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

Raffi Khatchadourian, *Member*, *IEEE*, Phil Greenwood, Awais Rashid, *Member*, *IEEE*, and Guoqing Xu

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 that 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 three of these programs. The results show that our parameterized heuristic algorithm was able to accurately and automatically infer the majority of new join points in subsequent software versions that were not captured by the original pointcuts.

Index Terms—Software development environments, software maintenance, software tools.

1 Introduction

Aspect-oriented Programming (AOP) [23] 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). Join point sets are described by pointcut expressions (PCEs), which are predicate-like expressions over various characteristics of "events" that occur during the program's execution. In AspectJ [22], such characteristics may include calls to certain methods, accesses to particular fields, and modifications to the runtime stack.

Consider an example PCE **execution**(* m*(..)) that 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 one base code version 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 in its unaltered form it is able to *continue* to capture the correct set of join points in future base code versions. 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 begins 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 cases, we say that the PCE is *fragile*. Hence, the *fragile pointcut problem* [25] 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 (e.g., [6], [24], [31], [36], [37]), limiting the scope of where advice may apply through more clearly defined interfaces (e.g., [14]), or enforcing structural and/or behavioral constraints on advice application (e.g., [13], [19], [40]). Yet others make points where advice may apply more explicit in the base code [17], or remove PCEs altogether [33]. 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.

Programmer-defined source code annotations can also be used to "mark" relevant locations in the source code where a CCC applies. PCEs then use these annotations to accurately select the appropriate join points. If used properly, i.e., if all

- *R. Khatchadourian and G. Xu are with the Department of Computer Science and Engineering, Ohio State University, Columbus, OH 43210. E-mail: {khatchad, xug}@cse.ohio-state.edu.*
- *P. Greenwood and A. Rashid are with Computing Department, Lancaster University, Lancaster LA1 4WA, United Kingdom. E-mail: {greenwop, awais}@comp.lancs.ac.uk.*

Manuscript received 27 Feb. 2010; revised 23 Aug. 2010; accepted 26 Jan. 2011; published online 7 Feb. 2011.
Recommended for acceptance by T. Tamai.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2010-02-0054.
Digital Object Identifier no. 10.1109/TSE.2011.21.

locations where the CCC applies are correctly annotated and if the corresponding PCE correctly selects these elements, this scheme can produce PCEs that are robust to changes such as refactorings since names and organization of program elements may change but the associated annotations remains intact. However, refactoring is not the only reason for PCE breaks. For example, adding a new element but neglecting to annotate it properly with *all* CCCs that apply to it will break an annotation-based pointcut.

It is important to note that, although this paper deals with the particular case of AspectJ, a system written in *any* language that allows developers to declare composition specifications (like PCEs in AspectJ) is susceptible to this predicament. Furthermore, this problem unfortunately develops into a vicious cycle where these new PCEs may also exhibit similar fragility problems.

To alleviate such problems, we propose an approach that provides automated assistance in rejuvenating PCEs upon changes 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, m_1, m_2, and m_3. 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 input PCE, 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 insight into the fragile pointcut problem is as follows: CCCs tend to crosscut traditional module boundaries. Thus, CCCs affect many heterogeneous modules across a software system. Despite their differences, these modules have at least one facet in common, i.e., that a particular CCC applies to them. Our hypothesis is that places in the source code corresponding to where a CCC applies share a similar structure, and that this information can be leveraged to maintain PCEs.

Our key contributions are as follows:

1. **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.
2. **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 with low α (false positive) and β (false negative) error rates of 18 and 16 percent, respectively.
3. **Expression recovery**. To ensure the applicability and practicality of our approach, we implemented our algorithm as an Eclipse (http://eclipse.org) IDE plug-in and evaluated its usefulness by rejuvenating PCEs in multiple versions of three of the aforementioned programs, which were of varying sizes and domains and representative of typical AO software. We found that, in exploiting the extracted patterns, our tool was able to accurately and automatically infer 90 percent of new join points that were selected by PCEs in subsequent software versions that were not selected by the original PCE, with a standard deviation of 24 percent. This demonstrates that the approach is indeed useful in alleviating the burden of recovering PCEs upon base-code modifications that took place in our subject programs, and the results advance the state of the art in automated tool support for coping with the evolution of AO programs.

A brief introduction of this work originally appeared in [20], and a demonstration of our preliminary tool, along with details of the implementation, appeared in [21]. In this paper, we fully describe our complete approach, which has been built upon the aforementioned previous work. This complete approach includes thoroughly developed ideas that have been incorporated into our initial algorithm. We also present a new dimension of our experimental results to comprehensively and accurately assess the overall usefulness of our approach.

The remainder of this paper is organized as follows: Section 2 presents a motivating example that features a fragile PCE. Section 3 highlights the key algorithmic facets of our approach, while Section 4 discusses the details of our implementation and evaluation. In Section 5, we compare our proposal with related work and explore future work, as well as conclude, in Section 6.

2 Pointcut Fragility Example

Fig. 1 shows an example AspectJ code snippet for a hypothetical drive-by-wire programming of an all-wheel drive, hybrid vehicle (line 2) which draws power from two different sources, namely, a diesel engine (line 25) and an electric motor (line 30), both of which contribute to the overall speed (line 3).[1] Fuel is distributed to the engine via

1. This example was inspired by one of the authors' work at the Center for Automotive Research (CAR) at Ohio State University.

```
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     this.car.notifyChangeIn(fuel);}}
29
30 class ElectricMotor {
31   private HybridAutomobile car;
32   public void increase(Current current) { //...
33     this.car.notifyChangeIn(current);}}
34
35 class Dashboard {
36   private HybridAutomobile car; //...
37   public void update() {
38     this.display(car.getOverallSpeed()); }}
```

Fig. 1. Hybrid automobile example.

the method DieselEngine.increase(Fuel) (line 27), while electricity is distributed to the motor via the method ElectricMotor.increase(Current) (line 32), whose method bodies are abbreviated. The classes conform to the *Observer* pattern, 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., the Dashboard (line 35). An accessor method (line 22) retrieves the value of the private instance field overallSpeed, which the method Dashboard.update() invokes (line 38) 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.[2] The points at which this advice is to apply are specified by its bound PCE (line 3) that selects join points corresponding to the execution of two of the aforementioned methods, namely,

2. For this advice to properly function, the pointcut must expose appropriate *context* pertaining to the join point. Specifically, both the implicit and explicit arguments of the increase(Energy) method would need to be exposed, perhaps by using the pointcut designators **this()** and **args()** to perform the checks. We have omitted these designators for presentation purposes.

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

Fig. 2. Speeding prevention aspect.

DieselEngine.increase(Fuel) and ElectricMotor.increase (Current). These methods have been underlined in Fig. 1. 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 to focus on the applicability of the advice.

Further suppose that the base code (Fig. 1) evolves to accommodate a new vehicle energy source, namely, a fuel cell, resulting in the creation of a FuelCell class (Fig. 3). 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 5).

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. It would also be helpful to mechanically suggest join points that should no longer be selected by a PCE. We will continue to use this example to demonstrate how our proposed approach can assist developers with these issues in an automated fashion.

3 Harnessing Commonality

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 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, namely, DieselEngine.increase(Fuel) and ElectricMotor.increase(Current), whose corresponding method executions were selected by the PCE listed on line 3, Fig. 2, are both

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

Fig. 3. A new fuel cell class.

