Detecting Broken Pointcuts Using Structural Commonality and Degree of Interest

Raffi Khatchadourian
City University of New York
rkhatchadourian@citytech.cuny.edu

Awais Rashid
Lancaster University
awais@comp.lancs.ac.uk

Hidehiko Masuhara
Tokyo Institute of Technology
masuhara@acm.org

Takuya Watanabe
Edirium K.K.
sodium@edirium.co.jp

***Abstract*—Pointcut fragility is a well-documented problem in Aspect-Oriented Programming; changes to the base-code can lead to join points incorrectly falling in or out of the scope of pointcuts. Deciding which pointcuts have broken due to base-code changes is a daunting venture, especially in large and complex systems. We present an automated approach that recommends pointcuts that are likely to require modification due to a particular base-code change, as well as ones that do not. Our hypothesis is that join points selected by a pointcut exhibit common structural characteristics. Patterns describing such commonality are used to recommend pointcuts that have potentially broken to the developer. The approach is implemented as an extension to the popular Mylyn Eclipse IDE plug-in, which maintains focused contexts of entities relevant to the task at hand using a Degree of Interest (DOI) model.**

I. Introduction

Although using Aspect-Oriented Programming (AOP) [1] can be beneficial to developers in many ways [2], such systems have potential for new problems unique to the paradigm. A key construct that allows code to be situated in a single location but affect many system modules is a query-like mechanism called a pointcut expression (PCE). PCEs specify well-defined locations (join points) in the execution of the program (base-code) where code (advice) is to be executed. In AspectJ [3], an AOP extension of Java, join points may include calls to certain methods, accesses to particular fields, and modifications to the run time stack. In this way, AOP allows for localized implementations of so-called crosscutting concerns (or aspects), e.g., logging, persistence, security. Without AOP, aspect code would be scattered and tangled with other code implementing the core functionality of the modules.

As the base-code changes with possibly new functionality being added, PCEs may become invalidated. That is, they may fail to select or inadvertently select new places in the program's execution, a problem known as PCE fragility [4]. Deciding which PCEs have broken is a daunting venture, especially in large and complex systems. In software with many PCEs, seemingly innocuous base-code changes can have wide effects. To catch these errors early, developers must manually check all PCEs upon base-code changes, which is tedious (potentially distracting developers), time-consuming (there can be many PCEs), error-prone (broken PCEs may not be fixed properly), and omission-prone (PCEs may be missed).

Several approaches combat this problem by proposing new PCE languages with more expressiveness [5,6], limiting where advice may apply [7], or enforcing constraints on advice application [8]. Others make advice applicability more explicit [9] or do not use PCEs [10]. However, each of these tend to require some level of anticipation and, consequently, when using PCEs, there may nevertheless exist situations where PCEs must be manually updated. Furthermore, when using more expressive PCE languages, the rules that the base-code must respect may be complex. Hence, although these languages may reduce fragility, they may render *detection* of broken PCEs more difficult [11].

Other approaches [4,12,13] analyze changes in join points between versions so that the difference in behavior is well-understood. However, PCEs that likely broke as a result of the change must be manually determined. The AspectJ Development Tools (AJDT; http://eclipse.org/ajdt), which displays current join point and PCE matching information, does not indicate which PCEs do *not* select a given join point nor which are likely broken due to a new join point. Ye and Volder [14] provide *almost matching* join points via developer-minded heuristics, and Nguyen et al. [15] incorporate missed join points into PCEs, but neither detect situations where join points are unintentionally selected by PCEs. Wloka et al. [16] automatically fix PCEs broken by refactorings, however, manual base-code edits may also break PCEs. Khatchadourian et al. [17] suggest join points that may require inclusion by a PCE. Yet, developers must *manually* detect broken PCEs, as well as determine how frequently to check.

In this paper, we present an automated approach that recommends a set of PCEs that are likely to require modification due to a particular base-code change. Our approach has been implemented as an automated AspectJ source-level inferencing tool called FRAGLIGHT, which is a plug-in to the popular Eclipse IDE (http://eclipse.org). FRAGLIGHT identifies, as the developer is making changes to the base-code, PCEs that have likely broken within a degree of *change confidence*. Based on how "confident" we are in the PCE being broken, FRAGLIGHT presents the results to the developer by manipulating the Degree of Interest (DOI) model of the Mylyn context [18].

