2008

Pointcut Rejuvenation: Recovering Pointcut Expressions in Evolving Aspect-Oriented Software

Raffi T. Khatchadourian
CUNY Hunter College

Phil Greenwood
Relative Insight

Awais Rashid
University of Bristol

Guoqing Harry Xu
University of California. Los Angeles

Pointcut Rejuvenation: Recovering Pointcut Expressions in Evolving Aspect-Oriented Software

Raffi Khatchadourian, Phil Greenwood, Awais Rashid, and Guoqing Xu

Technical Report
COMP-001-2008
August 2008
Revised March 2009, May 2009

Computing Department
InfoLab21
South Drive
Lancaster University
Lancaster LA1 4WA UK

Pointcut Rejuvenation: Recovering Pointcut Expressions in Evolving Aspect-Oriented Software*

Raffi Khatchadourian†
Ohio State University
khatchad@cse.ohio-state.edu

Phil Greenwood
Lancaster University
greenwop@comp.lancs.ac.uk

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

Guoqing Xu
Ohio State University
xug@cse.ohio-state.edu

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. In this paper, we present an automated approach which limits fragility problems by providing mechanical assistance in pointcut maintenance. The approach is based on harnessing arbitrarily deep structural commonalities between program elements corresponding to join points selected by a pointcut. The extracted patterns are then applied to later versions to offer suggestions of new join points that may require inclusion. To illustrate that the motivation behind our proposal is well-founded, we first empirically establish that join points captured by a single pointcut typically portray a significant amount of unique structural commonality by analyzing patterns extracted from 23 AspectJ programs. Then, we demonstrate the usefulness of our technique by rejuvenating pointcuts in multiple versions of 3 of these programs. The results show that our parameterized heuristic algorithm was able to automatically infer new join points in subsequent versions with an average recall of 0.93. Moreover, these join points appeared, on average, in the top 4^{th} percentile of the suggestions, indicating that the results were precise.

Keywords

Aspect-Oriented programming, tool-supported software evolution

1. INTRODUCTION

Aspect-Oriented Programming (AOP) [28] has emerged to reduce the scattering and tangling of crosscutting concern (CCC) implementations. This is achieved through specifying that certain behavior (advice) should be composed at specific (join) points during the execution of the underlying program (base-code). Sets of join points are described by pointcuts (PCEs) which are predicate-like expressions over various characteristics of "events" that occur during the program's execution. In AspectJ [27], an extension of Java with support for aspects, for instance, such characteristics may include calls to certain methods, accesses to particular fields, and modifications to the run time stack.

Consider an example PCE **execution**(∗ m∗(..)) which selects the execution of all methods whose name begins with *m*, taking any number and type of arguments, and returning any type of value. Suppose that in a particular version of the base-code, the above PCE selects the correct set of join points in which a CCC applies. As the software evolves, this set of join points may change as well. We say that a PCE is *robust* if it, in its unaltered form, is able to *continue* to capture the correct set of join points in future versions of the base-code. Thus, the PCE given above would be considered robust if the set of join points in which the CCC applies *always* corresponded to executions of methods whose name beings with *m*, taking any number and type of arguments, and so forth. However, with the requirements of typical software tending to change over time, the corresponding source code may undergo many alterations to accommodate such change, including the addition of *new* elements in which existing CCCs should also apply. Without *a priori* knowledge of future maintenance changes and additions, creating robust PCEs is a daunting task. As such, there may easily exist situations where the PCE itself must evolve *along* with the base-code; in these case we say that the PCE is *fragile*. Hence, the *fragile pointcut problem* [30] manifests itself in such circumstances where join points incorrectly fall in or out of the scope of PCEs.

Several approaches aim to combat this problem by proposing new pointcut languages with improved expressiveness [8, 29, 35, 40,41], limiting the scope of where advice may apply through more clearly defined interfaces [1,21], enforcing structural and/or behavioral constraints on advice application [20, 26, 44], making points where advice may apply more explicit in the base-code [24], or by removing PCEs altogether [37]. 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 to capture new join points as the software evolves. This process unfortunately develops into a vicious cycle where these new PCEs may also exhibit similar problems.

In order to alleviate such problems, we propose an approach that provides automated assistance in rejuvenating PCEs upon changes

*This material is based upon work supported in part by European Commission grants IST-33710 (AMPLE) and IST-2-004349 (AOSD-Europe).

†This work was administered during this author's visit to the Computing Department, Lancaster University, United Kingdom.

to the base-code. The technique is 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. To illustrate, again consider the example PCE given earlier and suppose that, in a certain base-code version, the PCE selects the execution of three methods, m1, m2, and m3. Further suppose that facets pertaining to these methods exhibit structural commonality, e.g., each of the methods' bodies may (textually) include a call to a common method y, or that each includes a call to three other methods x, y, and z, respectively, all of which have method bodies that include an assignment to a common field f. Likewise, each method may be declared in three different classes A, B, and C, respectively, all of which are contained in a package p. Moreover, if such characteristics are shared between program elements corresponding to join points selected by a PCE in one base-code version, it is conceivable that these relationships persist in *subsequent* versions. Consequently, our proposal involves constructing patterns that describe these kinds of relationships, assessing their expressiveness in comparison with the PCE used to construct them, and associating them with the PCE so that they may be applied to later base-code versions to offer suggestions of new join points that may require inclusion.

Our key contributions are as follows:

Commonality identification. We present a parameterized heuristic algorithm that automatically derives arbitrarily deep structural patterns inherent to program elements corresponding to join points selected by the original PCE. This allows join points to be suggested that may require inclusion into a revised version of the PCE, ensuring that evolutionary changes can be correctly applied by mechanically assisting the developer in maintaining PCEs.

Correlation analysis. We empirically establish that join points selected by a single PCE typically portray a significant amount of unique structural commonality by applying our algorithm to automatically extract and, thereafter, analyze patterns using PCEs contained within *single* versions of 23 AspectJ programs. We found that the derived patterns, on average, were able to closely produce the majority of join points selected by the analyzed PCE in the original base-code version.

Expression recovery. To ensure the applicability and practicality of our approach, we implemented our algorithm as an Eclipse IDE[1] plugin and evaluated its usefulness by rejuvenating PCEs in multiple versions of 3 of the aforementioned programs. We found that, in exploiting the extracted patterns, our tool was able to automatically infer new join points that were selected by PCEs in subsequent software versions that were not selected by the original PCE at an average recall of 0.93. Moreover, these join points appeared, on average, in the top 4^{th} percentile of the ranked list of suggested join points, indicating that the results were precise. This demonstrates that the approach is indeed useful in alleviating the burden of recovering PCEs upon base-code modifications.

Roadmap. §2 presents a motivating example that features a fragile PCE. §3 highlights the key algorithmic facets of our approach, while §4 discusses the details of our implementation and evaluation. In §5, we compare our proposal with related work and explore future work, as well as conclude, in §6.

2. POINTCUT FRAGILITY EXAMPLE

[1] http://www.eclipse.org

```
1  package p;
2  class HybridAutomobile {
3    private double overallSpeed;
4    //...
5    //Sets the new speed for changes in fuel.
6    public void notifyChangeIn(Fuel fuel) {
7      this.overallSpeed +=
8        fuel.calculateDelta(this);
9      /* Update attached observers ... */}
10
11   //Sets the new speed for changes in electricity.
12   public void notifyChangeIn(Current current) {
13     this.overallSpeed +=
14       current.calculateDelta(this);
15     /* Update attached observers ... */}
16
17   //Sets the new speed directly.
18   public void notifyChangeIn(double mph) {
19     this.overallSpeed += mph;
20     /* Update attached observers ... */}
21
22   public double getOverallSpeed() {
23     return overallSpeed;}}
24
25 class DieselEngine {
26   private HybridAutomobile car;
27   public void increase(Fuel fuel) {
28     //...
29     this.car.notifyChangeIn(fuel);}}
30
31 class ElectricMotor {
32   private HybridAutomobile car;
33   public void increase(Current current) {
34     //...
35     this.car.notifyChangeIn(current);}}
36
37 class Dashboard {
38   private HybridAutomobile car;
39   //...
40   public void update() {
41     this.display(car.getOverallSpeed()); }}
```

Figure 1: Hybrid automobile example.

Fig. 1 shows an example AspectJ code snippet for hypothetical drive-by-wire programming of an all-wheel drive, hybrid vehicle (line 2) which draws power from two different sources, viz., a diesel engine (line 25) and an electric motor (line 31), both of which contribute to the overall speed (line 3)[2]. Fuel is distributed to the engine via the method DieselEngine.increase(Fuel) (line 27), while electricity is distributed to the motor via the method ElectricMotor.increase(Current) (line 33), whose method bodies are abbreviated. The classes conform to the *Observer* pattern [17], with the DieselEngine and ElectricMotor notifying the HybridAutomobile of any change made to the energy consumption of the respective components. The HybridAutomobile in turn computes its new overall speed (lines 7–8, 13–14) and updates any attached observers, e.g., possibly the Dashboard (line 37). An accessor method (line 22) retrieves the value of the private instance field overallSpeed, which the method Dashboard.update() invokes (line 41) as part of the design pattern to refresh the driver's display.

