Using Atomic Changes to Explain Pointcut Deltas

Maximilian Stoerzer  
University of Passau  
stoerzer@fmi.uni-passau.de

ABSTRACT
Aspect oriented programming has been proposed as a way to improve modularity of software systems by allowing encapsulation of cross-cutting concerns. To do so, aspects specify where new functionality should apply using pointcuts. Unfortunately expressions written using today's mainstream pointcut languages are fragile, as non-local changes may easily change pointcut semantics. This is a major obstacle for evolution of aspect oriented software.

This technical report extends pointcut delta analysis proposed in earlier work to lighten these problems and allow programmers informed decisions if matching behavior changed expectedly or not.

1. INTRODUCTION
One of the main goals of aspect-orientation is to improve the modularity of a system by allowing to encapsulate cross-cutting concerns thus facilitating system maintenance. On the one hand this goal can be reached, as the crosscutting concern is indeed localized in the aspect, and additionally the tangled code is removed from the base system, thus improving internal cohesion of base modules. On the other hand, aspect and base are tightly coupled and furthermore this coupling is only implicit as it depends on the results of the pointcut expressions.

In earlier work [18, 19] we analyzed this problem and termed it the fragile pointcut problem. We further proposed a pointcut delta analysis to address this problem. This analysis is now extended to also capture base code edits as reasons for pointcut deltas.

2. POINTCUTS AND COUPLING
An important quality criterion for software systems is their modularity. Modular systems tend to be easier to understand and maintain as they localize a single concern in a single module and optimally allow to independently change different modules. This is especially important as estimated 80% of the total cost during the lifetime of a software system stem from the maintenance phase. Rules of thumb to achieve a good modularity are to provide modules which have a high internal cohesion and are loosely coupled among each other.

Crosscutting concerns per definition are concerns not properly modularizable in a single place by using traditional programming techniques. Implementing such a concern using an aspect promotes the modularity of the system, as both the crosscutting concern is localized in the aspect and the cohesion of those modules where the tangled code is removed from is improved as well. However, this improvement of modularity is not without cost. A major problem of current aspect languages is that aspect and base tend to be tightly coupled. Even worse, this coupling is only implicit, as aspect influence is not directly visible in the affected base modules.

These problems in part have been discussed in literature, for example refer to [8]. We will analyze the crosscutting mechanism in AspectJ-like languages in the following to give a detailed overview of the problem.

2.1 Pointcuts and Aspect-Base Coupling
The ability to reason about a program to select joinpoints has been identified as a core property of aspect-orientation and termed quantification in [3]. Recall that aspects in general provide two constructs to specify new behavior and where it should apply, called advice and pointcuts in AspectJ. Pointcuts are expressions allowing the programmer to specify where advice should be executed. So pointcuts implement this quantification property. The aspect weaver evaluates pointcut expressions and uses this information to combine advice with the base system and so finally produce the executable system. If we follow [3] and accept quantification as a core property of aspect-orientation, then any aspect-oriented language has to provide a comparable mechanism.

Although there is a broad spectrum of different joinpoint selection languages, a pointcut expression can be seen abstractly as a function taking the (base) program as input and calculating a set of matched joinpoints as output. The main differences are which joinpoints can be selected and—more important for system evolution—how this function can be specified. The latter describes the expressiveness of a pointcut selection language.

Optimally a pointcut expression by itself should transport its semantics, i.e. one would like to write a pointcut like “update the observer whenever an observed subject changes”. If today's systems would allow programmers to write such pointcuts, then indeed aspects relying on these pointcuts would be semantically stable, and also easy to formulate. For
an observer aspect for example this means that the observer is notified of all relevant subject changes.

Unfortunately many—if not all—of todays main stream joinpoint selection languages only allow to select joinpoints based on lexical or syntactical properties of the code. Pointcuts explicitly name elements in the code to address joinpoints. I.e. the programmer has to specify that the observer has to be updated after a call to method setPos(...) in one of the observed objects.

As it is rather inconvenient to specify a larger set of joinpoints by explicitly “naming” each joinpoint, AspectJ introduced wildcards allowing to exploit naming conventions. So it is possible to formulate pointcuts like “update the observer whenever a set*(..) method is called on one of the observed subjects”. However, using wildcards results in a new problem. Pointcuts using this mechanism exploit naming conventions. As such conventions are hard to check by a compiler, they are never guaranteed.

Naming and wildcards are natural things for programmers, so these pointcut languages are relatively intuitive and convenient to use (for programmers!). However, there is also an important drawback: specifying joinpoints by using wildcards and exploiting naming conventions can easily result in unwanted or accidentally lost matches. So developing a correct pointcut not trivial. For small programs a pointcut mismatch due to wildcards can easily be seen. However, aspects have been proposed for large or distributed system scenarios, where it is much harder to find spurious or missed matches. In general, the aspect programmer needs global system knowledge to assure that his pointcut works as expected.

The problem is that joinpoint selection based on lexical as well as syntactical or structural properties of a program do not allow to write pointcuts like “update the observer whenever an observed subject changes”—the programmer rather has to (lexically) enumerate all relevant joinpoints. Wildcards are just a (dangerous?) means to reduce the effort of writing the pointcut down. They do not provide additional robustness for pointcut definitions as even strict coherence to naming conventions does not guarantee that no additional joinpoints are selected.

This enumeration of joinpoints in direct consequence results in aspects tightly coupled with a specific version of the base system. This is an important observation: while aspect-orientation allows to improve internal cohesion of modules it introduces additional modules tightly coupled with the rest of the system.

To reduce this tight coupling, aspect inheritance is a way to split advice and pointcut definitions. Advice is defined in the abstract aspect, and pointcuts are specified later in a concrete sub-aspect. However, while this approach allows to define reusable functionality in an aspect, the concrete aspect is still tightly coupled with the base system. Thus aspect-oriented systems only provide half the solution they promised to offer—they allow to localize crosscutting concerns in separate modules with a high internal cohesion. However for aspect-oriented systems there is always the problem of a high aspect-base coupling.

2.2 Evolution in AO Systems

While the tight coupling of aspects up to now has been discussed from a theoretical point of view, analyzing its concrete effects for evolution of aspect-oriented systems is necessary. In the above we compared pointcut expressions with a function using a given (base) program as input and producing the set of selected joinpoints as output. While for functions in general one expects changed results for changed inputs, for pointcuts there are important differences. While one might even expect changes in the set of selected joinpoints in this context, one does not expect changes in the aspect semantics. Or, in the context of the above, the actual pointcut results should optimally change to match the original pointcut idea. Unfortunately, the actual pointcut results are not cooperative in general, but instead show some inbetween mixture of expected and unexpected additional and lost matches. We will justify this claim in the following.

Let us once again interpret a pointcut as a function. The set of matched joinpoints depends on the whole program as an input to this function, i.e. an aspect is potentially coupled with all modules of the system. Although this seems exaggerated, this actually is the case, if the pointcuts are not restricted to specific parts of the system. In general a pointcut is an all-quantified expression over a program—and thus indeed its semantics depend on the program as a whole. Even worse, some pointcut languages, including AspectJ, allow to specify pointcut expressions which only result in the selection of a joinpoint if another joinpoint has been previously selected (cmp. cflow). Consequently, to evaluate the pointcut expression, one has to actually calculate a fixpoint. This is clearly opposed to the statement that the result of a pointcut expression in general is “obvious” for the programmer.

