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

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

Phil Greenwood, Awais Rashid
Lancaster University
{greenwop,awais}@comp.lancs.ac.uk

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

***Abstract*—Pointcut fragility is a well-documented problem in Aspect-Oriented Programming; changes to the base-code can lead to join points incorrectly falling in or out of the scope of pointcuts. 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. We demonstrate the usefulness of our technique by rejuvenating pointcuts in multiple versions of several open-source AspectJ programs. The results show that our parameterized heuristic algorithm was able to automatically infer new join points in subsequent versions with an average recall of 0.93. Moreover, these join points appeared, on average, in the top 4th percentile of the suggestions, indicating that the results were precise.**

***Keywords*-Software development environments; Software maintenance; Software tools**

I. Introduction

Aspect-Oriented Programming (AOP) [1] has emerged to reduce the scattering and tangling of crosscutting concern (CCC) implementations. This is achieved through specifying that certain behavior (advice) should be composed at specific (join) points during the execution of the underlying program (base-code). Sets of join points are described by pointcuts (PCEs), which are predicate-like expressions over various characteristics of "events" that occur during the program's execution. In AspectJ [2], an extension of Java with support for aspects, for instance, such characteristics may include calls to certain methods, accesses to particular fields, and modifications to the run time stack.

Consider an example PCE **execution**($*$ m$*$(..)) 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 a particular version of the base-code, the above PCE selects the correct set of join points in which a CCC applies. As the software evolves, this set of join points may change as well. We say that a PCE is *robust* if it, in its unaltered form, is able to *continue* to capture the correct set of join points in future versions of the base-code. Thus, the PCE given above would be considered robust if the set of join points in which the CCC applies *always* corresponded to executions of methods whose name beings with *m*, taking any number and type of arguments, and so forth. However, with the requirements of typical software tending to change over time, the corresponding source code may undergo many alterations to accommodate such change, including the addition of *new* elements in which existing CCCs should also apply. Without *a priori* knowledge of future maintenance changes and additions, creating robust PCEs is a daunting task. As such, there may easily exist situations where the PCE itself must evolve *along* with the base-code; in this case, we say that the PCE is *fragile*. Hence, the *fragile pointcut problem* [3] manifests itself in such circumstances where join points incorrectly fall in or out of the scope of PCEs.

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 PCE used to construct them, and

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

*A portion of this work was administered during this author's visit to the Computing Department, Lancaster University, United Kingdom.

associating them with the PCE so that they may be applied to later base-code versions to offer suggestions of new join points that may require inclusion.

Our key contributions are as follows. First, we briefly overview 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 changes can be correctly applied by mechanically assisting the developer in maintaining PCEs. Next, we 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 analyze patterns using PCEs contained within *single* versions of 23 AspectJ programs. Lastly, to ensure the applicability and practicality of our approach, we implemented our algorithm as an Eclipse IDE (http://eclipse.org) plug-in and evaluated its usefulness by rejuvenating PCEs in multiple versions of 3 of the aforementioned programs.

II. Harnessing Commonality

Due to space limitations, we briefly overview 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; we invite to reader to refer to [4] for a thorough treatment. The algorithm works by discovering structural commonality between program elements corresponding to join points captured by a PCE in a particular software version. We capture such commonality by constructing patterns that abstractly describe the kinds of relations that the program elements have in common. The extracted patterns are then applied to later versions to offer suggestions of new join points that require inclusion as similar commonality may be exhibited in the future.

Our approach is divided into two conceptual phases: *analysis* and *rejuvenation*. The analysis phase is triggered upon modifications to or creation of PCEs. A graph is then computed which depicts structural relationships among program elements currently residing in the base-code. Next, patterns are derived from paths of the graph in which vertices and/or edges representing program elements and/or relationships are associated with join points selected by the PCE. This is done by matching the *shadows* of the join points captured by the input PCE. A join point shadow corresponds to a point in the program text where the compiler *may* actually combine the advice code with the base-code [5].

PCEs may be associated with certain dynamic conditions placed upon the behavior of the base-code. Presently, we conservatively treat these conditions as always being satisfied; future work may entail more accurately approximating truth values associated with dynamic conditions, perhaps through dynamic analysis and/or by leveraging techniques from [6]. Furthermore, the maximum length of the paths used to generate the patterns is an additional input to our algorithm. Although the parameter value is arbitrary, larger values result in greater analysis time. Consequently, the parameter value is trade-off to be considered in practice. We refer to this length as the *maximum analysis depth*.