Suppose now that roadways exhibit a new feature that notifies traveling vehicles of the speed limit. As a result, an aspect SpeedingViolationPrevention (Fig. 2) is introduced to augment the existing functionality of the programming depicted in Fig. 1 by limiting the vehicle's energy intake by declaring appropriate **around** advice (lines 2–4) which conditionally bypasses the execution of methods that contribute to the vehicle's overall speed. The points at

[2] This example was inspired by one of the authors' work at the Center for Automotive Research at Ohio State University.

```
1  aspect SpeedingViolationPrevention {
2    void around() :
3      execution(void increase(Energy+))
4        {/*...*/}}
```

Figure 2: Speeding prevention aspect.

```
1  package p;
2  class FuelCell {
3    private HybridAutomobile car;
4    public void increase(double mph) {
5      //...
6      this.car.notifyChangeIn(mph);}}
```

Figure 3: A new fuel cell class.

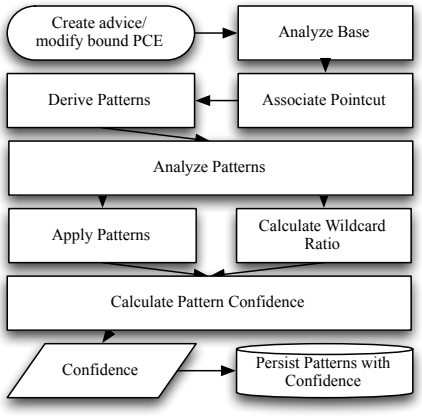

Figure 4: Phase I: Pointcut analysis.

which this advice is to apply are specified by its bound PCE (line 3) which selects join points corresponding to the execution of two of the aforementioned methods, viz., DieselEngine.increase(Fuel) and ElectricMotor.increase(Current). Class Energy (not shown) is an abstract super class of which both classes Fuel and Current (also not shown), parameters to the methods, extend. The type pattern Energy+ is a wild-card that denotes object references of type Energy and its subclasses. Note that facets related to the advice body are abbreviated here due to our focus on the applicability of the advice.

Further suppose that the base-code (Fig. 1) evolves to accommodate a new vehicle energy source, viz., a fuel cell, resulting in the creation of a FuelCell class (Fig. 3). In contrary to the existing energy sources, requests to increase power from the FuelCell require passing a numerical (**double**) parameter, which is the amount of acceleration (in miles/hour) that should result from the FuelCell internally generating power, to a method (line 4) that, in turn, notifies the HybridAutomobile of the change directly (line 6).

Intuitively, the SpeedingViolationPrevention aspect should also apply to the execution of this method, however, the PCE fails to select this new but *semantically equivalent* join point. Although the new method's signature is consistent with the other join points with only the parameter type differing, i.e., **double** is a primitive type that could not hold references to type Energy or any of its subclasses, this difference causes the PCE not to select this method's execution. Worse, many such join points may silently exhibit similar problems in evolving software with larger code bases. It would be helpful to developers if join points that may have been overlooked when manually updating PCEs to reflect new changes in the base-code could be mechanically suggested. We will continue to use this example to demonstrate how our proposed approach can help identify such new join points in an automated fashion.

3. HARNESSING COMMONALITY

In this section, we present a parameterized heuristic algorithm that assists developers in maintaining PCEs upon changes to the base-code by inferring new join points that may require inclusion. The algorithm works by discovering structural commonality between program elements corresponding to join points captured by a PCE in a particular software version. For instance, notice in the previous example that the two methods, viz., DieselEngine.increase(Fuel) and ElectricMotor.increase(Current), whose corresponding method executions were selected by the PCE listed on line 3, Fig. 2 are both declared in classes contained in package p. Additionally, considering solely the code snipped characterized in Fig. 1, both method bodies contain calls to methods, viz., notifyChangeIn(Fuel fuel) and notifyChangeIn(Current current), respectively, that read from the field HybridAutomobile.overallSpeed. We capture such commonality by constructing patterns that abstractly describe the kinds of relations that the program elements have in common. The extracted patterns are then applied to later versions to offer suggestions of new join points that require inclusion as similar commonality may be exhibited in the future.

3.1 High-level Overview and Assumptions

Our approach is divided into two conceptual phases: *analysis* and *rejuvenation*. The analysis phase, whose flow diagram is depicted in Fig. 4, is triggered upon modifications to or creation of PCEs[3]. A graph is then computed which depicts structural relationships among program elements currently residing in the base-code. Next, patterns are derived from paths of the graph in which vertices and/or edges representing program elements and/or relationships are associated with join points selected by the PCE. The patterns are then themselves analyzed to evaluate the *confidence* (as inspired by [10]) we have in using the pattern to identify join points that should be captured by a revised version of the PCE upon base-code evolution. Consequently, results produced by the pattern are correlated with and ranked by this value when presented to the developer. Finally, the patterns along with their confidence are linked with the PCE and persisted for later use in the next phase.

We envision our approach to be most helpful in scenarios where a developer performs a series of changes to the base-code and then proceeds to update PCEs to reflect those changes so that new join points are captured correctly. Thus, the rejuvenation phase, whose flow diagram is depicted in Fig. 5, is triggered prior to the developer manually altering the PCE so that automated assistance in performing the updates correctly can be provided. At this juncture, the patterns previously linked with the PCE are retrieved from storage and ran against a graph computed from the new base-code version to unveil the suggested join points. These join points are ones related to program elements that share structural similarities with program elements related to join points previously selected by the PCE in the original base-code version. Each suggestion is presented to the developer with the confidence of the pattern used to produce the suggestion, and the list of all suggestions are sort in decreasing order of confidence. In the remainder of this section, we discuss

[3]*Named*-PCEs are analyzed when they are referred to in advice-bound PCEs.

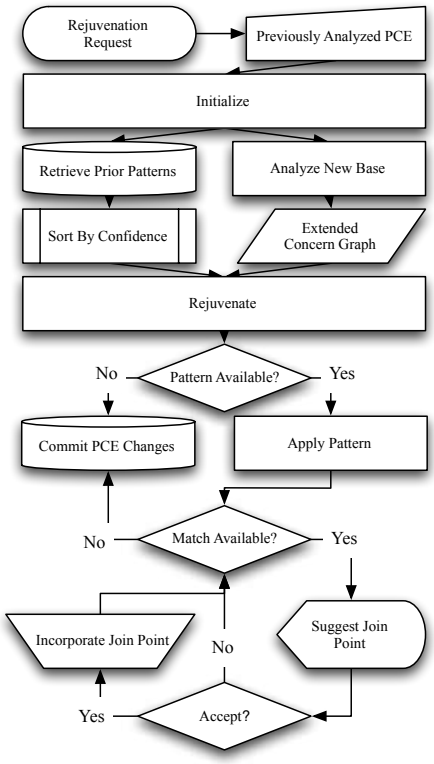

Figure 5: Phase II: Pointcut rejuvenation.

the algorithmic details of these steps, while the succeeding section describes our implementation.

Prior to continuing our discussion, we first state several simplifying assumptions about the underlying source code to be analyzed; we discuss in §4.1 how much of these have been relaxed in our implementation and how others can be dealt with in future work in §6. Firstly, we assume that the input PCE is initially correctly specified, i.e., it selects (and only selects) intended join points of the original program. Furthermore, we assume that inter-type declarations (static crosscutting) are not utilized by the analyzed aspects. Inter-type declarations allow aspects to introduce and modify facets of the base-code, e.g., member introduction, class hierarchy alteration, interface implementation injection, exception softening, existing at compile-time. Also, although it is possible for a PCE to select join points associated within an advice body (possibly the one it is bound to), we adopt the perspective that aspects are indeed separate from the base-code; advice may only apply to join points associated with classes, interfaces, and other Java types. Lastly, we assume that we can accurately resolve the declaration of the advice a PCE is bound to across varying versions of the software. This assumption may be invalidated via the use of refactorings, e.g., member relocation, being applied in between software versions. Section 6 discusses future plans on how our tool can be made to cope with this issue.

3.2 Concern Graphs

To abstract the details of the underlying source code, a representation of the program is first built using an adaptation of a *concern graph* [39]. Concern graphs have been used in previous work [38] to discover, describe, and track concerns in evolving source code as they allow for succinct representations of the underlying program.