For these complex functions, a change in any code artifact of the system can result in different results, manifested as additional or lost matched joinpoints, so influencing semantics of the system as a whole. The same effects are possible by mere addition of new program elements; so there is even no closed world pointcuts reason on.

Example 2.1 (System Evolution Scenario). For illustration consider the following scenario: A programmer might have correctly specified a pointcut (i.e. pointcut idea, expected and actual pointcut results are the same). The corresponding aspect works as intended, all tests are successful. Afterward the base code evolves, e.g. by trivial changes like renaming some methods, changing some method signatures and adding new methods.

If the programmer misses to update calls to changed methods, the compiler will issue a compile error. However, if we consider a pointcut referencing a method by its former name or by its former signature the set of joinpoints picked out by the—unchanged!—pointcut definition is silently altered. As a consequence, we now have a mismatch in pointcut ideas and actual pointcut result. The pointcut expression is no longer a valid implementation of the original pointcut idea.

In general there are several (trivial) non-local base code changes possibly modifying pointcut semantics in terms of actually selected joinpoints, for example:

Rename: Renaming classes, methods or fields influences semantics of call, execution, get/set and other pointcuts. Wildcards can only provide limited protection against these effects.
Move method/class: Pointcuts can pick out joinpoints by their lexical position, using within or within-code. Moving classes to another package or methods to another class obviously changes matching semantics for such pointcuts.

Add/delete method/field/class: Pointcut semantics are also affected by adding or removing program elements. New elements can (and sometimes should) be matched by existing pointcuts, but in general pointcuts cannot anticipate all possible future additions. Removal of program elements naturally results in 'lost' joinpoints.

Signature Changes: call- and execution-pointcut designators allow to pick joinpoints based on method signatures including method visibility. Thus signature-based pointcut definitions—although propagated as the more robust mechanism—are nonetheless fragile.

If a code artifact is changed, other artifacts depending on it in general have to be adapted. (Automated) refactoring based pointcut definitions—although propagated as the more robust mechanism—are nonetheless fragile.

We will address this mismatch with the term "fragile pointcut problem." In this chapter we show how tool support can help to lighten this problematic in general as dynamic pointcut designators (like the if- or cflow-constructs) cannot be evaluated statically. Thus in general it is not feasible to determine if a specific dynamic match has been added or lost and thus if a pointcut has to be adapted by a refactoring or not.

Additionally, automated refactoring requires that the user explicitly requests a refactoring, thus refactoring does not address system evolution in general. As an example just consider adding new methods, classes or packages due to new functionality. In these cases, a refactoring tool cannot help, as no existing code artifacts are modified and thus per definition no refactoring of an existing code artifact has been requested. But as we have seen above, addition of new code artifacts might result in additional adapted joinpoints and thus in changed aspect-program semantics. Finally, the new set of relevant joinpoints in terms of the pointcut idea in general is not formally specified, i.e. for the refactoring the set of joinpoints to select in general is not computable.

We in detail discussed the problem of aspect-oriented software evolution and state that this problem is one of the major problems of aspect-orientation today. Support for System Evolution is crucial for the long-term usability of any languages, as a major share of total costs of software development arise during software maintenance. For AspectJ and similar languages, on the languages level this problem is in general unsolved (and may not be solvable at all).

We refer to the problem of changed program semantics due to a mismatch in pointcut ideas and actual pointcut results by using the term "fragile pointcut problem." In this chapter we show how tool support can help to lighten this particular problem, by recapitulating and extending our own previously published results.

3. CALCULATING CHANGES

In this technical report we will show how pointcut deltas can be calculated to make changed matching behavior visible and furthermore how these deltas can be explained by relating them to changes between the original and the edited version. Therefore, we need to decompose the respective edit to a set of atomic changes, similar to the change impact analysis of Chianti introduced in [14]. There, an edit is decomposed into a set of atomic changes with a granularity roughly at the method level. In this section we show, how an AspectJ program edit can be decomposed into a set of atomic changes.

From a theoretical point of view, AspectJ programs are a superset of Java programs. As such, all change categories present in Java can also occur in AspectJ programs. However, due to the aspect-oriented language extensions in AspectJ, we also need some additional change categories compared to [14]. The following briefly reviews the new language constructs in AspectJ (compared to Java) and discussed how we derive atomic changes for them, if appropriate. For a set of changes derived for plain Java, please refer to [14].

3.1 Handling Inter Type Declarations

Handling of inter type declarations in terms of changes is straightforward, as inter type declarations simply add new methods and fields to a class, although not in its lexical scope. Consequently, we generate add method AM, change method CM, add field AF, add field initializer AFI, and finally lookup changes LC for inter type declarations accordingly. Similarly to [14], LC-changes model changes in the behavior of dynamic dispatch in case overriding or overloading methods are added. To calculate these lookup changes we use a techniques as published in [20]. This modeling reduces inter type declarations to traditional member additions, and thus allows to easily handle them.

3.2 Hierarchy Changes

Besides inter type declarations, static crosscutting in AspectJ also allows to change the direct superclass and add additionally implemented interfaces (using the declare parent... extends construct). This is however also possible by using traditional source code modifications.

The Chianti system currently does not capture such changes in the type hierarchy directly, but reports CM changes for methods containing instanceOf statements referring to the modified classes. However, as these changes are relevant for aspect-oriented programs—just consider pointcut semantics containing within, this or target predicates—we additionally explicitly model such changes to capture potential semantical changes. Table 1 gives an overview over the new change categories introduces to record respective changes.

We generate a respective change each time the superclass (ARS, DRS), super aspect (ARA, DRA) or the set of implemented interfaces (ARI, DRI) changes, either due to static crosscutting or due to regular edits of extends or implements-statements in the source code.

We also report virtual CP (change pointcut) changes for pointcuts now potentially selecting additional or less joinpoints (using the 'TypeName+' syntax). These CP changes syntactically depend on the hierarchy changes.
Table 1: New change categories to capture Hierarchy Modifications

<table>
<thead>
<tr>
<th>Category</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARI</td>
<td>Add Reference to Interface</td>
</tr>
<tr>
<td>DRI</td>
<td>Delete Reference to Interface</td>
</tr>
<tr>
<td>ARS</td>
<td>Add Reference to Superclass</td>
</tr>
<tr>
<td>DRS</td>
<td>Delete Reference to Superclass</td>
</tr>
<tr>
<td>ARA</td>
<td>Add Reference to Aspect</td>
</tr>
<tr>
<td>DRA</td>
<td>Delete Reference to Aspect</td>
</tr>
</tbody>
</table>

Table 2: New change categories to capture Hierarchy Modifications

<table>
<thead>
<tr>
<th>Category</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>AA</td>
<td>Add an empty aspect</td>
</tr>
<tr>
<td>DA</td>
<td>Delete an empty aspect</td>
</tr>
<tr>
<td>AAD</td>
<td>Add Advice Definition</td>
</tr>
<tr>
<td>CAH</td>
<td>Change Advice Header</td>
</tr>
<tr>
<td>CAB</td>
<td>Change Advice Body</td>
</tr>
<tr>
<td>DAD</td>
<td>Delete Advice Definition</td>
</tr>
<tr>
<td>AP</td>
<td>Add new Pointcut Definition</td>
</tr>
<tr>
<td>CP</td>
<td>Change Pointcut Definition</td>
</tr>
<tr>
<td>DP</td>
<td>Delete Pointcut Definition</td>
</tr>
</tbody>
</table>

3.3 Pointcuts and Advice

Besides static crosscutting constructs we have to handle changes in pointcuts and advice. As advice is a method-like construct, it is tempting to handle advice similar to methods. This is however insufficient, as for advice two components have to be considered: the advice header and the advice body. The advice header contains information about the available joinpoint context and the pointcut this piece of advice is bound to.

