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 changes made to the base-code is a daunting task, 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. We show that it is useful in revealing broken pointcuts by applying it to multiple versions of several open source projects and evaluating the quality of the recommendations produced against actual modifications.**

I. Introduction

Although using Aspect-Oriented Programming (AOP) [1] can be beneficial to developers in many ways [2–5], 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 [6], 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 [7]. 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 [8–13], limiting where advice may apply [14,15], or enforcing constraints on advice application [16–19]. Others make advice applicability more explicit [20] or do not use PCEs [21–23]. 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 in which the base-code must respect may be complex. Hence, although these languages may reduce fragility, they may render *detection* of broken PCEs more difficult [24].

Other approaches offer tool-support for detecting broken PCEs. The AspectJ Development Tools (AJDT) [25], 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 [26] augment the AJDT with *almost matching* join point information by relaxing PCEs using developer-minded heuristics but do not detect situations where join points are unintentionally selected by PCEs. Wloka et al. [27] automatically fix PCEs broken by refactorings, however, manual base-code edits may also break PCEs. In our previous work [28], we periodically 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 [29]. 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 [30].

Mylyn [31] 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 [32]. 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 the file,

edits the 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. For example, 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. In this way, possibly broken PCEs are presented to the developer in a variably invasive way. In other words, PCEs that likely need to the attention of the developer are presented more prominently in the IDE than ones that are less likely. The developer can then make alteration decisions based on FRAGLIGHTś recommendations, possibly adjusting the PCE or the base-code to rectify the problem. Our approach enables developers to discover problematic PCEs early in development so that they may be fixed before causing bugs that may compound over time. FRAGLIGHT alleviates much of the burden associated with identifying broken PCEs, making these systems easier to maintain.

FRAGLIGHT's recommendations are based on harnessing unique and arbitrarily deep structural commonalities between program elements corresponding to join points selected by a PCE in a particular software version. In [28], we showed that the majority of program elements corresponding to join points selected by a PCE in one base-code version shared such characteristics between them, and that these relationships persisted in subsequent versions. In this paper, we use this premise to detect broken PCEs on-the-fly.

This paper goes beyond [28] in that it:

Solves a different problem. Our previous approach, geared towards *aspect* developers[1], periodically suggests join points that may require inclusion in to a revised version of a PCE. Aspect developers may revise *PCEs*, possibly after *course-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. Here, we provide base-code developers with *immediate* feedback following *fine-grained* base-code changes that may have broke PCEs using a new, incremental algorithm. Base-code developers may then revise the *base-code* depending on the suggestion provided by this new approach.

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. A new, incremental algorithm is developed to enable on-the-fly, broken PCE detection. A new *confidence* equation for *PCEs* is presented that corresponds to the probability that the PCE has broken due to a base-code change.

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

[1]The distinction between aspect and base-code developers has been well documented. This is particularly relevant in regards to reusable aspects [33].

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;}}
```

Our key contributions can be summarized as follows:

Algorithm design. We present an automated approach that programmatically manipulates the Mylyn DOI model to bring broken PCEs to the base-code developer's attention early. The developer is informed, with a subtly that varies on likelihood, when their code is likely to break PCEs as it is being written.

Implementation and experimental evaluation. To ensure real-world applicability, we implemented our approach was as a seamless, publicly available extension to the Mylyn. A study on 14 version changes consisting of 5,711 base-code edits of AspectJ programs indicates that the technique is effective and practical in detecting broken PCEs as the base-code developer types. Upon completion of the experiments, the average DOI value of PCEs that *actually* broke were, on average, 2.5 times greater than that of PCEs that *did not* break throughout versions. This indicates that using our approach results in broken PCEs being 2.5 times more prominently displayed in the IDE than unbroken PCEs, bringing broken PCEs to the forefront while keeping unbroken PCEs in the background. These results advance the state of the art in automated tool-support for AOP evolution.

II. MOTIVATING EXAMPLE

We motivate our approach using a simple yet classic graphics application inspired by Kiczales et al. [6]. Though the example is small, we use its simplicity to detail our approach.

Listing 1 portrays a code snippet of a simple Point class (line 1) that implements a Figure (interface not shown) on a Cartesian plane. There are two instance fields, x and y, declared on line 2. Also, there are two mutator instance methods for field x (mutators for y have been omitted for presentation purposes), 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 (the accessor for x has been omitted for presentation purposes), declared on line 5, that returns the field value.

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 getX() 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 either changing the base-code 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.

[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.

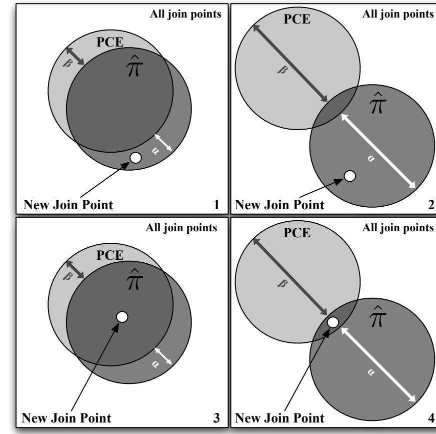

Fig. 1. Determining PCE breakage due to a JPS addition.