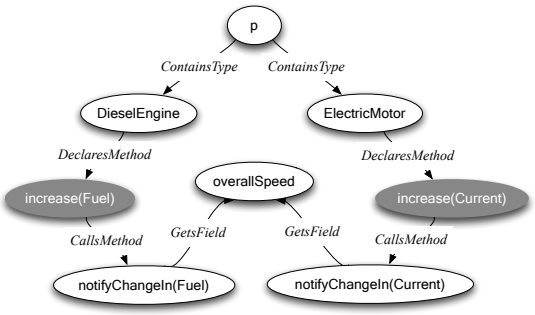

Figure 6: A graph subset computed from the example.

In this work, we extended concern graphs with several elements found in current Java languages, e.g., annotations, and adapted them for use with AOP.

More formally, we specify an extended concern graph CG^+ to be a labeled multidigraph consisting of a 4-tuple $CG^+ = (V, E, R, \ell)$. The set of vertices V represents program elements contained within the analyzed program, specifically, packages, classes, interfaces, enumeration types, annotations, methods, and fields[4]. Set E is a multiset of directed edges which connect the vertices in V depending on various relations that may hold between them as depicted in the source code. For example, the program entities HybridAutomobile and overallSpeed from the code snippet given in Fig. 1 are related in that the class HybridAutomobile *declares* the field overallSpeed. In this case, there would exist an edge in E connecting the vertex in V representing the class HybridAutomobile to the vertex representing the field overallSpeed. The set of all such (binary) relations between program elements that we consider make up the elements of the set R. Since two given vertices may be related in several different ways, i.e., they satisfy more than one relation, there may exist multiple edges between them. As such, $\ell\colon E \rightarrow R$ serves as a labeling function which distinguishes edges by labeling them with the satisfied relations they represent. Fig. 6 portrays a subset of the graph computed from the motivating example given in §2 diagrammatically.

Table 1 portrays the complete set of binary relations[5] that we consider as well as the program entity types they relate as derived from a recent version of the Java language specification [18] (*Enum* is an abbreviation for Enumerated Types). These relations may either hold in a structural sense, e.g., field declarations, or possibly during a particular execution of the program, e.g., method calls. §4.1 discusses how we conservatively approximated the truth value of these relations in our implementation by using exclusively static information, i.e., through examination of the program text, while §6 touches upon future work which could result in a more accurate approximations. Moreover, many kinds of relations may be formulated, however, we mainly focus on popular relation as used in previous work [6, 10, 39] with the addition of relations useful for AO languages, e.g., $Annotates$. §4 reports on the appropriateness of using such relations for PCE rejuvenation in AspectJ programs; future work may involve investigating the existence of other relations, e.g., $HandlesException$, that may contribute to our results.

[4]We do not consider local variables and other parameters in our analysis as crosscutting concerns tend to crosscut a larger granularity of program elements.

[5]For simplicity, we group class instance creations and constructor calls with method calls.

Relation	From Entity	To Entity
GetsField	Methods	Fields
SetsField	Methods	Fields
CallsMethod	Methods	Methods
OverridesMethod	Methods	Methods
ImplementsMethod	Methods	Methods
DeclaresMethod	Classes, Enums Interfaces	Methods
DeclaresField	Classes, Enums Interfaces	Fields
DeclaresType	Classes, Interfaces Annotations	Classes, Annotations Interfaces, Enums
ExtendsClass	Classes	Classes
ExtendsInterface	Interfaces	Interfaces
ImplementsInterface	Classes, Enums	Interfaces Annotations
ContainsType	Packages	Classes, Annotations Interfaces, Enums
Annotates	Annotations	Packages, Fields Interfaces, Enums Classes, Methods Annotations

Table 1: Analyzed program entity types and relations.

3.3 Concern Graph-Pointcut Association

The next step in our approach involves discovering graph elements (vertices and edges) that represent program elements corresponding to join points captured by the input PCE so that patterns capturing commonality existing between these elements can be later extracted. Recall that a PCE is a means to describe a set of join points and that a join point is a well-defined point in the execution of the base-code. Thus, the definition of a join point is very much dynamic in nature. A join point *shadow*, on the other hand, refers to base-code corresponding to a join point, i.e., a point in the program text where the compiler may actually perform the weaving [34]. Whether or not the base-code is advised at that point is dependent on advice being applicable at that point and possible dynamic conditions being met. In this paper, we treat a program as consisting of a set of join point shadows that *may* or *may not* be currently under the influence of advice[6]. Moreover, we treat a PCE as selecting a subset of these shadows; i.e., we assume that the PCE is free of dynamic conditions which allows us to exploit solely static information in our analysis. §4.1 discusses how our implementation conservatively relaxes this assumption so that PCEs utilizing dynamic conditions may nevertheless be used as input to our tool.

For a graph $CG^+ = (V, E, R, \ell)$, we say that a vertex $v \in V$ is associated with (or *enabled* w.r.t.) a PCE iff v represents a method whose corresponding method *execution*-join point shadow is selected by the PCE. For a graph built from the motivating example found in Fig. 1, the vertices representing the methods DieselEngine .increase(Fuel) and ElectricMotor.increase(Current) would be considered enabled w.r.t. the PCE found on line 3, Fig. 2. The graph subset shown in Fig. 6 illustrates this; the vertices representing these methods are shaded. Likewise, we say that an edge $(u, v) \in E$ is enabled w.r.t a PCE iff:

- the edge is labeled as either a method call, i.e., $CallMethods(u, v)$ holds, a field read, i.e., $GetsField(u, v)$ holds, or a field write, i.e., $SetsField(u, v)$ holds, and
- there exists a corresponding method *call*-, field *get*-, or field *set*-join point shadow selected by the PCE such that the called method, the read field, or the written field, respectively, is the one represented by vertex v, and the shadow resides within the body of the method represented by vertex u.

For example, an edge representing a call from a method m to a method n would be considered enabled w.r.t a PCE selecting a method call shadow for n originating in the body (or in AspectJ terminology, **withincode**) of m. Note that the difference between a method *execution*-join point and a method *call*-join point is that in the former, the corresponding shadow would lie at the *declaration* of the invoked method, while in the latter, it would lie at the *site* of the method invocation, i.e., the client code. §4.1 discusses how our implementation leverages existing tool support to deduce enabled graph elements. Possible future work entails considering additional AspectJ join point types such as *handler*-join points.

3.4 Pattern Extraction

Once that we are able to associate various graph elements with the input PCE, we may being to analyze commonality between these elements with the hope that *future* elements whose shadows should be included in a new version of the PCE may exhibit similar commonality with a particular level of confidence. For instance, recall from our motivating example that the methods (increase(Fuel) and increase(Current)) whose corresponding execution was selected by the PCE (**execution**(**void** increase(Energy+))) both contained calls to methods (notifyChangeIn(Fuel) and notifyChangeIn(Current)) which read from a common field (overallSpeed). Deliberately, this information is expressed by two paths, i.e., the sequences of connected edges, increase(Fuel) $\rightsquigarrow$ overallSpeed and increase(Energy) $\rightsquigarrow$ overallSpeed in the graph snippet portrayed in Fig. 6. We capture commonality associated with such graph elements by extracting *patterns* from paths in which they are contained. These patterns, which convey general "shapes" (in terms of paths) of the graph surrounding the enabled graph elements, will ultimately be applied to graphs computed from subsequent software versions to uncover new elements displaying the captured commonality.

For each enabled (w.r.t. the input PCE) vertex v and edge (u, v) we extract a set of patterns from finite, acyclic paths of length (in terms of edges) $\leq k$ passing through v and along (u, v), respectively. The *maximum analysis depth* parameter k, an input to the algorithm, controls tractability by restricting the depth of satisfied relations analyzed, and, consequently, limits the length of the patterns derived. §4.2 discusses our choice for k in our evaluation. One example of such a path when taking the enabled vertex $v =$ increase(Fuel) and $k = 2$ is increaseFuel(Fuel) $\xrightarrow{cm}$ notifyChangeIn (Fuel) $\xrightarrow{gf}$ overallSpeed, where edge labels cm and gf refer to the satisfied relations $CallsMethod$ and $GetsField$, respectively.

Intuitively, patterns are constructed in such a way that paths that match the pattern are ones that have common origins or sinks which are connected via similar (in terms of labels) edges. We consider two kinds of patterns, those derived from enabled vertices and those from enabled edges. A *vertex*-based pattern is obtained from a path by replacing certain vertices along the path with *vertex wild-cards*, while an *edge*-based pattern is obtained by not only replacing vertices with vertex wild-cards, but also certain edges with *edge wild-cards*. Vertex wild-cards only match vertices, while edge wild-cards only match edges. Wild-cards serve to express points of variation in paths the encompassing pattern is matched against, as well

[6]This definition differs slightly from those typically given in the literature [23, 48].

as to select shadows which are ultimately suggested for incorporation. As such, wild-cards may be *enabled* as determined by their position relative to the enabled graph element in the path used to create the pattern. Shadows associated with graph elements (cf. §3.3) matched by enabled wild-cards are those that eventually become suggested.