Patterns are then themselves analyzed to evaluate the *confidence* (as 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 (see [4] for details). 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 for later use in the next phase.

We envision our approach to be most helpful in scenarios where the base-code is modified prior to updating PCEs to reflect those changes so that new join points are captured correctly. Thus, the rejuvenation phase is triggered before the developer manually alters the PCE so that automated assistance in performing the updates correctly can be provided. Now, patterns previously linked with the PCE are retrieved from storage and applied to a graph computed from the new base-code version to unveil the suggested join points. These join points are ones related to program elements that share structural similarities with program elements related to join points previously selected by the PCE in the original base-code version. A list of suggestion, confidence pairs is presented to the developer, with the confidence component being the confidence of the pattern producing the suggestion. Lastly, the list is sorted in decreasing order of confidence.

III. Experimental Evaluation

In this section, we overview an experimental study conducted to ascertain the usefulness of our rejuvenation approach in terms of its ability to accurately suggest shadows to be incorporated into a revised version of a PCE given evolutionary changes made to the base-code. Again, interested readers are invited to refer to [4] for further details.

A. 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. Program element structural relationships were resolved with the aid of the JayFX (http://tinyurl.com/mbwreh) fact extractor, which we extended for use with Java 1.5 and AspectJ. JayFX generates "facts, " using class hierarchical analysis (CHA) [8], pertaining to structural properties and relationships, e.g., field accesses, method calls. These facts were used to construct the graph mentioned in §I. 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 associate the graph with a PCE. For a given PCE, the AJDT compiler produces the Java program elements, e.g., method declarations, method calls, field sets, correlated with selected shadows. Both pattern extraction and pattern-path matching was implemented via the Drools (http://jboss.org/drools) rules engine, which makes use of a modified version of the RETE algorithm [9]. The Drools framework not only provides an efficient solution to the many-to-many matching problem the tool is faced with, as well as a natural query language, but also performance benefits such as the caching. Pattern descriptions were persisted as XML files, which were read and written to using the Java Domain Object Model (JDOM; http://jdom.org) translation framework.

B. Study Configuration

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

First, we aimed to show that the motivation behind our proposal is well founded by demonstrating that shadows selected by a single PCE typically portray a significant amount of unique structural commonality. We did so by generating and, subsequently, studying patterns from single versions of 23 publicly available AspectJ benchmarks, applications, and libraries (including open-source projects) of varying size, in terms of non-blank, non-commented lines of code (LOC), and domain. Complete source code and descriptions of the studied subjects can be found on our website http://tinyurl.com/6ewl2r. To ensure that a certain level of quality was maintained, we purposefully selected subjects that have been used previously in the literature including empirical studies [10]. This ensures that the subjects have achieved a particular level of acceptance.

Table I lists the subjects along with associated KLOC (column *KL*; excludes code contained within aspect files), ranging from 0.07 for Quicksort to 44.0 for MySQL Connector/J, PCEs (column *PC*) analyzed (includes only PCEs bound to advice bodies), total selected shadows (column *shd.*), and thousands of patterns (column *KP.*) extracted (averaging 6.99 per shadow) and thereby evaluated. For each subject, the pattern generation was repeated five times using a 2.16 GHz Intel Core 2 Duo machine with a maximum Java heap size of 1GB. Column *t* depicts the total running time (excludes AST construction) in seconds, which itself averaged 8.22 secs per KLOC and 4.80 secs per PCE, indicating that the time required to generate our patterns is practical even for large applications. The remaining columns will be discussed later in §III-C.

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

Table I
PHASE I: CORRELATION ANALYSIS EXPERIMENT RESULTS.

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

Table II
PHASE II: REJUVENATION EXPERIMENT RESULTS.

Our goal in the second phase of the experiment was to demonstrate the usefulness of our technique in a real-world setting. We did so by rejuvenating PCEs in multiple versions of 3 of the aforementioned subjects. As this task was rather involved, we chose a proper subset of the subjects listed in Table I that were ripe for the analysis in a number of ways. These subjects, listed in Table II, 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. 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 non-trivial modifications.