III. APPROACH

A. Overview

Fig. 1 depicts Venn diagrams of four canonical situations and how FRAGLIGHT applies. In each, the universe is all of 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 [34].

We treat a program as a set of JPSs that *may* or *may not* be under the influence of advice, which helps simplify the presentation. 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 (discussed in §IV-A) so that PCEs utilizing dynamic conditions may nevertheless be used as input to our tool. The impact of this limitation is minimal [28]. Moreover, there is evidence that suggests that most PCEs do not take advantage of dynamic conditions [35].

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.

In the scenarios depicted in Fig. 1, a single PCE in the program is portrayed to simplify the presentation. The region labeled *PCE* represents the set of all JPSs selected by the PCE. The region labeled $\hat{\pi}$ represents all JPSs corresponding to program elements matched by a particular *structural pattern*. Structural patterns depict organizational relationships between

program elements, e.g., all methods declared by a class (a single depth pattern), all methods whose bodies textually contain a call to methods whose bodies include a statement that writes to a particular field (a multi-depth pattern).

Again for simplification, Fig. 1 presents a single structural pattern, whereas many structural patterns may exist in a given program. The α and β metrics, which are related to type I and type II errors, respectively, are used for measuring the similarity between a particular PCE and structural pattern (discussed in §III-B1). α measures how *closely* a pattern resembles a PCE, while β measures its *completeness*.

Program elements corresponding to JPSs selected by a PCE share a *high* degree of structural commonality iff there exists structural patterns that have *small* α and β errors w.r.t. the PCE. Conversely, elements corresponding to JPSs selected by a PCE do *not* share a high degree of commonality iff there exists patterns that have *large* α and β errors w.r.t. the PCE.

We now describe how predictions are made in each of the four situations in Fig. 1, where we consider a single PCE, structural pattern, and input JPS. The numbers at the bottom right corner of the diagrams correspond to the numbers below:

1) If a new JPS is added to the base-code that *shares* a *high* degree of structural commonality with JPSs selected by an existing PCE and is *not* selected by the PCE, we consider the PCE to be *more* "interesting," as it *may* need to be altered to *include* the new join point.
2) If a new JPS is added to the base-code that does *not* share a *high* degree of structural commonality with JPSs selected by the PCE and is *not* selected by the PCE, we consider the PCE to be *less* "interesting," as it is *unlikely* that the PCE needs to *include* the new join point.
3) If a new JPS is added to the base-code that *shares* a *high* degree of structural commonality with JPSs selected by the PCE and *is* selected by the PCE, we consider the PCE to be *less* "interesting," as it is *unlikely* that the PCE needs to *exclude* the new join point.
4) If a new JPS is added to the base-code that does *not* share a *high* degree of structural commonality with JPSs selected by the PCE and *is* selected by the PCE, we consider the PCE to be "more" interesting as it *may* need to be altered to *exclude* the new join point.

B. Workflow Details

1) Phase I: Analysis:

a) Pointcut Analysis Scope: The analysis phase (Fig. 2) is triggered when a Mylyn task is activated (step 1). At this time, a set of advice-bound PCE representations is collected from the current *pointcut analysis scope* (PAS), which is based on the *degrees-of-separation* concept used by Kersten and Murphy [32] in their *Active Search* feature (step 2). PCEs bound to advice in this set are those that FRAGLIGHT will later consider during the detection phase (described in §III-B2) when predicting broken PCEs due to added JPSs. Thus, it is these and only these PCEs that can possibly be included in our change predictions. This helps control tractability by allowing only a subset of PCEs to be analyzed. In our

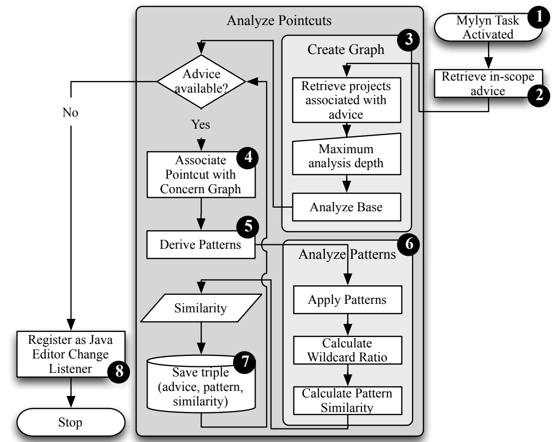

Fig. 2. Flowchart of the analysis phase that starts when a task is activated.

implementation (described in §IV-A), and later used during our experiments (described in §IV-C, IV-D), the *workspace* scope, which considers all PCEs available in all projects in a developer's workspace, is the default.