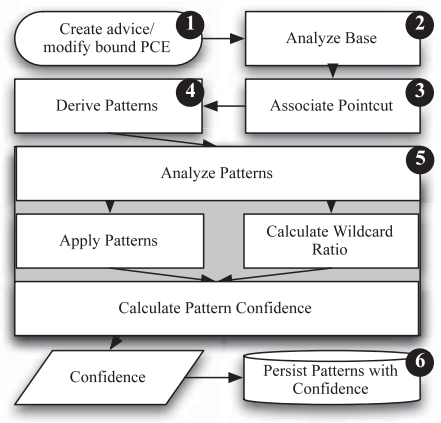

Fig. 4. Phase I: Pointcut analysis.

declared in classes contained in package p. Additionally, considering solely the code snipped characterized in Fig. 1, both method bodies contain calls to methods, namely, 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 kinds of relations that program elements have in common. 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

Our approach is divided into two conceptual phases: *analysis* and *rejuvenation*. The analysis phase (Fig. 4) is triggered upon modifications to or creation of advice-bound PCEs (step 1). Named-PCEs are analyzed when they are referred to in advice-bound PCEs. A graph is then computed which depicts structural relationships among program elements currently residing in the base code (step 2). Next, patterns are derived from paths in which vertices and/or edges representing program elements and/or relationships are associated with join points selected by the PCE (steps 3 and 4). The patterns are then analyzed to evaluate the *confidence* (inspired by [7]) 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 (step 5). Subsequently, results produced by the pattern are correlated with and ranked by this value when presented to the developer. Finally, patterns along with their confidence are linked with the PCE and persisted (step 6) for later use in the next phase.

Our approach is most helpful in scenarios where a developer performs a series of changes to the base code and then, prior to deployment/testing, proceeds to update PCEs to reflect those changes, ensuring that new join points are captured correctly. The rejuvenation phase (Fig. 5) is triggered previously to the developer manually altering the PCE so that automated assistance in performing the updates correctly can be provided (1). Patterns previously linked with the PCE are retrieved from storage and matched against a graph computed from the new base code version to unveil the suggested join points (2). These join points are the ones related to program elements that share structural

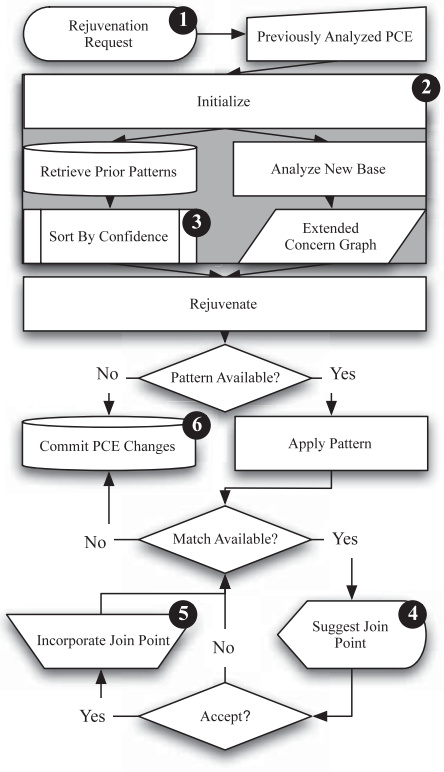

Fig. 5. Phase II: Pointcut rejuvenation.

commonality 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 (4), and the list of all suggestions is sorted in decreasing order of confidence (as a result of 3). The developer then adjusts the PCE to either incorporate or exclude the desired join points (5, 6) based on the suggestions.

3.2 Assumptions

We first state several simplifying assumptions about the underlying source code to be analyzed; we discuss in Section 4.1 how much of these have been relaxed in our implementation and how others can be dealt with in future work in Section 6. First, we assume that the input PCE is initially correctly specified, i.e., it selects (and only selects) intended join points. This ensures that the structural commonality exhibited by the corresponding program elements is correctly related to the input PCE. Furthermore, we assume that intertype declarations (ITDs) (static crosscutting) are not utilized by the analyzed aspects. Intertype declarations allow aspects to introduce and modify facets of the base code, e.g., member introduction, existing at compile time. This assumption helps simplify the algorithm presentation. Adding intertype declarations to the current algorithm would be reasonably straightforward.

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 separate from the base code; advice may only apply to join points associated with classes, interfaces, and other Java types. This assumption also helps simplify the algorithm presentation

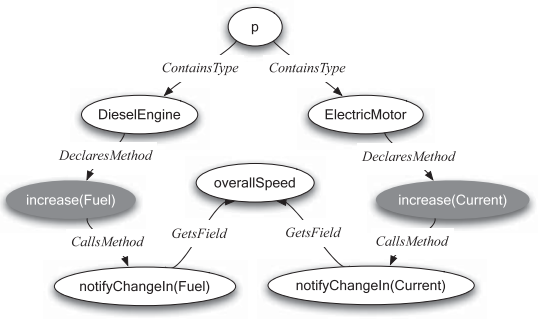

Fig. 6. A graph subset computed from the example.

since it reduces both the kinds of entities and relations between the entities existing in the input program that need to be considered. Moreover, it frees us from resolving the targets of **proceed** calls, which may exist in around advice. We discuss adding advice bodies in Section 6. Last, we assume that we can accurately resolve the declaration of the advice a PCE is bound to across varying versions of the software. This may be invalidated via the use of refactorings, e.g., member relocation, being applied in between software versions. Section 6 discusses plans for how our approach can be made to cope with this issue.

3.3 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* [35]. Concern graphs have been used in previous work [34] to discover, describe, and track concerns in evolving source code as they allow for succinct program representations. We have chosen to use concern graphs since they include information about the structure of programs, and we are interested in unveiling underlying structural patterns. We extended concern graphs with several elements found in current Java languages, e.g., annotations, and adapted them for use with AOP.

We specify an extended concern graph CG^+ to be a labeled multidigraph consisting of a 4-tuple $CG^+ = (V, E, R, \ell)$. The vertices V represent program elements contained within the analyzed program, specifically, packages, classes, interfaces, enumeration types, annotations, methods, and fields. We do not consider local variables and other parameters in our analysis as crosscutting concerns tend to crosscut a larger granularity of program elements. E is a multiset of directed edges that connect vertices in V depending on various relations that may hold between them as depicted in the source code. For example, HybridAutomobile and overallSpeed (Fig. 1) are related in that the class HybridAutomobile *declares* the field overallSpeed. In this case, there would exist an edge connecting the vertex that represents HybridAutomobile to the vertex representing overallSpeed. R is the set of all such (binary) relations that we consider. Since two vertices may be related in several ways, i.e., they satisfy more than one relation, there may exist multiple edges between them. As such, $\ell : E \rightarrow R$ serves as a labeling function that distinguishes edges by labeling them with the satisfied relations. Fig. 6 portrays a subset of the graph computed from the example given in Section 2.

TABLE 1
Analyzed Program Entity Types and Relations

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 portrays the complete set of binary relations that we consider as well as the program entity types in which they relate. Either of these relations may hold in a structural sense, e.g., field declarations, or possibly during a particular execution of the program, e.g., method calls. Section 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 Section 6 touches upon future work which could result in a more accurate approximation. Many kinds of relations may be formulated; however, we mainly focus on popular relations as used in previous work [5], [7], [35], with the addition of relations useful for AO languages, e.g., $Annotates$. Section 4 reports on the appropriateness of using such relations for PCE rejuvenation in AspectJ programs; adding additional relations is discussed in Section 6.