The advice body is similar to a method body and is also handled similarly. If the advice body is changed, we generate a CAB change (change advice body). However, if only the binding in the advice header is changed, we generate a CAH change (change advice header) instead. Similar to methods, we also have change categories to indicate addition or removal of a piece of advice (AAD (add advice definition) and DAD (delete advice definition), respectively).

Pointcut definitions can occur as a part of the advice header or as individual aspect elements. Consequently we also have change categories to indicate addition, change, and removal of pointcut definitions, AP, CP, and DP, respectively.

3.4 Aspects

Besides classes AspectJ finally also introduces aspects as modules containing these new constructs. Changes for new or removed aspects are generated similarly to classes, i.e. we generate AA and DA changes (add, delete aspect). Table 2 gives an overview of changes induced by pointcuts, advice and aspects.

Note that these changes—equivalently to the changes discussed in the Chianti work—also have dependencies among each other. If for example a piece of advice is deleted, we not only generate a DAD change, but also a CAB and a CAH change, where the DAD change depends on, to illustrate that first the advice body and the advice header has to be deleted, before this piece of advice can be removed.

Similar to Chianti, a change a1 is syntactically dependent on a change a2, if adding a1 without also adding a2 yields a syntactically incorrect AspectJ program. As these simple syntactic dependences are straightforward we will not discuss them any further here.

The calculation of atomic changes A between an original program P and an edited program P′ as presented in this section has been implemented by Jurgen Graf as a part of his master thesis [5].

3.5 Other AspectJ Constructs

Note that there are other language constructs in AspectJ which we did not explicitly discuss up to now. These include declare precedence, declare soft and declare error/warning. The declare error/warning construct results in compiler errors or warnings, but has no semantics in terms of runtime behavior and is thus not relevant in this context.

The declare soft statement however can be used to virtually convert a checked Java exception into a runtime exception, thus changing the type and error handling behavior for this exception. While defining respective changes is simple, calculating there semantical effects requires to analyse control flow graphs. Capturing these effects using call graphs however is difficult, and has thus been excluded for the following.

4. POINTCUT DELTA ANALYSIS

We advocate tool support for aspect-oriented programming, and in the following a delta analysis technique addressing the fragile pointcut problem is proposed. The approach presented here addresses the fragile pointcut problem for current languages and systems written in these languages. It is intended to ease maintenance and avoid rising costs for systems written in languages similar to AspectJ which suffer from fragile pointcut constructs. Although future pointcut languages might lighten the problem by narrowing the gap between intended pointcuts and pointcut constructs in aspect languages, completely closing the gap might not be possible. Thus the proposed analysis will still be helpful allowing to double-check programmer expectations and actual pointcut semantics.

The basic idea for the approach presented here is that programmers should be alerted of changes in the matching behavior of advice (i.e. the expected and actual pointcut results) if the underlying system changes. Unfortunately this approach is necessarily incomplete as there is still no information about the set of joinpoints to-be-matched to maintain aspect semantics. But focusing on actual changes can also help to check for expected changes in a pointcut set. While this is no perfect solution, it considerably improves the situation compared to manual inspection of matching behavior in the new program version.

4.1 Detecting Failures and their Reasons

Assume that due to joinpoint mismatches a new program version suffers from a bug. Although unintended semantical differences introduced into a system are (hopefully) revealed by rerunning a regression test suite (failing test), in general the programmer needs more information. Test failures do not explain failure reasons. Thus for a failing test, test results (e.g. an exception) have to be further analyzed to actually track down the bug.
Listing 1: Original and edited program version. Code added in the edited program version is underlined, code canceled out is deleted in the edited program version.

```java
aspect A {
    pointcut dynamic():
    within(C) && if(isTrue()) {
        pointcut setField(): set(Int *)
        & & dynamic();
    }
    before(): setField() {
        System.out.print("Changing field value");
    } /*[161-0-1108388203184-8132904]*/
    after(): call(* update()) && if(isTrue()) {
        System.out.print("Field update done");
    } /*[276-1-1108388538145-4535112]*/
    private boolean isTrue() {
        ... // some dynamic predicate
    }
}

class C {
    int x;
    static void main(String[] args) {
        D d = new D();
        d.setX(4);
        d.setX(5);
        d.update();
        void setX(int x) {
            this.x = x;
        }
        void update() {
            x = 2 * x;
        }
    }
}

class D extends C {
    void setX(int x) {
        this.x = x;
    }
    void update() {
        x = 2 * x;
    }
    /* deleted in edited version */
}
```

Finding failure inducing code modifications is hard if changed pointcut semantics due to non-local base edits are responsible, especially as a trace—if available at all—not necessary points to the failure inducing edit. If a joinpoint mismatch is failure inducing, actually finding this mismatch is very hard when examining a single program version. A programmer has to be aware of affecting aspects and their semantics to reason about system correctness. In large systems doing this manually is infeasible.

For the remaining of this chapter we will use the program shown in figure 1 as a running example to demonstrate our delta analysis. The figure shows both versions of example code. The underlined code is added, the code canceled out removed in the edited program version. Note that the two pieces of advice are marked up with a special kind of comment, which gives us a unique identifier for advice. We will discuss it’s necessity in section 4.3. We will compare those two versions using our pointcut delta analysis in the follow-

The difference between these two versions includes moving a method (update from class D to class C), modification of pointcut setField and addition of a new piece of advice. Although this program is tiny, the resulting changes in advice matching behavior are already hard to see without support.

4.2 Calculating PC-Deltas

Once a problem is known, it is often half solved. This is especially valid for pointcuts unexpectedly changing their semantics due to e.g. base code edits. Consider the following scenario: we have two versions of a program: an original version \( P \) and an edited program version \( P' \). To detect semantical differences in program behavior due to changed pointcut semantics, we propose an analysis which detects changes in matching behavior (called pointcut delta) and also—partly—traces these differences back to their corresponding code modification(s).

To derive the pointcut delta the following analysis is used: Informally, we calculate the set of matched joinpoints for both versions of the program and compare the resulting sets, producing delta information for pointcut matching. This approach is possible for any AspectJ-like language where the set of matched pointcuts is (at least partly) statically computable.

For cases where joinpoint matching cannot be decided statically, the matching is conservatively approximated and the resulting match marked accordingly. As the weaver also needs this information to generate the woven program, static matching information in general is available for most if not all compiled languages. Dynamic pointcut expressions result in runtime checks associated with runtime penalties, which are avoided if joinpoint matching information is statically known. For purely dynamic approaches however this might not be true. This is clearly one of the limitations of the approach presented here.

So, for a given aspect-oriented program \( P \), function

\[
\text{match : } P \rightarrow JP \times ADV \times Q
\]

determines the set of all aspect-joinpoint relations, where

- \( JP \) is the set of all joinpoints in \( P \),
- \( ADV \) is the set of all advice in \( P \) and
- \( Q \) is the quality of the matching relation, either dynamic or static.

