Techniques for Automated Software Evolution

Dissertation

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Raffi Khatchadourian
Graduate Program in Computer Science & Engineering

The Ohio State University

2011

Dissertation Committee:

Neelam Soundarajan, Advisor
Atanas Rountev
Paul A. G. Sivilotti
Paul Evans
ABSTRACT

For a variety of reasons, modern, non-trivial software systems must evolve to cope with change, including alterations in stakeholder requirements, environments in which the software is deployed, and dependent technologies, e.g., frameworks. Unfortunately, evolution and maintenance is an expensive, time-consuming, and error-prone task, especially when the system in question is large and complex. Typically, a change to a single program element requires changes to related, and often seemingly unrelated, elements scattered throughout the source code.

To address this problem, approaches have emerged to mechanically assist developers with a wide range of software evolution and maintenance tasks, including migrating code to a new framework version, translating existing code to a new platform, and restructuring code to mirror an improved design. This assistance is typically provided in the form of extensions (plug-ins) to integrated development environments (IDEs) that afford (semi-) automated aid in carrying out these tasks, thus easing the burden associated with evolution and maintenance. In some approaches, the corresponding plug-in keeps track of the elements relevant to the change being implemented, with the IDE displaying only those elements. Other approaches attempt to automatically restructure code to improve such features as type safety while preserving semantics.

Although existing approaches are useful in alleviating some of the burden associated with software evolution and maintenance, there are a number of situations where
developers are still required to complete evolution and maintenance tasks manually. These include but are not limited to upgrading legacy Java software to take advantage of many other available features of the modern Java language, replacing certain usages of Java collections with custom type hierarchies, and updating software composition specifications to cope with change. Automated approaches to assist developers with such cumbersome and error-prone tasks would be extremely useful in evolving and maintaining large, complex systems.

In this thesis, I explore and develop a number of new techniques that can be of great value to software developers in evolving code to accommodate change. The first of these is an automated refactoring which upgrades legacy Java code to use proper language enumeration (enum) types, a feature of the modern Java language. I have developed an approach that preserves semantics and that allows us to migrate legacy applications by automatically replacing a predominantly used pattern with suitable use of enums.

For the second technique, I explore and develop an automated approach to assist developers in maintaining pointcuts in evolving Aspect-Oriented (AO) programs. AO languages enable developers to better encapsulate crosscutting concern (CCC) implementations by allowing them to create an expression (a pointcut) which specifies well-defined points (join points) in a program’s execution where code corresponding to a CCC (an aspect) should apply. However, changes to the underlying program (base-code) may invalidate pointcuts, leaving developers to manually update pointcuts to capture the intended join points. I have developed an approach that mechanically aids developers in suitably updating pointcuts upon changes to the base-code by analyzing arbitrarily deep structural commonalities between program elements associated with
pointcuts in a particular software version. The extracted patterns are then applied
to later versions to suggest additional join points that may require inclusion.

The third technique I explore in this thesis pertains to reasoning about the behav-
ior of AO programs. As previously noted, AOP facilitates localized implementations
of CCCs by allowing developers to encapsulate code realizing a CCC that would oth-
erwise be scattered throughout many system modules and/or intertwined with code
realizing the primary functionality of a module. Therefore, AOP allows developers to
think about “one concern at a time.” However, reasoning about AOP presents unique
challenges, particularly as such programs evolve. I have developed an approach to
specify the behavior of aspects in such a way that they can be combined with the
behavior of the base-code to arrive at the overall system behavior without invalida-
tion. Although the technique is not currently automated and planned future work
does not involve fully automating this technique, I nevertheless explore ways to assist
developers by mechanizing portions of the associated reasoning activities.
To Aunt Mida and Uncle Tommy
ACKNOWLEDGMENTS

There have been many people that have helped, in one form or another, along the way in writing this thesis. I acknowledge a subset of them here, as it is inevitable that I inadvertently fail to mention everyone. For this, I apologize in advance.

I would like to thank my advisor, Dr. Neelam Soundarajan, for his support throughout my many years in graduate school. He has written numerous recommendation letters for my award applications, grant proposals, conference volunteering applications, and job applications. He has never denied a request for a letter and has been prompt in delivering them. Without this, I would not have been available to achieve and participate in many of the awards and activities listed on my curriculum vitae today.

Dr. Soundarajan seemed a bit hesitant to accept me as a student at first. After several months of inquiries and many trips to his office, I finally got him to agree. I never quite knew the reason for his hesitancy, but I suspect that is was mainly due to our difference in backgrounds. I came from a small liberal arts school with a solid yet industry focused Computer Science program. I had also spent some time as a software developer before entering my full-time graduate program at Ohio State. Thus, my background was very practical. Dr. Soundarajan, on the other hand, had a theoretical and mathematics-based background with respect to Software Engineering. This is a side of Software Engineering that I was not initially familiar with.
Despite our background differences, Dr. Soundarajan and I both benefited from our differences and found a way to compliment them to achieve a good mix of theory and practicality that the greater Software Engineering research community desired at the time.