3.4 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 describes a set of join points, which are well-defined points in the program's execution. Thus, a join point is very much dynamic in nature. A join point *shadow*, conversely, 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 [29]. Whether the base code is advised at that point is dependent on advice being applicable and possible dynamic conditions being met. We treat a program as consisting of a set of join point shadows that *may* or *may*

not be currently advised. This definition differs slightly from those typically given in the literature [16], [43] and helps simplify the algorithm presentation. Moreover, we treat a PCE as selecting a subset of these shadows, i.e., we assume that the PCE is free of dynamic conditions. This allows us to exploit solely static information in our analysis. Section 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. The evaluation results reported in Section 4.4 indicate that the impact of this limitation is minimal and that our current approach can be useful. There is evidence that suggests that most PCEs do not take advantage of dynamic conditions [2].

Shadows corresponding to method declarations enable method vertices, i.e., 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. Thus, a vertex representing the method m would be considered enabled w.r.t. a PCE that selects a method execution-join point for m.

For a graph built from the example 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 in Fig. 6 illustrates this; the vertices representing these methods are shaded.

While shadows corresponding to method declarations enable method vertices, shadows corresponding to sites (call-sites, field access, etc.) enable edges. 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., $CallsMethod(u, v)$ 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 lies at the *declaration* of the invoked method, while in the latter, it lies at the *site* of the method invocation, i.e., the client code. Section 4.1 discusses how our implementation leverages existing tool support to deduce enabled graph elements.

3.5 Pattern Extraction

Once we associate (enable) graph elements with the input PCE, (see Section 3.4), we analyze structural 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 structural characteristics with a particular level of confidence. Note that we only take advantage of structural commonality between program elements and not other kinds of commonality, e.g., string similarity of method names. We are interested in exploiting information pertaining to the structure and organization of the base code when related to PCEs.

Recall that (increase(Fuel) and increase(Current)), whose corresponding execution was selected by (**execution(void** increase(Energy+))), both contained calls to (notifyChangeIn(Fuel) and notifyChangeIn(Current)), which read from (overallSpeed). Deliberately, this information is expressed by two paths (sequences of connected edges) increase(Fuel) $\rightsquigarrow$ overallSpeed and increase(Energy) $\rightsquigarrow$ overallSpeed 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, i.e., graph elements representing program elements corresponding to join point shadows selected by the input PCE, will ultimately be applied to graphs computed from subsequent 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 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. Section 4.2 discusses our choice for k in our evaluation. An example of such a path when taking the enabled vertex $v = \text{increase(Fuel)}$ and $k = 2$ is $\text{increaseFuel(Fuel)} \xrightarrow{cm} \text{notifyChangeIn(Fuel)} \xrightarrow{gf} \text{overallSpeed}$, where edge labels cm and gf refer to the satisfied relations $CallsMethod$ and $GetsField$, respectively.

Intuitively, patterns are constructed from paths so that paths matching the pattern are ones that share common origins or sinks with the original path. Also, vertices in the matching paths are connected via similar (in terms of labels) edges as the vertices in the original path.

We consider two kinds of patterns, those derived from enabled vertices, called *vertex*-based patterns, and those from enabled edges, called *edge*-based patterns. A vertex-based pattern is obtained from a path by replacing vertices along the path with *vertex wild cards*. Vertex-based patterns are used for suggesting method execution join points. An *edge*-based pattern is obtained by not only replacing vertices with vertex wild cards, but also a certain edge with an *edge wild card*. Edge-based patterns are used for suggesting site-based (e.g., call-site, field-set) join points. The replacing edge wild card is related to the site-based shadow to be suggested.

Vertex-based patterns will contain all but one (nonwild) concrete vertex; this is the element representing the common source or sink. Every edge in a vertex-based pattern is concrete so that paths containing similarity connected vertices can be matched with the pattern. There are no edge wild cards in a vertex-based pattern. Edge-based patterns are similar to vertex-based patterns with the exception of the single edge wild card mentioned above.

While pattern matching is covered in Section 3.6, we briefly discuss wild card matching here. 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 as to select shadows that 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., Section 3.4) matched by enabled wild cards are eventually suggested.

We extract vertex-based patterns from a path $\pi = \langle e_1, e_2, \ldots, e_n \rangle$ and an enabled vertex v along π. Details of the algorithm can be found in Fig. 10 in the Appendix. 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 path

$$\text{increaseFuel(Fuel)} \xrightarrow{cm} \text{notifyChangeIn(Fuel)} \xrightarrow{gf} \text{overallSpeed}$$

where the vertex increaseFuel(Fuel) is enabled w.r.t. the PCE. $\{?^* \xrightarrow{cm} ? \xrightarrow{gf} \text{overallSpeed}\}$ would be the singleton extracted from this path, 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 the 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 manner. Details of the algorithm can be found in Fig. 11 in the Appendix. 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.

3.6 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.

To illustrate, suppose we augmented the graph in Fig. 6 with new vertices and edges representing facets of the FuelCell class in Fig. 3. The resulting situation is depicted in Fig. 7, where a new path

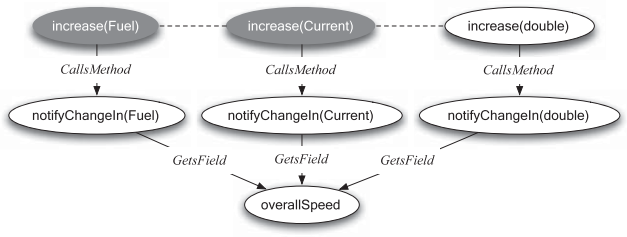

Fig. 7. Evolving the base code with a FuelCell class.

$$\text{increase}(\textbf{double}) \xrightarrow{cm} \text{notifyChangeIn}(\textbf{double}) \xrightarrow{gf} \text{overallSpeed}$$

matches the previously extracted pattern $?^* \xrightarrow{cm} ? \xrightarrow{gf} \text{overallSpeed}$.

Given that a pattern matches a path, suggested shadows are ones represented by graph elements along the path that 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 $?^* \xrightarrow{cm} ? \xrightarrow{gm} \text{overallSpeed}$ against the path

$$\text{increase}(\textbf{double}) \xrightarrow{cm} \text{notifyChangeIn}(\textbf{double}) \xrightarrow{gf} \text{overallSpeed},$$

the method FuelCell.increase(**double**) is represented by a vertex that matches an enabled wild card element. The situation is emphasized in Fig. 7 by a dashed line through the vertices that induced the wild card. As a result, we suggest that the CCC being realized by the advice on lines 2-4 of Fig. 2 applies to the shadow corresponding to the execution of this method due to its semantic equivalence with other shadows to which the same CCC applies, i.e., the ones selected by the PCE on line 3. In AspectJ, however, multiple advice declarations may to be responsible for realizing a particular CCC, similar to how multiple methods may be responsible for realizing a particular concern in Java. Such is the case here since applying the CCC to the suggested shadow would entail creating a new advice declaration to expose context from incompatible parameter types in the same position (in this case, Energy and **double**, both being the first parameter). Thus, upon our suggestion, the developer would proceed to create a new advice declaration bound to the PCE **execution**(**void** FuelCell. increase(**double**)) that properly implements the CCC corresponding to speeding violation prevention.

3.7 Suggestion Sorting

Shadows suggested for incorporation are presented to the developer in descending order of the degree of *confidence*

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

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

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

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

Fig. 8. Pattern attribute equations.

we have in the shadow being applicable to a revised version of the input PCE. The confidence value (real in $[0, 1]$) paired with each suggestion is inherited from the pattern that 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 is performed on three different dimensions, as depicted by the equations listed in Fig. 8, and referred to as pattern *attributes*.