4.3 Defining an Equality Relation for Pointcuts and Advice

To calculate the delta from \( \text{match}(P) \) and \( \text{match}(P') \) it is necessary to identify corresponding joinpoints and advice in both versions of the program, \( P \) and \( P' \), respectively. More formally speaking we need equality joinpoints and advice defined for joinpoints (or better on joinpoint representations) and advice of both program versions.

However, while this is trivial for methods, both joinpoints and advice are unnamed constructs (at least for AspectJ) and thus matching is problematic. What is needed is an identifying representation for joinpoints and advice which is stable across different versions, comparable to a method signature.
The lexical position of a joinpoint/advice in the source code ("source handle") can be used to identify a joinpoint in a given system version. Unfortunately this is no longer true if subsequent versions are considered, as functionally irrelevant modifications like adding some blank lines or comments change the joinpoint/advice source position and thus makes identification of corresponding items in both program versions impossible. Reordering of code artifacts in a file raises similar problems.

For advice this problem can be solved by (automatically) naming a new piece of advice once it is introduced in the system, as advice is a first class item in a program. This can be done by simply attaching an identifying comment to each advice body when a piece of advice is first encountered (as also visible in figure 1). While naming as a standard solution reliably solves this issue for advice, joinpoints are more complicated as they are no first class code artifacts. A joinpoint is only implicitly defined by the program statements.

However, similar to method signatures it is possible to identify joinpoints using joinpoint signatures composed of relevant code artifact signatures at the joinpoint. For example a call-joinpoint can be identified by the signature of the caller method, called method and a counter; similar a field set/get is identified by the accessed field and the, for example, the method-signature the access is located in.

Joinpoint signatures consist of three parts: (i) the joinpoint type, (ii) a referenced program element and (iii) a containing program element.

Definition 4.1 (Joinpoint Signatures). For a joinpoint \(jp\)

- \(env(jp)\) indicates the program item containing this joinpoint,
- \(ref(jp)\) indicates the program item referenced by this joinpoint,
- \(kind(jp)\) indicates the joinpoint kind.

For a detailed descriptions how joinpoint signatures are derived for the AspectJ joinpoint model refer to table 3.

Example 4.1 (Joinpoint Signatures). Consider the joinpoint \(jp\) represented by the call to \(D.setX(int)\) in \(C.main(String[])\) in the original program version of listing 1. The signature of this joinpoint is \((call.D.setX(int),C.main(String[])), \) Respectively, for this joinpoint, \(env(jp) = C.main(String[])\), \(ref(jp) = D.setX(int)\), and \(kind(jp) = call\).

Note that this joinpoint identification scheme is a heuristic, as in general a single method can contain multiple calls to the same callee all forming different joinpoints. In this case joinpoint signatures are not able to distinguish these joinpoints (apart from a counter).

Example 4.2 (Failure of Joinpoint Signatures). Consider the edited program version shown in listing 1. In method \(C.main()\) a second call to \(D.setX()\) has been inserted before the call to \(setX()\), also present in the original program versions. This additional call forms a new joinpoint which is assigned the same signature as the original call. As is has been inserted before the original call, the joinpoint signature will match the signature of the call in the original program.

To deal with multiple joinpoints with the same signature, we augment the joinpoint signatures with a trailing counter. This counter allows to keep track of the multiplicity of same-signature joinpoints. This allows us to detect changes in the number of same-signature joinpoints, although it is not possible to directly match respective joinpoints in two versions, as is also the case in the above example.

Although only a heuristic, this model of joinpoint identification using joinpoint signatures is descriptive enough for our purposes,

- as we keep track of the number of matched joinpoints by adding a counter to the respective joinpoint signature, thus enforcing unique identifiers and
- as the joinpoint model of AspectJ itself is not able to directly distinguish such joinpoints either.

Thus in practice this identification schema works very well, even if this model cannot guarantee that indeed equal joinpoints are identified by the same signature in both versions. The relevant information for the user—concerning the above example—is that in method \(C.main()\) an additional joinpoint is matched, potentially affecting system semantics.

With these two notions of equality for advice and joinpoints across different program versions it is now straightforward to calculate the delta set for \(match(P)\) and \(match(P')\) by using standard set operators, assuming that all matching information is statically known.

4.4 Dynamic Pointcut Designators

Up to now we did not explicitly consider dynamic pointcut designators. For these designators, the set of selected joinpoints can not be completely evaluated at compile time. Examples are the \(if\) or \(cfollow\) pointcut designators. Statically one has to conservatively approximate these constructs by assuming \(true\) for each such predicate as evaluation of pointcut expressions requires runtime values.

For the delta analysis this results in the comparison of supersets rendering the derived information less reliable. To deal with this problem we refine the delta analysis to exploit the associated matching quality information (static/dynamic) and mark up resulting delta entries correspondingly. By adding this knowledge six different cases can be distinguished:

New matches: A new statically determined advice association appeared in \(P'\):

\[
new_{static} = \{ (jp, adv, static) \mid \exists (jp, adv, static) \in match(P') \land \exists (jp, adv, \bullet) \in match(P) \}
\]

New potential matches: A new advice association has to be conservatively assumed in \(P'\), although evaluation is not possible at compile time:

\[
new_{dynamic} = \{ (jp, adv, dynamic) \mid \exists (jp, adv, dynamic) \in match(P') \land \exists (jp, adv, \bullet) \in match(P) \}
\]

Lost matches: A statically determined advice association is no longer present in \(P'\):

\[
lost_{static} = \{ (jp, adv, static) \mid \exists (jp, adv, static) \in match(P) \land \exists (jp, adv, \bullet) \in match(P') \}
\]

In the following, \(\bullet\) will indicate any possible value for a tuple variable (wildcard).
lost potential matches: A conservatively assumed advice association is no longer present in \( P' \):

\[
\text{lost}_{\text{dynamic}} = \{ (jp, adv, \neg \text{dynamic}) \mid \exists (jp, adv, \text{dynamic}) \in \text{match}(P) \land \exists (jp, adv, \star) \in \text{match}(P') \}
\]

dynamic → static: The set of associated advice did not change, but in contrast to \( P \) the responsible pointcut expression can be statically evaluated in \( P' \):

\[
\text{change}_{\text{d} \rightarrow s} = \{ (jp, adv, d \rightarrow s) \mid \exists (jp, adv, \text{dynamic}) \in \text{match}(P) \land \exists (jp, adv, \text{static}) \in \text{match}(P') \}
\]

static → dynamic: The set of associated advice did not change, but in contrast to \( P \) pointcut evaluation needs conservative approximations in \( P' \):

\[
\text{change}_{s \rightarrow d} = \{ (jp, adv, s \rightarrow d) \mid \exists (jp, adv, \text{static}) \in \text{match}(P) \land \exists (jp, adv, \text{dynamic}) \in \text{match}(P') \}
\]

Finally the pointcut delta is defined as the union of the classified delta sets, thereby also capturing dynamic pointcut designators:

\[
\text{pcDelta}(P, P') = \text{new}_{\text{static}} \cup \text{new}_{\text{dynamic}} \cup \text{change}_{d \rightarrow s} \cup \text{lost}_{\text{static}} \cup \text{lost}_{\text{dynamic}} \cup \text{change}_{s \rightarrow d}
\]

Note that in the above definitions we map aspect structure triples associating joinpoints and advice (with a given certainty; either static or dynamic) to delta triples, which have a similar structure, but different semantics.

