Using Pointcut Delta Analysis to Support Evolution of Aspect-Oriented Software

Maximilian Stoerzer and Juergen Graf
University of Passau
Passau, Germany
{stoerzer, grafj}@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 todays 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. In this paper we introduce a pointcut delta analysis to lighten these problems.*

1 Motivation

Aspect-oriented programming (AOP) as first introduced in [7] is a programming technique extending traditional programming techniques to improve software modularity. Its basic idea is to encapsulate so called *cross-cutting concerns* not properly modularizable using traditional programming techniques in a new kind of module called *aspect*.

Aspects provide two constructs to specify new behavior and where it should apply: *advice* and *pointcuts*. Advice is a method-like construct defining new functionality which is bound to a pointcut identifying a set of well-defined points during the execution of a program called *joinpoints*. Thus pointcuts specify *where* advice should be executed. The *aspect weaver* finally uses pointcut information to combine advice with the base system producing an executable system.

Unfortunately todays main stream pointcut languages are *based on lexical properties* of the code which change during system evolution. As a consequence, the *result* of an unchanged pointcut expression *can change* due to modifications of the base system, thus also changing system semantics. We consider this a crucial problem for evolution of aspect-oriented systems, as these semantical changes are *silent*, i.e. the programmer is not alerted if such changes happen. We use the term *fragile pointcut problem* to refer to this problem.

For the remaining of this paper, we will use AspectJ [6] as an example, although the observations made are also valid for other currently available AspectJ-like languages.

The rest of this paper is organized as follows. Section 2 examines the *fragile pointcut problem*, section 3 introduces our delta analysis. Section 4 describes our implementation, the tool `PCDiff`, and section 5 demonstrates its usefulness with a case study. We finally discuss related work, outline future work and conclude in section 6.

2 The Fragile Pointcut Problem

The first generation of pointcut designators explicitly selected joinpoints by *naming* elements of their corresponding context in the base program. These explicit references obviously introduce a *high coupling* between the base system and the aspect, making aspect reuse harder.

As a reaction to reduce coupling, AspectJ introduced *wildcards* allowing to exploit naming conventions. However, this results in a new problem. Pointcuts using this mechanism depend on these *naming conventions*. As such conventions are not checkable by a compiler, they are *never guaranteed*. As a result, programmers have to be very careful with their pointcuts to avoid *spurious or missed matches*.

For small programs a pointcut mismatch 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. Additionally, humans tend not to look for *unexpected* things, and mismatches in general are unexpected.

While the problems described above have to be handled when initially developing a pointcut, wildcards and explicit naming are especially problematic when considering system evolution as such pointcuts in general are *fragile*. A programmer might have correctly specified a pointcut. The corresponding aspect works as intended, all tests are suc-

cessful. Afterward the code evolves, e.g. by 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 an compiler 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*. In general there are several (trivial) non-local base code changes possibly modifying pointcut semantics in terms of actually selected joinpoints:

- ***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 `withincode`. Moving classes to another packages or methods to another class obviously changes matching semantics for such pointcuts.
- ***Add/delete method/field/class:*** Pointcuts 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[1].

Code modifications in general require to also update code referencing modified code[2]. (Automated) refactoring [2], as it is available in IBM's Eclipse IDE for Java or the Smalltalk Refactoring Browser, might be a way to avoid breaking pointcuts in some cases, but for AspectJ refactorings are currently not available and might be problematic in general when considering dynamic joinpoints like the `if`- or `cflow`-constructs. More important, automated refactorings require 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.

Aspects influencing a given base class are *not directly visible* in the code. As a result, a programmer modifying e.g. a class of the base system is not necessarily aware of all the aspects possibly matching joinpoints in this class. Tool support lightens this problem [1], but in our opinion this does not resolve the problems for *evolution of aspects, classes and their dependencies*, as here a tool has to keep track of differences between subsequent versions.

So currently system evolution as well as "refactorings" are done manually, so suffering from fragile pointcuts. Compared to traditional programming this problem is more serious as pointcut semantics in general *change silently*. There is only limited support to alert programmers if code modifications change the set of joinpoints matched by their pointcuts.

The issues demonstrated here are crucial for evolution of programs written using current AspectJ-like pointcut languages and thus for the long-term usability of languages like AspectJ. We refer to these problems as the *fragile pointcut problem* in [14].

3 Pointcut delta analysis

Software written today using available pointcut languages potentially will be maintained for years. So a way to deal with the fragile pointcut problem *for current languages* is needed.