We extract a set of vertex-based patterns from a path (a sequence of edges) $\pi = \langle e_1, e_2, \ldots, e_n \rangle$ and an enabled vertex v along π as depicted by the algorithm listed in Fig. 7. Text appearing in the figure between /*...*/ offer descriptions of each of the algorithm's steps. For reference, the helper functions $s\colon E \rightarrow V$ and $t\colon E \rightarrow V$ map an edge to its constituent source and target vertices, respectively. The algorithm proceeds as follows. If v occurs in π as the source vertex of the first edge, we extract a single pattern by replacing this vertex with an enabled wild-card. The remaining vertices along the path are replaced by disabled wild-cards except for the target vertex of the last edge. To illustrate, recall the previously considered path increaseFuel(Fuel) $\xrightarrow{cm}$ notifyChangeIn (Fuel) $\xrightarrow{gf}$ overallSpeed where the vertex representing the method increaseFuel(Fuel) is enabled w.r.t. the PCE on line 3, Fig. 2. The set of patterns extracted from this path would consist of the singleton $\{?^* \xrightarrow{cm} ? \xrightarrow{gf}$ overallSpeed$\}$ where ? denotes a disabled wild-card and $?^*$ an enabled wild-card.

Continuing, if v occurs in π as the target vertex of the first edge, a similar action is performed as in the previous case, however, we retain the source vertex of the first edge and instead replace the target vertex of the first edge with an enabled wild-card. For the case that v occurs in π as either the source or target vertex of the last node, the reverse process is performed. Finally, for the case in which v is not involved with either the first or last edge of the path, we split the path to extract two patterns, one with v as the target vertex of the last edge and one with v as the source vertex of the first edge and proceed as before.

Edge-based patterns are handled in a similar manor as depicted by the algorithm shown in Fig. 8. Here, parameter e represents the enabled arc to which to base the derived patterns by. An enabled edge wild-card is denoted by raising a pair of vertices to the enabled wild-card symbol $?^*$. Again, text appearing between /*...*/ offer descriptions of each of the algorithm's steps. The key difference between the vertex and edge pattern extraction algorithms is that, in the case of edges, the corresponding algorithm is intended to construct patterns which produce other edges exhibiting commonality related to the input (enabled) edge. This requires accounting for locations of where edges appear in paths, as well as the labels of the edges.

Deciding on this specific scheme (for both vertices and edges) was pragmatic; there is a trade-off to be considered between the abstractness of patterns, i.e., the ratio of constituent wild-cards to that of concrete elements, and the quality of the results produced. Particularly, highly abstract patterns are more likely to produce more (possibly spurious) results. We discuss how we dealt with this trade-off in our ranking scheme in §3.6.

3.5 Pattern Matching

We say that a pattern $\hat{\pi}$ matches a path π iff

- for each vertex u along π at position i there is a vertex v along $\hat{\pi}$ at position i s.t. either $u = v$ or v is a wild-card, and
- for each edge (p, q) along π at position j there is an edge (s, t) along $\hat{\pi}$ at position j s.t. either $\ell(p, q) = \ell(s, t)$ or (s, t) is a wild-card.

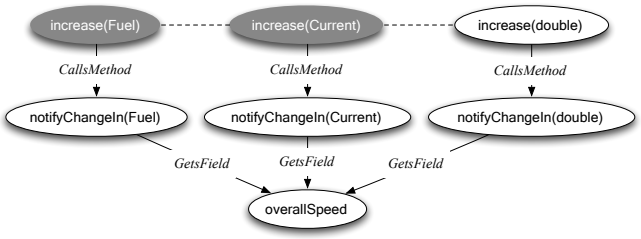

Figure 9: Evolving the base-code with a FuelCell class.

To illustrate, suppose we augmented the graph found in Fig. 6 with new vertices and edges representing facets of the code of the FuelCell class in Fig. 3. The resulting situation is depicted in Fig. 9 where a new path increase(**double**) $\xrightarrow{cm}$ notifyChangeIn(**double**) $\xrightarrow{gf}$ overallSpeed matches the previously extracted pattern $?^* \xrightarrow{cm} ? \xrightarrow{gf}$ overallSpeed.

Given that a pattern matches a path, suggested shadows are ones represented by graph elements (vertices and/or edges) along the path which matched enabled wild-cards in the pattern. Vertices representing methods matched by enabled wild-cards produce suggested shadows corresponding to the execution of those methods. Likewise, edges representing satisfied relations, e.g., method calls, field reads, field writes, between program elements matched by enabled wild-cards produce suggested shadows corresponding to the relation which reside in the body (**withincode**) of the method represented by the source vertex and operate (**call**, **get**, or **set**) on program element represented by the target vertex. For example, when matching the pattern $?^* \xrightarrow{cm} ? \xrightarrow{gf}$ overallSpeed against the path increase(**double**) $\xrightarrow{cm}$ notifyChangeIn(**double**) $\xrightarrow{gf}$ overallSpeed, the method FuelCell.increase(**double**) is represented by a vertex that matches an enabled wild-card element. The situation is emphasized in Fig. 9 by a dashed line through the vertices that induced the wild-card. As a result, the shadow corresponding to the execution of this method would be suggested to be included in a revised version of the PCE listed on line 3, Fig. 2, perhaps resembling **execution**(**void** increase(Energy+)) || **execution**(**void** FuelCell.increase(**double**)).

3.6 Suggestion Sorting

Shadows suggested for incorporation are presented to the developer in descending order of the degree of *confidence* we have in the shadow being applicable to a revised version of the input PCE. The confidence value, a real number in the interval $[0, 1]$, paired with each suggestion is inherited from the pattern which produced it. We evaluate our confidence in a pattern's ability to match shadows contained in a subsequent version of the base-code that should be captured by a revised version of the input PCE by applying the pattern to the *current* version of the base-code and assessing its performance. This assessment is performed on three different dimensions as depicted by the equations listed in Fig. 10. We refer to each of these as *pattern attributes* w.r.t. the PCE to be rejuvenated.

To describe the attributes more precisely, we first define a function $Match(\hat{\pi}, \Pi)$ where $\hat{\pi}$ ranges over the set of patterns and Π the power set of paths that given a pattern and a set of paths, matches the pattern against the paths resulting in a set of suggested shadows as detailed in §3.5. Then, we define the err_α rate attribute, equation (1), to be the ratio of the number of shadows captured by both the PCE and the pattern when matched against finite, acyclic paths in the graph $Paths(CG^+)$ to the number of shadows solely captured

function $ExtractVertexPatterns(\pi = \langle e_1, e_2, \ldots, e_n \rangle, v)$

1: $\hat{\Pi} \leftarrow \varnothing$ /*The set of patterns to be returned, initially empty*/
2: $\hat{\pi} \leftarrow \langle\rangle$ /*A single pattern to be built, initially the empty sequence of edges.*/
3: **for** $i \leftarrow 1, n$ **do** /*For each edge along path π*/
4: **if** $i = 1 \wedge s(e_i) \neq v \wedge t(e_i) \neq v$ **then** /*If it is the first edge and both the source nor target vertices are disabled*/
5: $\hat{\pi} \leftarrow \hat{\pi} + (s(e_i), ?)$ /*Append a new edge consisting of the old source as the source vertex and a disabled wild-card as the target vertex to the pattern.*/
6: **else if** $i = 1 \wedge t(e_i) = v$ **then** /*Otherwise, if it is the first edge and the target vertex is enabled*/
7: $\hat{\pi} \leftarrow \hat{\pi} + (s(e_i), ?^*)$ /*Append a new edge consisting of the old source as the source vertex and a enabled wild-card as the target vertex to the pattern.*/
8: **else if** $i = n \wedge s(e_i) \neq v \wedge t(e_i) \neq v$ **then** /*Otherwise, if it is the last edge and both the source and target vertices are disabled*/
9: $\hat{\pi} \leftarrow \hat{\pi} + (?, t(e_i))$ /*Append a new edge consisting of a disabled wild-card as the source vertex and the old target as the target vertex to the pattern.*/
10: **else if** $i = n \wedge s(e_i) = v$ **then** /*Otherwise, if it is the last edge and the source vertex is enabled*/
11: $\hat{\pi} \leftarrow \hat{\pi} + (?^*, t(e_i))$ /*Append a new edge consisting of a enabled wild-card as the source vertex and the old target as the target vertex to the pattern.*/
12: **else if** $i \neq 1 \wedge i = n \wedge t(e_i) = v$ **then** /*Otherwise, if it is neither the first nor the last edge and the target vertex is enabled*/
13: $\hat{\pi} \leftarrow \hat{\pi} + (?, ?^*)$ /*Append a new edge consisting of a disabled wild-card as the source vertex and an enabled wild-card as the target vertex to the pattern.*/
14: **else if** $i = 1 \wedge i \neq n \wedge s(e_i) = v$ **then** /*Otherwise, if it is the first but not the last edge and the source vertex is enabled*/
15: $\hat{\pi} \leftarrow \hat{\pi} + (?^*, ?)$ /*Append a new edge consisting of an enabled wild-card as the source vertex and a disabled wild-card as the target vertex to the pattern.*/
16: **else if** $i \neq 1 \wedge i \neq n \wedge s(e_i) \neq v \wedge t(e_i) \neq v$ **then** /*Otherwise, if it is neither the first nor the last edge and both the source and target vertices are disabled*/
17: $\hat{\pi} \leftarrow \hat{\pi} + (?, ?)$ /*Append a new edge consisting a disabled wild-card as the source vertex and an enabled wild-card as the target vertex to the pattern.*/
18: **else if** $i \neq 1 \wedge i \neq n \wedge s(e_i) = v$ **then** /*Otherwise, if it is neither the first nor the last edge and the source vertex is enabled*/
19: $\hat{\pi} \leftarrow \hat{\pi} + (?^*, ?)$ /*Append a new edge consisting of an enabled wild-card as the source vertex and a disabled wild-card as the target vertex to the pattern.*/
20: $\hat{\Pi} \leftarrow \hat{\Pi} \cup \{\hat{\pi}\}$ /*Add the completed pattern to the set to be returned.*/
21: $\hat{\pi} \leftarrow \langle\rangle$ /*Reset $\hat{\pi}$ to be the empty sequence of edges.*/
22: **else** /*Otherwise, it must be that it is neither the first nor the last edge and the target vertex is enabled*/
23: $\hat{\pi} \leftarrow \hat{\pi} + (?, ?^*)$ /*Append a new edge consisting of a disabled wild-card as the source vertex and an enabled wild-card as the target to the pattern.*/
24: $\hat{\Pi} \leftarrow \hat{\Pi} \cup \{\hat{\pi}\}$
25: $\hat{\pi} \leftarrow \langle\rangle$
26: **end if**
27: **end for**
28: **return** $\hat{\Pi} \cup \{\hat{\pi}\}$ /*Return the accrued set of patterns along with the last completed pattern.*/