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 Section 3.6. Then, we define the err_α rate attribute, (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 by the pattern. Note that CG^+ refers to the graph computed from the code in which the pattern was constructed (original, unrevised program). Furthermore, $|\text{PCE}|$ is the number of shadows selected by PCE.

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 the region marked α in Fig. 9. 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 among shadows captured by the PCE; we would expect *future* shadows to exhibit *similar* characteristics, a claim that is validated by our experiment reported in Section 4.4. If a pattern matches no shadows, its err_α rate is 0. For example, applying the pattern $?^* \xrightarrow{cm} ? \xrightarrow{gf} \text{overallSpeed}$ to the original base code version in Fig. 1 would produce three shadows corresponding to the execution of methods DieselEngine.increase(Fuel), ElectricMotor.increase(Current), and Dashboard.update() (due to the pattern matching the path $\text{update}() \xrightarrow{cm} \text{getOverallSpeed}() \xrightarrow{gf} \text{overallSpeed}$). Thus, the err_α rate for this pattern w.r.t. the PCE found on line 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 $\frac{1}{3}$.

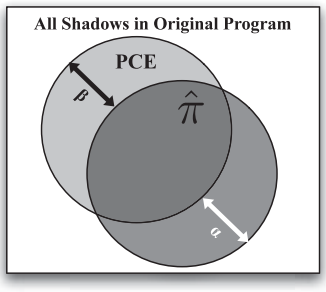

Fig. 9. Comparing a PCE with a pattern $\hat{\pi}$ in the original program.

The err_β rate attribute, (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 solely by the given PCE. The difference between the err_α and err_β rates is subtle but important; 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. 9 by the region marked β). 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 PCE, *regardless* of whether or not those properties are common to the captured shadows. 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 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 (3). Note that an empty pattern has no concrete elements; thus such a pattern has 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 is typically less likely to hone in on shadows that are *only* selected by the given PCE. Conversely, a pattern that is less abstract 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 (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 FuelCell.increase(**double**) produced when applying the pattern to the new version of the base code (cf., Fig. 3).

4 Experimental Evaluation

4.1 Implementation

We implemented our algorithm as a plug-in, called REJUVENATE POINTCUT (http://code.google.com/p/rejuvenate-pc), 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 (http://cs.mcgill.ca/~swevo/jayfx) fact extractor, which we extended for use with Java 5 and AspectJ. JayFX generates "facts," using class hierarchical analysis (CHA) [8] pertaining to structural properties and relationships, e.g., field accesses, method calls. Source code and transitively referenced libraries (possibly in binary format) are analyzed during graph building. The AJDT compiler (http://eclipse.org/ajdt) 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 shadows. Both pattern extraction and pattern-path matching were implemented via the Drools (http://jboss.org/drools) rules engine, which uses a modified RETE algorithm [11]. The Drools framework provides a natural query language and an efficient solution to the many-to-many matching problem. Pattern descriptions were persisted as XML files using JDOM (http://jdom.org).

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 runtime predicates, is always applied. If the tool encounters any intertype declarations or any other form of the static crosscutting, the associated PCE is still processed, but these constructs are not factored into the analysis. That is, any program element introduced to the base code via an ITD, as well as any program element relation induced by static crosscutting, is not represented in the extended concern graph. Thus, there may be shadows related to program elements introduced by ITDs that will not be suggested by our tool. Moreover, there may be relations induced by static crosscuts between program elements that our tool does not use, which may reduce pattern precision. While it would be reasonably straightforward to implement this technology, the limitation did not seem to have a significant impact on the performance of our tool, as the following sections demonstrate. Also, there is evidence that static crosscutting is not prevalent in AspectJ programs [2, Section 4].

4.2 Study Configuration

Our evaluation was conducted in two phases. For both phases, the maximum analysis depth parameter k was set at 2. Although setting k to be less than 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. For example, consider the following PCE taken from [26] that is intended to select all join points corresponding to JDBC (http://java.sun.com/jdbc) connection creations calls originating from mypackage:

```
pointcut connectionCreation(String url,
    String username, String password)
      : call(public static Connection
        DriverManager.getConnection(String,
          String, String))
        && args(url,username, password)
        && within(mypackage.*);
```

This PCE is too specific since there are two additional getConnection methods in the DriverManager interface that can be used to create database connections [43]. Suppose that client code is added to the system that calls these methods instead. Since join points corresponding to these new method calls would not be captured by the above PCE, it would be helpful if our approach suggested them upon rejuvenation. To do so, patterns would need to be constructed that effectively capture the structural relationships exhibited by program elements related to the currently selected shadow. It is conceivable that methods responsible for creating database connections call a common method for establishing a network connection. However, a pattern of length 1 (having a single edge) would be insufficient to capture such a relationship since the pattern must incorporate elements from the client code, the methods responsible for creating the database connection, and the method responsible for creating a network connection. A pattern of length 2, on the other hand, would suffice. CCCs that apply to methods that delegate tasks to intermediate methods are common in OOP. Thus, in general, we deemed it necessary to drive the analysis reasonably deep through these layers, which, for example, corresponds to analyzing longer method call chains. Setting k greater than 2 may result in effective rejuvenation for a wider variety of situations, but

TABLE 2
Phase I: Correlation Analysis Experiment Results

subject	LOC	class.	PCE	shad.	patt.	err_α	err_β	t (s)
AJHotDraw	21750	298	32	90	3362	0.32	0.06	73.93
Ants	1572	33	22	297	1254	0.15	0.23	9.06
Bean	121	2	2	4	16	0.24	0.23	1.75
Cactus	7573	93	4	222	2151	0.21	0.52	4.36
Contract4J	10722	199	15	350	1809	0.26	0.44	59.18
DCM	1680	29	8	343	2472	0.15	0.45	2.16
Figure	94	5	1	6	22	0.11	0.45	2.01
Glassbox	25940	430	55	208	2620	0.1	0.29	120.65
HealthWatcher	5716	76	27	122	1004	0.21	0.16	7.85
LawOfDemeter	1586	29	5	164	540	0.15	0.41	9.49
MobileMedia	3806	52	25	25	775	0.23	0.00	3.67
MySQL[a]	44016	187	2	3016	17564	0.12	0.58	336.29
NullCheck	1474	27	1	112	92	0.17	0.55	104.66
N-Version	552	15	4	9	80	0.19	0.24	0.51
Quicksort	73	3	4	7	56	0.19	0.15	2.78
RacerAJ	576	13	4	9	15	0.23	0.09	1.82
RecoveryCache	222	3	4	14	72	0.11	0.21	1.93
Spacewar	1415	21	9	58	225	0.15	0.22	17.09
StarJ-Pool	38218	511	1	3	67	0.25	0.00	29.78
Telecom	277	10	4	5	32	0.21	0.02	3.21
Tetris	1043	8	18	27	498	0.16	0.01	6.46
TollSystem	5195	88	35	85	1677	0.26	0.06	9.47
Tracing	366	5	16	132	676	0.17	0.4	2.32
Totals:	173987	2137	298	5308	37079	0.18[b]	0.16[c]	810.43

Legend	
LOC	Total number of non-blank, non-commented lines of code.
class.	Total number of class files after compilation.
PCE	Total number of pointcuts analyzed.
shad.	Total number of selected shadows.
patt.	Total number of patterns extracted by our tool.
err_α	Average pattern err_α rate.
err_β	Average pattern err_β rate.
t	Total pattern extraction time in seconds.

a. MySQL Connector/J
b. Average rate weighted by number of patterns.
c. Average rate weighted by number of PCEs.

our experiments described in the forthcoming sections suggest that it would likely result in a large runtime overhead with little gain in precision and recall. We discuss this issue further in Section 6.

