Analysis of AspectJ Programs

Maximilian Störzer*

December 11, 2003

Abstract

Program Analysis is increasingly used to enhance program understanding and find flaws in programs. In contrast to testing, it can *guarantee* properties of a program. Up to now, in the context of program analysis, aspect oriented programming (AOP) has mostly been used for program instrumentation (tracing), but has not been itself subject to analytical methods. This paper identifies sources of flaws in AOP and suggests that program analysis could be used to avoid these pitfalls.

The subject of this paper is not to present any solution for identified problems, but presents ideas how a solution might be approached.

1 Motivation

AOP is a new paradigm in programming extending traditional programming techniques, first introduced in [4]. Its basic idea is to encapsulate *crosscutting concerns* influencing many modules of a given software system in a separate 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 an 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*, a prototype aspect-oriented language extending Java, is considered. Main features of AspectJ are *introduction*, modification of class hierarchies and *advice*; each of these will be examined in detail later.

AOP is a very attractive and powerful technique, but holds new risks, too. Changes introduced with AspectJ are not visible *directly* in the the base system's source code, making program comprehension more difficult. Aspects are a new modularization unit, and are usually stored in separate files. But the effects of this code can influence the whole system. Tool support is necessary to make this aspect influence visible [14].

Some AOP-related problems are discussed in detail in this paper, with examples given for AspectJ. Organization is as follows: Section 2 addresses static elements, section 3 `advice`. Section 4 deals with aspect interference. Section 5 concludes, outlines future work and gives an overview of related work.

2 Static Program Modification

AspectJ offers a set of mechanisms designed to modify behavior of an existing system: Introduction allows to insert new members (both methods and data fields) into classes, classes can be pushed down the class hierarchy using `declare parent`.

2.1 Binding Interference and Class Introduction

Introducing new members into existing classes can result in name clashes (or *static interference*) if a member with the same name already exists. Name clashes are reported as errors by the AspectJ-compiler, *ajc*, and can thus easily be avoided.

Binding interference—i.e. changes in class behavior by introducing new members in a hierarchy, called dynamic interference in [12]—however has to be examined. Consider the example program from figure 1.

Aspect `introduceM` introduces a method `m` to class `B`. Any call[1] from class `C` now results in a call of `B.m()` and not in `X.m()` as before; the program output for call `c.m()` is `Aspect:B.m()`.

Class `B` might work with this change as intended, as this is the class where the method is introduced to.

[1]University of Passau, Lehrstuhl for Software-Systems, 94032 Passau, `stoerzer@fmi.uni-passau.de`

[1]As AspectJ extends Java, any call mentioned in this paper means a *virtual* call.

```
class X {
    void m() { System.out.println("X.m()"); }
    void n() { System.out.println("X.n()"); }
    public static void main(String[] args) {
        C c = new C();
        X a = new A();

        c.m();   // show binding interference
        if (a instanceof B) {
            System.out.println("B object!");
        } else {
            System.out.println
              ("Unknown class - no B!");
        }        // show type change
        a.n();   // show binding interference
                 // due to hierarchy change
    }
}
```

```
class A extends X {}

class B extends X {
    void n() { System.out.println("B.n()"); }
}

class C extends B {}

aspect introduceM {
    void B.m() {
        System.out.println("Aspect: B.m()");
    }
}

aspect ChangeHierarchy dominates introduceM {
    declare parents: A extends B;
}
```

Figure 1: Binding Interference and Change of type hierarchies

Anyhow, effects of introductions modify any subclass of `B`, which does not itself redefine `m`. If the introduced method `B.m()` redefines `X.m()` 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 guarantee this kind of method redefinition. Code of a (unknown) client might rely on calls to `X.m()`; the clients functionality then is broken.

The described problem is a special case of the *fragile base class problem* [7], as here the behavior of a subclass suffers from—oblivious—base class modifications. Although tracking down bugs introduced by changing a base class is difficult (only a superclass of the class where the error shows up has been changed), 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 find bugs emerging from binding interference, impact analysis of aspect application can reveal method calls with *changed dynamic lookup*.

To avoid flaws the compiler could prohibit binding interference altogether, but this is too strict as the changed behavior of subclasses might be intended as well. A reasonable approach could be that the compiler warns if binding interference occurs and leaves actions to the programmer.

In [11], a method to compute changes in dynamic lookup has been suggested which breaks source code modifications down into atomic changes. This impact analysis can be used here, as a subset of atomic changes (add method, change lookup and add field) can be easily derived from the introduction definition of the aspect. With the set of atomic changes at hand, the set of changed dynamic lookups and necessary regression test cases can be determined.

