Unweaving the Impact of Aspect Changes in AspectJ

Luca Cavallaro
Politecnico di Milano
Piazza L. da Vinci, 32 – 20133 Milano, Italy
cavallaro@elet.polimi.it

Mattia Monga
Università degli Studi di Milano
Via Comelico 39 – 20135 Milano, Italy
mattia.monga@unimi.it

ABSTRACT

Aspect-oriented programming (AOP) fosters the coding of tangled concerns in separated units that are then woven together in the executable system. Unfortunately, the oblivious nature of the weaving process makes difficult to figure out the augmented system behavior. It is difficult, for example, to understand the effect of a change just by reading the source code. In this paper, we focus on detecting the run time impact of the editing actions on a given set of test cases. Our approach considers two versions of an AspectJ program and a test case. Our tool, implemented on top of the *abc* weaver and the AJANA framework is able to map semantics changes to the atomic editing changes in the source code.

Categories and Subject Descriptors

D.2.3 [**Software Engineering**]: Coding Tools and Techniques

General Terms

Language, Verification

Keywords

AspectJ, Software Maintenance, Change Impact Analysis

1. INTRODUCTION

Software maintenance of an evolving code base is a complex problem. A major source of complexity is understanding the effect of source code changes on the behavior of the program. Even small changes can have non-local effects that can make difficult to grasp the impact of the global properties of the system. In object-oriented programs polymorphism and dynamic binding may affect the behavior of virtual method calls that are not lexically near the allocation site. The problem is even critical in aspect-oriented software, due to the intrinsic obliviousness [4] of join points. In fact, the woven aspect may change control and data dependencies of the base code, and a change in the aspect code can significantly affect the semantics of the whole system. This is actually the very motivation of introducing an aspect: in practice, however, especially when both aspects and classes evolved separately, it is very easy to get unexpected results. We aim at supporting the programmer in tracking down the causal chain from his/her changes to the surprising effect, in order to ease the conception of a solution. An important contribution to this problem is offered by *change impact analysis* [2], a collection of techniques for determining the effects of a source code editing action on the behavior of a set of test cases for a program. Recently, many techniques have been proposed to support change impact analysis of object-oriented software [9, 11] but very little effort has been done to apply this technique to Aspect Oriented Software

In this paper we present an application of change impact analysis to AspectJ programs. Our approach considers two versions of an AspectJ [6] program and captures their syntactical differences, breaking them in atomic changes. Then we observe the behavior of the program under a test case that gives unexpected results. We build a representation of the executed parts of the two versions and we compare them, in order to understand how the semantics has changed and how this maps to the atomic changes introduced. We implemented our approach on the top of *abc* [1], an extensible aspect compiler, and AJANA [16], a framework for AspectJ analysis.

2. MOTIVATING EXAMPLE

We illustrate the need for change impact analysis with an example for which we report the source code in Listings 1 2, and 3: it implements a simple system composed of two Java classes and one AspectJ aspect. The class `Point` (see Listing 1) has two fields, and the proper getters and setters for those fields; moreover it exposes a `setRectangular` method, which sets both fields. The class `PointExt` (see Listing 2) should be considered an evolution of `Point`, in fact it is a subclass of it that overrides two methods.

Listing 3 shows an aspect that is expected to be woven to the above base system. This aspect declares some "introductions" to the base system, two pointcuts and six pieces of advice. The evolution of the aspect consists in adding a new field and modifying the advice marked as `before2`.

A test case for the system is listed in Listing‘4.

The code reported in the example implements an observer pattern by using the aspect `BoundPoint`. The latter defines

FOAL'09, March 2, 2009, Charlottesville, Virginia, USA.