Mylyn (http://eclipse.org/mylyn) is a standard Eclipse plug-in that facilitates software evolution by focusing graphical components of the IDE so that only artifacts related the currently active task are revealed to the developer. The context is comprised of the relevant elements, along with information pertaining to how *interesting* the elements are to the related task. The more a developer interacts with an element (e.g., navigates to a file), when working on a task, the more interesting the element is deemed to be, and vice-versa.

In Mylyn, elements may also become interesting implicitly,

Listing 1. A point on a Cartesian plane.

```
1  public class Point implements Figure {
2      private double x;private double y;
3      public void setX(double x) {this.x=x;}
4      public void setTwiceX(double x) {this.x=2*x;}
5      public double getY() {return y;}}
```

Listing 2. An aspect managing how Figures are displayed.

```
1  public aspect DisplayManipulation {
2    after():
3      execution(* Figure+.set*(..))
4        {Display.update();}
5    double around():
6      execution(double Figure+.get*(..))
7        {return proceed()*0.5;}}
```

e.g., a package may become interesting if a class within the package is edited. FRAGLIGHT implicitly makes PCEs that are *more* likely broken *more interesting*, i.e., by *increasing* its DOI value, while implicitly making PCEs that are *less* likely broken to be *less interesting*, i.e., by *decreasing* its DOI value.

FRAGLIGHT's recommendations are based on harnessing unique and arbitrarily deep structural commonality between program elements corresponding to join points selected by a PCE in a particular software version. The majority of program elements corresponding to join points selected by a PCE in one base-code version share such characteristics between them, and these relationships persist in subsequent versions [17]. Here, we use this premise to detect broken PCEs on-the-fly.

This paper goes beyond [17] in that it:

Solves a different problem. Our previous approach, geared towards *aspect* developers[1], periodically suggests join points that may require inclusion into a revised version of a PCE. Aspect developers may revise *PCEs*, possibly after *coarse-grained* base-code changes, depending on the provided *join point* suggestions. Our new approach, geared towards *base-code* developers, however, suggests *PCEs* that may have broken due to a single revision to the base-code.

Presents a new, incremental algorithm. While our previous approach works with only a single PCE at a time, in this paper, our incremental approach avoids rebuilding and analyzing the base-code for each PCE.

Integrates with Mylyn. Our new approach is integrally tied to the Mylyn DOI, a proven, successful, and familiar model.

II. MOTIVATING EXAMPLE

We motivate our approach using a simple yet classic graphics application [3]. Listing 1 portrays code for a simple Point class (line 1) that implements a Figure (interface not shown) on a Cartesian plane. Two instance fields, x and y, are declared on line 2. There are two mutator instance methods for field x (mutators for y omitted for presentation), namely, setX(**double**), declared on line 3, which assigns field x to be the argument, and setTwiceX(**double**), declared on line 4, which assigns field x to be double the argument. Furthermore, there is an accessor instance method for field y (accessor for x omitted for presentation), declared on line 5, that returns the field value.

[1]The distinction between aspect and base-code developers has been well documented. This is particularly relevant in regards to reusable aspects [19].

As Figures may be maneuvered in many different editor modules, the DisplayManipulation aspect snippet (Listing 2) localizes the code for manipulating how Figures are displayed. The **after** advice (line 2) refreshes the Display (line 4, code not shown) whenever the state of a Figure is altered. This advice is implicitly executed **after** control leaves any join point selected by its bound PCE (line 3). These join points correspond to the **execution** of any method implementing a method of the Figure interface (Figure+) whose name begins with **set**, takes any number and type of parameters, and returns any type of value. In Listing 1, this corresponds to the execution of the setX(**double**) and setTwiceX(**double**).

Likewise, the **around** advice (line 5) scales Figures by 50%. The advice body (line 7) is implicitly executed *around* join points matching its bound PCE (line 6). Such join points correspond to the execution of methods implementing a method in the Figure interface whose name begins with **get**, taking any number and types of parameters, and returning any value. In Listing 1, this corresponds to the execution of the getY() method. When executed, the advice body first **proceed** s to execute the selected join point, multiplies the return value by the scaling factor, and returns the resulting value in its place.

Suppose that in this version, both PCEs are correct, i.e., they select all and only the intended join points. Now suppose that in a subsequent version, a new method move(**double**,**double**), which moves figures according to the specified coordinates, is added to the Figure interface. A corresponding implementation is then added to the Point class:

Listing 3. A new method is added to move Figures using coordinates.

```
1  public void move(double x,double y)
2    {this.x=x;this.y=y;}
```

Clearly, this new method alters the state of Figures, however, the PCE bound to the **after** advice, which refreshes the Display following state changes to Figures, on line 3 of Listing 2 fails to select this new join point. As a result, this PCE breaks.[2] Notice, however, that the PCE bound to the **around** advice, which scales figures, does not break and thus continues to select all and only the desired join points.

