierarchies, with the intention to move classes (together with all their subclasses) 'down' the inheritance hierarchy, so that original type relations still hold[5].

4.1 Impact of Changing the Inheritance Hierarchy

The impact of changes in the inheritance relations is demonstrated in figure 3. The changes presented in thie example are due to application of the following simple aspect:

```
aspect O {
    declare parents: D extends G;
}
```

At first sight any client using classes with a modified inheritance hierarchy should still work as any type relation is still correct. However, there are two problems. Let `d` be an object of type `D`:

instanceof: In example of figure 3, class `D` is moved down the inheritance hierarchy by aspect `O`. Any predicate `d instanceof G` now changed value—from *false* to *true*. More generally, the **type** of class `D` has changed. This allows additional up-casts (`(G)d`), which resulted in a `ClassCastException` before. These exceptions might have been caught and so control flow might have changed.

binding interference: Change of inheritance hierarchies might possibly change the method actually executed by a virtual call. Figure 3 gives an example of this situation with method call `d.n()`: Without application of the aspect, `A.n()` is called; with `O` applied, the virtual call evaluates to `G.n()`.

4.2 Hierarchy Modification as Hierarchy Composition

Modification of the inheritance hierarchy can again be viewed as a hierarchy combination. In this case, the hierarchy induced by `declare parents ...extends` statements contains an empty class for any class in the base hierarchy and an inheritance relation $\leq'$ modified by the aspect statement as follows:

Definition 4.1 (Induced Hierarchy) *Let $\mathcal{H} = (\mathcal{C}, \leq)$ be a hierarchy an aspect `A` is applied to. Let `D` be the set*

[5] It is not possible to move classes 'up' in the inheritance hierarchy (AspectJ accepts this declaration without effect).

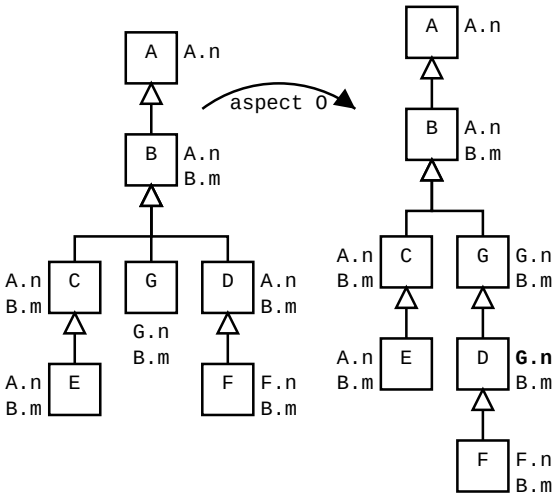

Figure 3: Effects of hierarchy modification.

of tuples derived from `declare parents ... extends` *statements of this aspect. Then* $\leq'$ *is defined as follows:*

$$(\leq') = (\leq \cup D)$$

The hierarchy defined by `A` *is* $\mathcal{H}' = (\mathcal{C}', \leq')$*, where* $\mathcal{C}' = \mathcal{C}$*,* $\forall \mathtt{C} \in \mathcal{C}'$ *:* $members(\mathtt{C}) = \emptyset$*.*

As hierarchy modifications in AspectJ are restricted—it is only allowed to declare that a class now is a subclass of a sibling (or a subclass thereof)[6]in the inheritance tree—the following always holds:

- $(\leq) \subseteq (\leq')$
- $(\mathtt{D},\mathtt{C}) \in (\leq') \Rightarrow (\mathtt{C},\mathtt{D}) \notin (\leq')$ (no conflicts in $\leq'$)

With this properties, the simplified hierarchy combination operator can be applied as no collapsing of equivalence classes due to conflicts is necessary. The resulting hierarchy is given by $\mathcal{H} = (\mathcal{C}, \leq')$.

4.3 Impact of Type Changes

To prove that any client still works as before, the interference criterion of [10] is a necessary but *not sufficient* condition. If a language contains statements for run time type identification (*RTTI*), control flow might change although the above noninterference criterion is met. Java contains such statements with the predicate `instanceof`, which allows to make control flow dependent of the type of an object.

To guarantee that behavior of a client is preserved, all `instanceof` statements have to evaluate to the same value. To calculate the value of such expressions, the type of each reference involved in an `instanceof` predicate has to be known. Approximations with points-to analysis are possible but precise points-to analysis is undecidable. Thus in general only a superset of the type of an object a reference points to can be calculated.

Preservation of behavior can only be guaranteed iff points-to sets of references involved in an instanceof-statement before and after the hierarchy modification evaluate to the *same single type*—a very rigid requirement. In general, when using static analysis, many predicates will evaluate to type-sets with a cardinality bigger than one. In this case, conservative approximation requires to assume that the behavior of the client has changed.