A common technique to reveal unintended semantical changes is testing. Semantical differences introduced into a system are (hopefully) revealed by rerunning a regression test suite through failing test. Testing however only shows the presence of bugs, but can never prove their absence. Additionally test failures do not explain the failure reason. Thus for a failing test, test results (e.g. an exception) have to be further analyzed to actually track down the bug. Finding the failure inducing code modifications is hard if changed pointcut semantics due to non-local edits are responsible.

In the following we will use the program shown in figure 1 as a running example. Figure (a) shows the original program version, figure (b) an edited version. We will compare those versions using delta analysis.

The difference between these two versions includes moving a method (`update` from `D` to `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 not obvious.

3.1 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 $\mathcal{P}$ and an edited program version $\mathcal{P}'$. To detect semantical differences in program behavior due to

[1] This seems less relevant as public interfaces should be stable. However, aspects are not restricted to public interfaces.

[2] Renaming a method for example requires modifying all calls to this method to match the new name.

```
aspect A {

  pointcut setField():  set(int *);

  before():  setField() {
    print("Changing field value");
  } /*[161-0-1108388203184-8132904]*/



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

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

(a)

```
aspect A {
  pointcut dynamic():  within(C) && if(4 < 5);
  pointcut setField():
    set(int *) && dynamic();
  before():  setField() {
    print("Changing field value");
  } /*[161-0-1108388203184-8132904]*/
  after():  call(* update()) && if(4 < 5) {
    print("Field update done");
  } /*[276-1-1108388538145-4535112]*/
}
class C {
  int x;
  static void main(String[] args){
    D d = new D();
    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; }
  /* deleted */
}
```

(b)

Figure 1. (a) Original version of example program. (b) Edited version of example program (underlining is used to show added/changed code fragments).

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.

So, for a given aspect-oriented program $\mathcal{P}$, function

$$match : \mathcal{P} \rightarrow JP \times ADV \times Q$$

determines the set of all aspect-joinpoint relations, where

- *JP* is the set of all joinpoints in $\mathcal{P}$,
- *ADV* is the set of all advice in $\mathcal{P}$ and
- *Q* is the quality of the matching relation, either *dynamic* or *static*.

As this information is also used by the weaver when composing aspects and base-system, it is in general available.

To calculate the delta from $match(\mathcal{P})$ and $match(\mathcal{P}')$ it is necessary to identify *corresponding joinpoints and advice* in both versions $\mathcal{P}$ and $\mathcal{P}'$ of the program. More formally speaking we need *equality relations* 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") is *no* adequate representation, as even adding some blank lines changes the joinpoint/advice source position and thus would make identification of corresponding items in both program versions impossible.

For advice this problem can be solved easily by automatically naming a new piece of advice once it is introduced in the system. To do so, our tool inserts an identifying comment at the end of each piece of advice when encountered first (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 objects in a program.

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

Note that this joinpoint identification scheme is only 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 clearly distinguish these joinpoints (apart from the counter). But as we keep track of the *number* of matched same signature joinpoints (using the counter) and the *joinpoint model* of AspectJ itself is not able to directly distinguish such joinpoints either in practice this identification schema works very well.

With these two notions of equality for advice and joinpoints across different program versions it is now straight forward to calculate the delta set for $match(\mathcal{P})$ and $match(\mathcal{P}')$.

3.2 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 widely available `if`- or `cflow` pointcut designators. Statically one has to conservatively approximate these constructs by assuming `true` for each such predicate as evaluation 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 refined our 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 $\mathcal{P}'$:

$$new_{static} = \{(jp, adv, +_{static}) \mid \exists (jp, adv, \texttt{static}) \in match(\mathcal{P}') \wedge \nexists (jp, adv, \bullet^{3}) \in match(\mathcal{P})\}$$

- ***New potential matches:*** A new advice association has to be conservatively assumed in $\mathcal{P}'$, although evaluation is not possible at compile time:

$$new_{dynamic} = \{(jp, adv, +_{dynamic}) \mid \exists (jp, adv, \texttt{dynamic}) \in match(\mathcal{P}') \wedge \nexists (jp, adv, \bullet) \in match(\mathcal{P})\}$$

[3] In the following, '•' will indicate any possible value for a tuple variable (wildcard).

- ***Dynamic → Static:*** The set of associated advice did not change, but in contrast to $\mathcal{P}$ the responsible pointcut expression can be statically evaluated in $\mathcal{P}'$:

$$change_{d \to s} = \{(jp, adv, d \to s) \mid \exists (jp, adv, \texttt{dynamic}) \in match(\mathcal{P}) \wedge \exists (jp, adv, \texttt{static}) \in match(\mathcal{P}')\}$$

- ***Lost matches:*** A statically determined advice association is no longer present in $\mathcal{P}'$:

$$lost_{static} = \{(jp, adv, -_{static}) \mid \exists (jp, adv, \texttt{static}) \in match(\mathcal{P}) \wedge \nexists (jp, adv, \bullet) \in match(\mathcal{P}')\}$$

- ***Lost Potential matches:*** A conservatively assumed advice association is no longer present in $\mathcal{P}'$:

$$lost_{dynamic} = \{(jp, adv, -_{dynamic}) \mid \exists (jp, adv, \texttt{dynamic}) \in match(\mathcal{P}) \wedge \nexists (jp, adv, \bullet) \in match(\mathcal{P}')\}$$

- ***Static → Dynamic:*** The set of associated advice did not change, but in contrast to $\mathcal{P}$ pointcut evaluation needs conservative approximations in $\mathcal{P}'$:

$$change_{s \to d} = \{(jp, adv, s \to d) \mid \exists (jp, adv, \texttt{static}) \in match(\mathcal{P}) \wedge \exists (jp, adv, \texttt{dynamic}) \in match(\mathcal{P}')\}$$

We thus finally define the pointcut delta as the union of the classified delta sets, thereby also capturing dynamic pointcut designators:

$$pcDelta(\mathcal{P}, \mathcal{P}') = \bigcup \{new_{static}, new_{dynamic}, change_{d \to s}, lost_{static}, lost_{dynamic}, change_{s \to d}\}.$$

Note that the above assumes that *jp* and *adv* alone identify a tuple $(jp, adv, \bullet)$. This of course depends on the chosen joinpoint representation. As joinpoint signatures are extended with a counter this requirement is fulfilled for the joinpoint signatures introduced 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 by further analyzing dynamic joinpoints, 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[13, 9]. We consider this out of the scope of this paper.

3.3 Explaining Deltas

The benefit of calculating the delta set is that it tends to be *small* compared to the system's overall number of advice. If $pcDelta(\mathcal{P}, \mathcal{P}') = \emptyset$, the programmer can assume that an edit did not affect any applying aspect. If $pcDelta(\mathcal{P}, \mathcal{P}')$ 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 alone is already valuable information as unexpected matches can be seen more easily.

The inverse problem is to find *expected but not experienced matches*. Unfortunately this is considerable harder to do automatically as here an analysis would need information about the programmers expectations. These expectations 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 search through a small delta than through the whole program, 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. Certainly, if a *pointcut itself* has been *modified*, we expect differences in its matching behavior[4].

2. Aspect evolution can add additional or remove some pieces of advice. This also includes addition or removal of a complete aspect.

3. *Base Code Edits* are most problematic and most likely the reason for *unexpected* changes in the matching behavior, as outlined in the motivation.

To explain causes for a delta, we enriched each delta entry $(jp, adv, \bullet)$ with additional information giving the reasons *why* this entry exists by associating pointcut, advice and joinpoint changes.

3.3.1 Modified Pointcuts

Finding modified pointcut definitions for a specific delta is relatively easy as joinpoints are associated with adapting advice. Analyzing the source code of advice and referenced pointcuts allows to access the referenced pointcut definitions[5]. Pointcuts can also reference other pointcuts. We express these dependencies using two relations

$$reference(\mathcal{P}) \subset PC_{\mathcal{P}} \times PC_{\mathcal{P}}$$

and

$$bind(\mathcal{P}) \subset ADV_{\mathcal{P}} \times PC_{\mathcal{P}}.$$

where $PC_{\mathcal{P}}$ is the set of pointcuts and $ADV_{\mathcal{P}}$ is the set of advice defined in $\mathcal{P}$. These two relations can be computed by a simple (syntactical) analysis. 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.

The union of $reference(\mathcal{P})$ and $bind(\mathcal{P})$ is represented as a directed acyclic graph reflecting the syntactic dependencies of advice and pointcut definitions. For each *adv* with a corresponding delta element $(jp, adv, \bullet)$ and each program version $\mathcal{P}$ and $\mathcal{P}'$ we calculate the set of all pointcut definitions the advice depends on by by running a breadth first search starting at the graph node corresponding to *adv*. The resulting trees are merged[6]. Nodes representing pointcuts where lexical differences exist are annotated as "changed", additional or removed nodes are annotated respectively. By presenting this merged and annotated tree, the user gets a structured overview of pointcut edits.