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

b) Concern Graphs: In step 3, an *extended concern graph* is built from projects that include the aspects whose advice-bound PCEs are in the PAS. A concern graph is a directed multigraph depicting structural relations (e.g., calling, declarations, package containment) between program elements (e.g., types, methods, fields) [36]. We extended 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.

c) Maximum Analysis Depth: A maximum analysis depth (k) is also a parameter to control tractability. It controls the depth of the structural relations considered. In §IV-B, we discuss our choice for the analysis depth for our experiments.

d) Pattern Extraction: Next, each PCE in the PAS is associated with the graph (step 4). 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 [28].

Algorithmically, pattern extraction works by first enumerat-

ing 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" (step 5).

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[3]. Note that the enablement is w.r.t. the PCE.

e) 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.

We next detail how patterns and PCEs are compared.

f) Pattern Analysis: Step 6 is responsible for comparing the derived patterns with the PCE (as demonstrated above) and 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. 3.

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 step 3. 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 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.

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

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 (cf. α discussed in §III-A), 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* (in fact, 0) err_{β} w.r.t. the PCE declared on line 3 of 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* (in fact, 1) err_{β} w.r.t. the PCE declared on line 6 of Listing 2, as *none* of the method 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.

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

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

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

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

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

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

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

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

Fig. 3. PCE change confidence equation.

On the other hand, a pattern that is less abstract is less likely to match all JPSs selected by a PCE [28].

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 (step 7) for later use in the (next) detection phase. When all PCEs have been processed, the FRAGLIGHT is registered as a *Java Editor Change Listener* [37] (step 8). 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.

2) Phase II: Detection: In the detection phase (Fig. 4), FRAGLIGHT determines new JPSs when keystrokes are entered by the developer in the IDE (step 1). For method execution JPSs, it finds new method declarations using Eclipse [38], 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 [39]. A

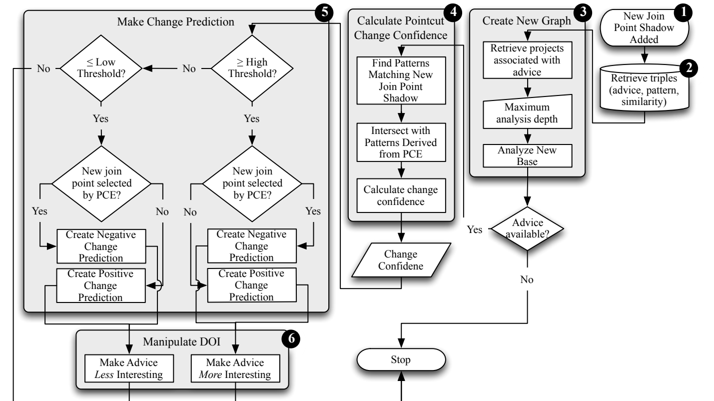

Fig. 4. Flowchart depicting the detection phase that commences when a new join point is added.

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 in step 2. 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', step 3).

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 (step

4). 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, $\mu(jps) = \{?^* \xrightarrow{sf} \text{Point.x}, ?^* \xrightarrow{sf} \text{Point.y}, \text{Point} \xrightarrow{dm} ?^*\}$.

In equation (7), $\delta(PCE)$ is all patterns derived from PCE (obtained from step 2 of Fig. 4).

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

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. This situation corresponds to the top left (1) Venn diagram in Fig. 1. Each of the other Venn diagrams correspond to situations where the limits of sel and sim go to 0 and 1, respectively. 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. 3 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

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

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

$$= \frac{1}{2}\Big(|sel(jps, PCE) - sim(?^* \xrightarrow{sf} \text{Point.x}, PCE)| + |sel(jps, PCE) - sim(\text{Point} \xrightarrow{dm} ?^*, PCE)|\Big)$$

$$= \frac{1}{2}\left(|0 - 1| + |0 - \frac{7}{9}|\right) = \frac{8}{9} \text{ (per Ex. 11 and 13)}$$

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

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

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

$$= |sel(jps, PCE) - sim(\text{Point} \xrightarrow{dm} ?^*, PCE)|$$

$$= |0 - \frac{5}{9}| = \frac{5}{9} \text{ (per Ex. 11 and 13)}$$

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

3) PCE Change Prediction: A PCE *change prediction* is created for PCEs with change confidences either below a *low* or above a *high* threshold (step 5). 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.

4) Mylyn DOI Model Manipulation: FRAGLIGHT manipulates the Mylyn DOI model (step 6) 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. Experimental Evaluation

A. Implementation

FRAGLIGHT is implemented as a relation provider extension to the standard Mylyn Eclipse plug-in. The extended concern graph was constructed with the aid of the JayFX fact extractor [36], which we extended for use with modern Java languages and AspectJ (with the latter being as part of our previous work [28]). JayFX generates "facts," using class hierarchical analysis (CHA) [40], pertaining to structural properties and relationships between program elements, e.g., field accesses, method calls, in a particular project. Its lightweight representation of program elements makes for an efficient analysis. Source code and transitively referenced libraries (possibly in binary format) are analyzed during graph building.