In general, each incremental change to the base-code can potentially break PCEs and thus cause bugs. If developers wait until many such changes, problems may be compounded and more difficult to find. To alleviate this, developers could perform a global analysis of all aspects and verify that each PCE is correct after every incremental change. However, not only would such an activity be distracting to base-code developers, it could also be non-trivial. Although this simple example contains only two PCEs, larger, more realistic systems may contain many more PCEs whose correctness would need to be verified. It would thus be helpful for developers if broken PCEs could be brought to their attention early. It would also be helpful if *un*broken PCEs were kept in the "background" as no action would be required. That way, the base-code developers may continue coding when an error is less likely and pause work otherwise. Rectifying such a problem would involve

[2]This PCE could have instead selected field **set** join points, which would have seemingly solved the problem. However, interfaces do not contain variable instance fields. Moreover, in the case of the Point class, the Display would have been refreshed twice, which could be inefficient.

either changing the base-code[3] so that it is correctly selected (or not selected) by the problematic PCE, or by altering the PCE itself. In the following sections, we will demonstrate how FRAGLIGHT can automatically alleviate such problems.

III. APPROACH

Overview. Our approach deals with the program's join point *shadows* (JPSs), which are the static counterparts of join points, i.e., points in the program text where the compiler may insert advice code [20]. Specifically, we treat a program as a set of JPSs that *may* or *may not* be under the influence of advice. Furthermore, we define a PCE to be a subset of JPSs, thus eliminating the need to consider complex expression constructs. We also assume that the PCE is free of dynamic conditions, which allows us to exploit solely static information in our analysis. Our implementation conservatively relaxes this assumption so that PCEs utilizing dynamic conditions may nevertheless be used as input to our tool. The impact of this limitation is minimal [17], and most PCEs do not use dynamic conditions [21].

FRAGLIGHT predicts how likely each PCE is to change given a change in the base-code. We model base-code changes as a series of JPS additions and removals, with each added JPS in the series being used as input. Changing a JPS, e.g., renaming a method, is modeled as the addition of a new JPS, e.g., the new method's **execution**.

Example 1. Adding the move() method in Listing 3 would result in three new JPSs, namely, **execution**(**void** Point.move(**double**,**double**)), **set**(Point.x), and **set**(Point.y), with the latter two being on line 2 in Listing 3.

Workflow Details. *Phase I: Analysis.* The analysis phase is triggered when a Mylyn task is activated. At this time, advice-bound PCE representations are collected.

Example 2. If the aspect in Listing 2 was the only aspect in all of the projects in the workspace, the PCEs bound to the **after** () advice declared on line 2 and the **around**() advice declared on line 5 would be analyzed.

Concern Graphs: An *extended concern graph* is built from projects that include the aspects whose advice-bound PCEs were analyzed. A concern graph is a directed multigraph depicting structural relations (e.g., calling, declarations, package containment) between program elements (e.g., types, methods, fields) [22]. We extend the graph with relations and entity types found in modern Java languages.

Example 3. Vertices for Point, Point.y, and Point.getY() would be in a graph built from Listing 1. Arcs would include Point $\xrightarrow{df}$ Point.y and Point $\xrightarrow{dm}$ Point.getY() $\xrightarrow{gf}$ Point.y, where df, dm, and gf refer to field declaration, method declaration, and field retrieval ("gets field") relations, respectively.

Maximum Analysis Depth: A maximum analysis depth (k) is a parameter to control tractability. It controls the depth of the structural relations considered.

Pattern Extraction: Next, each PCE is associated with the graph. This involves identifying portions of the graph (vertices or arcs) that are related to the JPSs selected by a PCE.

Example 4. Recall that the PCE declared on line 3 of Listing 2 selects executions of methods (and overriding methods via the + designator in Figure+) implementing the Figure interface and whose name begins with "set," etc. This PCE would be associated with the vertices representing the methods Point.setX() and Point.setTwiceX(). Graph elements (e.g., vertices) that represent such methods are "enabled" w.r.t. a PCE [17].

Algorithmically, pattern extraction works by first enumerating acyclic, finite paths of maximum length k in the graph.