To check the impact of changes to any client of the modified hierarchy the noninterference criterion can be applied if RTTI is excluded. Finding the method calls with changed lookup is easy: Only calls to methods (re)defined in a class between (and including) the new and the former superclass can be influenced, if those methods are not redefined by the affected class itself.

4.4 Detection of Binding Interference due to Hierarchy Modifications

Detection of changes in lookup due to hierarchy modification can be achieved by a simple algorithm. The idea is that any method call has a changed target iff now the virtual call evaluates to a newly assigned superclass. This change in lookup again has to be propagated to any subclass not redefining the affected method.

Calculation of the necessary data can be performed in three steps:

1. Get the set of classes D affected by hierarchy modification.
2. $\forall \mathtt{d} \in D$ calculate the intermediate classes IC between this class and the newly assigned superclass.
3. For any method `m` known in `d`, check if a call now actually evaluates to a class $\mathtt{C} \in IC$. If this is the case, the behavior of the call to `m` possibly changed and `m` has to be added to $\Delta lookup(\mathtt{d})$.

Again, any (transitive) subclass of `d` which does not redefine `m` is affected by the change as well.

[6] If u, v are siblings $\Rightarrow (u,v) \notin (\leq^*) \wedge (v,u) \notin (\leq^*) \wedge \exists w \in \mathcal{C} : (u,w) \in (\leq^*) \wedge (v,w) \in (\leq^*)$, $(\leq^*)$ indicates the transitive closure of $(\leq)$.

5 Impact Analysis of Changes

In [9], a method to compute impact of system modifications on a set of given test drivers has been suggested. It breaks modifications down into atomic changes like *add method* (AM) and *add field* (AF). These atomic changes can be easily derived from the aspects; dependent changes like *change lookup* are calculated by the analysis presented in sections 3 and 4.

With the set of changed lookups at hand, impact analysis can be used to choose a set of test drivers which has to be rerun to check whether the system still works as intended. Only a short summary is presented here, for details refer to [9].

The classes of the hierarchy $\mathcal{H}$ under consideration are now associated with a set of test drivers $\mathcal{T} = \{t_1, \ldots, t_n\}$, where each $t \in \mathcal{T}$ calls a subset of methods defined by classes in $\mathcal{H}$. For each test driver t_i, impact analysis is performed using the call graph of t_i to determine if the test driver (or client) is affected. This is done by checking if t_i calls (maybe transitively) any method with changed lookup.

This check uses calculated information about changed lookups when traversing an edge in the call graph. If the call matches a call in the set of changed lookups $\Delta lookup(\mathtt{C})$ the test driver has to be rerun.

To create the call graph, the type of the calling object at runtime has to be determined for each method call to decide whether the call changed its behavior. This is the case if the object reference may have a type with changed behavior as indicated by the analysis presented above.

Unfortunately, calculation of the exact type at runtime is undecidable. However, points-to analysis can be used to calculate an approximation: the set of possible types for an object reference in the test driver. If a call of any type in this set is contained in the set of methods with changed semantics, conservative approximation demands that the semantics of this call have to be considered as changed. In this case, the test driver containing this method call has to be rerun. The results of this regression tests show if the program still works as intended.

So, the analysis proposed here can provide different results:

- A set of introductions and hierarchy modifications with no effect *on a given set* of test-drivers can be determined. These changes can be incorporated safely into the system as the semantics of the system are not changed.
- For atomic changes modifying system behavior, the subset of test cases which must be rerun can be determined. Impact of these changes can be checked by the results of these regression tests only.
- For the given hierarchy $\mathcal{H}$, impact of static features of aspect application on the semantics of the hierarchy can be determined.

This information can be used by the programmer to avoid unexpected changes and specifically examine results of intended changes.

6 An Example Analysis

To see how the proposed algorithms work, the analysis is applied to an example using all static modification features of AspectJ.

Program 6.1 Combined Aspect Applied to Hierarchy.

```
class Main{
  public static void main(String[] args) {
    print("A: "); A a = new A(); a.n();
    print("B: "); B b = new B(); b.n(); b.m();
    print("C: "); C c = new C(); c.n(); c.m();
    print("D: "); D d = new D(); d.n(); d.m();
    print("E: "); E e = new E(); e.n(); e.m();
    print("F: "); F f = new F(); f.n(); f.m();
    print("G: "); G g = new G(); g.n(); g.m();
    println();
  }
}

aspect MNO {
  // declare parent extends / implements
  declare parents: D extends G;
  declare parents: C implements I;
  declare parents: D implements I;

  // introductions
  public void I.y() { print("I.y()"); }
  void B.n() { print("B.n()"); }

  public static void main(String[] args) {
    print("A: "); A a = new A(); a.n();
    print("B: "); B b = new B(); b.n(); b.m();
    print("C: "); C c = new C(); c.n();
      c.m(); c.x(); c.y();
    print("D: "); D d = new D(); d.n();
      d.m(); d.x(); d.y();
    print("E: "); E e = new E(); e.n();
      e.m(); e.x(); e.y();
    print("F: "); F f = new F(); f.n();
      f.m(); f.x(); f.y();
    print("G: "); G g = new G(); g.n(); g.m();
    println();
  }
}
```