2.2 Impact of Changing the Inheritance Hierarchy

Besides introduction, AspectJ allows to modify the structure of inheritance hierarchies within certain boundaries[2], intended to move classes (together with all their subclasses) 'down' the inheritance hierarchy, so that original type relations still hold.

Consider figure 1. At first glance any client using classes with a modified inheritance hierarchy should still work. However, there are some problems:

instanceof: In example of figure 1, `A` is moved down the inheritance hierarchy by aspect `ChangeHierarchy`. Any predicate `a instanceof B` for a `a` of type `A` now changed value—from *false* to *true*. More generally, the **type** of class `A` has changed. This allows additional up-casts, which resulted in a `ClassCastException` before (e.g. `(B)a`).

binding interference: Change of inheritance hierarchies might possibly change the method actually executed by a virtual call. Figure 1 gives an example of this situation with method call `a.n()`: without application of the aspect, `X.n()` is called, with aspect `ChangeHierarchy` active, `B.n()` is executed.

The output of the described example is without aspect

```
X.m()
Unknown class - not a B instance!
X.n()
```

[2]The newly assigned superclass has to be a subclass of an old superclass.

and with aspect

```
Aspect: B.m()
B object found!
B.n()
```

This demonstrates both binding interference due to *introduction* and due to *changes in class hierarchy*.

As a first step to reveal this subtle changes in program behavior, the set of classes from a hierarchy $\mathcal{C}$ with changed type information has to be determined by examining the `declare parent` statements of all aspects `A` in a given set of aspects, $\mathcal{A}$. Change of type information can be caused by changing the position of a class in the inheritance hierarchy or by defining that a class implements an interface (`declare ... implements`). All subclasses of a modified class are affected by these changes as well. Let $ChangeType(\mathcal{A}, \mathcal{C})$ be the set of all these classes.

Using *points-to analysis* [2], for each '<`ref`> `instanceof` <`type`>' predicate of a client `K`, the type set reference '`ref`' might point to, has to be determined. If the intersection of this set with $ChangeType(\mathcal{C})$ is not empty, behavior of client `K` might have changed. As exact points-to analysis is undecidable, false alarms are possible.

To reveal binding interference, `declare parent` statements have to be examined if method lookups might have changed by redefinition of inheritance hierarchies. Let $ChangedLookup(\mathcal{C})$ be set of *possibly changed method lookups*. Clients of any class in $ChangedLookup(\mathcal{C})$ possibly changed their behavior.

3 Changing Runtime Behavior of Programs: `advice`

Apart from static program modifications described up to now, AspectJ `advice` allows to modify *state and control flow* of the program *at run time*. The effects of these modifications are difficult to foresee, their analysis requires data flow analysis (DFA).

3.1 Defining Pointcuts

AspectJ defines a set of *hooks* in the execution of a program, where dynamic modification of program flow and state are applied, so called *joinpoints*. A set of joinpoints can be named for further use in the aspect, by defining a *pointcut*.

As the idea of aspects is used to encapsulate crosscutting concerns, a pointcut potentially comprises a large set of joinpoints (dependent on the size of the system). The use of wildcards is very comfortable to specify necessary pointcuts, but their application can be problematic, especially in large systems. Relying on *naming conventions* is dangerous here—conventions are not guaranteed. So wildcards may easily miss necessary or accidentally include unwanted joinpoints.

Avoiding name-based pointcuts completely might solve the problem but is obviously too restrictive. A logical consequence is to provide tool support to allow better control of aspect application by the programmer as is available for AspectJ (XEmacs mode/some IDE plug-ins). Aspects are mapped to every joinpoint they apply to (and vice versa) and visualized. However, these tools are only *passive*, the user must explicitly examine, *if and where* there are changes in the set of matched joinpoints. This is insufficient, as programmers work with large systems and will not examine the whole system for *unexpected* changes. Tools should only display *changes* in sets of affected joinpoints. This restriction is necessary as this is exactly the information a programmer needs to see the impact of changes to a pointcut modification.

Another interesting approach might be to develop a high level specification language, allowing to specify exactly *which parts* of a given system are allowed *under which circumstances* to be modified by an aspect. A tool could then take pointcut definitions and a specification and check if these definitions are valid, thus avoiding unexpected aspect influence. Pointcut definitions using wildcards too generously would be revealed.

3.2 Requirements for Data Flow Analysis

To understand effects of `advice`, advanced program analysis might help. As a starting point, many algorithms in this context require a dedicated representation of programs. A standard data structure used in this context are *Dependence Graphs (DGs)*. Their efficient construction has been a research topic for years [3], but especially for object-oriented languages like Java it is still discussed.