Example 5. A path of length one is Point.setX() $\xrightarrow{sf}$ Point.x, where sf represents a field manipulation ("sets field") relation.

Next, paths that contain enabled vertices or arcs are used to construct patterns.

Example 6. The vertex Point.setX() in the path shown in Ex. 5 is enabled w.r.t. the PCE declared on line 3 in Listing 2.

Wild cards are then substituted for various graph elements (either vertices or arcs), with the enabled graph elements being substituted with "enabled wild cards".

Example 7. We derive the pattern $?^* \xrightarrow{sf}$ Point.x from the PCE declared on line 3 in Listing 2 using the path depicted in Ex. 5, where $?^*$ is an enabled wild card.[4] Note that the enablement is w.r.t. the PCE.

Pattern Matching: Pattern matching identifies paths with common sources and sinks as those containing enabled graph elements. Graph elements matching enabled wild cards are those whose represented JPS exhibit similar structural commonality with the JPSs selected by the PCE.

Example 8. The pattern in Ex. 7 would match (and only match) the paths Point.setX() $\xrightarrow{sf}$ Point.x and Point.setTwiceX() $\xrightarrow{sf}$ Point.x in Listing 1. Notice that the enabled wild card $?^*$ matches Point.setX() and Point.setTwiceX(), which corresponds to *all* and *only* the selected JPSs. This indicates that this pattern describes similar structural characteristics as the PCE from which it was derived. Note, though, that while the enabled wild card of the pattern Point $\xrightarrow{df}$ $?^*$ also matches both Point.setX() and Point.setTwiceX(), it also matches Point.getY(), whose corresponding JPS is not selected by the PCE. This indicates that, while this pattern expresses similar structural characteristics as the PCE, it is too broad.

Pattern Analysis: Patterns are compared with the PCE, producing a pattern *similarity* metric, which quantifies how closely the pattern resembles a PCE in terms of structural properties related to selected JPSs. The closer a pattern's similarity is to 1 (its range is in $[0, 1]$), the more closely the pattern matches similar structural commonality as that of the PCE. The equation to calculate the pattern-PCE similarity is depicted in equation (4) of Fig. 1.

Details of the pattern similarity metric are as follows. CG refers to the extended concern graph built from the original base-code when the Mylyn task is activated. In our motivating example, this graph would represent the code in Listing 1. Next, we define a function $match(\hat{\pi}, \Pi)$, where $\hat{\pi}$ ranges over the set of patterns and Π the power set of paths in CG. This

[3]It may not always be possible to fix the problem using a base-code change as doing so may break other PCEs.

[4]Patterns of greater lengths may contain wild cards that are not enabled.

$$err_\alpha(\hat{\pi}, PCE) = \begin{cases} 0 & \text{if } match(\hat{\pi}, paths(CG)) = \varnothing \\ 1 - \dfrac{|PCE \cap match(\hat{\pi}, paths(CG))|}{|match(\hat{\pi}, paths(CG))|} & \text{o.w.} \end{cases} \quad (1)$$

$$err_\beta(\hat{\pi}, PCE) = \begin{cases} 1 & \text{if } PCE = \varnothing \\ 1 - \dfrac{|PCE \cap match(\hat{\pi}, paths(CG))|}{|PCE|} & \text{o.w.} \end{cases} \quad (2)$$

$$abs(\hat{\pi}) = \begin{cases} 1 & \text{if } |\hat{\pi}| = 0 \\ \dfrac{|\mathcal{W}(\hat{\pi})|}{|\hat{\pi}|} & \text{o.w.} \end{cases} \quad (3)$$

$$sim(\hat{\pi}, PCE) = 1 - \left[err_\alpha(\hat{\pi}, PCE)(1 - abs(\hat{\pi})) + err_\beta(\hat{\pi}, PCE)abs(\hat{\pi})\right] \quad (4)$$

$$sel(jps, PCE) = \begin{cases} 1 & \text{if } jps \in PCE \\ 0 & \text{o.w.} \end{cases} \quad (5)$$

$$\mu(jps) = \left\{\hat{\pi} \;\middle|\; jps \in match(\hat{\pi}, paths(CG'))\right\} \quad (6)$$

$$\delta(PCE) = \left\{\hat{\pi} \;\middle|\; \hat{\pi} \text{ was derived from } PCE\right\} \quad (7)$$

$$chconf(jps, PCE) = \begin{cases} sel(jps, PCE) & \text{if } \mu(jps) \cap \delta(PCE) = \varnothing \\ \dfrac{1}{|\mu(jps) \cap \delta(PCE)|} \displaystyle\sum_{\hat{\pi} \in \mu(jps) \cap \delta(PCE)} |sel(jps, PCE) - sim(\hat{\pi}, PCE)| & \text{o.w.} \end{cases} \quad (8)$$

Fig. 1. PCE change confidence equation.

function, given a pattern and a set of paths, matches the pattern against the paths, resulting in a set of JPSs. These are the JPSs whose corresponding program elements exhibit the structural commonality represented by the pattern.