6.1 The System to Analyze

As a starting point, the class hierarchy defined by program 1.1 is given, together with aspect `MNO`, which combines

the effects of former aspects. It introduces a new method `n` to class `B`, changes the inheritance relation (`declare parents:D extends G`) and declares that classes `C` and `D` implement interface `I`. Methods are inserted to class interface `I`. Additionally, the aspect defines an own `main`-method which is necessary to test the results of interface declaration. Effects of aspect application are a changed structure as well as a changed lookup for some methods.

The classes of this example are quite simple: All methods only print their name and the class they are defined in, but this setting is already sufficient to show how the aspect affects the existing system. Figure 4 presents the output of the system. The figure contains three sections. The output of the original system without application of the aspect is marked with '(a)'. The effects of binding interference are visible in section '(b)', which shows the output of the original main method with aspect `MNO` applied to the system. The set of known methods is identical, but the dispatch has changed for classes `B`, `C`, `E`, and `D`. The first three classes are affected by the introduction of `n` to `B`, class `D` by the change of the hierarchy.

All effects of the aspect are visible in section '(c)', where the effects of the `declare parents: ...implements I` statements become visible. No 'old' base system code uses this effects as in the original hierarchy `C` and `D` did not implement `I`. So, for `C`, `D` and all their subclasses, methods `x` and `y` can be called. For class `C` only an implementation of `x` is provided, for `y` the default implementation of `I` is used—as is visible in the output.

6.2 Applying the Proposed Analysis

The analysis revealing classes only using the default implementation of an interface, like e.g. `E` does, is quite simple and not considered. The example concentrates on changes in lookup. Changes due to introduction can be found by applying algorithm 3.1. For the example hierarchy, table 5 summarizes the gathered information. The example application of the algorithm traverses all classes of a given hierarchy according to a bfs-order determined by the structure of the class hierarchy *after* applying hierarchy modifications of the aspect.

Step 7 is interesting as at this position the changed lookup results from the change of hierarchy structure, *not* from introduction (the father of `D` now is `G`, which has an own definition of method `m`; so introduction of `m` to `B` has no longer any effect on `D`). When calculating changes in lookup, these effects must be considered. The algorithm reproduces the results visible when comparing sections (a) and (b) of figure 4.

6.3 Using these Results—Impact Analysis

The calculated information about changed lookups can be used for impact analysis to determine whether a given test driver has to be rerun. For illustration consider the set of (quite simple) test drivers associated with the example hierarchy presented in program 6.3.

To decide if control flow has been changed by introductions, the call graph has to be constructed. Note that points-to analysis is necessary as the types of caller and callee of a virtual call has to be identified or at least re-

```
a) : original system
javac demo.java
java Main
A: A.n()
B: A.n()    B.m()
C: A.n()    B.m()
D: A.n()    B.m()
E: A.n()    B.m()
F: F.n()    B.m()
G: G.n()    B.m()

b) : changes due to dynamic
     interference
ajc demo.java demo.aj
java Main
A: A.n()
B: B.n()    B.m()
C: B.n()    B.m()
D: G.n()    B.m()
E: B.n()    B.m()
F: F.n()    B.m()
G: G.n()    B.m()

c) : including introduction
     to interface
ajc demo.java demo.aj
java M
A: A.n()
B: B.n()    B.m()
C: B.n()    B.m()   C.x()   I.y()
D: G.n()    B.m()   D.x()   D.y()
E: B.n()    B.m()   C.x()   I.y()
F: F.n()    B.m()   D.x()   D.y()
G: G.n()    B.m()
```

Figure 4: Example: Produced output.

Step	v	declared methods	members(v)	Intr(v)	Δ*lookup*(v)	queue
1	-	-	-	-	-	{A}
2	A	n	n	-	-	{B}
3	B	n, m	m	n	B.n	{C, G}
4	C	n, m	-	-	B.n	{G, E}
5	G	n, m	n	-	-	{E, D}
6	E	n, m	-	-	B.n	{D}
7	D	n, m	-	-	**G.n**	{F}
8	F	n, m	n	-	-	∅

Figure 5: Results produced by the algorithm (x, y omitted).

Program 6.2 Test Drivers for the Example Hierarchy.

```
class T1 {
    public static void main(String[] args) {
        F f = new F();
        f.m();   // calls B.m()
        f.n();   // calls F.n()
    }
}
class T2 {
    public static void main(String[] args) {
        B b = new B();
        b.n();   // calls B.n(), changed lookup
    }
}
class T3 {
    public static void main(String[] args) {
        G d;
        if (args.length != 0) d = new D();
        else d = new G();
        d.n();   // calls G.n(), caller: D or G
    }
}
```

stricted to a as-small-as-possible type set.