Figure 7: Vertex-based pattern extraction algorithm.

by the pattern. Note that CG^+ refers here to the graph computed from the base-code in which the pattern was constructed, i.e., the original, unrevised program. Furthermore, recall from §3.3 that we treat a PCE as a set of shadows, thus, |PCE| refers to the cardinality of the set denoted by PCE, i.e., the number of shadows it selects.

The α signifies the metric's association with the rate of type I (or α) errors which relates to the number of false positives resulting from applying the pattern to the original version of the base-code, as portrayed by region marked α in the Venn diagram depicted in Fig. 11. Conceptually, the err_α rate quantifies the pattern's ability in matching *solely* the shadows contained within the PCE; the closer the err_α rate is to 0 the more likely the shadows matched by the pattern are also ones contained within the PCE. It refers to the *quality* of results that the pattern is likely to produce in the future. A pattern with a low err_α rate is one that expresses a strong relationship amongst shadows captured by the PCE; we would expect *future* shadows to exhibit *similar* characteristics. If a pattern matches no shadows, its err_α rate is 0. For example, applying the pattern $?^* \xrightarrow{cm} ? \xrightarrow{gf}$ overallSpeed to the original base-code version in Fig. 1 would produce three shadows corresponding to

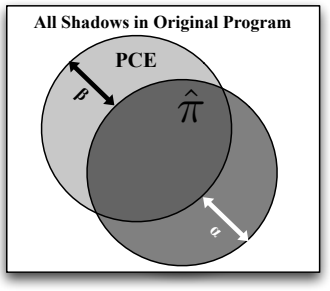

Figure 11: Comparing a PCE with a pattern $\hat{\pi}$ in the original program.

the execution of methods DieselEngine.increase(Fuel), ElectricMotor.increase(Current), and Dashboard.update() (due to the pattern matching the path update() $\xrightarrow{cm}$ getOverallSpeed() $\xrightarrow{gf}$ overallSpeed). Thus, the err_α rate for this pattern w.r.t. the PCE found on line

function $ExtractEdgePatterns(\pi = \langle e_1, e_2, \ldots, e_n \rangle, e)$

1: $\hat{\Pi} \leftarrow \varnothing$ /*The set of patterns to be returned, initially empty*/
2: $\hat{\pi} \leftarrow \langle\rangle$ /*A single pattern to be built, initially the empty sequence of edges.*/
3: **for** $i \leftarrow 1, n$ **do** /*For each edge along path π*/
4: **if** $i = 1 \wedge e_i \neq e$ **then** /*If it is the first edge and it is disabled*/
5: $\hat{\pi} \leftarrow \hat{\pi} + (s(e_i), ?)$ /*Append a new edge consisting of the old source as the source vertex and a disabled wild-card as the target vertex to the pattern.*/
6: **else if** $i = n \wedge e_i \neq e$ **then** /*Otherwise, if it is the last edge and it is disabled*/
7: $\hat{\pi} \leftarrow \hat{\pi} + (?, t(e_i))$ /*Append a new edge consisting of a disabled wild-card as the source vertex and the old target as the target vertex to the pattern.*/
8: **else if** $i \neq 1 \wedge i \neq n \wedge e_i \neq e$ **then** /*Otherwise, if it is neither the first nor the last edge and it is disabled*/
9: $\hat{\pi} \leftarrow \hat{\pi} + (?, ?)$ /*Append a new edge consisting of disabled wild-cards as both the source and target vertices to the pattern.*/
10: **else if** $(i = 1 \vee i = n) \wedge i \neq n \wedge e_i = e$ **then** /*Otherwise, if it is the first edge or the last edge but not the only edge and it is enabled*/
11: $\hat{\pi} \leftarrow \hat{\pi} + (?, ?)^{?^*}$ /*Append a new enabled edge wild-card consisting of disabled wild-cards as both the source and target vertices to the pattern.*/
12: **else if** $i \neq 1 \wedge i \neq n \wedge e_i = e$ **then** /*Otherwise, if it is neither the first nor the last edge and it is enabled*/
13: $\hat{\pi} \leftarrow \hat{\pi} + (?, ?)^{?^*}$
14: $\hat{\Pi} \leftarrow \hat{\Pi} \cup \{\hat{\pi}\}$ /*Add the completed pattern to the set to be returned.*/
15: $\hat{\pi} \leftarrow \langle (?, ?)^{?^*} \rangle$ /*Reset $\hat{\pi}$ to be a sequence consisting of a new enabled edge wild-card with disabled wild-cards as both the source and target vertices.*/
16: **else** /*Otherwise, it must be that it is the only edge and it is enabled*/
17: $\hat{\pi} \leftarrow \hat{\pi} + (s(e_i), ?)^{?^*}$ /*Append a new enabled edge wild-card consisting of the old source as the source vertex and a disabled wild-card as the target vertex to the pattern.*/
18: $\hat{\Pi} \leftarrow \hat{\Pi} \cup \{\hat{\pi}\}$
19: $\hat{\pi} \leftarrow \langle (?, t(e_i))^{?^*} \rangle$ /*Append a new enabled edge wild-card consisting of a disabled wild-card as the source vertex and the old target as the target vertex to the pattern.*/
20: **end if**
21: **end for**
22: **return** $\hat{\Pi} \cup \{\hat{\pi}\}$ /*Return the accrued set of patterns along with the last completed pattern.*/

Figure 8: Edge-based pattern path extraction algorithm.

3 of Fig. 2, which selects the execution of methods DieselEngine .increase(Fuel) and ElectricMotor.increase(Current) in the original base-code version, would be $\frac{1}{3}$.

The err_β rate attribute, equation (2), is the ratio of the number of shadows captured by both the PCE and the pattern when applied to paths in the graph to the number of shadows captured by solely by the given PCE. The difference between the err_α and err_β rates is subtle but important; the β signifies the metric's association with the rate of type II (or β) errors which relates to the number of false negatives produced by the pattern (also depicted in Fig. 11 by the region marked β). Conceptually, the err_β rate quantifies the pattern's ability in matching *all* of the shadows contained within the PCE; the closer the err_β rate is to 0 the more likely the pattern is to match all the shadows contained within the PCE. It refers to the *quantity* of *correct* results that the pattern is likely to produce in the future. A pattern with a low err_β rate expresses properties similar to the ones expressed by the given PCE, *regardless* of whether or not those properties are common to the captured shadows. Naturally, if the given PCE does not contain any shadows, the pattern's corresponding err_β rate is 1 since it could not possibly match *any* of the join points contained within PCE. For example, the above considered pattern would display an err_β of 0 w.r.t. the PCE found on line 3, Fig. 2 since it, when applied to the original base-code version, produces all the shadows captured by the PCE.