Equations (1), (2), and (3) are combined in the similarity calculation to measure patterns on three dimensions. Equation (1) is the err_α error rate attribute, which is akin to the ratio of the number of JPSs selected by both the PCE and the pattern when matched against finite, acyclic paths in the graph $paths(CG)$ to the number of JPSs solely selected by the pattern ($|PCE|$ refers to the number of JPSs selected by PCE). It is subtracted from 1 to create an error ratio in the statistical sense. It quantifies the pattern's ability in matching *solely* the JPSs within the PCE; the closer the err_α rate is to 0 the more likely the JPSs matched by the pattern are also ones within the PCE. If $\hat{\pi}$ does not match any JPSs, the err_α is 0 as it is vacuously precise.

Example 9. The pattern depicted in Ex. 7 would have a *small* (in fact, 0) err_α w.r.t. the PCE declared on line 3 of Listing 2, as both express exactly the same methods, namely, Point.setX() and Point.setTwiceX(). On the other hand, the pattern Point $\xrightarrow{dm}$?* would have a *larger* err_α w.r.t. the PCE declared on line 6 as the executions of Point.setX() and Point.setTwiceX() would be matched by the pattern but not selected by the PCE. Particularly, err_α here would be $\frac{2}{3}$ because, of the three method **execution**s matched by the pattern, only one of them is also selected by the PCE ($1 - \frac{1}{3}$).

Equation (2) is the err_β error rate attribute, which is akin to the ratio of the number of JPSs selected by both the PCE and the pattern when applied to paths in the graph to the number of JPSs selected solely by the PCE. Similar to err_α, the quantity is subtracted from 1 and its range is in $[0, 1]$. It quantifies the pattern's ability in matching *all* of the JPSs selected by the PCE; the closer the err_β rate is to 0 the more likely the pattern is to match all the JPSs selected by the PCE. If there are no JPSs selected by the PCE, the err_β is vacuously 1 (any pattern matches no JPSs).

Example 10. The pattern shown in Ex. 7 would have a *small* (0) err_β w.r.t. the PCE declared on line 3, Listing 2, as the pattern matches *all* of the methods selected by the PCE (i.e., the pattern "covers" the PCE). However, the same pattern would have a *large* (1) err_β w.r.t. the PCE declared on line 6, Listing 2, as *none* of the executions matched by the pattern are selected by the PCE (i.e., it does not cover the PCE).

Finally, equation (3) is the pattern abstractness (abbreviated abs), i.e., the ratio of wild card to concrete elements. $\mathcal{W}(\hat{\pi})$ projects the wild cards from a pattern $\hat{\pi}$, with $|\mathcal{W}(\hat{\pi})|$ being the number of wild cards in the pattern $\hat{\pi}$ and $|\hat{\pi}|$ being the total number of graph elements. An empty pattern has no concrete elements, thus, it has an abs of 1. For instance, the pattern in Ex. 7 has an abs of $\frac{1}{3}$.

We use abs because patterns containing many wild cards are more likely to match a greater number of concrete graph elements and vice versa. Thus, we combine the err_α and err_β rates by use of a weighted mean weighted by abs in equation (4). The reason is that a pattern that is very abstract is less likely to match JPSs that are *only* selected by a PCE. On the other hand, a pattern that is less abstract is less likely to match all JPSs selected by a PCE [17].

Example 11. Let $\hat{\pi}$ be the pattern from Ex. 7, PCE be the PCE declared on line 3 of Listing 2, and CG be the graph representing the base-code in Listing 1. Then, $sim(\hat{\pi}, PCE) = 1 - [(0)(\frac{2}{3}) + (0)(\frac{1}{3})] = 1$. Let $\hat{\pi}$ be Point $\xrightarrow{dm}$?* and PCE be the PCE declared on line 6. Then, $sim(\hat{\pi}, PCE) = 1 - [(\frac{2}{3})(\frac{2}{3}) + (0)(\frac{1}{3})] = \frac{5}{9}$.