3.3.2 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. We assume existence of a function $exists : ADV \times \mathcal{P} \rightarrow \{true, false\}$ to derive whether or not advice *adv* exists in $\mathcal{P}$ (or $\mathcal{P}'$ respectively). For each delta entry $(jp, adv, \bullet)$ we also add information whether *adv* can be found in both versions or not.

3.3.3 Base Code Edits

Finally lost or additional matches can also be due to modifications of the base code, as such edits can result in addition or removal of joinpoints to match.

- 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).

[4] This also includes modification of anonymous pointcuts.

[5] For AspectJ, advice and pointcuts can again reference multiple other pointcuts (results are combined using logical operators not '!', or '||' and and '&&').

[6] We can correlate the *adv*-nodes in both trees; pointcuts are named constructs.

To capture information about existing joinpoints, we define a function $exists : JP \times \mathcal{P} \rightarrow \{true, false\}$ reflecting this information. Compared to the above, implementation of this function is considerably more complex as therefore a detailed comparison of both program versions on the statement level is necessary.

Our current implementation of *exists*, as a heuristic, checks if program elements containing or referenced by a joinpoint (i.e. recorded in the joinpoint signature) exist in both versions. For example `call`- or `execution`-joinpoints require the called/executed method to exist; field access joinpoints are similar.

Thus the granularity of *exists* currently is only–roughly speaking–on the method level. More precise results are however possible if we analyze the source code of $\mathcal{P}$ and $\mathcal{P}'$ on the statement level. Future work will first address this topic. Although e.g. deletion of a single call is not captured by this method, this approach already captures adding, moving, renaming or deleting program elements. Thus out tool can considerably help to lighten evolution of aspect-oriented programs.

Note that source code changes potentially change the value of dynamic predicates (compare e.g. AspectJ's pointcut designators `if`, `this` or `target`) and thus the actual matching behavior for advice. However, if dynamic predicates are approximated, such effects are not visible and consequently our analysis oblivious to such changes–a potential match is present for both program versions. But if such effects are captued by the calculation of *match* our analysis automatically benefits from them.

To summarize, a delta entry $(jp, adv, \bullet)$ is associated with a structured delta of pointcut definitions by analyzing *adv* and its referenced pointcuts, with information about new or deleted advice, and finally–although limited–with information about new or removed joinpoints. Thus the programmer gets detailed information *if and why* joinpoint matching behavior has changed. This information of course considerably helps when trying to find the reasons for failures due to changed pointcut semantics.

4 The PCDiff Plugin

The pointcut delta analysis outlined in section 3 has been implemented as an Eclipse plugin extending the AspectJ Development Tools (ajdt). Our implementation uses available information from the ajdt-plugin [1] and the AspectJ compiler (the *abstract structure model (asm)*) and does not calculate any matching information itself. An advantage of this strategy is that our plugin will automatically benefit from improvements in the resolution of dynamic joinpoints added to the AspectJ compiler.

The asm works well for static pointcuts, but for pointcuts including dynamic joinpoints (`if`, `cflow`, ...), the model (conservatively) approximates possible matches (i.e. `if(..)` is approximated as `true`). So the model reports spurious matches. However, the model also reports if a pointcut includes dynamic designators allowing our tool to mark up corresponding delta information.

4.1 Usage Scenario

We envision our tool to be used in quality assurance during the development process. We assume the following setting: we have a base programmer, an aspect programmer and a quality assurance agent (QA agent), who could be the person who is also responsible for running tests, etc.

Finding bugs due to conflicting edits of the base system is hard. It is even harder if the aspect and the base developer are different people. In this case this also raises another interesting (management) question: If failures due to changed pointcut semantics occur, who is responsible? The aspect developer or the base developer? Both answers are not satisfactory. The aspect programmer developed the aspect for a given version of the system using the program elements at hand, and cannot anticipate all future edits of the base system.