The AJDT compiler was leveraged to conservatively (explained next) associate the graph with a PCE. For a given PCE, the AJDT compiler produces the Java program elements, e.g., method declarations, method calls, field sets, correlated with selected JPSs. Both pattern extraction and pattern-path matching were implemented via the Drools Rules Engine [41], which uses a modified RETE algorithm [42]. Drools provides a natural query language and an efficient solution to the many-to-many matching problem. A prototype implementation of FRAGLIGHT is publicly available [43].

B. Study Configuration

To assess the usefulness and effectiveness of our approach in detecting broken (and unbroken) PCEs, we examined the final DOI values of PCEs (as manipulated by only FRAGLIGHT) after replaying a series of base-code changes from software version histories. We assume that elements with a higher DOI value, thereby being more prominent in the IDE, will be more noticeable by developers and vice-versa. Note that an assessment using a confusion matrix, precision, and recall do not directly apply here as these notions are built in to the DOI model [30,32]. That is, the DOI model is a scale; adding weight to one elements removes it from another. It is very much a sorting mechanism that strives to display the most relevant IDE UI elements to the developer for a specific task. As such, comparing the ratio of DOI values between broken and unbroken PCEs suffices as an effective assessment.

Although a user study would be useful, we chose a software evolution simulation using version histories for several reasons. Firstly, user studies have a number of barriers [44]. Secondly, we desired to isolate the DOI manipulation that was due to our tool and not by other UI interactions performed by the developer. This way, we can ensure that we accurately assess FRAGLIGHT's change predictions. Thirdly, using version histories provides an noninvasive, unbiased way to assess our approach. Nevertheless, we consider a user study for future work to enhance the results presented here.

With the initial DOI value of each PCE at 0, we say that a successful DOI manipulation is one where broken PCEs had a higher DOI value than those that did not break. In this case, broken PCEs are brought to the developer's attention, whereas PCEs that did not break remain in the background. Note that it is important to, during the experiments, not manually manipulate the DOI, e.g., by manually clicking on any of the IDE elements so that we can be sure that FRAGLIGHTś predictions are solely responsible for these values. We visit possible drawbacks of this approach in §IV-E.

For this experiment, we set $k = 1$ (see §III-B1c), which keeps the tool run time short so that predictions can be made as quickly as possible since the analysis runs while the developer is typing. Moreover, we set the low and high confidence thresholds parameters to 0.15 and 0.55, respectively, meaning that PCEs assigned a *chconf* ≤ 0.15 resulted in a decreased DOI, while ones of ≥ 0.55 resulted in an increase. We empirically found that these thresholds worked the best with our corpus. In the future, we plan to more thoroughly assess trade-offs between analysis depth and prediction time, as well as optimal threshold values.

Table I includes our subjects along with associated number of discrete releases (column *vers.*) analyzed, total non-blank, non-commented lines of code (counted using SLOCCount [45]), which excludes code contained within aspect files, between all versions (column *LOC*), ranging from an average of $\approx$ 1.4K per version for MobilePhoto and $\approx$ 6K for HealthWatcher, and total number of analyzed (advice-bound) PCEs throughout versions (column *aPCE*). Subject source code and other information can be found on our website [46], as well as in the literature [47,48].

Column *at (s)* depicts the total PCE analysis time in secs for all versions. Analysis occurs when the developer activates a Mylyn task (normally at the start of working on a particular bug or feature). Then, all PCEs in the PAS are analyzed. For each version, the analysis was repeated three times, with the results of each averaged, using a 2.83 GHz Intel machine. The JVM heap size was 5 GB. The average was $\approx$ 1.05 secs per KLOC and $\approx$ 0.14 secs per PCE, which indicates that the analysis time is practical for even large applications.

Column *JPS* is the total number of JPSs added between subject versions. These are the JPSs used as input, some of which broke PCEs and others that did not. Since we collected PCE statistics after inputting all JPSs added between versions to our tool, is was not important to identify precisely which JPSs caused particular PCEs to break. Instead, classifying which PCEs broke and which did not was sufficient.

As noted in §III, the output of our tool is a PCE change prediction, which, in turn, manipulates the Mylyn DOI. All of this is done in the background as the developer is working. We should note also that the predictions occur in a separate thread, which fortunately does not interrupt the developer's workflow. However, having short prediction times (i.e., the

subject	vers.	LOC	aPCE	at (s)	JPS	pt (s)	bPCE	uPCE	bDOI	σ_{bDOI}	uDOI	σ_{uDOI}
HealthWatcher	8	47537	217	47.20	2648	1.1e4	6	29	1.17	0.98	0.21	0.77
MobilePhoto	6	8331	196	11.18	3063	3.5e3	29	58	2.21	2.54	1.14	1.78
Totals:	14	55868	413	58.39	5711	1.5e4	35	87	2.03[a]	2.37[a]	0.83[a]	1.58[a]

[a] Arithmetic mean

TABLE I
EXPERIMENTAL RESULTS.