Listing 1: Source code for Point class

```
public class Point {
    // Y part omitted to save space
    int x = 0;
    public int getX() { return x; }
    public void setRectangular(int newX, int newY)
        throws Exception {
        setX(newX);
        setY(newY);
    }
    public void setX(int newX)
        throws Exception {
        if (newX < 0) throw new Exception();
        x = newX;
    }
}
```

Listing 2: Source code for PointExt class

```
public class PointExt extends Point {
    //Overridden in modified version
    public void setRectangular(int newX, int newY)
        throws Exception {
         setX(newX + 1);
         setY(newY + 1);
    }
     //Overridden in modified version
    public void setX(int newX) throws Exception {
        if (newX < 0) throw new Exception();
        x = (int)newX /2;
    }
}
```

some pieces of advice woven into join points identified by the calls to the method setX. These pieces of advice check the parameter passed to the called method and keep a history of the older values of the x field. We imagine to slightly modify the initial version of the program by overriding two methods in the class *PointExt*, which is a subclass of *Point*; moreover, we add a field in the aspect BoundPointand a line in the before2 advice.

From the viewpoint of editing changes, the overriding of the methods in *PointExt* can be decomposed in two actions: a first step that brings a new method in the subclass, then a second step that changes the method body of the added method. Finally we also consider that the overriding brings a change in the lookup table of the program: before the change when a method *setX* was invoked on an object with dynamic type *PointExt* the superclass method was invoked. After the overriding the method that is actually invoked in this case is the overridden one.

Finally in listings 4 we show a test case for the example program. Of course the outcome of the test for the first and for the modified version of the program will be different, but not all the introduced changes could be responsible for the result change.

The main purpose of our analysis is to help a developer understand which code edits originated the change and which actions he should take to bring the program back into a state in which it gave the previous output.

3. CHANGE IMPACT ANALYSIS FOR AOP

Our approach considers two versions v_0 and v_1 of the same program and a set of test cases that should apply on both. We aim at mapping source code changes to the semantic differences induced by a test case t. In order to achieve this goal, we proceed as follows:

Listing 3: Source code for BoundPoint aspect

```
public aspect BoundPoint {
    //Added in modified version
    private int previousValue;

    // a reference to a Point object
    PropertyChangeSupport support =
        new PropertyChangeSupport(this);
    public void addPropertyChangeListener
        (PropertyChangeListener l){
        support.addPropertyChangeListener(l);
    }
    void firePropertyChange(Point p,
                            String property,
                            double oldval,
                            double newval) {
        p.support.firePropertyChange(property,
            new Double(oldval),new Double(newval));
    }

    // ====== pointcuts =======
    pointcut setterX(Point p):
    call(public void Point+.setX(*)) && target(p);

    pointcut setterXonly(Point p):
    setterX(p) &&
    !cflow(
     execution(void Point+.setRectangular(int,int)));
    // ====== advices ======
    before(Point p, int x) throws InvalidException:
     setterX(p) && args(x) { // before1
        if (x < 0) throw new InvalidException("bad");
    }
    void around(Point p): setterX(p) { // around1
      int oldX = p.getX(); proceed(p);
      firePropertyChange(p,"setX",oldX,p.getX());
    }
    void around(Point p): setterXonly(p) { // around2
      int oldX = p.getX(); proceed(p);
      firePropertyChange(p,"onlysetX",oldX,p.getX());
    }
    before (Point p): setterX(p){ // before2
      //added in modified version
      this.previousValue = p.getX();
      System.out.println("start setting p.x");
    }
    after(Point p) throwing (Exception ex):
     setterX(p) { // afterThrowing1
      System.out.println(ex);
    }
    after(Point p): setterX(p){ // after1
      System.out.println("done setting p.x");
    }

}
```

1. we compare the source code of v_0 and v_1 and find the textual changes between the two. The difference is decomposed into *atomic changes* detailed in Section 3.1;
2. we build a control flow representation ($AJIG$) for each version of the program and we mark the differences we found between $AJIG_0$ and $AJIG_1$ as *dangerous edges* (Section 3.2);
3. the two programs are instrumented and run whit the test set, in order to collect dynamic pieces of information that we use to decorate $AJIG_0$ and $AJIG_1$ by marking the executed paths (Section 3.3);
4. for each test case t that gives different results when applied to v_0 and v_1 we consider the decorated graphs $AJIG_0$ and $AJIG_1$ and we finally map dangerous edges to source code changes.