The base programmer in general should not be responsible to modify an affected aspect as an aspect might affect many other modules as well. So the base class programmer definitely is no expert to adapt an aspect potentially influenced by his code changes.

Although assigning responsibilities for fragile pointcuts is problematic, our tool can help to resolve this issue. The QA agent in general has the knowledge who is responsible for which components of the system, as he also assigns tasks to resolve traditional quality issues. When a new version of a software system is built, the QA agent runs our tool on the last and the new system version. He examines the results and reports potentially problematic deltas to the responsible aspect *and* base programmer.

So the QA agent is responsible to reveal editing conflicts, but aspect and base programmer *together* are responsible for solving the issue. Neither aspect nor base programmer alone have sufficient knowledge to detect or even deal with the problem, as this would require the aspect programmer to know the base code or the base programmer to be aware of all aspects. Both is not realistic for large team-developed systems. However *together* they are able to quickly repair broken aspects.

A second advantage of this approach is that it would nicely fit into the development process as widely used in industry. It is well-known that the programmer in general is a bad tester, thus a separate QA group in general is considered a good thing.

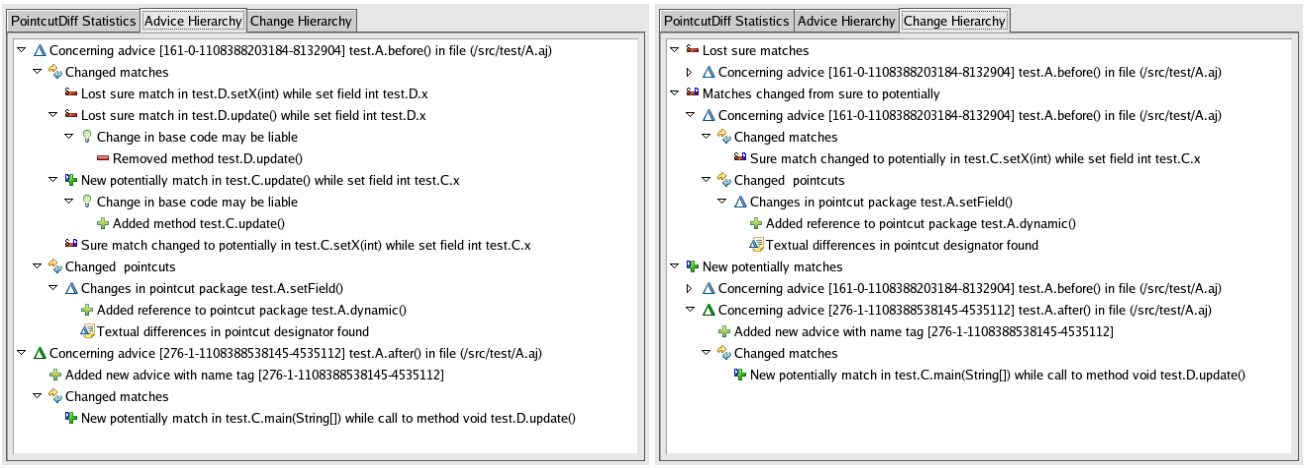

Figure 2. *Diff Visualizer* **showing** *Advice Hierarchy* **and** *Change Hierarchy* **view**

4.2 Using PCDiff

Because our implementation relies on unique names assigned to all advice declarations, the initial project version has to be prepared accordingly. Selecting a single AspectJ project we are able to trigger the "*Create unique names for all advice*" action in the *Pointcut Diff* sub-menu, which will add the unique name tag at the end of each unnamed advice declaration. In future program versions derived from this prepared version new advice is automatically captured and named.

Assuming two fitting versions in the Eclipse workspace the PCDiff plugin is ready to compare them. Therefore we select two build-configuration files (.ajproperties) - one for each version of the project and select "*Compare projects in Diff Visualizer*" in the *Pointcut Diff* sub-menu. The project containing the resource selected at first is considered to be the older version during the calculation of the differences.

When the analysis is finished the results are displayed in the *Diff Visualizer* view (figure 2). This view subdivides into three parts, *PointcutDiff Statistics*, *Advice Hierarchy* and *Change Hierarchy*. The *Statistics* tab shows a short textual summary about the differences found between both versions, including the number of additional joinpoint matches or the number of advice involved in a change and similar values.

While the *Advice Hierarchy* visualizes the differences sorted by the advice they are associated with, the *Change Hierarchy* splits the result by the kind of the change (section 3.2).

In addition to the *Diff Visualizer* view all changes are also illustrated by markers in the source code utilizing the *Eclipse marker mechanism*.

4.3 A first Example

We will now demonstrate the PCDiff plugin with our example from figure 1. At first we select version **a** followed by version **b** of our example project in the Eclipse resource view and trigger the "*Compare projects in Diff Visualiser*" option. As expected some changes in the advice-joinpoint matching have been detected and are summarized in the *Statistics* pane. Switching to the *Advice Hierarchy* (left image on figure 2) there are two pieces of advice involved in the changes as shown in the tree view. A double-click on each element in this view opens a text editor showing the corresponding element.

Denoted by the *green delta* in front of the advice named "*276-1-1108388538145-4535112*" we see that this advice is new and has been added in version **b**. Removed advice is marked by a *red delta*. All new or lost matches belonging to those pieces of advice are obviously due to the creation or removal of the appropriate advice.

In contrast advice "*161-0-1108388203184-8132904*" has not been added or removed and thus the changes in the advice-joinpoint matches have to be due to *Modified Pointcuts* (section 3.3.1) or *Base Code Edits* (section 3.3.3). Taking a closer look at the elements belonging to this advice we notice modifications in the matching of joinpoints, symbolized by the elements in the *Changed matches* subtree. For some of them our tool was able to identify a base-code change as a possible reason, shown in a subtree below the affected joinpoint match. But there have also been changes in the pointcut designators of our advice that could have caused the difference. They are outlined in the *Changed pointcut* subtree. Currently, added or removed references to named pointcut constructs as well as textual changes in the designator are detected and visualized.