In the first phase, 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 nonblank, noncommented lines of code (LOC) and domain. LOC was counted using the Eclipse Metrics tool found at http://metrics.sourceforge.net. Complete source code and descriptions of as well as references to the studied subjects can be found on our website, http://sites.google.com/site/pointcutrejuvenation. The authors were not involved in the development of any of the subject applications. To ensure that a certain level of quality was maintained, we purposefully selected subjects that have been used previously in the literature, including empirical studies. This ensures that the subjects have achieved a particular level of acceptance within the community.

Table 2 lists the subjects along with associated LOC, which excludes code contained within aspect files, (column *LOC*), ranging from 73 for Quicksort to 44K for MySQL Connector/J, number of class files after compilation (column *class.*), PCEs (column *PCE*) analyzed, which includes only PCEs bound to advice bodies, total selected shadows (column *shad.*), and patterns (column *patt.*) extracted (averaging 6.99 per shadow). For each subject, the pattern generation was repeated three times, with the results of each run averaged, using a 2.83 GHz Intel machine. The JVM heap size was set to 5 GB. Column t depicts the running time in seconds, which excludes intermediate representation (ASTs) construction time. The average was 4.66 seconds per KLOC, 0.15 seconds per shadow and 2.72 seconds per PCE. This indicates that the time required to generate our patterns is practical for even large applications. The remaining columns will be discussed in Section 4.3.

In Phase II, we demonstrate the usefulness of our technique in a real-world setting by rejuvenating PCEs in multiple versions of three of the aforementioned subjects. As this task was rather involved, we chose a proper subset of the subjects listed in Table 2 that were ripe for the analysis in a number of ways. These subjects, listed in Table 3, were comprised of a series of discrete 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 versions.

We briefly introduce the subjects here; more information pertaining to the subjects can be found on our website mentioned in Section 4.2. HealthWatcher is a web-based application that provides various medical-related support

TABLE 3
Phase II: Rejuvenation Experiment Results

subject	vers.	PCE	targ.	sugg.	rec.	$\sigma_{\mathbf{rec.}}$	$\uparrow_{TP}$	$\sigma_{\uparrow_{TP}}$	t (s)
Contract4J	5	13	317	4542	0.81	0.35	10.40	15.41	365.47
HealthWatcher	8	6	30	536	1.00	0.00	5.72	7.19	55.59
MobilePhoto	7	30	33	1714	0.97	0.18	0.83	3.64	115.31
Totals:	20	49	380	6792	0.93[a]	0.24[a]	3.97[a]	9.51[a]	536.37

a. Arithmetic mean

vers. is the number of versions analyzed, PCE is the number of pointcuts analyzed, targ. is the number of shadows in the target region, sugg. is the total number of suggested shadows, rec. is the average recall, σ_{rec} is the corresponding standard deviation, $\uparrow_{TP}$ is the average number of suggestions appearing before true positives, $\sigma\uparrow_{TP}$ is the corresponding standard deviation, and t is the total rejuvenation time in seconds

to patients. MobileMedia is a software product line consisting of applications that manipulate photo, music, and video on mobile devices. Last, Contract4J is a framework that facilitates Design by Contract (DbC) [30] style programming in Java (version 5 and later).

For our approach to be successfully evaluated, a complete set of changes was 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 as units of evolution. Moreover, we were solely interested in rejuvenating PCEs between versions that exhibited nontrivial (defined next) modifications.

We define the following conditions for PCEs regarding subsequent versions, which ensures that the performance of our tool is evaluated only in situations where the PCE recovery due to modifications to the base code is nontrivial. We say that a PCE contained in a version A *evolved* between a version B iff

1. the textual representation of the PCE in A differs from the textual representation of the PCE in B,
2. the shadows selected by the PCE in A differ from the shadows selected by the PCE in B, and
3. the shadows selected by the PCE in B differ from the shadows selected by the old representation of the PCE in B.

Criterion 1 asserts that a developer rewrote the PCE between the two versions, i.e., they textually differ. Criterion 2 excludes the situation where the developer unnecessarily rewrote the PCE between versions, i.e., the situation where two expressions capture the same exact shadows. Last, criterion 3 excludes the situation where the PCE remained robust between versions. As such, we 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 join points. Column *PCE*, Table 3 shows the number of PCEs across versions that met these criteria and were consequently selected to be rejuvenated by our tool. Column t represents the total rejuvenation time in seconds, which averaged 10.95 seconds per PCE. This indicates that our tool is practical to use, especially since users will most likely rejuvenate their pointcuts between releases of their software. We discuss ways to possibly reduce rejuvenation time in Section 6. The remaining columns are discussed in Section 4.4.

4.3 Phase I: Correlation Analysis Results

In Phase I, we assess the amount of unique structural commonality typically portrayed by shadows selected by a single PCE by studying attributes (cf., Fig. 8) of patterns extracted from a single version of the subjects listed in Table 2. Recall that a pattern with a low err_α, cf., (1), is one that expresses unique structural commonalities between shadows selected by the PCE from which it was extracted. 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 among shadows selected by the PCE that are *not* exhibited by other shadows. Recall from Section 3 that our definition of *shadow* is such that a shadow corresponds to a join point that may or may not be currently under the influence of advice. Column err_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, cf., (2), rate (column err_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

In Phase II, we assess the accuracy of our technique to mirror human-produced results by rejuvenating PCEs in multiple versions of the subjects listed in Table 3. We then evaluate 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 version. We are especially interested in exploring our tool's performance in precisely suggesting 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. The total number of these targeted shadows across all rejuvenations is listed by column *targ.*, Table 3.

4.4.1 Quantitative Analysis

As success metrics, we defined a *promising* rejuvenation to be one where our tool suggests the majority of targeted shadows, i.e., a high recall. Moreover, as suggestions are ranked by confidence (cf., Section 3.7), the traditional notion of precision (for unranked results) does not apply to our situation [28]. Instead, we defined a *precise* rejuvenation to be one where targeted shadows appeared near the top of the list of suggestions. In other words, the closer the true positives appear near the top of the list, the more effective we deem our tool would have been in these situations.

Column *rec.*, Table 3 shows the average recall at which our tool was able to suggest targeted shadows, while column $\sigma_{rec.}$ shows the corresponding standard deviation. The average recall across all subjects was found to be 0.90, which is normalized using a standard error of 0.03, with a standard deviation of 0.24. This indicates that, on average, our tool suggested 90 percent of targeted shadows with a standard deviation of 24 percent, demonstrating that our tool typically resulted in promising rejuvenation. In a real-world situation, however, the developer would be left to manually discover the remaining shadows (10 percent on

average) that our tool did not identify. In the worst case, this activity would require a whole program analysis. Thus, for such situations where our tool does not find all shadows that must be incorporated into updated versions of pointcuts, the usefulness of our tool is limited.

While both HealthWatcher and MobileMedia had similar recall values, the average recall for ContractJ was 0.81, which was distinctly lower. We conjecture several reasons for this difference. First, unlike the other subjects, Contract4J is a framework; thus, client code was not analyzed. Analyzing client code along with the framework could potentially result in higher performing patterns as more structural commonality may have existed between the framework and client code. Second, Contract4J makes heavy use of annotation types in defining PCEs, which is not typical of AO programs. In using annotations, locations in the base code where advice should apply are "marked." This is likely to result in program elements corresponding to selected join points that do not portray widespread structural commonality. This fact was verified in the first phase of our experiment (Section 4.3) when we found that the patterns produced from Contract4J had relatively high err_β (0.44 on average) rates, especially in comparison to HealthWatcher (0.16) and MobileMedia (0.00).