Note that the above assumes that \( jp \) and \( adv \) alone identify a tuple \((jp, adv, \star)\). This of course depends on the chosen joinpoint representation. As joinpoint signatures as proposed here include a counter this requirement is fulfilled here. Using these six categories, the derived matching delta is enriched with confidence information. Static information can be trusted, dynamic information still requires programmer investigation, but offers hints where to start.

Clearly a goal must be to reduce uncertain information as much as possible. Program analysis can be used to evaluate some dynamic expressions at compile time (i.e. by using partial evaluation, abstract interpretation or related techniques) so reducing the amount of spurious matches, but an exact calculation of matching information in general is not computable. As this is also a relevant problem for performance of AOP software, this is a current research topic [17, 12]. However, this is not in the scope of the work presented here.

## 5. Explaining Deltas

The benefit of calculating the delta set is that these sets tends to be small compared to the system’s overall number of matched joinpoints, at least if we do not consider adding or removing aspects at the moment. If \( \text{pcDelta}(P, P') = \emptyset \), the programmer can assume that an edit did not affect semantics of any static pointcut expression in terms of matched joinpoints (i.e. here the actual pointcut result did not change). Note however that joinpoint mismatch due to dynamic pointcut expressions can occur and is not covered ((\( (jp, adv, \text{dynamic}) \in \text{match}(P) \setminus \text{match}(P') \)). If \( \text{pcDelta}(P, P') \) contains differences, these differences can easily be traced back to the affected aspects, so the aspect programmer can be notified of this change. As a result, the delta is valuable information as unexpected matches can be found more easily.

The inverse problem is to find expected but not experienced matches. This corresponds to a change in the pointcut definitions (and thus the expected pointcut result). Unfortunately this is considerably harder to do automatically as here an analysis would need information about the expected pointcut results. These expected results would have to be checked against the actual matching behavior. However, although this can’t be done automatically, for the programmer it is easier to check a small delta than the whole program in order to find out is expected matches are actually present. Thus our analysis also offers support in this case.

While the delta set alone is valuable, we refined our analysis to identify causes for these deltas, to allow a programmer to immediately see why a specific delta entry exists. Potential changes resulting in pointcut deltas are threefold:

1. Aspect evolution can add additional or remove some pieces of advice. This also includes addition or removal of a complete aspect (new expected pointcut result).
2. If a pointcut itself has been modified, we expect differences in its matching behavior\(^6\) (new expected pointcut result).
3. Base Code Edits, or more precisely their effects on joinpoints are most problematic and most likely the reason for unexpected changes in the matching behavior, as outlined before (new actual pointcut result).

\(^{6}\)This also includes modification of anonymous pointcuts.
To explain why a joinpoint match is in the delta, we enriched each delta entry \((jp,adv,\bullet)\) with additional information explaining the reasons why this entry exists by associating relevant pointcuts, advice and joinpoints with atomic changes derived from comparing the two program versions \(P\) and \(P'\). In the following we assume that the set of atomic changes \(A\) has been calculated as described in section 3.

5.1 New or Removed Advice

Most obvious, additional or lost matches can result from added or removed advice. Note that this also includes adding or removing a whole aspect. For each delta entry \((jp,adv,\bullet)\) we check if atomic changes exist which reference the advice \(adv\).

**Definition 5.1 (Advice Changes).** For \((jp,adv,\bullet)\in pcDelta(P,P')\), we calculate atomic changes associated with the advice \(adv\) as

\[
(adv,\Delta_{adv}) = advChanges((jp,adv,A(P,P'))) \\
\text{where}
\]

\[
advChanges((jp,adv,\bullet),A(P,P')) = \\
\text{case } \bullet = \text{static } \lor \text{+dynamic } \to \text{+adv} \in A(P,P') \\
\text{case } \bullet = \text{static } \lor \text{-dynamic } \to \text{-adv} \in A(P,P') \\
\text{otherwise } \to \emptyset
\]

Associating atomic changes with advice is straightforward. A tuple can be in the delta, if advice has either been added to \((AAD(adv))\) or removed from \((DAD(adv))\) the system. However, this is also the most simple case. More interesting are changes in pointcut definitions, which we will examine next.

5.2 Modified Pointcuts

Finding modified pointcut definitions only requires a simple analysis of textual differences in the respective code. The information which joinpoints and advice are linked by this pointcut expression is in general also derivable from the system, as this information is also needed by the weaver. The delta tuples associate joinpoints with adapting advice.

For AspectJ analyzing the source code of advice directly from \(P\) is the set of pointcuts and \(ADV\) is the set of advice defined in \(P\). It is thus possible to compute all modified and referenced pointcut definitions for a given piece of advice in the delta set for two given program versions.

For the special case of hierarchical pointcut dependencies present in AspectJ, the presentation of the differences is best presented to the user in an annotated graph. The union of \(reference(P)\) and \(bind(P)\) defines a directed acyclic graph reflecting the syntactic dependencies of advice and pointcut definitions.

\[
G(P) = reference(P) \cup bind(P)
\]

For each \(adv\) with a corresponding delta element \((jp,adv,\bullet)\) we calculate the merged pointcut dependence graph.

\[
G_{merged}(P,P') = \{\{a,b,\pm\}\mid (a,b) \in G(P') - G(P)\} \cup \{\{a,b,-\}\mid (a,b) \in G(P) - G(P')\} \cup \{\{a,b,=\}\mid (a,b) \in G(P) \cap G(P')\}
\]

**Example 5.1 (Structural Delta).** We illustrate this structural comparison with an example shown in figure 1. While we do not show code matching this dependence of advice and pointcuts, this code can be easily generated, as the reader may verify. Note that the dependence graph for advice and pointcuts in general is an acyclic\(^8\) rooted graph.

Sub-figure (a) shows syntactical pointcut dependences in the original program version, sub-figure (b) in the edited version and sub-figure (c) the merged dependence graph. Note that in sub-figure (c) the edges are labeled to show if a edge has been added (denoted by ‘+’), removed (denoted by ‘−’) or is present in both program versions (denoted by ‘=’). Figure (d) finally shows the resulting merged dependence graph (or rather the edge set) \(G_{merged}(P,P')\) in set representation.

While \(G_{merged}(P,P')\) already gives the programmers hints to find out how a pointcut expression has been changed, we can further improve this information by also associating information about atomic changes referring to these pointcut definitions.