amount of time needed for FRAGLIGHT to generate a PCE change prediction) is advantageous so that broken PCEs are brought to the developer's attention as early as possible. Column *pt (s)* portrays the total prediction time in secs during our experiment, which averaged ≈ 2.61 per added JPS. This indicates that the developer would see programmatic changes in the DOI made by FRAGLIGHT on average ≈ 2.61 secs after adding a new JPS to the base-code, which is practical. The remaining columns will be discussed shortly.

The order in which JPSs were used as input to our tool is insignificant. This may be unintuitive as applying patterns to different base-codes is likely to match different JPSs, however, the only base-code we are interested in the patterns matching is the JPS being added. As such, the order in which JPSs are added to the old base-code version to obtain the new base-code version is irrelevant as each JPS is considered in isolation.

Columns *bPCE* and *uPCE* are the total number of broken and unbroken PCEs between versions, respectively. For this, we use the conditions for a PCE to be considered broken between subsequent base-code versions from [28]. We say that a PCE in version v_i *broke* in version v_j where $i < j$ iff:

1) the textual representation of the PCE in v_i differs from its textual representation in v_j,
2) the JPSs selected by the PCE in v_j differs from the JPSs selected by its old representation in v_j.

Criterion 1 asserts that the PCE was rewritten between versions, i.e., they textually differ. Criterion 2 excludes situations where the PCE selects the same JPSs between versions.

PCEs that meet these criteria are those that required textual modification to allow the PCE to continue to capture intended join points. We discuss possible drawbacks for using these criteria to identify broken PCEs in version history in §IV-E.

C. Quantitative Analysis

After simulating the addition of JPSs between versions of our subjects, we then collected the resulting PCE DOI values. The hope is that broken PCEs resulted in a higher DOI value than that of unbroken PCEs. In this case, broken PCEs would appear more prominently in the IDE than unbroken PCEs, so that developers can direct their attention to the problem early. Columns *bDOI* and *uDOI* depict the average final DOI value of broken and unbroken PCEs, respectively, while σ_{bDOI} and σ_{uDOI} portray the corresponding standard deviations. These columns show the average final PCE DOI values after adding all the new JPSs between versions v_i and v_{i+1} to v_i for all $i = 1 \ldots k-1$ where k is the number of subject versions.

From Table I, the average DOI value of PCEs that *actually* broke are, on average, 2.5 times greater than the average DOI value of PCEs that *did not* break. Recall that the resulting DOI values are completely and only due to FRAGLIGHT's manipulation. These results indicate that FRAGLIGHT is promising in bringing broken PCEs to the developers' attention, while hiding unbroken PCEs, all while they are typing. Particularly, using our approach results in broken PCEs being 2.5 times more prominently displayed in the IDE than unbroken PCEs. Moreover, FRAGLIGHT presents its results to the developer in a familiar way using existing, well-integrated IDE mechanisms (i.e., Mylyn). Because of Mylyn, FRAGLIGHT's results are propagated throughout all UI elements where PCEs are visible, making the results consistent among views.

D. Qualitative Analysis

We now analyze several situations where our tool performed as expected and vice-versa. For succinctness, we draw examples from only the HealthWatcher subject. We begin with a scenario where our tool assigned a *high* DOI to a PCE that *broke* between versions. The aspect synchronizes a methods in "record" types using a concurrency manager. The PCE broke between versions 8 and 9 due to adding a new record types (representing diseases and symptoms), which resulted in the PCE selecting two additional join points. Adding these new join points caused FRAGLIGHT to produce 2 predictions, averaging a $chconf = 0.67$. One pattern that was used was one matching accesses to a field. The JPSs added in the subsequent version were methods that also accessed this field, and being that they were not selected by the original PCE, FRAGLIGHT increased the PCE's DOI value to 2. The DOI value was not higher because there is another method in an unrelated class that also accesses this field but is not selected by the PCE.

We now discuss an instance where our tool assigned a *low* DOI to a PCE that *broke* between versions. Changes made in versions 1 to 2 involved introducing the Command pattern [49] to replace the individual servlets that implemented each of the operations provided by HealthWatcher. Consequently, a PCE in an aspect responsible for computation distribution broke. To rectify the problem, the PCE was rewritten to no longer select join points contained in classes derived from a particular servlet but instead to select join points contained in classes derived from a servlet following the Command pattern. Unfortunately, the final DOI value for this PCE was 0 as FRAGLIGHT produced no predictions for this PCE. The reason was that all patterns that were derived from the PCE in version 1 were invalidated by version 2. That is, the program

elements referred to in the structural patterns derived from the PCE in the first version were no longer present in the second. As such, applying the patterns from version 1 produced no matches in version 2. Thus, no predictions were made, which suggests that our approach may not be effective in situations involving widespread, atomic refactorings. However, not all refactoring (especially to patterns) can be fully automated in an atomic fashion. We predict that our tool would perform well in situations where the changes involve several intermediate steps, which would provide FRAGLIGHT the opportunity to match more existing patterns against old base-code.