The *Change Hierarchy* view (figure 2, right side) illus-

trates the changes of joinpoint matches sorted by their kind. So this view provides an overview which change types have been detected. This also allows the user to easily get an impression which matches e.g. changed from a sure to a possible match.

Although this example is simple, it gives an impression how our tool can help to find program flaws introduced due to accidentally matched or lost joinpoints. Note that the tool captures differences due to modified pointcut definitions as well as differences due to changes in the base code.

5 Case Studies

As unfortunately only very limited AspectJ code is publicly available, we were only able to evaluate our tool with two case studies.

5.1 AspectJ Examples

Naturally the first subjects of our evaluation are the AspectJ example projects, more precisely Telecom and Spacewar. These projects are small and easy to understand, so the way how `PCDiff` finds and displays its 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 introduction or removal of advice.

The result of the comparison between the different build configurations is displayed in figure 3 for the *telecom* example and figure 4 for the *spacewar* example.

The table is divided into 5 columns, starting with the build configurations that have been compared, the number of aspects and pieces of advice involved in any change, a summary of all changed matches and finally the reason for those changes.

Looking at the *Aspects* column it is subdivided into 3 columns, $+$, Δ and $-$. These symbols have a very intuitively meaning: $+$ denotes the number of added aspects, Δ the number of aspects involved in any change and $-$ the number of removed aspects. Below the summary all relevant aspects are listed separately together with the appropriate symbol in front of their name. All further statistics in these lines refer to the named aspect.

The *Advice* column is also split into 3 columns, $+$, Δ and $-$ with a similar meaning. $+$ is the number of newly introduced advice, Δ the number of advice involved in any joinpoint matching changes and $-$ the number of deleted advice.

Possibly of most interest is the fourth column, *Changed Matches*, containing the number of new sure matches ($+_S$), new potential matches ($+_D$), possible matches changed to sure ones ($_D \rightarrow_S$), sure matches changed to possible ones ($_S \rightarrow_D$), lost possible matches ($-_D$) and lost sure matches ($-_S$).

Finally the reason for all found changes is shown in the fifth and last column. As we were only comparing different build configurations naturally this is the reason for all differences. This column will be of greater interest when analyzing a more sophisticated project in the following section.

5.2 AspectJ development tools

As second subject for our analysis we finally found an example in the source of the AspectJ development tools Eclipse plugin itself. Since November 2004 aspects have been used in *org.eclipse.ajdt.ui*. Comparing the changes of the advice joinpoint matching between versions of this project separated by approximately 15 days, we analyzed the evolution of the matching behavior in this code.

An interesting finding of our analysis is that pointcut definitions rarely change. Most changes are due to the removal of old or introduction of new advice. Modifications in the base code affect pointcuts by creating new statements that match an existing pointcut bound to advice. Dynamic pointcut definitions have been used very seldom in ajdt and joinpoint matches that changed from a dynamic to a static nature (or the other way round) could not be observed.

Most interesting for the purpose to evaluate the benefit of our tool is that there are indeed several cases where the matching behavior changed due to base code edits, as can be seen in table 5 (column reason).

Although we just reported our experience gained from two examples and these single data points do not yet allow to draw conclusions, the results are promising. The two case studies we made give a good impression of the benefits of our tool, especially as the data shows that modified pointcut semantics or more precisely lost or additional matches due to base code edits are not a theoretical problem.

6 Related Work & Conclusions

Compared to the prior work reported in [14] the delta analysis proposed here has considerably been extended as we improved the delta analysis, added the handling of dynamic pointcut designators and the explanation of the resulting deltas. We also examined “real” AspectJ code in our case studies.

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.

Compared	Aspects			Advice			Changed Matches						Reason