We defined the following conditions for PCEs regarding

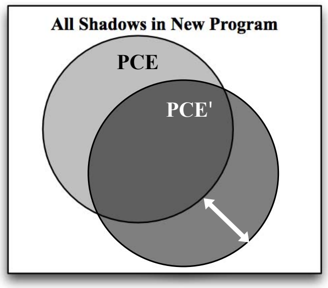

Figure 1. Comparing a PCE with its revision in the new program.

subsequent versions, which ensured that the performance of our tool was evaluated only in situations where the PCE recovery due to modifications to the base-code was non-trivial. We say that a PCE contained in a version A *evolved* between a version B iff

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

The last criterion asserts that the region designated by the light-shaded arrow in the Venn diagram depicted in Fig. 1, where the outer region symbolizes all shadows in B, *PCE* the shadows in B selected by the old representation, and *PCE*$'$ the shadows selected in B by the new representation, is non-empty. 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 *PC*, Table II shows the number of PCEs across versions which met this criteria and were, consequently, selected to be rejuvenated by our tool. Determining the region marked as *PCE* in Fig. 1 required carefully copying the original PCE to the subsequent version and binding it to an empty advice body. Column *t* designates the total rejuvenation time in secs. Although the average was ˜4 secs per KLOC, producing a single suggestion occupied 3.84 secs on average, which could result in a slow rejuvenation time for PCEs with large target regions. As the analysis is conservative in a number of ways (q.v. §II), future work may entail exploring graph reduction technique as in [11] or swapping the AJDT with [12] for a simpler intermediate representation to possibly reduce the rejuvenation time. The remaining columns are discussed in §III-D.

C. Phase I: Correlation Analysis Results

In first phase, we assessed the amount of unique structural commonality typically portrayed by shadows selected by a single PCE by studying attributes of patterns extracted from a single version of the subjects listed in Table I. A pattern with a low type I (false positive) error rate produced by applying the pattern to the base-code in which it was derived is one that expresses unique structural commonalities between shadows selected by the PCE in a particular version. 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 type I error rate is one that expresses common structural characteristics amongst shadows selected by the PCE that are *not* exhibited by other shadows. Column α depicts the average type I error rate for all patterns extracted from the associated subject. We found the average, weighted by the number of patterns extracted, the type I error 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 rate the type II (false negative) rate (column β) whose average, weighted by the number of PCEs analyzed, among all subjects was found to be 0.16. The combination of these two findings show that shadows selected by a single PCE typically display a significant amount of unique structural commonality.

D. Phase II: Expression Recovery Results

During the second phase, we assessed the accuracy of our technique by rejuvenating PCEs in multiple versions of the subjects listed in Table II. We then evaluated the relationship between the shadows that were suggested for inclusion by our tool and those that were actually included in (human) revised PCEs residing in a subsequent version. We were 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 version. These are exactly the shadows that the developer would have had to *manually* determine to be applicable to the PCE, which coincide with those that our tool could be most helpful in mechanically discovering. This "target" set of shadows is represented by the region surrounding the light-shaded arrow in Fig. 1. The total number of shadows occupying this regions across all rejuvenations is listed by column *trg.*, Table II.

1) Quantitative Analysis: As success metrics for evaluating our approach, we defined a *promising* rejuvenation to be one where our tool suggests the majority of shadows contained within the target region, i.e., a high recall. Moreover, as suggestions are ranked by confidence, the traditional notion of precision (for unranked results) does not apply to our situation [13]. Instead, we defined a *precise* rejuvenation to be one where targeted shadows appeared near the top of the list of suggestions after sorting by confidence.

Column *rec.*, Table II shows the average recall at which our tool was able to suggest targeted shadows. The average recall across all subjects was found to be 0.93, indicating that, on average, our tool suggested 93% of shadows that resided in this region. This demonstrates that our tool typically resulted in promising rejuvenation. Column *pr.*, on the other hand, portrays the average percentile rank, a means to divide an ordered list into sections, of targeted shadows. A low percentile rank indicates that the suggested shadow appears towards the top of (or first on) the list and vice versa. Our results show that targeted shadows, on average, appeared in the top 4th percentile of the list of suggested shadows produced by our tool. Since there is a direct correlation between the number of original shadows and the number of derived patterns (and, consequently, the number of suggested shadows), this would have allowed the developer to easily identify the target shadows. Therefore, the performance of our tool was exceptionally precise.