Recall that a pattern $\hat{\pi}$ is derived from a path π by replacing concrete elements in the path with wild-card elements. Wild-card graph elements may match a number of elements contained in the graph as detailed previously. When predicting a pattern's future ability to help rejuvenate a given PCE, we would like to take into account its *abstractness* (abbreviated abs), i.e., the ratio of the number of constituent wild-card elements to concrete elements. Let $|\hat{\pi}|$ denote the number of elements (vertices and edges), including wild-cards, at unique positions in the pattern $\hat{\pi}$. Moreover, let $\mathcal{W}(\hat{\pi})$ denote the multiset projection of wild-card elements contained in pattern $\hat{\pi}$. Likewise, $|\mathcal{W}(\hat{\pi})|$ represents the number of wild-card elements contained within pattern $\hat{\pi}$. Then, the abs of a pattern $\hat{\pi}$, which is independent of any particular PCE, is given by equation (3). Note that an empty pattern has no concrete elements, thus, we consider such a pattern to be completely abstract, i.e., having an abstractness of 1. To exemplify, the aforementioned pattern would be considered $\frac{2}{5}$ abstract.

The intuition behind abs is that patterns containing many wild-card elements 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 for the following reasons. A pattern that is very abstract, i.e., containing many wild-cards, is typically less likely to hone in on shadows that are *only* contained within the given PCE. Conversely, a pattern that is less abstract, i.e., more *concrete*, containing fewer wild-cards, is less likely to cover all shadows selected by the given PCE. The combined metrics are used to derive the *confidence* (abbreviated $conf$) pattern attribute depicted in equation (4), which is a convenient, single metric in judging the *confidence* we have in the pattern accurately detecting shadows to be included in a future, rejuvenated version of the related PCE. The closer a pattern's confidence is to 1 the more likely it will produce accurate suggestions in the future. In the case of our previous example, the pattern exhibits a $conf$ of 0.60 which, in turn, would be paired with the suggested shadow

$$err_{\alpha}(\hat{\pi}, \text{PCE}) = \begin{cases} 0 & \text{if } |Match(\hat{\pi}, Paths(CG^{+}))| = 0 \\ 1 - \dfrac{|\text{PCE} \cap Match(\hat{\pi}, Paths(CG^{+}))|}{|Match(\hat{\pi}, Paths(CG^{+}))|} & \text{otherwise} \end{cases} \quad (1)$$

$$err_{\beta}(\hat{\pi}, \text{PCE}) = \begin{cases} 1 & \text{if } |\text{PCE}| = 0 \\ 1 - \dfrac{|\text{PCE} \cap Match(\hat{\pi}, Paths(CG^{+}))|}{|\text{PCE}|} & \text{otherwise} \end{cases} \quad (2)$$

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

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

Figure 10: Pattern attribute equations.

FuelCell.increase(**double**) produced when applying the pattern to the new version of the base-code (cf. Fig. 3).

4. EXPERIMENTAL EVALUATION

In this section, we provide an overview of the experimental study conducted to quantitatively and qualitatively ascertain the usefulness of our rejuvenation approach in terms of its ability to accurately suggest shadows to be incorporated into a revised version of a PCE given evolutionary changes made to the base-code.

4.1 Implementation

We implemented our algorithm as a plugin, called REJUVENATE POINTCUT[7], to the popular Eclipse IDE. Eclipse abstract syntax trees (ASTs) with source symbol bindings were used as an intermediate program representation. The extended concern graph was constructed with the aid of the JayFX[8] fact extractor, extended for Java 1.5 and AspectJ, which generates "facts, " using class hierarchical analysis (CHA) [11], pertaining to structural properties and relationships, e.g., field accesses, method calls. Immediate project source code as well as transitively referenced libraries (possibly in binary format) are analyzed during graph building. The AJDT compiler[9] was leveraged to conservatively (see below) 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 shadows. Both pattern extraction and pattern-path matching was implemented via the Drools[10] rules engine, which makes use of a modified version of the RETE algorithm [16]. The Drools framework not only provides an efficient solution to the many-to-many matching problem the tool is faced with, as well as a natural query language, but also performance benefits such as the caching of results. Pattern descriptions were persisted as XML files, which were read and written to using the Java Domain Object Model (JDOM)[11] translation framework.

To increase applicability to real-world applications, we relaxed several assumptions described in Section 3. For example, we conservatively assume that dynamic advice, i.e., advice bound to a PCE containing run time predicates is always applied. If the tool encounters any inter-type declarations or any other form of the static crosscutting, the associated PCE is still processed but these constructs are ignored.

[7] `http://code.google.com/p/rejuvenate-pc`
[8] `http://www.cs.mcgill.ca/~swevo/jayfx`
[9] `http://www.eclipse.org/ajdt`
[10] `http://www.jboss.org/drools`
[11] `http://jdom.org`

4.2 Study Configuration

Our evaluation was conducted in two phases where subject source code was used as input to our tool "as-is" with no remarkable modifications made by the study designers. For both phases, the maximum analysis depth parameter was set at 2. Although setting the parameter to a value < 2 would theoretically improve performance, we chose a greater value due to the inherent nature of PCEs to capture join points that crosscut many heterogeneous architectural modules; thus, we deemed it necessary to drive the analysis reasonably deep through these layers. Evaluating trade-offs between performance and analysis depth has been designated for future work.

First, we aimed to show that the motivation behind our proposal is well-founded by demonstrating that join point shadows selected by a single PCE typically portray a significant amount of unique structural commonality. We did so by generating and, subsequently, studying patterns from single versions of 23 publicly available AspectJ benchmarks, applications, and libraries (including open-source projects) of varying size, in terms of non-blank, non-commented lines of code (LOC), and domain. Complete source code and descriptions of the studied subjects can be found on our website `http://tinyurl.com/6ewl2r`. To ensure that a certain level of quality was maintained, we purposefully selected subjects that have been used previously in the literature [5,7,9,12,31,33,46,47,49] including empirical studies [14,19]. This ensures that the subjects have achieved a particular level of acceptance within the community.

Table 2 lists the subjects along with associated KLOC[12] (column *KL*), ranging from 0.07 for Quicksort to 44.0 for MySQL Connector/J, number of class files after compilation (column *cls.*) and PCEs (column *PC*) analyzed[13], total selected shadows (column *shd.*), and thousands of patterns (column *KP.*) extracted (averaging 6.99 per shadow) and thereby evaluated. For each subject, the pattern generation was repeated five times using a 2.16 GHz Intel Core 2 Duo machine with a maximum Java heap size of 1GB. Column *t* depicts the total running time[14] in seconds, which itself averaged 8.22 secs per KLOC and 4.80 secs per PCE, indicating that the time required to generate our patterns is practical even for large applications. The remaining columns will be discussed in §4.3.

Our goal in the second phase of the experiment was to demonstrate the usefulness of our technique in a real-world setting. We did so by rejuvenating PCEs in multiple versions of 3 of the afore-

[12] Excludes code contained within aspect files.
[13] Includes only PCEs bound to advice bodies.
[14] Excludes intermediate representation (ASTs) construction time.

subject	KL.	cls.	PC	shd.	KP.	α	β	t (s)
AJHotDraw	21.8	298	32	90	3.36	0.32	0.06	101
Ants	1.57	33	22	297	1.25	0.15	0.23	43
Bean	0.12	2	2	4	0.02	0.24	0.23	4
Contract4J	10.7	199	15	350	1.80	0.26	0.44	115
DCM	1.68	29	8	343	2.47	0.15	0.45	4
Figure	0.10	5	1	6	0.02	0.11	0.45	8
Glassbox	26.0	430	55	208	2.62	0.1	0.29	228
HealthWatcher	5.72	76	27	122	1.00	0.21	0.16	22
Cactus	7.57	93	4	222	2.15	0.21	0.52	8
LoD	1.59	29	5	164	0.54	0.15	0.41	46
MobilePhoto	3.80	52	25	25	0.78	0.23	0.00	11
MySQL[a]	44.0	187	2	3016	17.6	0.12	0.58	379
NullCheck	1.47	27	1	112	0.10	0.17	0.55	293
N-Version	0.55	15	4	9	0.08	0.19	0.24	1
Quicksort	0.07	3	4	7	0.06	0.19	0.15	3
RacerAJ	0.58	13	4	9	0.02	0.23	0.09	5
RecoveryCache	0.22	3	4	14	0.07	0.11	0.21	6
Spacewar	1.42	21	9	58	0.23	0.15	0.22	37
StarJ-Pool	38.2	511	1	3	0.07	0.25	0.00	75
Telecom	0.28	10	4	5	0.03	0.21	0.02	7
Tetris	1.04	8	18	27	0.50	0.16	0.01	14
TollSystem	5.20	88	35	85	1.68	0.26	0.06	20
Tracing	0.37	5	16	132	0.68	0.17	0.4	1
Totals:	174	2137	298	5308	37.1	0.18	0.16	1431