Column $\uparrow_{TP}$ portrays the average number of suggestions that appeared before true positives in the ordered list of suggested shadows, while column $\sigma_{\uparrow_{TP}}$ portrays the corresponding standard deviation. As indicated by Table 3, the $\uparrow_{TP}$ value across all subjects averaged ~ 4, which is normalized using a standard error of 1.3, with a standard deviation of 9.51. This corresponds to the average number of suggested shadows the developer would have had to search through prior to discovering a true positive. Our results show that these target shadows appeared, on average, significantly close to the top of the list of suggestions, which would have allowed developers to easily identify target shadows.

Note that the results of each of the subjects vary in this category. The sample size of HealthWatcher (6 PCEs) compared to MobileMedia (30 PCEs) was too small to draw any significant conclusions as to why the $\uparrow_{TP}$ values for these subjects were different. However, notice that our tool did not perform as well when applied to the Contract4J subject once again. We found that the difference is due to the fact that Contract4J's PCEs contained many dynamic conditions, especially in comparison with the other two subjects. This use of dynamic PCEs naturally results in less accurate patterns due to the conservative nature of our algorithm. Since substantial use of dynamic PCEs is not typical, as previously discussed in Section 3.4, the results indicate that the performance of our tool would be precise for many AO programs.

4.4.2 Qualitative Analysis

We identify 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 $\mathrm{call}(*\mathrm{HttpSession} + .\mathrm{putValue}(\mathrm{String}, \mathrm{Subject}))$ was affected by a modification to the base code that involved introducing the *Adapter* design pattern [12]. 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 refactoring. For example, one modification introduced the *Command* design pattern [12], which required relocating the implementations of several Servlets to a series of Command classes. This 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. Therefore, 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 rare. Furthermore, having only a minimal set of patterns generated for these PCEs, we questioned their usefulness in the cases where 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 may have eluded a developer when manually updating PCEs.

4.5 Threats to Validity

Several possible threats can undermine the aforementioned evaluation results. We explain how we have minimized their effects. Recall that Phase II aimed to assess the ability of our approach to mimic human-produced results in a real-world setting. First, the subjects selected for our study may not be representative of the majority of AO programs, thus hindering the usefulness evaluation. We chose subjects that were publicly available open source projects, where a number of developers contributed to the source code. In addition, we chose mature projects so that ample time has been allotted to accumulate diverse coding styles and maintenance changes. This assures that subject pool was adequate in representing real-world AO projects.

Second, our choice for the delta in which our approach was applied to the base code may not have accurately coincided with when developers actually recovered their PCEs manually. For instance, they may have updated their PCEs to reflect changes in the base code numerous times prior to a release. However, we estimated that the upper bound on when this activity would take place is immediately prior to milestone releases, i.e., a new version would

not be publicly released until PCEs were verified to correctly capture all of and only intended join points. As such, we chose major release points for our delta, which practically represents a subset of when PCEs would be recovered manually. Moreover, we are unaware of situations that occurred during the development of our subjects where the developer evolved the base code but neglected to update pointcuts. Since our delta was taken at major release points, it was unlikely for such a situation to occur at this level of granularity. However, it is possible that this situation occurred in between these points.

Last, in Section 4.2, we expressed that in Phase II we were solely interested in rejuvenating PCEs between versions that exhibited nontrivial modifications. In this way, we assessed the usefulness of our tool only in situations where it was needed as to avoid possibly overly positive results in situations where it was not needed. However, we have no way of telling if the suggestions made by our tool when used in an unneeded situation would deter the developer from correctly updating PCEs. By design, we have excluded such scenarios from our analysis and thus do not have data pertaining to the behavior of the tool when used in these situations.

5 Related Work

5.1 Concern Traceability

The closest work resembling (and inspiring) ours involves tracking [34] and managing [7] 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. First, Dagenais et al. [7] derive expressive intensional patterns from enumeration-like extensional descriptions of where concerns apply in source code and proceed to compare the performance between the two. Our patterns are also intensional descriptions but derived from *other* intensional descriptions, namely, PCEs. Also, patterns produced by our approach have been made to compete with the expressiveness inherent to PCEs, which deal specifically with CCCs. For example, our confidence evaluation is obtained using three dimensions of analysis (cf., Fig. 8). Recall from Section 4.2 that, due to the nature of CCCs, our graph-based approach features a general analysis depth parameter. This parameter, along with the algorithmic considerations taken, allows us to derive patterns of a specified length. Thus, concepts pertaining to algorithm development are treated more fully in this work. Last, we present a thorough empirical evaluation of our technique applied to evolving AO software.

5.2 Aspects and Refactoring

Wloka et al. [42] present a technique for automatically updating PCEs upon various refactorings of the base code. The associated tool updates PCEs only when predefined refactorings are invoked, whereas our tool deals with general base-code modifications. Moreover, 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.

Anbalagan and Xie [1] present an approach that 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 Aspect-Oriented Software Development

Several techniques (e.g., [32], [39]) aim to automate AOP-based development. However, they analyze *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, our approach *infers* shadows that are likely to belong in a new version of the PCE based upon those changes. Automated tools, such as AJDT and PointcutDoctor [43], 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 Ye and De Volder [43] 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 sufficiently expressive to represent the developer's true intentions in capturing join points corresponding to a PCE [27], these difficulties being rooted at the inherent *fragility* of typical PCE languages [25]. Several approaches (e.g., [9], [18], [24], [31], [38]) attempt to add expressiveness to help combat this problem by altering or abstracting the underlying join point model. Others (e.g., [5], [15]) 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, our tool makes suggestions based off this rich model while affording the developer the luxury of using a familiar AO language. Yet others (e.g., [17], [33]) propose new, hybrid languages that feature facets from both paradigms. Thus, these languages would not be considered completely AO in a traditional sense [10].

6 Conclusion and Future Work

We have overviewed an approach that 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, where the maximum structural commonality analysis

depth set at two, were presented, indicating that our approach would be useful in rejuvenating pointcuts in real-world situations.

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 [42]. Moreover, we plan to incorporate techniques introduced by Anbalagan and Xie [1] to perform compact PCE representation rewriting. This approach takes as input a set of shadows and uses join point clustering and string analysis of program element names to produce a compact PCE, making it an appropriate approach to follow ours in a tool chain. In addition, a program element tracing mechanism, e.g., Java Annotations, may be useful in pinpointing PCE declarations across subsequent software versions.

Evaluating tradeoffs between performance (in terms of pattern accuracy and running time) and maximum analysis depth to more thoroughly evaluate our approach is an interesting area of future work. In particular, it would be interesting to find a saturation point where increasing the maximum analysis depth parameter does not improve precision and recall. Furthermore, it would be helpful to discover an optimal parameter value that has a desirable tradeoff between performance and accuracy. Exploring graph reduction techniques (e.g., those employed by Robillard and Murphy [35]) and leveraging the abc compiler [3] to possibly reduce rejuvenation time may be helpful in this situation.

Additional future work may involve investigating the existence of other program element relations, e.g., $HandlesException$, that may contribute to our results, and subsequently incorporating these relations in the extended concern graph. Pointcuts selecting *handler*-join points would then be associated these relations.