Next, we turn to an instance where our tool assigned a *low* DOI to a PCE that *did not* break between analyzed versions. A PCE in a synchronization aspect did not break between *any* subject versions. Furthermore, its final DOI value following the experiment was 0. As previously discussed, HealthWatcher was refactored to use the Command pattern between versions 1 and 2. However, this PCE selected join points not related to this refactoring and thus did not break. Four predictions were made between these versions, resulting in an average $chconf = 0.11$. The derived patterns did not exhibit strong structural commonality with that of the PCE as they expressed calls to such common methods as String.equals(). Moreover, the patterns did not match the added JPSs. This resulted in a low PCE DOI value.

Finally, we detail a scenario where our tool assigned a *high* DOI value to a PCE that *did not* break. One such instance occurred with a PCE in an aspect responsible for catching exceptions raised inside Observer pattern [49] implementations and displaying exception details in a web page. FRAGLIGHT produced 17 change predictions between all versions for the PCE, resulting in an average $chconf = 0.48$. The final DOI value for this PCE was 3. The PCE was **execution**(**void** Update∗Data.executeCommand(..)), meaning that executeCommand() methods declared in classes whose name starts with Update and ends with Data are selected. One of the extracted patterns matched calls to CommandRequest.isAuthorized(). An added JPS in version 8, namely, the execution of the method UpdateSymptomSearch.executeCommand(), includes a call to CommandRequest.isAuthorized(), as such, it matches the pattern. However, the PCE correctly selects base-code pertaining to the *Observer* pattern (i.e., the *Data* is "observed") and not that of *Search*es. Since the CommandRequest.isAuthorized() is called from within many of *Data* classes and that the new JPS matched the pattern, this mislead our tool to suggest that the PCE had broken.

E. Threats to Validity

Several threats may diminish our evaluation results. We discuss here how their effects have been minimized. Our evaluation aimed to simulate FRAGLIGHT's performance in a real-world setting. We drew data from multiple versions of two projects, which may not be representative of AO projects at large. However, these subjects have been extensively studied previously in the literature. Moreover, they comprise publicly available open source projects, which are contributed to by a number of developers. Although only two projects were used, they constitute fourteen versions with large deltas (a total of 5,711 added JPSs).

We assumed that all PCEs are correctly written between version deltas, which correspond to major subject release points. Our assumption is that, prior to a major release, all PCEs select and only select intended join points. This is essential in determining which PCEs broke and in which versions. Moreover, we assumed that broken PCEs were fixed by rewriting the PCE. Yet, there are other ways to "fix" a broken PCE, namely, by changing the *base-code* to conform to the PCE. For example, to fix the broken PCE portrayed in §II, we could change the name of the move method to setBothXandY.

It would be difficult to use base-code conformance as a reliable means to determine broken PCEs as there are many reasons that base-code can change, including fixing a broken PCE. However, it is reasonable to assume that the only reason PCEs change is because they are broken. Moreover, it is reasonable to assume that the majority of PCEs are fixed by altering the PCE itself.

When assessing the changes of DOI in our experiments, we began the DOI flat, i.e., 0, and then fed a series of added JPSs to the tool to obtain the subsequent version. No other factors affected the DOI other than the programmatic manipulation performed by FRAGLIGHT, which allowed us to focus on the quality of its predictions. However, in a real-world setting, the DOI may be affected by other events that occur within the IDE, such as developer clicks and navigation. As such, more investigation may be necessary to assess the effectiveness of FRAGLIGHT's programmatic DOI manipulation in combination with other events while the developer is typing, which we plan for future work.

V. RELATED WORK

Wang et al. [50] automatically create *analysis-based* PCEs [8] from traditional named-based ones, such as those in AspectJ, which may avoid fragility issues. However, there is no round trip support to convert these PCEs back in cases where they do need to change.

Nguyen et al. [51] propose an approach geared towards aspect *mining*, i.e., converting non-AO programs to AO ones, which also works on maintaining *existing* AO systems. However, their approach is focused on incorporating missed JPSs into PCEs, whereas our approach is for detecting broken PCEs, either by inclusion *or* exclusion of JPSs, as the developer is typing. Moreover, the results of our tool are incorporated into an existing system (Mylyn) for focusing developer attention on particular software elements.