Once the pattern similarity has been calculated, triples corresponding to an analyzed advice, a pattern derived using its bound PCE, and the pattern's similarity to the PCE are stored in memory for later use in the (next) detection phase. When all PCEs have been processed, the FRAGLIGHT is registered as a *Java Editor Change Listener*. In this way, it becomes an

"observer" of the editing pane where the base-code developer writes code. This allows FRAGLIGHT to observe keystrokes entered by the developer and detect when a new JPS is added; we explain this in more detail in the following section. Once a Mylyn task is deactivated, the tool is de-registered as a listener.

Phase II: Detection. In the detection phase, FRAGLIGHT determines new JPSs when keystrokes are entered by the developer in the IDE. For method execution JPSs, it finds new method declarations using Eclipse, which are the lowest level granularity whose addition information is available by this framework. FRAGLIGHT then includes its own code for JPSs residing within method bodies, e.g., method calls, adapting an AST differencing algorithm [23]. A new JPSs that FRAGLIGHT would detect is shown in Ex. 1.

Triples related to analyzed advice (PCEs), patterns, and similarity (calculated in the analysis phase) are retrieved. Then, the graph (CG) is augmented with information pertaining to the new base-code version using projects associated with the retrieved advice (resulting in CG').

Example 12. Adding the move() method in Listing 3 would result in new paths, e.g., Point $\xrightarrow{dm}$ move(), move() $\xrightarrow{sf}$ Point .x, move() $\xrightarrow{sf}$ Point.y, being added to CG, producing CG'.

Next, for each retrieved advice, its bound PCE *change confidence* (defined in equation (8)) value is calculated. First, we define a characteristic function sel in equation (5) s.t. $sel(jps, PCE) = 1$ if jps is selected by PCE and 0 otherwise. Recall that we treat a program as consisting of a set of JPSs that may or may not be currently selected by a PCE and treat a PCE as selecting a subset of these JPSs. As such, a jps is selected by PCE iff $jps \in PCE$.

Example 13. Let jps = **execution**(**void** Point.move(**double**, **double**)) and PCE be the PCE declared on line 3 of Listing 2. Then, we have that $sel(jps, PCE) = 0$ because, although move is a method of a class implementing Figure, its name does not begin with "set". Let jps = **execution**(**void** Point.setX(**double**)). Then, $sel(jps, PCE) = 1$.

In equation (6), $\mu(jps)$ is the set of all patterns that match jps when applied to the new base-code version CG'.

Example 14. Let jps = **execution**(**void** Point.move(**double**, **double**)), $k = 1$, and CG' be the graph representing the combined base-code of Listings 1 and 3. Then, $\mu(jps) = \{?^* \xrightarrow{sf} \mathsf{Point.x}, ?^* \xrightarrow{sf} \mathsf{Point.y}, \mathsf{Point} \xrightarrow{dm} ?^*\}$.

In equation (7), $\delta(PCE)$ is all patterns derived from PCE

Example 15. Let PCE be the PCE declared on line 3 of Listing 2. Then, $\delta(PCE) = \{?^* \xrightarrow{sf} \mathsf{Point.x}, \mathsf{Point} \xrightarrow{dm} ?^*\}$. Let PCE be the PCE declared on line 6. Then, $\delta(PCE) = \{?^* \xrightarrow{gf} \mathsf{Point.y}, \mathsf{Point} \xrightarrow{dm} ?^*\}$.

Finally, Equation (8) depicts the PCE change confidence equation, which produces a real number in $[0, 1]$ that corresponds to the *confidence* we have that PCE will need to be *changed* (i.e., it breaks) due to adding jps to the base-code. The closer the value is to 1, the more likely the PCE breaks because of the new JPS and vice-versa.

We now discuss the individual cases within equation (8). The case in which $\mu(jps) \cap \delta(PCE)$ is non-empty implies that there is at least one pattern s.t. it is derived from PCE and it matches jps, which is part of the new base-code. We consider the *similarity* of all such patterns to PCE. If a pattern is very similar to the PCE in terms of matching and selected JPSs, respectively, and jps is not selected by the PCE, i.e., $sel(jps, PCE) = 0$, then we are very confident that PCE has broken as a result of adding jps. In this case, we have that $|sel(jps, PCE) - sim(\hat{\pi}, PCE)|$ will be close to 1. The equation is then the average of the values for all patterns meeting the earlier stated criteria. If no patterns meet this criterion, i.e., $\mu(jps) \cap \delta(PCE) = \varnothing$, then the change confidence is simply whether or not the JPS is selected by the PCE, i.e., $sel(jps, PCE)$. This is because there are no patterns derived from the PCE that also match jps.