As AspectJ is an extension to Java, an aspect oriented DG (ADG) is supposed to extend a Java DG. An operational semantics for subsets of Java[13] and AspectJ `advice`, modeled as method call interception [5] is available, which is a necessary precondition for construction of the control flow graph (CFG) and finally

```
class C {
  int x = 0;
  B b = new B();

  void m() {                                [A]
    b.n();
    System.out.println(x);
  }
}


class B {
  int y = 0;

  void n() {
      System.out.println(y);
  }
}
```

```
aspect A {                                  [C]
  // define pointcut
  pointcut callBn(B b, C c):
    target(b) && this(c) && call(void B.n());

  // define around advice - changes
  // member values of caller and callee
  void around(B b, C c): callBn(b, c) {
    c.x = 5;
    b.y = -3;
    proceed(b, c);
    // --> redefinition of B.n() if suppressed
  }
  // test driver
  public static void main(String[] args) {
      C c = new C();
      c.m();
  }
}
```

Figure 2: Using AspectJ `advice`—an example for wrapping a method.

the ADG as a CFG has to consider known aspect applications[3].

With the ADG at hand, traditional analytic techniques as well as new techniques analyzing impact of aspect code can be used with aspect oriented languages. Fulfillment of the requirements described in brief here allows to apply DFA to AspectJ. Unfortunately, construction of the ADG is far from trivial, a fitting infrastructure is not available.

3.3 Impact of Advice—changing program state and flow

Although analysis of `advice` is expensive, it would be a great achievement to improve confidence in AOP as especially `advice` can completely change the semantics of a program. DFA can prove whether control flow remains the same independent of aspect application.

Using `advice`, AspectJ allows insertion of arbitrary code (even wrapping of existing methods) at *joinpoints*. Consider the example from figure 2. Method `C.m()` is wrapped by around-`advice` of aspect `A`. The `advice` has access to any (visible and modifiable) members of caller and callee[4].

The example in figure 2 modifies data members `c.x` and `b.y` using around `advice`. Changing these values might alter program flow as the *state* of caller and callee objects changes. More important—these changes are applied *globally* at any matching pointcut and may corrupt the system if changes in state are applied accidentally.

Changes of system behavior due to advice are only comprehensible if influence of aspect code is visible directly in the source code of the base system. Tools (as mentioned before) displaying applied aspects at relevant joinpoints produce annotation as shown in figure 2 (markings `[A]`, `[C]`). However, these merely textual annotations are insufficient for two reasons:

- First, they only show that aspect `A` references method `C.m()`, but influence on `B` is not shown as there is no pointcut definition affecting `B` defined in the aspect. Note, that the around `advice` changes value of `b.y`, so there is influence on `B`. This influence is not captured by the annotation.

- Second, there is no information if the state of a class is changed by an applied aspect or not. Direct influence is easy to track, but influence can be hidden by a sequence of method calls of caller or callee, finally influencing values of some distant object.

This information is important for every programmer and should be visible at every affected class.

Using methods of program analysis, `advice` may be classified into *semantic changing* and *semantic preserving*. Using DGs, changes in control flow and/or program state can be revealed using e.g. slicing to find witnesses for influence on system data members from aspect code. Logging for example does not influence program execution (not considering performance and

[3] For DG construction, calculation of *Reaching Definitions* sets on the CFG is necessary.

[4] Around `advice` is an interesting feature of AspectJ—it allows wrapping and thus can be used to simulate Composition Filters [1]. This shows the relation between these at first glance very different AOSD approaches.

```
class Main {
  public static void main(String[] args) {
    start();
    end();
  }
  void start() {}
  void end() {}
}
aspect A {
  before(): call(Main.start()) {
    openDoor();
  }
  after(): call(Main.end()) {
    closeDoor();
  }
  void openDoor(){}
  void closeDoor(){}
}
aspect B {
  before(): call(Main.start()) {
    goInside();
  }
  after(): call(Main.end()) {
    goOutside();
  }
  void goInside(){}
  void goOutside(){}
}
```

Figure 3: Ordering of Aspects—Conflicting `advice` Order.

the log file) and could be ignored when tracking down bugs due to advice application.

4 Interfering Aspects

Up to now, possible problems of interference of aspects and base system have been discussed. Additionally, aspects can interfere with each other either directly—e.g. via introduction, aspect domination or advice—or indirectly by *manipulating and reading the same fields* of a class of the base system.