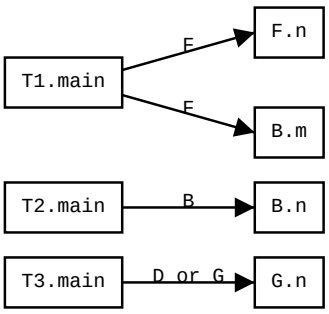

Figure 6: Call Graph of Simple Test drivers

To get a first impression how impact analysis works, consider test drivers T1 to T3 and their call graph. The edge labels of figure 6.3 indicate the type of the calling object. To evaluate the impact of an aspect using the call graph, we need the results of table 5.

Test driver T1 is obviously unaffected by changes due to aspect application as no lookup for an F-object changed. Test driver T2 calls n from a B-object. This lookup has changed from A.n() to B.n() due to introduction of method B.n. This test driver has to be rerun.

Test driver T3 is a little more complex as here the type of the calling object is statically unknown. Possible types are D and G. For a G-object, semantics would be preserved, but for a D-object, the call would evaluate to G.n() and not to A.n as in the original hierarchy. Conservative approximation demands to rerun test driver T3. Certainly this is a simple example, but there is no restriction to apply this analysis to real-world call graphs as it can be done by performing this simple check for every edge.

7 Preliminary Implementation and Future Work

A prototype of the analysis presented in sections 2 to 4 has been implemented and produces reasonable results for programs written in a subset of AspectJ, including the example of section 6 presented in this paper.

However, implementation of the impact analysis and extension of the set of analyzable programs still has to be done. A point of interest is the handling of Java import-statements as imported classes are necessary information to built up the hierarchy $\mathcal{H}$. For these classes, source code might not be available. To solve this problem, it is planned to reconstruct class information out of Java byte code using the BCEL API.

Evaluation of occurence of binding interference in 'real life' AspectJ programs is necessary to determine if this problem is actually relevant for AspectJ programmers. However, even if binding interference is not very frequent, the AspectJ compiler should issue a warning.

8 Conclusion and Related Work

This paper pointed out the problem of binding interference emerging from usage of the AspectJ features introduction and hierarchy modification. Definitions are given how AspectJ introduction and hierarchy modification can be interpreted as hierarchy combinations. With this definitions at hand, the noninterference criterion of [10] and the impact analysis of [9] can be applied to check if clients of the hierarchy under consideration possibly changed behavior. This analysis can help AspectJ programmers to examine the impact of aspects before application and avoids subtle flaws in their programs.

To improve separation of concerns, several different approaches besides aspect oriented programming have been suggested. Aksit et al. proposed composition filters [2, 1] to route incoing and outgoing messages through a filter queue, thus enabling similar functionality. Batory et al. proposed layered designs [4, 3].

Especially relevant for the approach presented here is [8]. Ossher and Tarr proposed multi-dimensional separation of concerns, leading to a separate implementation of different features and a composition of the resulting hierarchies according to user defined composition rules. Semantics of these compositions are a research topic addressed in [10].

Besides [10], very little work of program analysis for AOSD approaches is known, although impact analysis of [9] could be used for AOSD software as well.

Acknowledgements

Thanks to Silvia Breu for her valuable feedback.

References

[1] M. Aksit and B. Tekinerdogan. Solving the modeling problems of object-oriented languages by composing multiple aspects using composition filters, 1998.

[2] Mehmet Aksit, Ken Wakita, Jan Bosch, Lodewijk Bergmans, and Akinori Yonezawa. Abstracting Object Interactions Using Composition Filters. In Rachid Guerraoui, Oscar Nierstrasz, and Michel Riveill, editors, *Proceedings of the ECOOP'93 Workshop on Object-Based Distributed Programming*, volume 791, pages 152–184. Springer-Verlag, 1994.

[3] D. Batory and Y. Smaragdakis. Building product-lines with mixin layers, 1999.

[4] Don Batory and Sean O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, 1992.

[5] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[6] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. 1994.

[7] Leonid Mikhajlov and Emil Sekerinski. A study of the fragile base class problem. *Lecture Notes in Computer Science*, 1445:355–382, 1998.

[8] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach, 2000. Proc. Symposium on Software Architectures and Component Technology: The State of the Art in Software Development.

[9] Barbara G. Ryder and Frank Tip. Change impact analysis for object-oriented programs. *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering (PASTE 2001)*, pages 46–53, 2001.

[10] Gregor Snelting and Frank Tip. Semantics-based composition of class hierarchies. In *ECOOP*, page 562ff, 2002.