Potential future work also entails incorporating aspect types and corresponding relations into the construction of the graph, as well as intertype declarations. This process would be well facilitated by the new Java Development Tools (JDT) weaving feature introduced in AJDT 1.6.2. The JDT weaving feature allows deeper integration with Eclipse, especially with handling intertype declarations [41]. In adding aspect elements, advice can be represented as a new kind of vertex. Then, advice vertices can be connected to method (or advice) vertices via an edge representing an "advising" relation if the PCE bound to the advice captures join points within the method's (or advice's) body. Also, in the case of **around** advice, calls to **proceed** can be added to the graph by considering the join point shadows captured by the bound PCE. If the join point shadow is a method execution, then a $CallsMethod$ edge can connect the vertex representing the advice to the vertex representing the method. If the join point shadow is a method call, on the other hand, the procedure would be a bit more complex due to inheritance considerations. Particularly, a $CallsMethod$ edge would need to connect the vertex representing the advice to vertices representing methods that are in the class hierarchy (since CHA is being used) of the method being called at the call-join point. Adding such relations may help uncover new join points in advice bodies that should be included into existing PCEs.

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

1: $\hat{\Pi} \leftarrow \varnothing$ {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.}
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.}
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.}
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.}
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.}
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.}
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.}
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.}
20: $\hat{\Pi} \leftarrow \hat{\Pi} \cup \{\hat{\pi}\}$ {Add the completed pattern to the return set.}
21: $\hat{\pi} \leftarrow \langle\rangle$ {Reset $\hat{\pi}$.}
22: **else** {Otherwise, 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 vertex.}
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 return set along with the last completed pattern.}

Fig. 10. Vertex-based pattern extraction algorithm.

A more accurate assessment of the dynamic applicability of advice may be an interesting avenue, possibly using dynamic traces in the analysis. Dynamic analysis, as well as static analyses such as Rapid Type Analysis (RTA) [4], may also be valuable in more accurately estimating the truth values associated with relations.

In Phase II of our experiment, we assessed the level of automation achievable by our approach by measuring its ability to mimic human-produced results. Our approach, however, has the potential to help developers correctly specify PCEs and, perhaps, prevent bugs. In the future, we intend to explore the ability of our approach to help prevent bugs that are caused by PCE misspecification.

Appendix

Pattern Extraction Algorithms

We detail the vertex-based and edge-based pattern extraction algorithms introduced in Section 3.5. Note that these algorithms are written in an imperative style and are conceptual, i.e., their purpose is to convey the algorithm more precisely. As mentioned in Section 4.1, the pattern extraction process was actually implemented via the Drools rules engine. Drools rules files were created based on the

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

1: $\hat{\Pi} \leftarrow \varnothing$ {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.}
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.}
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.}
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.}
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 return set.}
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.}
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.}
20: **end if**
21: **end for**
22: **return** $\hat{\Pi} \cup \{\hat{\pi}\}$ {Return the return set along with the last completed pattern.}

Fig. 11. Edge-based pattern path extraction algorithm.

following algorithms. These files are used as input to the rules engine in our implementation, thus facilitating a declarative way of expressing how patterns are extracted from paths. The rules engine decides on the specifics of how the patterns are actually extracted.

The vertex-based pattern extraction algorithm is depicted in Fig. 10. Text appearing in the figure between $\{\ldots\}$ offers descriptions of each of the algorithm's steps. For reference, the helper functions $s : E \rightarrow V$ and $t : E \rightarrow V$ map an edge to its constituent source and target vertices, respectively.

The edge-based pattern extraction algorithm is depicted in Fig. 11. Here, parameter e represents the enabled arc to which to base the derived patterns. An enabled edge wild card is denoted by raising a pair of vertices to the enabled wild card symbol $?^*$. Again, text appearing between $\{\ldots\}$ offers descriptions of each of the algorithm's steps.

ACKNOWLEDGMENTS

This work was supported in part by European Commission grants IST-33710 (AMPLE) and IST-2-004349 (AOSD-Europe). The authors would like to thank Barthelemy Dagenais, Tao Xie, Alexander Egyed, Martin Robillard, Alfred V. Aho, Marc Eaddy, Linton Ye, and David Chiu for their answers to many technical and research-related questions and for referring them to related work. They would also like to thank the anonymous reviewers for their extremely useful comments and suggestions.

REFERENCES

[1] P. Anbalagan and T. Xie, "Automated Inference of Pointcuts in Aspect-Oriented Refactoring," *Proc. Int'l Conf. Software Eng.*, pp. 127-136, 2007.

[2] S. Apel, "How AspectJ Is Used: An Analysis of Eleven AspectJ Programs," *J. Object Technology*, vol. 9, no. 1, pp. 117-142, Jan. 2010.

[3] P. Avgustinov, A. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble, "ABC: An Extensible AspectJ Compiler," *Trans. Aspect-Oriented Software Development I*, pp. 293-334, 2006.

[4] D.F. Bacon and P.F. Sweeney, "Fast Static Analysis of C++ Virtual Function Calls," *Proc. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications*, pp. 324-341, 1996.

[5] M. Braem, K. Gybels, A. Kellens, and W. Vanderperren, "Automated Pattern-Based Pointcut Generation," *Proc. Int'l Symp. Software Composition*, pp. 66-81, Mar. 2006.

[6] W. Cazzola, S. Pini, and M. Ancona, "Design-Based Pointcuts Robustness against Software Evolution," *Proc. Workshop Reflection, AOP, and Meta-Data for Software Evolution*, W. Cazzola, S. Chiba, Y. Coady, and G. Saake, eds., Fakultät für Informatik, Universität Magdeburg, pp. 35-45, July 2006.

[7] B. Dagenais, S. Breu, F.W. Warr, and M.P. Robillard, "Inferring Structural Patterns for Concern Traceability in Evolving Software," *Proc. IEEE/ACM Int'l Conf. Automated Software Eng.*, pp. 254-263, Nov. 2007.

[8] J. Dean, D. Grove, and C. Chambers, "Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis," *Proc. European Conf. Object-Oriented Programming*, pp. 77-101, Aug. 1995.

[9] M. Eichberg, M. Mezini, and K. Ostermann, "Pointcuts as Functional Queries," *Proc. Programming Languages and Systems: Second Asian Symp.*, pp. 366-381, Nov. 2004.

[10] R. Filman and D. Friedman, "Aspect-Oriented Programming Is Quantification and Obliviousness," *Proc. Workshop Advanced Separation of Concerns*, Oct. 2000.

[11] C.L. Forgy, "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem," *Artificial Intelligence*, vol. 19, pp. 324-341, 1982.

[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Mar. 1995.

[13] W.G. Griswold, K. Sullivan, Y. Song, M. Shonle, N. Tewari, Y. Cai, and H. Rajan, "Modular Software Design with Crosscutting Interfaces," *IEEE Software*, vol. 23, no. 1, pp. 51-60, Jan./Feb. 2006.

[14] S. Gudmundson and G. Kiczales, "Addressing Practical Software Development Issues in AspectJ with a Pointcut Interface," *Proc. Workshop Advanced Separation of Concerns*, Oct. 2001.

[15] K. Gybels and J. Brichau, "Arranging Language Features for More Robust Pattern-Based Crosscuts," *Proc. Int'l Conf. Aspect-Oriented Software Development*, pp. 60-69, Mar. 2003.

[16] E. Hilsdale and J. Hugunin, "Advice Weaving in AspectJ," *Proc. Int'l Conf. Aspect-Oriented Software Development*, pp. 26-35, Mar. 2004.