The idea is that only the statements on a dangerous path solicited by the test case t can be responsible for the change

Listing 4: Source code for a test case for program in listings 1 2 3

```
public class Demo implements
                    PropertyChangeListener {
    public void propertyChange
        (PropertyChangeEvent arg0) { /* ... */ }
    public static void main(String[] a)
        throws Exception {
        Point p1 = new Point();
        p1.addPropertyChangeListener(new Demo());
        p1.setRectangular(5,2);
        if(p1.x > 5) { p1.setX(6);}
        Point p2 = new PointExt();
        p2.addPropertyChangeListener(new Demo());
        p2.setRectangular(5,2);
        p2.setX(5);
    }
}
```

of results of t: we are thus interested in them when we are trying to understand the impact of our source code editing.

3.1 Atomic changes

The first step in change impact analysis is to decompose the difference between v_0 and v_1 into a set of *atomic changes.* We consider the atomic changes as proposed by [10, 8] for the base system, and, by [18] for the aspect part. A set of atomic changes that is able to reproduce the difference between v_0 and v_1 is computed by comparing the abstract syntax trees of the two versions of the program and by finding their differences. Since an AspectJ program is composed by a plain Java code base and some aspect-oriented code units, one can apply the different catalogs of atomic changes separately.

In general, a given change may depend on some previous one. Intuitively, an atomic change A_1 is dependent on another atomic change A_2 if applying A_1 to the original version of the program without also applying A_2 results in a syntactically invalid program: in order to be possible to change a method (CM), that method has to exist: if it was added by the new version (AM), the atomic change CM depends on the specific AM.

Three types of dependence can be considered: *structural,* which captures the necessary sequences that occur when new elements are added or deleted in a program; *declarative,* which captures all the necessary element declarations that are required to create a valid intermediate version; *mapping,* which captures implicit dependencies introduced by changing the class hierarchy or overriding methods. An example of dependencies between atomic changes is shown in Figure 1. Here we report the dependencies computed for the example of Section 2. In the picture the arrows denotes the interdependencies between changes. The overriding of the methods in the class PointExt is broken into four atomic changes: the addition of an empty method (AM) and the corresponding modification of the method body (CM). The latter is structurally dependent on the former. Moreover we also have two changes for the virtual methods lookup: let's consider for example the overriding of the method setX. Each time someone is going to call the method setX on a reference statically typed as Point, if the runtime object will be of type PointExt the call will reach the method defined in the subclass. The same will happen if the reference has as static type PointExt. These changes are represented as lookup changes (LC) and the addition of the method setX in PointExt has a mapping dependency on them. The changes in the aspect BoundPoint are decomposed in an added field (AF) and a changed advice body (CAB) changes. The latter is declaration dependent on the former, because it uses the field definition added by the AF change.

3.2 Control flow representation of the program

To capture the differences in behavior we need to build a representation of the program. To accurately model AspectJ semantics, we use a control-flow representation, the AspectJ Inter-module Graph (AJIG), presented in [15]. This representation is an extension to aspect-oriented encapsulation units of the Java Interclass Graph [5]. The main goal of this graph is to make explicit the interactions between the base system and the aspect part of an AspectJ program with particular regard to the weaving of multiple advice at the same join point.