The reasoning behind equation (8) in Fig. 1 is as follows. When $\mu(jps) \cap \delta(PCE) = \varnothing$, none of the patterns derived from PCE, i.e., $\delta(PCE)$, matches jps as a result of applying them to CG'. In other words, jps shares *no* structural commonality with JPSs selected by PCE. Being that our hypothesis is that JPSs selected by a PCE typically share significant structural commonality, and this JPS shares *no* structural commonality with such JPSs, we suggest that jps *not* be selected by PCE. Then, the confidence we have in PCE breaking as a result of adding jps is just $sel(jps, PCE)$, i.e., 1 if jps is selected by PCE and 0 otherwise. In contrast, when $\mu(jps) \cap \delta(PCE) \neq \varnothing$, there exists a pattern derived from PCE that matches jps as a result of applying it to the new base-code. Here, we average the $chconf$ for all such patterns.

Example 16. Let jps = **execution**(**void** Point.move(**double**, **double**)), PCE be the PCE declared on line 3 of Listing 2, $k = 1$, and CG' be the graph representing the combined base-code of Listings 1 and 3. Per Ex. 14 and 15, we have that

$$|\mu(jps) \cap \delta(PCE)| = |\{?^* \xrightarrow{sf} \mathsf{Point.x}, \mathsf{Point} \xrightarrow{dm} ?^*\}| = 2$$

As such, we have that $chconf(jps, PCE)$

$$= \frac{1}{2}\Big(|sel(jps, PCE) - sim(?^* \xrightarrow{sf} \mathsf{Point.x}, PCE)| + |sel(jps, PCE) - sim(\mathsf{Point} \xrightarrow{dm} ?^*, PCE)|\Big)$$
$$= \frac{1}{2}\left(|0 - 1| + |0 - \frac{7}{9}|\right) = \frac{8}{9} \text{ (per Ex. 11 and 13)}$$

Let PCE be the PCE declared on line 6. Then,

$$|\mu(jps) \cap \delta(PCE)| = |\{\mathsf{Point} \xrightarrow{dm} ?^*\}| = 1$$

As such, we have that $chconf(jps, PCE)$

$$= |sel(jps, PCE) - sim(\mathsf{Point} \xrightarrow{dm} ?^*, PCE)|$$
$$= |0 - \frac{5}{9}| = \frac{5}{9} \text{ (per Ex. 11 and 13)}$$

Notice that the $chconf$ of the *broken* PCE (line 3) is *greater* than the $chconf$ of the *unbroken* PCE (line 6).

PCE Change Prediction. A PCE *change prediction* is created for PCEs with change confidences either below a *low* or above a *high* threshold. As a convenience, we add additional

information regarding the prediction depending on whether the newly added JPS is currently selected by the corresponding PCE. It is meant to guide the developer in determining how a broken PCE should be fixed, i.e., whether the new JPS should be removed from (a *negative* change prediction) or added to (a *positive* change prediction) the PCE.

Mylyn DOI Model Manipulation. FRAGLIGHT manipulates the Mylyn DOI model using the low and high confidence thresholds. If the PCE change confidence falls in the *low* confidence interval, the PCE is made *less* "interesting" in the DOI model, moving the developer's attention *away* from the PCE so that they may focus on the base-code. Conversely, if the change confidence falls in the *high* interval, the PCE is made *more* "interesting," bringing the developer's attention *towards* the PCE, so that they may focus on PCEs that may have broken as a result of their newly added base-code.

Example 17. Due to the small size of our example, let the low $chconf$ threshold be 0.6 and the high be 0.8. The scenario described in Ex. 16 results in a positive change prediction for the PCE declared on line 3 of Listing 2 as its $chconf$ is above the high threshold, thereby increasing the PCE's DOI value. Conversely, the PCE declared on line 6 has a $chconf$ below the low threshold, which results in a negative change prediction and a decrease in its DOI value. As such, the broken PCE receives a higher DOI value than the unbroken one.

IV. IMPLEMENTATION

FRAGLIGHT is implemented as a relation provider extension to the standard Mylyn Eclipse plug-in. The extended concern graph was constructed using the JayFX fact extractor [24], which we extended for use with modern Java languages and AspectJ. JayFX generates "facts," using class hierarchical analysis (CHA) [25], pertaining to structural properties and relationships between program elements, e.g., field accesses, method calls, in a particular project. Source code and transitively referenced libraries (possibly in binary format) are analyzed during graph building.