[a]MySQL Connector/J

Table 2: Phase I: Correlation analysis experiment results.

subject	vers.	PC	trg.	rec.	pr.	t (s)
Contract4J	5	13	317	0.81	0.05	1046
HealthWatcher	8	6	30	1.00	0.13	146
MobilePhoto	7	39	33	0.97	0.02	266
Totals:	20	49	380	0.93	0.04	1458

Table 3: Phase II: Rejuvenation experiment results.

mentioned subjects. These subjects, listed in Table 3, were comprised of a series of releases (column *vers.*) which allowed the accuracy of the shadows mechanically suggested by our tool to be evaluated against *actual* modifications to PCEs, in terms of included shadows, made by human developers in subsequent software versions. For our approach to be successfully evaluated, a complete set of changes were required to be considered in isolation. It was often the case that subsequent versions in SVN/CVS repositories did not contain complete changes, e.g., the base-code was modified and committed with the PCE modified and committed in a later version. This made reasoning about units of discrete modifications difficult, thus, we considered major releases of the analyzed software as units of evolution. Moreover, we were solely interested in rejuvenating PCEs between versions that exhibited "significant" modifications. As a result, we defined the following conditions for PCEs regarding subsequent software versions which ensured that the performance of our tool was evaluated only in situations where the PCE recovery due to modifications to the base-code was non-trivial. We say that a PCE contained in a software version A *evolved* between a version B iff

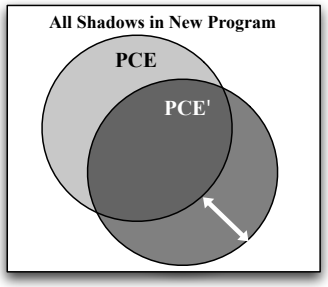

Figure 12: Comparing a PCE with its revision in the new program.

- the textual representation of the PCE in A differs from the textual representation of the PCE in B,
- the set of shadows selected by the PCE in A is disjoint from the set of shadows selected by the PCE in B, and
- the set of shadows selected by the PCE in B is disjoint from the set of shadows selected by the old representation of the PCE in B.

The last criterion asserts that the region designated by the light-shaded arrow in the Venn diagram depicted in Fig. 12, where the outer region symbolizes all shadows in B, *PCE* the shadows in B selected by the old representation, and *PCE*$'$ the shadows selected in B by the new representation, is non-empty. Thus, our experiment evaluated the performance of our tool only in situations

where a textual modification to the PCE was required to allow the PCE to continue to capture intended shadows. Column *PC*, Table 3 shows the number of PCEs across versions which met this criteria and were, consequently, selected to be rejuvenated by our tool. Determining the region marked as *PCE* in Fig. 12 required carefully copying the original PCE to the subsequent software version and binding it to an empty advice body. Column *t* designates the total rejuvenation time in secs taken which averaged ~4 secs per KLOC, indicating that the tool is practical to use. The remaining columns are discussed in §4.4.

4.3 Phase I: Correlation Analysis Results

In first phase, we assessed the amount of unique structural commonality typically portrayed by join point shadows selected by a single PCE by studying the attributes (cf. Fig. 10) of patterns extracted from a single version of the subjects listed in Table 2. Recall that a pattern with a low err_α is one that expresses unique structural commonalities between shadows selected by the PCE it was extracted from. In this situation, applying the pattern to the original version of the base-code would result in a set of suggested shadows that matched closely with those selected by the PCE itself. Thus, a pattern with a low err_α rate is one that expresses common structural characteristics amongst shadows selected by the PCE that are *not* exhibited by other shadows. Column α depicts the average err_α rate for all patterns extracted from the associated subject. We found the average, weighted by the number of patterns extracted, err_α rate among all subjects to be 0.18, demonstrating that a high correlation exists. Moreover, we found this correlation to be exceptionally widespread, i.e., not only was the commonality unique to shadows selected by a particular PCE, but *many* of these shadows shared these characteristics. This is indicated by the average err_β rate (column β) whose average, weighted by the number of PCEs analyzed, among all subjects was found to be 0.16. The combination of these two findings show that shadows selected by a single PCE indeed typically display a significant amount of unique structural commonality.

4.4 Phase II: Expression Recovery Results

During the second phase, we assessed the accuracy of our technique by rejuvenating PCEs in multiple versions of the subjects listed in Table 3. We then evaluated the relationship between the shadows that were suggested for inclusion by our tool and those that were actually included in (human) revised PCEs residing in a subsequent software version. In particular, we were interested in exploring our tool's performance in being able to precisely suggest shadows that were selected by the revised PCE but would not have been selected by the original PCE had we applied it to the new base-code version. These are exactly the shadows that the developer would have had to *manually* determine to be applicable to the PCE, which coincide with those that our tool could be most helpful in mechanically discovering. This "target" set of shadows is represented by the region surrounding the light-shaded arrow in Fig. 12. The total number of shadows occupying this regions across all rejuvenations is listed by column *trg.*, Table 3.

4.4.1 Quantitative Analysis

As success metrics for evaluating our approach, we defined a *promising* rejuvenation to be one where our tool suggests the majority of shadows contained within the target region. Moreover, as suggestions are ranked by confidence (cf. Section 3.6), we defined a *precise* rejuvenation to be one where targeted shadows appeared near the top of the list of suggestions. Column *rec.*, Table 3 shows the average recall at which our tool was able to suggest targeted shadows. The average recall across all subjects was found to be 0.93, indicating that, on average, our tool suggested 93% of shadows that resided in this region. This demonstrates that our tool typically resulted in promising rejuvenation. Column *pr.*, on the other hand, portrays the average percentile rank, a means to divide an ordered list into sections, of targeted shadows. A low percentile rank indicates that the suggested shadow appears towards the top of (or first on) the list and vice versa. Our results show that targeted shadows, on average, appeared in the top 4^{th} percentile of the list of suggested shadows produced by our tool, which would have allowed the developer to easily identify them. This establishes that the rejuvenation performed by our tool was exceptionally precise.

4.4.2 Qualitative Analysis

In this section, we identifying potential reasons for both accurate and inaccurate suggestions made by our tool. For succinctness, we draw examples from only the HealthWatcher subject. The major contributing factor that was found to cause patterns derived by our approach to be ineffective when applied to subsequent versions relates to modifications made to the base-code that involved removing program elements appearing in patterns. For example, the PCE **call**(∗ HttpSession+.putValue(String, Subject)) was affected by a modification to the base-code which involved introducing the *Adapter* design pattern. Consequently, the HttpSession class was replaced, invalidating all patterns containing references to this class. Fortunately, however, our tool was able to compensate by producing other patterns that were effective in rejuvenating the aforementioned PCE.

Common base-code modifications involved structural refactorings. For example, one modification encompassed introducing the *Command* design pattern which required relocating the implementations of several Servlets to a series of Command classes. This activity induced the need to rejuvenate several PCEs. As the modifications made to the base-code were minimal and purely structural, i.e., the method bodies remained intact, our patterns encouragingly but expectedly proved completely effective in this situation, suggesting only and all of the targeted shadows.

We found several PCEs in the subjects to be very specific, often selecting only a single join point. Ergo, patterns, although few, constructed using these PCEs were generally associated with a high confidence value. However, it was not clear such patterns would prove useful as base-code modifications that break the PCE could be few and far between. Furthermore, having only a minimal set of patterns generated for these PCEs, we questioned their usefulness in the cases that such change does occur. Despite this, we did find scenarios involving updates to these PCEs and, surprisingly, our patterns were able to produce accurate suggestions in these situations. One particular PCE that related to synchronization required rejuvenation due to *new* types introduced. An obscure pattern that centered upon references to an exception raised by classes that required the managed synchronization behavior caused shadows associated with the new types to be accurately suggested. This demonstrates a benefit of our approach in its ability to discover obscure structural characteristics that a developer may not have been immediately aware of when manually updating PCEs.

5. RELATED WORK

5.1 Concern Traceability

The closest work resembling (and inspiring) ours involves tracking [38] and managing [10] concerns in source code throughout evolution. These approaches do not specifically deal with AOP, and our approach may be seen as their adaption and extension to the

paradigm. However, there are several key differences. Firstly, [10] derives expressive intensional patterns from enumeration-like extensional descriptions of where concerns apply in source code and proceeds to compare the performance between the two. Our patterns are also intensional descriptions but derived from *other* intensional descriptions, viz., PCEs. Also, patterns produced by our approach have been made to compete with the expressiveness inherent to PCEs which deal specifically with CCCs, e.g., our confidence evaluation is obtained using three dimensions of analysis (cf. Fig. 10). Recall from §4.2 that due to the nature of CCCs, our graph-based approach features a general analysis depth parameter and corresponding algorithmic considerations to derive patterns of a parameterized length. Thus, concepts pertaining to algorithm development are treated more fully in this work. Lastly, we present a thorough empirical evaluation of our technique's performance in the realm of evolving AO software.