The AJIG is designed to represent precisely all interactions involving the pieces of advice that are anonymous pieces of code analogous to object-oriented methods that are executed when a specific dynamic join point occurs: such interactions are at therefore the core of aspect-oriented programming. Unlike explicit method calls, an advice is invoked implicitly at the shadow of a certain join point. The execution of the advised Java code is completely replaced by the combination of pieces of advice and the join point shadow that matches it. Before- and after- advice can be considered as special cases of around-advices with an implicit **proceed** statement. Thus, for each advised piece of Java code its control-flow subgraph is replaced with the representation of woven pieces of advice called Interaction Graph (IG).

To create an interaction graph we need to compute the precedence the woven pieces of advice are executed with, and to do this the AJIG uses an *advice nesting tree*, which represents the run-time advice nesting relationships. Each tree level contains at most one around-advice, which is the root of all pieces of advice in the lower levels of the tree. With each around-advice A the algorithm associates (1) a possibly-empty set of before-pieces of advice and after-pieces of advice, (2) zero or one around-pieces of advice, and possibly (3) the actual call site that could be invoked by the call to **proceed** in A. These pieces of advice and the call site appear as if they were nested within A.

For example, a call to Point.setX gives the advice nesting tree shown in Figure 2. The information captured by the advice nesting tree is used to weave advice bodies at a shadow and can be exploited to build the AspectJ Interaction Graph. A condensed picture of the AJIG for Point.setX is shown in Figure 3.

The AJIGs built for the two versions of the program are then compared to find *dangerous edges,* i.e., edges that are different in the two versions. The algorithm for comparing AJIGs is a depth first traversal of the graphs that aims at finding the differences between paths in the graphs and it was proposed and described in detail in [15]. Basically, if an edge is present in the first graph and is missing in the second or if it was added in the second or if the edges of the path changed their label, then they are marked as "dangerous". An example of a dangerous path is shown in Figure 4. The example refers to the AJIG of Figure 3 and it corresponds to the impact of adding a field and modifying the body for advice before2 in BoundPoint aspect and overriding the method setX in the class PointExt. There are two dangerous edges (marked in red) in the figure. The first one is the one

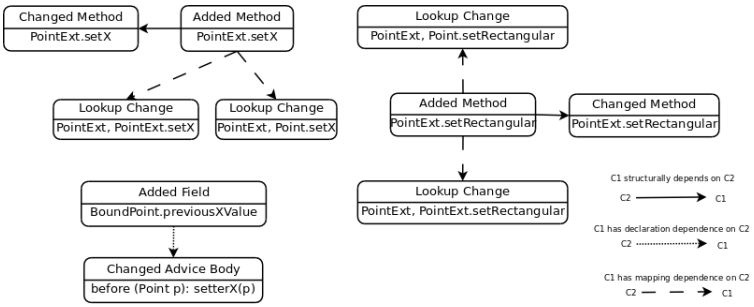

Figure 1: Representation of atomic changes dependencies for the example

Figure 2: Advice nesting tree for BoundPoint at shadow this.setX() in the example presented in section 3.1

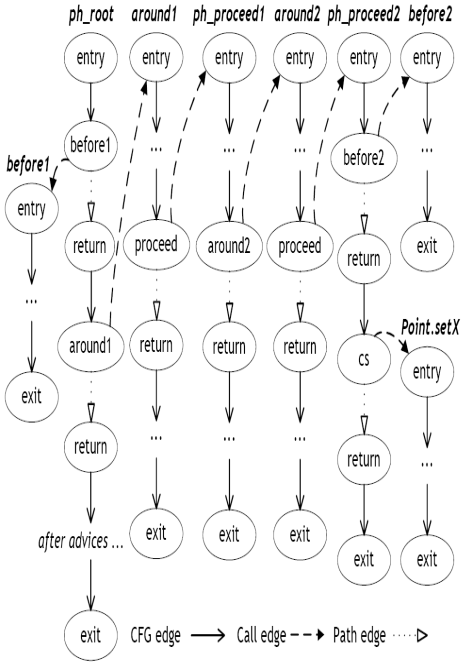

Figure 3: AspectJ Interaction Graph for the example presented in section 3.1

starting from the before2 entry point and determined by the addition of the statement **this**.previousValue = p.getX() in Listing 3. The second one is determined by the addition of the path going through the method PointExt.setX, overridden in class PointExt.