Tools like XEmacs AspectJ-mode can display different aspects applied in parallel at join-points. But again, they do not show any semantical interference of these aspects. The problems emerging from interfering aspects are similar to those emerging from aspect-base interference and will not be considered here.

In [9], an additional problem only affecting the application of multiple aspects at one joinpoint has been stated: aspect order. AspectJ permits definition of a partial order on aspects using `dominates` language construct: `aspect A dominates B { ... }`, indicating that `A` is applied after application `B` has been applied, possibly overriding effects of `B`. If `advice` from both aspects demand a conflicting execution order, like in example 3 (taken from [9]), the granularity of this ordering is too coarse as single pieces of advice have to be ordered, not whole aspects.

The only sound execution order is obviously $\ll start - openDoor - goInside - goOutside - closeDoor - end \gg$. So neither `A dominates B` nor vice versa is sound with necessary `advice` order.

Detection of contradictory aspect order is difficult, as semantics of the code is relevant here. A workaround might be to request an annotation determining order: A before B, vice versa or don't care. This can be achieved as a special comment in code at the joinpoint where different pieces of `advice` apply. The presence of such annotations can be checked automatically and conflicts (cycles) in this user given dependencies can easily be found.

5 Conclusion and Related Work

Many software engineers shiver when thinking about modifying the behavior of their system by some aspect like 'by magic'. Indeed, AspectJ introduces language elements which can be sources of subtle flaws in a program. These problems have to be solved, before techniques like AspectJ will be commonly used.

Information calculated by applying program analysis to AspectJ can be used to dramatically improve tool support available for AspectJ—either as compiler warnings or separate analysis toolkits—so reducing flaws in AspectJ programs.

Ideas to solve the problems demonstrated in this paper suggest, that an ADG and points-to analysis for aspect oriented programs are needed to develop algorithms to enable analysis of `advice`. A better control over the effects of AspectJ languages constructs will help to improve confidence, as the lack of control of global modification features is one of the most problematic drawbacks of AOP.

Current work concentrates on the static modification features of AspectJ. Impact analysis of introduction and hierarchy modification can be performed by adapting available algorithms [11]. Analysis of `advice` is a major challenge to develop an analytical framework for AOP software. The main effort that has to be made is the creation of the necessary infrastructure for AspectJ source code. Working with Java byte code is no appropriate approach, as aspects no longer exist at byte code

level.

In [12] an impact analysis for hierarchy composition, as used in Hyper/J is proposed. [10] examines aspect interference, but focuses on aspects applying to the same joinpoint and resulting nondeterminism in application order only. In [5], an semantical approach for method call interception is presented.

To improve separation of concerns, several different approaches besides aspect oriented programming have been proposed. Aksit et al. proposed transparent *Composition Filters* [1] intercepting and rerouting method calls through filter queues. These filters are able to redirect, reject, or pass messages on to the original receiver object. Redirection of objects involves reification of messages in the filter queue. In [8], Harris and Ossher proposed multi-dimensional separation of concerns. This leads to a separate implementation of different features and a composition of the resulting hierarchies according to user defined composition rules.

Acknowledgments

Thanks to Silvia Breu, Jens Krinke and the unnamed reviewers for their 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] Michael Hind and Anthony Pioli. Which pointer analysis should i use? In *International Symposium on Software Testing and Analysis*, pages 113–123, 2000.

[3] Susan B. Horwitz and Thomas W. Reps. The use of program dependence graphs in software engineering. In *Proceedings of the Fourteenth International Conference on Software Engineering*, pages 392–411, 1992.

[4] 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.

[5] R. Lämmel. A Semantical Approach to Method-Call Interception. In *Proc. of the 1st International Conference on Aspect-Oriented Software Development (AOSD 2002)*, pages 41–55, Twente, The Netherlands, April 2002. ACM Press.

[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] K. Ostermann and G. Kniesel. Independent extensibility – an open challenge for aspectj and hyper/j, 2000. Position paper for the ECOOP'2000 Workshop on Aspects and Dimension of Concerns, C. V. Lopes (ed.), 2000.

[10] Mario Südholt Rémi Douence, P.Fradet. A framework for the detection and resolution of aspect interactions. In *Proceedings of the ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering (GPCE'02)*, October 2002.

[11] 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.

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

[13] Don Syme. Proving java type soundness. In *Formal Syntax and Semantics of Java*, pages 83–118, 1999.

[14] Detlef Vollmann. Visibility of join-points in aop and implementation languages. In *Second Workshop on Aspect-Oriented Software Developement*, pages 65–69, Bonn, Germany, February 2002. GI SIG.