Zhao [52] presents a change impact analysis for AOP. They detail an "Advice Invocation Change" (AIC) that indicates which PCEs are *affected* by new JPSs, but such an affect may not be a PCE breakage. Conversely, a new JPS that does not produce an AIC could also result in a broken PCE.

VI. Conclusion and Future Work

We have detailed an approach that detects likely broken PCEs due to the addition of a new JPS. We showed how our approach works with the Eclipse editor, as well as its integration features with the popular Mylyn plug-in, so that these PCEs are brought to the base-code developer's attention, with likely unbroken PCEs moved to the background, in a standardized, consistent way, all while the developer is typing. Also, we showed, via an empirical evaluation, that our approach is effective in bringing broken PCEs to light, with such PCEs having DOI values that are, on average, 2.5 times greater than the average DOI value of unbroken PCEs.

In the future, we plan to persist the patterns along with the Mylyn context as to avoid rebuilding them if there are no changes in the base-code between task activations. This may have a performance impact in certain situations. Furthermore, the analysis phase commences only when a Mylyn task is activated, which occurs when a developer starts to work on a particular bug or feature. We plan to further investigate the optimal time to reanalyze the base-code.

References

[1] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin, "Aspect oriented programming," in *European Conference on Object-Oriented Programming*, 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] M. Kersten and G. C. Murphy, "Atlas: A case study in building a web-based learning environment using aspect-oriented programming," in *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1999.

[4] M. Lippert and C. V. Lopes, "A study on exception detection and handling using aspect-oriented programming," in *International Conference on Software Engineering*, 2000.

[5] R. Walker, E. Baniassad, and G. Murphy, "An initial assessment of aspect-oriented programming," in *International Conference on Software Engineering*, 1999.

[6] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An Overview of AspectJ," in *European Conference on Object-Oriented Programming*, 2001.

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

[8] T. Aotani and H. Masuhara, "Scope: an aspectj compiler for supporting user-defined analysis-based pointcuts," in *International Conference on Aspect-Oriented Software Development*, 2007.

[9] K. Ostermann, M. Mezini, and C. Bockisch, "Expressive Pointcuts for Increased Modularity," in *European Conference on Object-Oriented Programming*, 2005.

[10] W. Cazzola, S. Pini, and M. Ancona, "Design-Based Pointcuts Robustness Against Software Evolution," in *Workshop on Reflection, AOP, and Meta-Data for Software Evolution*, 2006.

[11] K. Sakurai and H. Masuhara, "Test-based pointcuts for robust and fine-grained join point specification," in *International Conference on Aspect-Oriented Software Development*, 2008.

[12] K. Klose and K. Ostermann, "Back to the Future: Pointcuts as Predicates over Traces," in *International Workshop on Foundations of Aspect-Oriented Languages*, 2005.

[13] L. M. Seiter, "Role annotations and adaptive aspect frameworks," in *International Workshop on Linking aspect technology and evolution*, 2007.

[14] J. Aldrich, "Open Modules: Modular Reasoning About Advice," in *European Conference on Object-Oriented Programming*, 2005.

[15] S. Gudmundson and G. Kiczales, "Addressing Practical Software Development Issues in AspectJ with a Pointcut Interface," in *Workshop on Advanced Separation of Concerns at the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Oct. 2001.

[16] R. Khatchadourian, J. Dovland, and N. Soundarajan, "Enforcing behavioral constraints in evolving aspect-oriented programs," in *International Workshop on Foundations of Aspect-Oriented Languages*, 2008.

[17] W. G. Griswold, K. Sullivan, Y. Song, M. Shonle, N. Tewari, Y. Cai, and H. Rajan, "Modular Software Design with Crosscutting Interfaces," *IEEE Softw.*, 2006.

[18] K. Sullivan, W. G. Griswold, Y. Song, Y. Cai, M. Shonle, N. Tewari, and H. Rajan, "Information hiding interfaces for aspect-oriented design," in *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2005.

[19] M. Bagherzadeh, H. Rajan, G. T. Leavens, and S. Mooney, "Translucid contracts: expressive specification and modular verification for aspect-oriented interfaces," in *International Conference on Aspect-Oriented Software Development*, 2011.

[20] K. Hoffman and P. Eugster, "Bridging Java and AspectJ through explicit join points," in *International Symposium on Principles and Practice of Programming in Java*, 2007.

[21] H. Rajan and G. Leavens, "Ptolemy: A Language with Quantified, Typed Events," in *European Conference on Object-Oriented Programming*, 2008.

[22] E. Bodden, É. Tanter, and M. Inostroza, "Join point interfaces for safe and flexible decoupling of aspects," *ACM Transactions on Software Engineering and Methodology*, 2014.

[23] F. Steimann, T. Pawlitzki, S. Apel, and C. Kästner, "Types and modularity for implicit invocation with implicit announcement," *ACM Transactions on Software Engineering and Methodology*, 2010.