We will formulate the respective algorithms using functional pseudo code. Therefore we need the simple auxiliary function shown below, which transforms a set to a list (with arbitrary order).

\[
\text{listOf}(\emptyset, x) = x \\
\text{listOf}(S, x) = let \ s \in S \text{ and } S' = S \setminus \{s\} \text{ in } \text{listOf}(S', s:x)
\]

**Definition 5.2 (Changes per Node).** For a referenced pointcut \(a\) with successors succ we derive the set of associated changes \(\Delta\) as

\[
\text{chgForNode}(a,\text{listOf}((a,b,\bullet)|b \in \text{succ}), A(P,P'), \Delta),
\]

where

\[
\text{chgForNode}(a, [], A(P,P'), \Delta) = (a,\Delta) \\
\text{chgForNode}(a, (a,b,\bullet): es, A(P,P'), \Delta) = case \bullet \in \{\text{+dynamic}, \text{-dynamic}\} \to \text{case } \Delta' = \Delta \cup \{\text{CAH}(a), \text{CP}(a)\} \cap A(P,P') \text{ then } \\
\quad \text{let } \Delta' = \Delta \cup \{\text{CAH}(a), \text{CP}(a)\} \cap A(P,P') \text{ in } \text{chgForNode}(a, es, A(P,P'), \Delta') \\
\text{otherwise } \text{let } \Delta' = \Delta \cup \{\text{CAH}(a), \text{CP}(a)\} \cap A(P,P') \text{ in } \text{chgForNode}(a, es, A(P,P'), \Delta')
\]

\(^8\)Recursive pointcut definitions are not allowed in AspectJ.
\(^9\)The respective piece of advice always is the root node.
Function \( \text{chgForNode} \) augments each node in the structural delta \( G_{\text{merged}}(P, P') \) with potentially affecting changes. For example if we have a node \( a \) with a single outgoing edge \((a, b, +)\), the additional reference from \( a \) to \( b \) can only occur if \( a \) itself has been changed, thus we check if \( A(P, P') \) contains a respective change (change pointcut \( \text{CP}(a) \) or change advice header \( \text{CAH}(a) \), as we do not know whether we deal with pointcut or advice). As the new reference can be due to the addition of a reference to \( a \) itself, we also check for add pointcut \( \text{AP}(a) \) changes in case of +.

We traverse the dependence graph using depth first search to capture all relevant changes. We want the derived set of changes to fulfill two properties: (i) no relevant change should be missing (i.e., we want a conservative solution), (ii) the associated sets should be as small as possible. To achieve this goal, we stop to traverse a path once we reach a removed node (changes further down are irrelevant). Function \( \text{getPCC} \) formalizes this strategy.

**Definition 5.3 (Pointcut Changes).** For a given merged annotated delta graph \( G_{\text{merged}}(P, P') \) and a piece of advice \( adv \), we associate pointcut definition reachable from \( adv \) with respective atomic changes by calling

\[
\text{getPCC}(G_{\text{merged}}(P, P'), [adv], A(P, P'))
\]

where

\[
\begin{align*}
\text{getPCC}(G_{\text{merged}}(P, P'), []) & = A(P, P') \\
\text{getPCC}(G_{\text{merged}}(P, P'), a : as, A(P, P')) & = \\
\text{let} \ succ = & \text{listOf} \{ \{ b | (a, b, \bullet) \in G_{\text{merged}}(P, P') \land \bullet \neq \cdot \} \} \\
\text{and} \ edges = & \text{listOf} \{ \{(a, b, \bullet) | (a, b, \bullet) \in G_{\text{merged}}(P, P') \land \bullet \neq \cdot \} \} \\
\text{and} \ A\Delta Set' = & A\Delta Set \cup \}
\end{align*}
\]

\[
\{ \text{chgForNode}(a, edges, \emptyset, A(P, P')) \} \text{ in getPCC}(G_{\text{merged}}(P, P'), succ\@as, A\Delta Set', A(P, P'))
\]

and '@' denotes the list concatenation operator.

To illustrate the above concepts, we show how the two defined functions work in practice. Therefore we apply them to the annotated delta graph we showed in figure 1.

### 5.3 Base Code Edits

Finally delta entries can also be due to modifications of the base code, as such edits can result in addition or removal of joinpoints matches by existing pointcut expressions.

- Additionally matched new joinpoints could be unexpected matches due to program extensions or rename/move operations and should be further examined.
- If a joinpoint has been removed from the program, this might be a lost match due to rename/move or deleted statements. This should be examined (also in the context of additional matches) to re-add the lost match if appropriate.
- If the joinpoint is present in both versions, the reason for a changed matching behavior must either be a pointcut modification or additional/removed advice (as captured by the first two cases).

In this context atomic changes allow us to capture these effects. To associate pointcut delta tuples with respective atomic changes, we exploit the information stored in the joinpoint signatures as described in table 3.

A joinpoint signature references the program item a joinpoint is contained in and also the program item this joinpoint references. If for a delta tuple \((jp, adv, \bullet)\) the qualifier
• is either + or − (either statically or dynamically), we can check if program items referenced by the joinpoint $jp$ have either been added, changed, or deleted. We formalize this idea in the following.

**Definition 5.4 (Item Affecting Changes).** Let $p$ be a program item in $P$, i.e. a method, initializer, etc. We then write $getChanges(p)$ to refer to all item, change item and delete item changes directly affecting this program item.

The above definition—on purpose—does not explicitly define the set of atomic changes affecting each given program item. We left this definition out as it is rather intuitive. For each program item there are in general changes indicating their addition, change or removal. For methods for example we have AM, CM and DM changes, for initializers AI, CI, and DI changes.

**Definition 5.5 (Associated Base Code Changes).** Let $(jp, adv, •) \in pcDelta$ be a pointcut delta tuple. Let $e = env(jp)$ be the joinpoint environment, and $r = ref(jp)$ the referenced program item. We then calculate the atomic changes associated with $jp$ as

$$(jp, \Delta) = getChanges(env(jp)) \cup getChanges(ref(jp))$$

Note that for some joinpoint kinds, e.g. execution and advice execution, $getChanges(r)$ and $getChanges(e)$ are identical.

### 5.4 Explained Delta Set

To summarize, a delta entry $(jp, adv, •) \in pcDelta$ is associated with changes affecting the bound outline, a structured delta of pointcut definitions by analyzing $adv$ and its referenced pointcuts, and finally with information about new or removed joinpoints.

Thus the programmer gets detailed information if and why joinpoint matching behavior has changed. This information considerably helps when trying to find the reasons for failures due to changed pointcut semantics.

We finally define function $explainDelta$, which collects all the above results.

**Definition 5.6 (Explained Pointcut Delta).** For each delta tuple $(jp, adv, •) \in pcDelta(P, P')$, we define the associated explained delta tuple as

$$((tp, getChanges(env(jp))) \cup getChanges(ref(jp))),$$

$$adv, getAdvChanges(jp, adv, •), A(P, P'),$$

$$getPCC(G_{merged}(P, P'), [adv], \emptyset, A(P, P')),$$

•).

Informally, the explained delta tuple consists of four components. First, we associate the affected joinpoint with base code edits, second the affected advice with advice changes, third we add information about changed pointcut dependencies and finally we also state the kind of joinpoint change.

With this structured association of atomic changes at hand, the programmer has considerably more information to examine and explain experienced changes in pointcut matching behavior.

### 5.5 Base Code Edits and Dynamic Predicates

Note that source code changes potentially change the value of dynamic predicates, i.e. even if a tuple $(jp, adv, dynamic)$ is present for both the original and the edited program version, we cannot be sure that the matching semantics did not change (compare e.g. AspectJ’s pointcut designators if, this or target). However, as dynamic predicates are conservatively approximated, such effects are not visible and consequently our analysis is oblivious to such changes—a potential match is present for both program versions.

Although our analysis is incomplete in presence of dynamic pointcut constructs, it is nevertheless useful.