2) Qualitative Analysis: In this section, we identifying potential reasons for both accurate and inaccurate suggestions made by our tool. For succinctness, we draw examples from only the HealthWatcher subject. The major contributing factor that was found to cause patterns derived by our approach to be ineffective when applied to subsequent versions relates to modifications made to the base-code that involved removing program elements appearing in patterns. For example, the PCE **call**(∗ HttpSession+.putValue(String, Subject)) was affected by a modification to the base-code that involved introducing the *Adapter* design pattern [14]. Consequently, the HttpSession class was replaced, invalidating all patterns containing references to this class. Fortunately, however, our tool was able to compensate by producing other patterns that were effective in rejuvenating the aforementioned PCE.

Common base-code modifications involved structural refactoring. For example, one modification encompassed introducing the *Command* design pattern [14], which required relocating the implementations of several Servlets to a series of Command classes. This activity induced the need to rejuvenate several PCEs. As the modifications made to the base-code were minimal and purely structural, i.e., the method bodies remained intact, our patterns encouragingly but expectedly proved completely effective in this situation, suggesting only and all of the targeted shadows.

We found several PCEs in the subjects to be very specific, often selecting only a single join point. 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 that such change does occur. Despite this, we did find scenarios involving updates to these PCEs and, surprisingly, our patterns were able to produce accurate suggestions in these situations. One particular PCE that related to synchronization required rejuvenation due to *new* types introduced. An obscure pattern that centered upon references to an exception raised by classes that required the managed synchronization behavior caused shadows associated with the new types to be accurately suggested. This demonstrates a benefit of our approach in its ability to discover obscure structural characteristics that may have eluded a developer.

IV. 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 were presented, indicating that our approach is particularly usefulness in rejuvenating PCEs.

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 [15]. Moreover, we plan to incorporate techniques outlined in [16] 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.

References

[1] G. Kiczales *et al.*, "Aspect oriented programming," in *ECOOP*, 1997.
[2] ——, "An overview of aspectj," in *ECOOP*, 2001.
[3] C. Koppen and M. Stoerzer, "PCDiff: Attacking the fragile pointcut problem." in *Eur. Int. Workshop on Aspects in Software*, 2004.
[4] R. Khatchadourian *et al.*, "Pointcut rejuvenation: Recovering pointcut expressions in evolving aspect-oriented software," Lancaster University, UK, Tech. Rep. COMP-001-2008, Aug. 2008, (rev Mar. 2009).
[5] H. Masuhara, G. Kiczales, and C. Dutchyn, "A compilation and optimization model for aspect-oriented programs," in *CC*, 2003.
[6] B. Dufour *et al.*, "Measuring the dynamic behaviour of AspectJ programs," in *OOPSLA*, 2004.
[7] B. Dagenais *et al.*, "Inferring structural patterns for concern traceability in evolving software," in *ASE*, 2007.
[8] J. Dean, D. Grove, and C. Chambers, "Optimization of object-oriented programs using static class hierarchy ana." in *ECOOP*, 1995.
[9] C. L. Forgy, "Rete: a fast algorithm for the many pattern/many object pattern match problem," *Artificial Intelligence*, pp. 324–341, 1982.
[10] P. Greenwood *et al.*, "On the impact of aspectual decompositions on design stability: An empirical study," in *ECOOP*, 2007.
[11] M. P. Robillard and G. C. Murphy, "Concern graphs: finding and describing concerns using structural program depend." in *ICSE*, 2002.
[12] P. Avgustinov *et al.*, "abc : An extensible aspectj compiler," *Trans. Aspect-Oriented Softw. Dev.*, vol. 1, pp. 293–334, 2006.
[13] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge University Press, 2008.
[14] E. Gamma *et al.*, *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.
[15] J. Wloka, R. Hirschfeld, and J. Hänsel, "Tool-supported refactoring of aspect-oriented programs," in *AOSD*, 2008.
[16] P. Anbalagan and T. Xie, "Automated inference of pointcuts in aspect-oriented refactoring," in *ICSE*, 2007.