[17] K. Hoffman and P. Eugster, "Bridging Java and AspectJ through Explicit Join Points," *Proc. Int'l Symp. Principles and Practice of Programming in Java*, pp. 63-72, Sept. 2007.

[18] A. Kellens, K. Mens, J. Brichau, and K. Gybels, "Managing the Evolution of Aspect-Oriented Software with Model-Based Pointcuts," *Proc. European Conf. Object-Oriented Programming*, pp. 501-525, July 2006.

[19] R. Khatchadourian, J. Dovland, and N. Soundarajan, "Enforcing Behavioral Constraints in Evolving Aspect-Oriented Programs," *Proc. Seventh Workshop Foundations of Aspect-Oriented Languages*, pp. 19-28, Apr. 2008.

[20] R. Khatchadourian, P. Greenwood, A. Rashid, and G. Xu, "Pointcut Rejuvenation: Recovering Pointcut Expressions in Evolving Aspect-Oriented Software," *Proc. IEEE/ACM Int'l Conf. Automated Software Eng.*, pp. 575-579, Nov. 2009.

[21] R. Khatchadourian and A. Rashid, "Rejuvenate Pointcut: A Tool for Pointcut Expression Recovery in Evolving Aspect-Oriented Software," *Proc. IEEE Int'l Working Conf. Source Code Analysis and Manipulation*, pp. 261-262, Sept. 2008.

[22] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W.G. Griswold, "An Overview of AspectJ," *Proc. European Conf. Object-Oriented Programming*, pp. 327-354, June 2001.

[23] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin, "Aspect-Oriented Programming," *Proc. European Conf. Object-Oriented Programming*, pp. 220-242, June 1997.

[24] K. Klose and K. Ostermann, "Back to the Future: Pointcuts as Predicates over Traces," *Proc. Workshop Foundations of Aspect-Oriented Languages,* C. Clifton, R. Lämmel, and G.T. Leavens, eds., Mar. 2005.
[25] C. Koppen and M. Stoerzer, "PCDiff: Attacking the Fragile Pointcut Problem," *Proc. European Interactive Workshop Aspects in Software,* K. Gybels, S. Hanenberg, S. Herrmann, and J. Wloka, eds., Sept. 2004.
[26] R. Laddad, *AspectJ in Action.* Manning, 2003.
[27] M. Lippert and C. Lopes, "A Study on Exception Detection and Handling Using AOP," *Proc. Int'l Conf. Software Eng.,* pp. 418-427, May 2002.
[28] C.D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval.* Cambridge Univ. Press, 2008.
[29] H. Masuhara, G. Kiczales, and C. Dutchyn, "A Compilation and Optimization Model for Aspect-Oriented Programs," *Proc. Int'l Conf. Compiler Construction,* pp. 46-60, Apr. 2003.
[30] B. Meyer, "Applying 'Design by Contract'," *Computer,* vol. 25, no. 10, pp. 40-51, Oct. 1992.
[31] K. Ostermann, M. Mezini, and C. Bockisch, "Expressive Pointcuts for Increased Modularity," *Proc. European Conf. Object-Oriented Programming,* pp. 214-240, July 2005.
[32] M.A. Perez-Toledano, A. Navasa, J.M. Murillo, and C. Canal, "Titan: A Framework for Aspect-Oriented System Evolution," *Proc. Int'l Conf. Software Eng. Advances,* p. 4, 2007.
[33] H. Rajan and G. Leavens, "Ptolemy: A Language with Quantified, Typed Events," *Proc. European Conf. Object-Oriented Programming,* pp. 155-179, July 2008.
[34] M.P. Robillard, "Tracking Concerns in Evolving Source Code: An Empirical Study," *Proc. IEEE Int'l Conf. Software Maintenance,* pp. 479-482, Sept. 2006.
[35] M.P. Robillard and G.C. Murphy, "Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies," *Proc. Int'l Conf. Software Eng.,* pp. 406-416, May 2002.
[36] K. Sakurai and H. Masuhara, "Test-Based Pointcuts for Robust and Fine-Grained Join Point Specification," *Proc. Int'l Conf. Aspect-Oriented Software Development,* pp. 96-107, Mar. 2008.
[37] L.M. Seiter, "Role Annotations and Adaptive Aspect Frameworks," *Proc. Int'l Workshop Linking Aspect Technology and Evolution,* p. 3, 2007.
[38] J. Sillito, C. Dutchyn, A.D. Eisenberg, and K.D. Volder, "Use Case Level Pointcuts," *Proc. European Conf. Object-Oriented Programming,* pp. 246-268, June 2004.
[39] M. Stoerzer and J. Graf, "Using Pointcut Delta Analysis to Support Evolution of Aspect-Oriented Software," *Proc. IEEE Int'l Conf. Software Maintenance,* pp. 653-656, 2005.
[40] K. Sullivan, W.G. Griswold, Y. Song, Y. Cai, M. Shonle, N. Tewari, and H. Rajan, "Information Hiding Interfaces for Aspect-Oriented Design," *Proc. Int'l Symp. Foundations of Software Eng.,* pp. 166-175, Sept. 2005.
[41] The Eclipse Foundation, "JDT Weaving Features," Mar. 2009, http://wiki.eclipse.org/JDT_weaving_features, July 2010.
[42] J. Wloka, R. Hirschfeld, and J. Hänsel, "Tool-Supported Refactoring of Aspect-Oriented Programs," *Proc. Int'l Conf. Aspect-Oriented Software Development,* pp. 132-143, Mar. 2008.
[43] L. Ye and K.D. Volder, "Tool Support for Understanding and Diagnosing Pointcut Expressions," *Proc. Int'l Conf. Aspect-Oriented Software Development,* pp. 144-155, Mar. 2008.

Raffi Khatchadourian received the BS degree in computer science from Monmouth University, New Jersey, in 2004, and the MS degree in computer science from Ohio State University, Columbus, in 2010, where he is currently working toward the PhD degree in the Department of Computer Science and Engineering. His research interests include automated software evolution, automated refactoring, information retrieval in the context of engineering software, and formal reasoning of aspect-oriented software. He has been awarded several research fellowships, including those from the US National Science Foundation (NSF) and JSPS, that have allowed him to conduct research in various parts of the globe, including the United Kingdom and Japan. He is a member of the IEEE.

Phil Greenwood received the PhD degree in computer science from Lancaster University. He is a senior research associate in the Computing Department of Lancaster University, United Kingdom. His main research areas involve Aspect-Oriented Software Development (AOSD), middleware, and empirical software analysis techniques throughout the entire software development life cycle.

Awais Rashid is a professor of software engineering at Lancaster University, United Kingdom, where he leads research on advanced software modularity and composition mechanisms. His current research interests are in Aspect-Oriented Software Development (AOSD) and empirical evaluation of novel modularity and composition techniques. He leads the European Network of Excellence on Aspect-Oriented Software Development and has served as a program committee member for the AOSD conference for several years as well as its organizing chair in 2004 and program cochair 2006. He is also a member of the AOSD Conference Steering Committee 2006-2011 and its Executive Committee 2007-2011. He is a member of the IEEE.

Guoqing Xu is currently working toward the PhD degree in the Department of Computer Science and Engineering, Ohio State University, Columbus. His research interests are programming languages, compilers, and software systems. His dissertation work has developed static and dynamic program analysis techniques to help programmers find, remove, and prevent runtime bloat in large-scale object-oriented applications.