- For static joinpoint selection languages such effects are captured by the calculation of $match$, thus our analysis is complete in these cases.
- Second, in our (limited) case studies we never experienced such a case, and even the use of dynamic predicates is rare.
- Finally, even if our tool misses these rare cases, in general our tool offers considerable support compared to the manual approach in the frequently occurring cases.

### 6. CASE STUDIES

We implemented this pointcut delta analysis as part of the AOPA. An initial preliminary prototype has been implemented as a student project by Christian Koppen, and Jürgen Graf finally refined this prototype by also implementing a Chianti-like calculation of atomic changes and integrated this analysis as his master project [5]. He also considerably helped in gathering the data these case studies are based on.

We used the resulting AJDiff-plugin to analyze the AspectJ example programs Telecom and SpaceWar as well as several of the programs available in the abc test suite. As for these programs only a single version is available, we could only compare different build configurations.

#### 6.1 AspectJ Examples

Our first subjects for evaluation are the AspectJ example projects, more precisely Telecom and SpaceWar. These projects are small and easy to understand, so the way how our tool derived results can be manually verified.

Unfortunately the CVS repository for the AspectJ examples does not contain different versions of the examples, thus our analysis can only be run against different build configurations. As expected we only found changes due to the addition or removal of advice and aspects.

##### 6.1.1 Telecom Example—Comparing basic and timing

We start with comparing the basic and the timing build configurations of the Telecom example. Compared to the minimal basic build configuration, timing adds class Timer and aspects Timing and TimerLog. Furthermore the test driver class BasicSimulation is replaced with class TimingSimulation. This addition resulted in the set of changes shown in figure 2.

When examining the pointcut delta for the Telecom example we note four additional, statically adapted joinpoints
in build configuration timing compared to basic, two in regular methods, and two within advice bodies. Figure 3 shows how changes are associated to this delta to explain it. In each case—not very surprising—our analysis found that the addition of a new piece of advice is the reason for the newly adapted joinpoint.

The first two delta entries show the two additional matches for the two pieces of advice defined in Timing, which advise the call to Connection.complete() in method Call pickups() and Connection.drop() in Call hangup(). The second two delta entries refer to advice defined by aspect TimerLog. Here, statements timer.start() in the first after-advice and timer.stop() in the second after-advice of Timing are advised by respective pieces of advice defined by aspect TimerLog. As for each delta entry a new piece of advice is found (none of the four pieces of advice is present when using the basic build configuration), the addition of the two aspects and the respective pieces of advice, modeled by the respective AA and AAD changes, are reported as reasons for the change in matching behavior. Note however that for the two pieces of advice defined in TimerLog however we also get the CAB change resulting from the removal of the target advice as additional potential delta reason. Figure 4 shows a screenshot of the Eclipse change view presenting this output to the user.

### 6.1.2 Telecom Example—Comparing basic and billing

The second example is a comparison of the basic and the billing build configurations of the Telecom example. Compared to the minimal basic build configuration, billing adds class Timer and aspects Timing and Billing (but not TimerLog). Furthermore the test driver class BasicSimulation is replaced with class BillingSimulation. These changes resulted in the set of changes shown in figure 5.

We experience five additional, statically adapted joinpoints in build configuration billing compared to basic, all of them in regular methods. Figure 6 shows how changes are associated to this delta to explain it. Similar to our first example our analysis found that the addition of the two aspects and the new pieces of advice are the reason for the newly adapted joinpoints. The first two delta entries show the two additional matches for the two pieces of advice defined in Timing, which advise the call to Connection.complete() in method Call pickups() and Connection.drop() in Call hangup, as seen before. The second two delta entries show the new matches of the Billing advice. Note that for the first entry for Billing actually refers to two delta entries. The after return-advice in Billing is bound to constructor calls for both class Local and class LongDistance. However, for both delta entries the same reason, i.e. the addition of aspect Billing and the after returning-advice is derived by our analysis. The second entry for Billing refers to the after-advice bound to pointcut endTiming. It is interesting to note that this is also the pointcut one of the two pieces of advice is bound to. However, as a different (second!) piece of advice is bound to this pointcut, we also get a different tuple entry. Again for each delta entry a new piece of advice is found (none of the four pieces of advice is present when using the basic build configuration), thus—similarly to the first example—the addition of the two aspects and the respective pieces of advice, modeled by the respective AA and AAD changes, is reported as reason for the change in matching behavior. Figure 7 shows a screenshot of the Eclipse view presenting this output to the user.

### 6.1.3 SpaceWar Example—Comparing debug and demo

For the SpaceWar example we compare the two build configurations debug and demo. Note that in contrast to the previous example we now remove an aspect. However, as we still only have a single version, delta entries are still only due to addition and removal of aspects. Compared to build configuration debug, file Debug.aj is omitted in the demo build configuration. This file contains the definition of aspect Debug and of class InfoWin (a simple text window to display debug output). Figure 8 gives an overview of the resulting changes.

While we will not discuss the results for SpaceWar in detail, we want to point out that this example is the first where we experienced changes in matching behavior for dynamic joinpoints. Compared to the Telecom example, the only difference is that SpaceWar defines more pieces of advice applying at more joinpoints and thus has more entries in the delta set. However, again each entry is straightforward matched with removed advice.

### 6.2 The abc Test Suite

As the pointcut delta analysis is a static analysis technique, we could use the abc test suite to evaluate our prototype. We present the results in this section. The abc test suite is a set of AspectJ tools which have been used to evaluate the performance of AspectJ programs as discussed in [2]. However, for brevity we only point out interesting peculiarities for the different case studies and otherwise only present the remaining data, i.e. the resulting changes and explained deltas in interesting cases. Comparable to the Telecom example, the abc test suite is not available in different versions and thus in general deltas are due to addition or removal of aspects.

#### 6.2.1 The Bean Example

![Figure 2: Atomic Changes for Telecom basic → timing.](image-url)
Figure 4: Screenshot of AOPA presenting the Explained Pointcut Delta for the Telecom example comparing build configurations basic and timing.

Figure 5: Atomic Changes for Telecom, basic → billing.

Figure 7: Screenshot of AOPA presenting the Explained Pointcut Delta for the Telecom example comparing build configurations basic and billing.
For this example we compared the two versions resulting from comparing a build with and without aspect BoundPoint. To make the non-aspectized version compilable, we also had to remove two lines in the Demo.main(...) method, which resulted in a respective CM change (i.e. we have an (artificial) base code edit in this case). Figure 9 gives an overview of the resulting changes. Note the high number of CM and DM changes, which are due to the various inter type declarations to Point to add and remove property change listeners (there is one additional CM change due to the modification of main(...)).

For the Bean example removal of aspect BoundPoint results in 4 lost static matches, 2 of which can be associated with each removed piece of advice. However, considering figure 10 also shows that for each advice, one joinpoint is associated with additional information. For this joinpoint, the CM change we introduced to make the non-aspectized version compilable is also reported as a potential change, as method Demo.main(...) also contains calls to setX and setY which are adapted by advice. In this case this is spurious information, however it gives a first impression how our analysis also captures base code edits (it is also possible that the two setX statements in main(...) had been deleted; in that case the CM change would have been the delta reason).

6.2.2 The DCM Example

For the DCM example we examined the effect of the removal of aspect AllocFree. Figure 11 shows the resulting changes. Note the high number of DRI changes. This is due to aspect AllocFree using the marker interface idiom to add a finalize() method to all classes in the system. Therefore it declares that all types implement the interface DCM.handleGC.Finalize and introduces a finalize method to the interface.