3.3 Combining dynamic information

Dangerous edges are only a potential cause of an unwanted effect. However, the execution of a test case that gives unexpected results can be *explained* in terms of dangerous path traversals. In other words, we use the information gathered from the instrumentation of the program to consider only dangerous paths executed by a test case and we map these paths on the atomic changes computed earlier. The mapping takes place by considering that nodes in the paths interested by dangerous edges represent instructions in the program, so we perform the mapping considering their line numbers in the compilation units they come from. At this point we have a set of atomic changes, representing syntactical changes in the source code of the program, mapped on a set of dangerous edges in the program control flow representation, which represents changes of behavior of a test case. This gives the causal chain between the editing and the unexpected result: the set of changes mapped onto the traversed dangerous edges are responsible for the observed behavior. In fact, undoing the atomic changes set will produce a version of the program that will not show anymore the altered test result. Moreover, by removing the set of changes, we would have a syntactically correct intermediate program version, since atomic changes interdependencies consider syntactical dependencies between changes. Atomic changes not mapped on dangerous edges for a given test are not responsible for the change of that test result and can be left in the modified version of the program. Finally we can consider that changes not mapped on any test case dangerous edges are not stressed at all by the test suite.

4. IMPLEMENTATION

For the implementation of our solution we relied on the *abc* compiler [1] and on AJANA framework [16].
The *abc* compiler is an extensible compiler for AspectJ based on Soot [12]. *abc* gave us the possibility to manipulate the abstract syntax trees of the programs we analyzed and to find atomic changes. Moreover it provided us with an easy to analyze intermediate representation of the program, Jimple, which we used to build our AJIG. Finally *abc* performs a two phases weaving process. In the first phase it computes the pointcuts shadows in the system code but keeps the weaving information separated from the system bytecode, then it performs some optimizations and finally produces code to insert pieces of advice at the proper shadow. The possibility to keep weaving information separated from the code gave us way to build our AJIG witout considering the extra code that AspectJ compiler generates to weave an advice at a shadow. In this way we could produce a more accurate representation of the program control flow.
AJANA is a general framework for ApectJ analysis. It provided us the AJIG representation. We had to slightly modify this representation to implement our change impact analysis, in order to keep track of some pieces of information originally not needed for AJIG construction.

5. RELATED WORK

Change impact analysis for object-oriented programs was introduced by Ryder et al. in [10]. In their paper they proposed a technique for Java software and in [9, 8] they

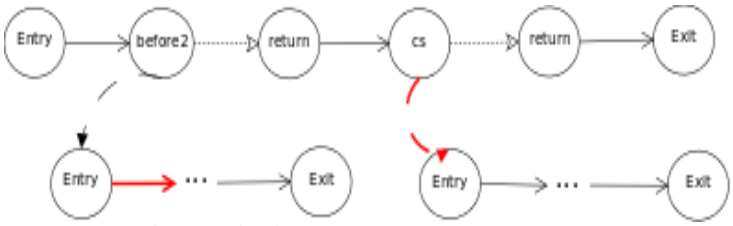

Figure 4: Dangerous edges determined by modification of before2 advice body in class BoundPoint and overriding of method setX in PointExt class

extended the applicability of the technique and provided tool support.

The problem of change impact analysis for AspectJ software was previously treated in [18]. This paper presents a lightweight approach to change impact analysis based on a static program representation. We based our approach on a dynamic representation of the program which captures the actually executed paths in the control flow. Since from our previous work [3] we noticed that for AspectJ static representations introduces a considerable quantity of dependencies that are not to be considered at run time we think that our approach could lead to an increased precision in the analysis.