5.2 Aspects and Refactoring

A technique for automatically updating PCEs upon various refactorings of the base-code is presented in [45]. The associated tool only updates PCEs only when predefined refactorings are invoked, whereas our tool deals with general base-code modifications. Moreover, in contrary to our technique, the approach is unable to update PCEs due to additions of new join points introduced in the new base-code version.

The approach presented in [3] clusters a set of given join points to a single PCE based on common characteristics in program element names, using lexical matching, for refactoring non-AO software to use aspects. The proposal does not consider the PCE maintenance upon base-code evolution in AO software. Nevertheless, we foresee an interesting scenario where the proposed tool may be integrated with our technique to automatically cluster suggested join points to be included in a revised PCE.

5.3 Automated AOSD

Several techniques [2,36,43] aim to automate AOP-based development. However, they focus on analyzing the *changes* in shadows between software versions so that the developer fully understands the impact of the alteration of the base-code on advice behavior. In contrast, the focus of our approach is to *infer* shadows that likely belong in a new version of the PCE based upon those changes. Automated tools such as AJDT and PointcutDoctor [48] display join points that currently and *almost*, respectively, match a given PCE, but do not analyze the differences exhibited by join points *between versions* of the base-code. Furthermore, the ranking scheme of [48] is hard-coded by a predefined, developer-minded heuristic, while our approach ranks join point suggestions in a more custom fashion using analysis results from the previous base-code version.

5.4 Pointcut Fragility

It is claimed that current PCE languages are not be sufficiently expressive to represent the developer's true intentions in capturing join points corresponding to a PCE [32], these difficulties being rooted at the inherent *fragility* of typical PCE languages [30]. Several approaches [13,25,29,35,42] attempt to add expressiveness to help combat this problem by altering or abstracting the underlying join point model. Others [6, 22] go even further by proposing approaches that combat fragility in these models. Our proposal confronts the problem from a fundamentally different perspective by combating pointcut fragility in a *current* language (AspectJ) and essentially maintaining a rich join point model underneath the given one. In this view, the tool makes suggestions based off this rich model while affording the developer the luxury of using a familiar AO language. Yet, others [24, 37] propose new, hybrid languages that feature facets from both paradigms. Thus, these languages would not be considered completely AO in a traditional sense [15].

6. CONCLUSION AND FUTURE WORK

In this paper, we have proposed an approach which limits the problems associated with pointcut fragility by providing automated assistance to developers in rejuvenating pointcuts as the base-code evolves. Arbitrarily deep structural commonalities between program elements corresponding to join points captured by a pointcut in a single software version are harnessed and analyzed. Patterns expressing this commonality are then applied to subsequent versions to offer suggestions of new join points that may require inclusion. The implementation of a publicly available tool was discussed, and the results of an empirical investigation were presented, indicating that our approach is particularly usefulness in rejuvenating PCEs in a real-word setting.

In its current state, our tool presents the developer with the suggested shadows that are to be manually integrated. In the future, once the selection is final, PCEs can be automatically rewritten using existing refactoring support [4] adapted for AspectJ constructs. Moreover, we plan to utilize tool-support from [3] in order to perform PCE rewriting via join point clustering and string analysis of program element names. Also, a program element tracing mechanism, e.g., Java Annotations, may be useful in pinpointing PCE declarations across subsequent software versions.

Potential future work also entails incorporating aspect types and semantics of inter-type declarations into the construction of the extended concern graph. Furthermore, a more accurate assessment of the dynamic applicability of advice may be an interesting avenue to explore, possibly using dynamic traces in the initial analysis. Dynamic analysis may also be valuable in more accurately estimating the truth values associated with the relations depicted by the concern graph.

Acknowledgments

We would like to thank Barthelemy Dagenais, Tao Xie, Alexander Egyed, Martin Robillard, Alfred V. Aho, Marc Eaddy, and Linton Ye for their answers to our many technical and research related questions and for referring us to related work.

7. REFERENCES

[1] J. Aldrich. Open modules: Modular reasoning about advice. In *ECOOP*, 2005.

[2] P. Anbalagan and T. Xie. Apte: automated pointcut testing for AspectJ programs. In *WTAOP*, 2006.

[3] P. Anbalagan and T. Xie. Automated inference of pointcuts in aspect-oriented refactoring. In *ICSE*, 2007.

[4] D. Bäumer, E. Gamma, and A. Kiezun. Integrating refactoring support into a Java development tool. In *OOPSLA*, 2001.

[5] E. Bodden and K. Havelund. Racer: effective race detection using AspectJ. In *ISSTA*, 2008.

[6] M. Braem, K. Gybels, A. Kellens, and W. Vanderperren. Automated pattern-based pointcut generation. In *SC*, 2006.

[7] N. Cacho, F. C. Filho, A. Garcia, and E. Figueiredo. Ejflow: taming exceptional control flows in aspect-oriented programming. In *AOSD*, 2008.

[8] W. Cazzola, S. Pini, and M. Ancona. Design-based pointcuts robustness against software evolution. In *RAM-SE*, 2006.

[9] R. Coelho, A. Rashid, A. Garcia, F. C. Ferrari, N. Cacho, U. Kulesza, A. von Staa, and C. J. P. de Lucena. Assessing the impact of aspects on exception flows: An exploratory study. In *ECOOP*, 2008.

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

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

[12] B. Dufour, C. Goard, L. Hendren, O. D. Moor, G. Sittampalam, and C. Verbrugge. Measuring the dynamic behaviour of AspectJ programs. In *OOPSLA*, 2004.

[13] M. Eichberg, M. Mezini, and K. Ostermann. Pointcuts as functional queries. In *APLAS*, 2004.

[14] 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 *ICSE*, 2008.

[15] R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Advanced Separation of Concerns*, 2000.

[16] C. L. Forgy. Rete: a fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, pages 324–341, 1982.

[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[18] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java™ Language Specification (3rd Edition)*. Addison-Wesley, 2005.

[19] 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 *ECOOP*, 2007.

[20] W. Griswold, K. Sullivan, Y. Song, M. Shonle, N. Tewari, Y. Cai, and H. Rajan. Modular software design with crosscutting interfaces. *IEEE Software*, 2006.

[21] S. Gudmundson and G. Kiczales. Addressing practical software development issues in aspectj with a pointcut interface. In *Advanced Separation of Concerns*, 2001.

[22] K. Gybels and J. Brichau. Arranging language features for more robust pattern-based crosscuts. In *AOSD*, 2003.

[23] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In *AOSD*, 2004.

[24] K. Hoffman and P. Eugster. Bridging java and aspectj through explicit join points. In *PPPJ*, 2007.

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

[26] R. Khatchadourian, J. Dovland, and N. Soundarajan. Enforcing behavioral constraints in evolving AO programs. In *FOAL*, 2008.

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

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

[29] K. Klose and K. Ostermann. Back to the future: Pointcuts as predicates over traces. In *FOAL*, 2005.

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

[31] U. Kulesza, C. Sant'Anna, A. Garcia, R. Coelho, A. von Staa, and C. Lucena. Quantifying the effects of aspect-oriented programming: A maintenance study. In *ICSM*, 2006.

[32] M. Lippert and C. Lopes. A study on exception detection and handling using AOP. In *ICSE*, 2002.

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

[34] H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In *CC*, 2003.

[35] K. Ostermann, M. Mezini, and C. Bockisch. Expressive pointcuts for increased modularity. In *ECOOP*, 2005.

[36] M. A. Perez-Toledano, A. Navasa, J. M. Murillo, and C. Canal. Titan: a framework for aspect oriented system evolution. In *Software Engineering Advances*, 2007.

[37] H. Rajan and G. Leavens. Ptolemy: A language with quantified, typed events. In *ECOOP*, 2008.

[38] M. P. Robillard. Tracking concerns in evolving source code: An empirical study. In *ICSM*, 2006.

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

[40] K. Sakurai and H. Masuhara. Test-based pointcuts: a robust pointcut mechanism based on unit test cases for software evolution. In *LATE*, 2007.

[41] L. M. Seiter. Role annotations and adaptive aspect frameworks. In *LATE*, 2007.

[42] J. Sillito, C. Dutchyn, A. D. Eisenberg, and K. D. Volder. Use case level pointcuts. In *ECOOP*, 2004.

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

[44] K. Sullivan, W. Griswold, Y. Song, Y. Cai, M. Shonle, N. Tewari, and H. Rajan. Information hiding interfaces for aspect-oriented design. In *FSE*, 2005.

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

[46] G. Xu and A. Rountev. Regression test selection for aspectj software. In *ICSE*, 2007.

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

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

[49] J. Zhao. Change impact analysis for aspect-oriented software evolution. In *Principles of Software Evolution*, 2002.