For this example removal of the aspects results in 41 delta matches. The second piece of advice is associated with only 2 of those 41 delta tuples (this piece of advice only prints a summary), the second piece of advice with 39 (these are the adapted constructor calls). For both pieces of advice the deletion of the aspect is identified as the delta reason.

6.2.3 The NullCheck Example

For the NullCheck example we removed aspect EnforceCodingStandards to generate the second version. This aspect defines a single piece of advice which is associated with 75 joinpoints. Figure 12 gives an overview of the resulting changes.

An interesting peculiarity is that this aspect also adapts 4 joinpoints within its own body. This is interesting as this case can lead to infinite recursion if calls within the advice actually return a null value (which is however not the case). For these 4 joinpoints within the advice body, we get a detailed feedback, as can be seen in figure 13. For these joinpoints, the change of the advice body is additionally reported as a potential delta reason, as the body edit potentially removed the respective joinpoints. For all 75 joinpoints our analysis determines the removal of the aspect as a delta reason.

7. RELATED WORK & CONCLUSIONS

Compared to the prior work reported in [18, 19] the delta analysis proposed here has been extended considerably as we improved the delta analysis by adding the explanation of the resulting deltas using atomic changes. We also considerably extended our case studies, now also be calculating and reporting the atomic changes associated with a delta entry. The AspectJ development tools ajdt [1] visualize relations between aspects and base, but the current version does not contain any support for pointcut deltas or delta analysis in general. While ajdt statically analyzes a single program version to provide valuable feedback for the user, we are using two (or more) versions to analyze their differences to support system evolution.

Our approach relies on a good approximation of dynamic pointcut designators. An approach to better approximate the cflow pointcut is presented in [17]. Partial evaluation [12] may also be useful to better approximate dynamic joinpoints.

Besides the work mentioned above we see our work related to many other efforts to improve program understanding, especially the work about Delta Debugging, Change Impact Analysis and the development of new pointcut languages.

7.1 Change Impact Analysis and Delta Debugging

The goal of Change Impact Analysis is to provide techniques to allow programmers to analyze the effects of changes they made. Examples are the work presented in [13, 7] or [16, 14]. In the latter work the edit between two program versions is decomposed in a set of Atomic Changes which we also use to explain delta entries.

Delta Debugging as introduced in [21] also focuses on finding failure inducing inputs or edits. However, this approach does not reveal any syntactical or semantical dependencies of the different program constructs as derived by our delta analysis. Second, Delta Debugging relies on executing intermediate versions. This however might not be possible for...
Aspect Timing Advice [timing-1]:
Changed matches (new sure: 1)
Reason: → AA(telecom.Timer<aspect> public) < AAD(telecom.Timer,[timing-
1].after(telecom.Connection) public)

Aspect Billing Advice [billing-1]:
Changed matches (new sure: 2)
Reason: → AA(telecom.Billing<aspect> public) < AAD(telecom.Billing,[billing-
1].afterReturning(Customer, Connection))

Aspect Billing Advice [billing-2]:
Changed matches (new sure: 1)
Reason: AA(telecom.Billing<aspect> public) < AAD(telecom.Billing,[billing-
2].after(telecom.Connection) public)

Aspect Timing Advice [timing-1]:
Changed matches (new sure: 1)
Reason: AA(telecom.Timer<aspect> public) < AAD(telecom.Timer,[timing-
1].after(telecom.Connection) public)

Aspect Billing Advice [billing-2]:
Changed matches (new sure: 1)
Reason: AA(telecom.Billing<aspect> public) < AAD(telecom.Billing,[billing-
2].after(telecom.Connection) public)

Figure 6: Explained Pointcut Delta for basic → billing.

Figure 9: Atomic Changes for Bean, removal of BoundPoint.

Aspect BoundPoint Advice bean.BoundPoint.around.#1:
Changed matches (lost sure: 2)
Specific reason at join- point Demo.main(String[][]) while
call to method void Point.setX(int): → CM (Demo.main(String[][])]
Reason:
→ DA (bean.BoundPoint) < DAD (bean.BoundPoint.around.#1)

Figure 10: Explained Pointcut Delta for Bean, removal of aspect BoundPoint.

picked in a more semantical way pointcuts tend to be less fragile.
An approach in-between these two extremes proposes
descriptive pointcuts, a set of descriptive pointcut designa-
tors which allows to specify joinpoints by their (semantic) properties [9]. This approach reduces the necessity to refer-
ence names or source locations and thus considerably light-
A detailed analysis of the fragile pointcut problem as outlined above - even if the goal of system evolution is the removal of aspect EnforceCodingStandards.around() while call to method String StringBuffer.append(String): Specific reason at joinpoint EnforceCodingStandards.around() while call to method String-Buffer StringBuffer.append(String): Specific reason at joinpoint EnforceCodingStandards.around() while call to method String-Buffer StringBuffer.append(String): Specific reason at joinpoint EnforceCodingStandards.around() while call to method Signature: JoinPoint$StaticPart.getSignature(): → DA (EnforceCodingStandards.around.$1) < DAD (EnforceCodingStandards.around.$1) Reason: → DA (EnforceCodingStandards<aspect> public) < DAD (EnforceCodingStandards.around.$1) Figure 13: Explained Pointcut Delta for NullCheck, removal of aspect EnforceCodingStandards.

ens the problem with fragile pointcuts. Unfortunately, although research produced first results [11] these pointcut designators are currently not widely available.

While we consider the improvement of pointcut languages important research, these languages will only lighten the problem in the future when the emerging constructs will become part of mainstream languages. However, by then we assume that there is a considerable amount of code written in e.g. AspectJ where evolution suffers from the problems outlined above - even if the goal of system evolution is the removal of the pointcut definitions with new, more declarative constructs. Additionally, even if new constructs are available the old constructs will be kept for compatibility reasons for some time. For this code our approach is valuable.

7.3 Conclusions

This chapter in detail examined currently available joinpoint selection mechanisms used in state-of-the-art aspect-oriented languages. We showed that these mechanisms all suffer from the fragile pointcut problem. Furthermore we claim that a purely language-based solution is a very ambitious goal.

Although improvement of pointcut languages is a research topic and might well reduce the fragile pointcut problem one day, we introduced an alternative tool-based approach: pointcut delta analysis is used to deal with changes in the set of actually selected joinpoints in an aspect-oriented program.

We showed that the calculated delta set together with associated responsible code changes can considerably help to reveal unexpected changes in the matching behavior of pointcuts by reporting the results of our case studies using our implementation. Thus the main contributions of this chapter are:

- A detailed analysis of the fragile pointcut problem as a major problem for evolution of aspect-oriented programs and aspect-orientation in general.
- The introduction of a pointcut delta analysis allowing to derive and explain differences in the set of selected joinpoints.
- Finally we also provided an implementation of our analysis as an Eclipse plugin extending ajdt and examined the benefits of our tool in several case studies.

To conclude, although we only have few data points to evaluate our tool, the results are promising and suggest that our tool might well help to avoid introduction of bugs into an aspect-oriented system due to accidentally matched or lost joinpoint deltas during system evolution. To best of our knowledge, up to now this is the only delta-analysis based tool for this purpose.

8. REFERENCES