	+	Δ	−	+	Δ	−	$+_S$	$+_D$	$_D \rightarrow_S$	$_S \rightarrow_D$	$-_D$	$-_S$	
basic → billing	2	0	0	4	0	0	5	0	0	0	0	0	change of build configuration
	+ Billing			2	0	0	3	0	0	0	0	0	
	+ Timing			2	0	0	2	0	0	0	0	0	
basic → timing	2	0	0	4	0	0	4	0	0	0	0	0	change of build configuration
	+ TimerLog			2	0	0	2	0	0	0	0	0	
	+ Timing			2	0	0	2	0	0	0	0	0	
basic → build	3	0	0	6	0	0	7	0	0	0	0	0	change of build configuration
	+ Billing			2	0	0	3	0	0	0	0	0	
	+ Timing			2	0	0	2	0	0	0	0	0	
	+ TimerLog			2	0	0	2	0	0	0	0	0	
billing → timing	1	0	1	2	0	2	2	0	0	0	0	3	change of build configuration
	− Billing			0	0	2	0	0	0	0	0	3	
	+ TimerLog			2	0	0	2	0	0	0	0	0	

Figure 3. Statistics for the *telecom* example project using different build configurations

Compared	Aspects			Advice			Changed Matches						Reason
	+	Δ	−	+	Δ	−	$+_S$	$+_D$	$_D \rightarrow_S$	$_S \rightarrow_D$	$-_D$	$-_S$	
debug → build	0	0	2	0	0	14	0	0	0	0	2	302	change of build configuration
	− Display2			0	0	1	0	0	0	0	0	4	
	− Debug			0	0	13	0	0	0	0	2	298	

Figure 4. Statistics for the *spacewar* example project using different build configurations

Compared	Aspects			Advice			Changed Matches						Reason
	+	Δ	−	+	Δ	−	$+_S$	$+_D$	$_D \rightarrow_S$	$_S \rightarrow_D$	$-_D$	$-_S$	
01.11.2004	2	0	0	3	0	0	355	0	0	0	0	0	new aspects introduced
15.11.2004	+ PreferencePageBuilder			1	0	0	4	0	0	0	0	0	
	+ FDDC			2	0	0	351	0	0	0	0	0	
15.11.2004	1	1	1	11	0	3	31	8	0	0	0	355	new/removed aspect, advice removed
01.12.2004	+ MarkerUpdating			2	0	0	2	0	0	0	0	0	
	− FDDC			0	0	2	0	0	0	0	0	351	
	Δ PreferencePageBuilder			9	0	1	29	8	0	0	0	4	
01.12.2004	0	2	0	0	1	1	1	0	0	0	0	8	advice removed, change in basecode
15.12.2004	Δ MarkerUpdating			0	0	1	0	0	0	0	0	1	
	Δ PreferencePageBuilder			0	1	0	1	0	0	0	0	7	
15.12.2004	0	0	0	0	0	0	0	0	0	0	0	0	no change
15.02.2005													
15.02.2005	0	1	1	0	2	1	2	0	0	0	0	1	removed aspect, change in basecode
01.03.2005	− MarkerUpdating			0	0	1	0	0	0	0	0	1	
	Δ PreferencePageBuilder			0	2	0	2	0	0	0	0	0	
01.03.2005	0	1	0	0	2	0	0	0	0	0	0	2	change in basecode
15.03.2005	Δ PreferencePageBuilder			0	2	0	0	0	0	0	0	2	

Figure 5. PointcutDiff has been used to calculate the changed matches in the *org.eclipse.ajdt.ui* project of the AspectJ development tools between 01.11.2004 and 15.03.2005 in steps sized ≈15 days.

Our approach relies on a good approximation of dynamic pointcut designators. An approach to better approximate the `cflow` pointcut is presented in [13]. Partial evaluation [9] 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.

6.1 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 [10, 4] or [12, 11].

In the latter work the edit between two program versions is decomposed in a set of Atomic Changes. These changes are then associated with nodes and edges of call graphs for tests drivers, allowing to calculate which tests are affected and which changes affect these tests. However, this work is in the context of classical object-oriented programming (Java) and up to now has not been extended for aspect constructs.

The techniques presented here are a first step to bring Change Impact Analysis to aspect-orientation and thus can be compliment with traditional Change Impact Analysis especially to get a more thorough analysis of changes in the set of joinpoints existing in a program. Future work will address this topic and try to combine both approaches.

Delta Debugging as introduced in [15] 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 resulting version. This however might not be possible for software under development. Our approach statically analyzes both versions and can be easily integrated in an IDE.

6.2 Improved Pointcut Languages

The improvement of the pointcut definition mechanism is an important research topic today. Several approaches have been proposed to attack the fragile pointcut problem using improved pointcut languages.

To reduce coupling, AspectJ [6] invented *abstract aspects*. These aspects can contain abstract pointcuts which are defined by inheriting aspects. Thus all the advice code is encapsulated in the abstract aspect and can be reused. The aspect can be applied to a concrete problem by inheriting from the abstract aspect and defining the pointcuts for the concrete base system. Unfortunately, although coupling is reduced, pointcuts in the concrete aspect still are fragile.

[3] proposes a logic pointcut language. In this language, a program is represented as a set of facts and pointcuts are defined in a Prolog like language as a query over these facts. Although this language is Turing-complete its expressions could be evaluated by our tool to acquire the necessary matching information. However, if a expression cannot be completely evaluated at compile time we would again have to conservatively approximate. However, as joinpoints are picked in a more semantical way pointcuts tend to be less fragile.