[24] A. Kellens, K. Mens, J. Brichau, and K. Gybels, "Managing the Evolution of Aspect-Oriented Software with Model-based Pointcuts," in *European Conference on*

Object-Oriented Programming, 2006.
[25] A. Clement, A. Colyer, and M. Kersten, "Aspect-oriented programming with ajdt," in *ECOOP Workshop on Analysis of Aspect-Oriented Software*, 2003.
[26] L. Ye and K. D. Volder, "Tool Support For Understanding and Diagnosing Pointcut Expressions," in *International Conference on Aspect-Oriented Software Development*, 2008.
[27] J. Wloka, R. Hirschfeld, and J. Hänsel, "Tool-supported refactoring of aspect-oriented programs," in *International Conference on Aspect-Oriented Software Development*, 2008.
[28] R. Khatchadourian, P. Greenwood, A. Rashid, and G. Xu, "Pointcut rejuvenation: Recovering pointcut expressions in evolving aspect-oriented software," *IEEE Transactions on Software Engineering*, 2012.
[29] The Eclipse Foundation, "The eclipse development environment," 2015, last checked January 20, 2015. [Online]. Available: http://www.eclipse.org
[30] M. Kersten and G. C. Murphy, "Mylar: a degree-of-interest model for IDEs," in *International Conference on Aspect-Oriented Software Development*, 2005.
[31] The Eclipse Foundation, "The mylyn task and application lifecycle management framework," 2015, last checked January 20, 2015. [Online]. Available: http://www.eclipse.org/mylyn
[32] M. Kersten and G. C. Murphy, "Using task context to improve programmer productivity," in *ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2006.
[33] S. Clarke and R. J. Walker, "Composition patterns: An approach to designing reusable aspects," in *International Conference on Software Engineering*, 2001.
[34] H. Masuhara, G. Kiczales, and C. Dutchyn, "A Compilation and Optimization Model for Aspect-Oriented Programs," in *International Conference on Compiler Construction*, 2003.
[35] S. Apel, "How AspectJ is Used: An Analysis of Eleven AspectJ Programs," *Journal of Object Technology*, 2010.
[36] B. Dagenais, S. Breu, F. W. Warr, and M. P. Robillard, "Inferring structural patterns for concern traceability in evolving software," in *International Conference on Automated Software Engineering*, 2007.
[37] IBM, "Eclipse Documentation - Interface IElementChangeListener," 2012, last checked January 23, 2015. [Online]. Available: http://help.eclipse.org/juno/topic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/ui/texteditor/IElementStateListener.html
[38] ——, "Eclipse Documentation - Interface IJavaElementDelta," 2012, last checked January 23, 2015. [Online]. Available: http://help.eclipse.org/galileo/topic/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/IJavaElementDelta.html
[39] B. Fluri, M. Wursch, M. Pinzger, and H. Gall, "Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction," *IEEE Transactions on Software Engineering*, 2007.
[40] J. Dean, D. Grove, and C. Chambers, "Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis," in *European Conference on Object-Oriented Programming*, 1995.
[41] Red Hat, Inc., "Drools business rule management system," 2015, last checked January 23, 2015. [Online]. Available: http://www.drools.org
[42] C. L. Forgy, "Rete: a fast algorithm for the many pattern/many object pattern match problem," *Artificial Intelligence*, 1982.
[43] Eclipse Labs, "fraglight: A tool for early detection of broken pointcuts in evolving aspect-oriented software," 2015, last checked January 23, 2015. [Online]. Available: http://code.google.com/a/eclipselabs.org/p/fraglight
[44] R. P. Buse, C. Sadowski, and W. Weimer, "Benefits and barriers of user evaluation in software engineering research," in *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2011.
[45] D. Wheeler. SLOCCount. Last checked May 15, 2015. [Online]. Available: http://www.dwheeler.com/sloccount
[46] New York City College of Technology, City Univeristy of New York, "Pointcut change prediction," 2015, last checked January 23, 2015. [Online]. Available: http://openlab.citytech.cuny.edu/pcp
[47] E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. C. Filho, and F. Dantas, "Evolving software product lines with aspects: an empirical study on design stability," in *International Conference on Software Engineering*, 2008.
[48] P. Greenwood, T. T. Bartolomei, E. Figueiredo, M. Dósea, A. F. Garcia, N. Cacho, C. Sant'Anna, S. Soares, P. Borba, U. Kulesza, and A. Rashid, "On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study," in *European Conference on Object-Oriented Programming*, 2007.
[49] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, Mar. 1995.
[50] L. Wang, T. Aotani, and M. Suzuki, "Improving the Quality of AspectJ Application: Translating Name-Based Pointcuts to Analysis-Based Pointcuts," in *International Conference on Quality Software*, 2014.
[51] T. T. Nguyen, H. V. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Aspect recommendation for evolving software," in *International Conference on Software Engineering*, 2011.
[52] J. Zhao, "Change impact analysis for aspect-oriented software evolution," in *International Workshop on Principles of Software Evolution*, 2002.