The AJDT was used to conservatively associate the graph with a PCE. Both pattern extraction and pattern-path matching were implemented via Drools (http://drools.org). A prototype implementation of FRAGLIGHT is publicly available (http://github.com/khatchad/fraglight).

V. CONCLUSION AND FUTURE WORK

We have detailed an approach, and corresponding implementation, that detects likely broken PCEs due to base-code changes. In the future, we plan to administer a full empirical evaluation of our proposed approach and other improvements.

REFERENCES

[1] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin, "Aspect oriented programming," in *ECOOP*, 1997.

[2] G. C. Murphy, R. J. Walker, E. L. A. Baniassad, M. P. Robillard, A. Lai, and M. A. Kersten, "Does aspect-oriented programming work?" *Commun. ACM*, 2001.

[3] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of aspectj," in *ECOOP*, 2001.

[4] C. Koppen and M. Stoerzer, "PCDiff: Attacking the fragile pointcut problem." in *Eur. Int. Workshop on Aspects in Software*, 2004.

[5] T. Aotani and H. Masuhara, "Scope: an aspectj compiler for supporting user-defined analysis-based pointcuts," in *AOSD*, 2007.

[6] L. Wang, T. Aotani, and M. Suzuki, "Improving the quality of aspectj application: Translating name-based pointcuts to analysis-based pointcuts," in *Int. Conf. Quality Software*, 2014.

[7] J. Aldrich, "Open modules: Modular reasoning about advice," in *ECOOP*, 2005.

[8] R. Khatchadourian, J. Dovland, and N. Soundarajan, "Enforcing behavioral constraints in evolving aspect-oriented programs," in *FOAL*, 2008.

[9] K. Hoffman and P. Eugster, "Bridging java and aspectj through explicit join points," in *PPPJ*, 2007.

[10] E. Bodden, É. Tanter, and M. Inostroza, "Join point interfaces for safe and flexible decoupling of aspects," *TOSEM*, 2014.

[11] A. Kellens, K. Mens, J. Brichau, and K. Gybels, "Managing the evolution of aspect-oriented software with model-based pointcuts," in *ECOOP*, 2006.

[12] M. Stoerzer and J. Graf, "Using pointcut delta analysis to support evolution of aspect-oriented software," in *ICSM*, 2005.

[13] J. Zhao, "Change impact analysis for aspect-oriented software evolution," in *Int. Workshop on Principles of Software Evolution*, 2002.

[14] L. Ye and K. D. Volder, "Tool support for understanding and diagnosing pointcut expressions," in *AOSD*, 2008.

[15] T. T. Nguyen, H. V. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Aspect recommendation for evolving software," in *ICSE*, 2011.

[16] J. Wloka, R. Hirschfeld, and J. Hänsel, "Tool-supported refactoring of aspect-oriented programs," in *AOSD*, 2008.

[17] R. Khatchadourian, P. Greenwood, A. Rashid, and G. Xu, "Pointcut rejuvenation: Recovering pointcut expressions in evolving aspect-oriented software," *IEEE Trans. Softw. Eng.*, 2012.

[18] M. Kersten and G. C. Murphy, "Mylar: a degree-of-interest model for ides," in *AOSD*, 2005.

[19] S. Clarke and R. J. Walker, "Composition patterns: An approach to designing reusable aspects," in *ICSE*, 2001.

[20] H. Masuhara, G. Kiczales, and C. Dutchyn, "A compilation and optimization model for aspect-oriented programs," in *Int. Conf. Compiler Construction*, 2003.

[21] S. Apel, "How aspectj is used: An analysis of eleven aspectj programs," *Journal of Object Technology*, 2010.

[22] M. P. Robillard and G. C. Murphy, "Concern graphs: finding and describing concerns using structural program dependencies," in *ICSE*, 2002.

[23] B. Fluri, M. Wursch, M. Pinzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Trans. Softw. Eng.*, 2007.

[24] B. Dagenais, S. Breu, F. W. Warr, and M. P. Robillard, "Inferring structural patterns for concern traceability in evolving software," in *ASE*, 2007.

[25] J. Dean, D. Grove, and C. Chambers, "Optimization of object-oriented programs using static class hierarchy analysis," in *ECOOP*, 1995.