Change impact analysis is related with regression testing and debugging. We based our work on a framework developed for a regression test selection technique [15]. Many other techniques were proposed for aspect oriented programs regression testing. In [13] an approach based on a wrapper class synthesis technique and a framework for generating test inputs for AspectJ programs are presented; in [14] the authors propose an approach to generate regression test cases starting from a specification of the program. All these approaches can be considered as complementary to ours, since they focus on discovering if a failure was introduced by changes while we want to unwind the relation between source code changes and failures found in the program.

Delta debugging was originally introduced by Zeller in [17]. This technique allows to create intermediate versions of the program by adding or removing a set of atomic changes from the program source. Change impact analysis for Java programs was successfully integrated with delta debugging in [9], we plan to do the same in our future work.

6. CONCLUSION AND FUTURE WORK

In this work we developed a prototype to perform change impact analysis on AspectJ programs. Our prototype can relate changes in the source code of an AspectJ program to changes in its behavior. The tool is based on a modified version of the *abc* aspect weaver and the AJANA framework. We tested our prototype on simple examples of evolving AspectJ programs that advise call and execution join points. Since our first results are promising, we plan to extend our experimentation to real world programs. Moreover, we think that the level of abstraction of atomic changes is too low in most cases, even if it worked well in our toy examples. In fact, Ryder et al. report in [11] that Java change impact analysis based only on a syntactical classification produces changes sets that are difficult to be interpreted by humans. Thus, we plan to study some higher level view to give to the programmer a more effective means of understanding his or her changes.

7. REFERENCES

[1] P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: an extensible AspectJ compiler. In *Proc. of AOSD'05*.

[2] S. A. Bohner. Software change impact analysis. *Wiley-IEEE Computer Society Pr*, 1996.

[3] A. C. D'Ursi, L. Cavallaro, and M. Monga. On bytecode slicing and AspectJ interferences. In *Proc. of FOAL'07*.

[4] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. Technical report, RIACS, 2000.

[5] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for Java software. In *Proc. of OOPSLA'01*, 2001.

[6] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proc. of ECOOP'01*, 2001.

[7] M. Marin, L. Moonen, and A. van Deursen. An integrated crosscutting concern migration strategy and its application to JHotDraw. In *Proc. of SCAM'07*, 2007.

[8] X. Ren, O. Chesley, and B. G. Ryder. Identifying failure causes in Java programs: An application of change impact analysis. *IEEE TSE.*, 32(9), 2006.

[9] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: a tool for change impact analysis of Java programs. In *Proc. of OOPSLA'04*, 2004.

[10] B. G. Ryder and F. Tip. Change impact analysis for object-oriented programs. In *Proc. of PASTE'01*, 2001.

[11] M. Stoerzer, B. G. Ryder, X. Ren, and F. Tip. Finding failure-inducing changes in Java programs using change classification. In *Proc. of FSE-14*. ACM, 2006.

[12] R. Vallée-Rai, L. Hendrena, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot – a Java optimization framework. In *Proc. of CASCON'99*, 1999.

[13] T. Xie and J. Zhao. A framework and tool supports for generating test inputs of aspectj programs. In *Proc. of AOSD'06*. ACM, 2006.

[14] D. Xu and W. Xu. State-based incremental testing of aspect-oriented programs. In *Proc. of AOSD'06*, 2006.

[15] G. Xu and A. Rountev. Regression test selection for AspectJ software. In *Proc. of ICSE'07*, 2007.

[16] G. Xu and A. Rountev. AJANA: a general framework for source-code-level interprocedural dataflow analysis of AspectJ software. In *Proc. of AOSD'08*, 2008.

[17] A. Zeller. Yesterday, my program worked. today, it does not. why? In *Proc. of ESEC'99*, 1999.

[18] S. Zhang, Z. Gu, Y. Lin, and J. Zhao. Change impact analysis for AspectJ programs. In *Proc. of ICSM 2008*.