An approach in-between these two extremes proposes *declarative pointcuts*, a set of *descriptive pointcut designators* which allows to specify joinpoints by their (semantic) properties [5]. This approach reduces the necessity to reference names or source locations and thus considerably lightens the problem with fragile pointcuts. Unfortunately, although research produced first results [8] 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 main stream 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 renewal 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.

6.3 Future Work

Future work will address two topics. Though first results are promising we plan a thorough evaluation of our tool to validate the benefits we expect from our tool. This is mainly hindered by the lack of subjects to analyze. Second, we will explore how additional, more detailed information about source edits can further improve our results. On the long term we will incorporate these efforts in a general impact analysis framework for evolution of aspect-oriented software, to also capture effects of aspect edits and add support known from traditional object-oriented impact analysis.

6.4 Conclusions

In this paper we claimed that current mainstream pointcut languages are not satisfactory, as they suffer from the *fragile pointcut problem*. Although improvement of pointcut languages is a research topic and might well solve this

problem one day, we introduced a delta analysis to deal with this problem for current languages, based on a comparison of the sets of matched joinpoints for two program versions. We showed that the calculated delta set together with associated responsible code constructs can considerably help to reveal unexpected changes in the matching behavior of pointcuts by reporting the results of our case studies using our implementation.

The main contributions of this paper are:

- A detailed analysis of the fragile pointcut problem as a major problem for evolution of programs written in currently available aspect-oriented languages.
- The introduction of a pointcut delta analysis allowing to derive a joinpoint matching delta across program versions which also handles dynamic pointcut designators. Furthermore our analysis also explaines which edits are responsible for the experienced delta.
- This paper furthermore shows that aspect-oriented languages need to be complemented with tool support.
- Finally we also provide an implementation of our analysis as an Eclipse plugin extending `ajdt` and examined the benefits of our tool in two case studies.

If requested, the Eclipse plugin PCDiff is available from the authors for evaluation purposes.

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.

References

[1] A. C. Andy Clement and M. Kersten. Aspect-oriented programming with ajdt. In *Proceedings of AAOS 2003: Analysis of Aspect-Oriented Software, held in conjunction with ECOOP 2003*, July 2003.

[2] M. Fowler. *Refactoring – Improving the Design of Existing Code*. Addison-Wesley, 1999.

[3] K. Gybels. Using a logic language to express cross-cutting through dynamic joinpoints.

[4] 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 the ACM SIGPLAN Conf. on Object Oriented Programming Languages and Systems (OOPSLA'01)*, pages 312–326, October 2001.

[5] G. Kiczales. The fun has just begun. Keynote AOSD 2003, Boston, March 2003.

[6] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.

[7] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[8] H. Masuhara and K. Kawauchi. Dataflow pointcut in aspect-oriented programming. In *Programming Languages and Systems, First Asian Symposium, APLAS 2003, Beijing, China, November 27-29, 2003, Proceedings*, pages 105–121. Springer-Verlag, 2003.

[9] H. Masuhara, G. Kiczales, and C. Dutchyn. Compilation Semantics of Aspect-Oriented Programs. In *Proc of workshop Foundations Of Aspect-Oriented Languages (FOAL) held in conjunction with AOSD 2002*. 2002.

[10] A. Orso, T. Apiwattanapong, J. Law, G. Rothermel, and M. J. Harrold. An empirical comparison of dynamic impact analysis algorithms. In *Proc. of the International Conf. on Software Engineering (ICSE'04)*, pages 491–500, Edinburgh, Scotland, 2004.

[11] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: a tool for change impact analysis of java programs. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 432–448. ACM Press, 2004.

[12] B. G. Ryder and F. Tip. Change impact analysis for object-oriented programs. In *PASTE '01: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 46–53. ACM Press, 2001.

[13] D. Sereni and O. de Moor. Static analysis of aspects. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 30–39. ACM Press, 2003.

[14] M. Stoerzer and C. Koppen. Pcdiff: Attacking the fragile pointcut problem. In *Proceedings of European International Workshop on Aspect Software (EIWAS'04), Berlin, Germany*, 2004.

[15] A. Zeller. Yesterday my program worked. Today, it does not. Why? In *Proc. of the 7th European Software Engineering Conf./7th ACM SIGSOFT Symp. on the Foundations of Software Engineering (ESEC/FSE'99)*, pages 253–267, Toulouse, France, 1999.