I am very grateful that Dr. Soundarajan finally decided to accept me as a student. He, and many other members of the Software Engineering group at Ohio State University, taught me to see problems for a different perspective. I learned to think of programs more mathematically. In solving problems, I thought about assertions rather than just examples. I learned to think abstractly about problems, which helped immensely in my progress towards a PhD degree. Indeed, it is this particular skill that allowed me to learn one of the most valuable skills any graduate student can learn, that is, the ability to learn. I learned how to properly not understand difficult and complex concepts at first so that I may fully understand them later. This, along with the skill of effective and thorough explanation, are not easily achievable and are the greatest assets I have obtained during my time as a graduate student.

Equally critical to my success was my girlfriend, Kelly McMeans, whom stood by my side during almost my entire time as a graduate student. I am utterly grateful for her tremendous support throughout all these years, and without it, I would have not completed this thesis. She picked me up when I was down, and she stood by my side when I felt alone. There were many times when I felt that I had no more energy nor resources to continue, and she convinced me otherwise. She waited countless months for me during my many trips overseas and always welcomed me back with open arms.
Kelly and her family made me feel at home in a place that was hundreds of miles away from the only place I knew as home and that was very different in many ways. For this, I will be forever grateful.

I am also grateful to my cat, Fluffy, and my dog, Pudding. I am grateful to Fluffy for providing me company when I lived alone for almost two years in my one bedroom apartment in Worthington. She would always welcome me with a purr after returning from many long nights of hard work at the local coffee shops. I thank Pudding for stealing my heart at the pet store and for the opportunity for many long walks in the park we had together where I could easily gather my thoughts and relax.

My career as a graduate student has given me the opportunity to work with many great people both around the globe and close to home. I would like to thank my numerous co-authors and colleagues, namely, Dr. Jason Sawin, Dr. Atanas Rountev, Dr. Phil Greenwood, Dr. Awais Rashid, Gouging Xu, and Dr. Hidehiko Masuhara, for their invaluable advice and support. I am especially grateful to Dr. Rashid for allowing me to visit Lancaster University and for Dr. Masuhara for allowing me to visit the University of Tokyo. These are experiences that I will treasure forever.

I am grateful to my uncles, Robert and Arthur Papazian, for handing me down my very first computer. It was one of the very first Apple Macintosh systems and, despite it’s monochrome screen, lack of a hard drive, outdated 3.5” single-density floppy drive, slow processing, lack of software, and my many sleepless nights where I cried because I could not get my latest games to execute, it remains in my heart to this day. Through this machine, I realized my love for pushing buttons and watching things subsequently happen. I will never forget the day that I realized that a mysterious external box with a telephone jack that I had not tested for years allowed me to communicate with
other computers through the telephone network at a stunning 1200 bps. I was simply amazed that my friend, Adrian from middle school, could see what I was typing on my computer screen on his computer screen from across town.

I am grateful to Dr. Guirgis and Dr. Needleman from the Ohio State University Medical Center for working with me though some extremely difficult times during the writing of this thesis. Their advice and suggestions proved to be extremely valuable in its completion.

There are many teachers I have had throughout the years that have helped contribute to the completion of this document in both positive and negative ways. I would like to thank Williard Bastian from Monmouth University for his inspiration to become a Computer Scientist. He was tough on me at first but, in the end, I learned a great deal about Computer Science from him. I thank Ms. Sutter from John P. Stevens High School in Edison, NJ for letting me know that I will never be successful in college. I also thank Ms. Hopkins from Menlo Park Elementary school in Edison, NJ for staying the extra hours before and after class to help me with my math homework, particularly, with fraction calculations. Tae Kim from Edison, NJ also helped me in a similar regard during high school. I thank him for this and his delightful family who fed me delicious kimchee and bulgogi for dinner following our study sessions.

Additionally, I am grateful to Frank Tip, Barthelemey Dagenais, Tao Xie, Alexander Egyed, Martin Robillard, Alfred V. Aho, Marc Eaddy, Linton Ye, Amir Aryani, Sai Zhang, Yu Lin, Mathew Parkinson, and Gary T. Leavens for their answers to my many technical and research related questions and for referring me to related work.
I am also grateful to the numerous anonymous referees from various workshops and conferences for their extremely useful comments and suggestions.

Lastly, I would like to thank my aunt, Mariam Khatchadourian, and my uncle, Tom Frank, for their unselfish and extremely generous support during my undergraduate education. This document would have never existed if it was not for them. Thus, it is to them that I dedicate this thesis.
VITA

March 2010 .......................... M.S. Computer Science & Engineering, Ohio State University, Columbus, Ohio, USA

May 2004 ............................ B.S. Computer Science, Monmouth University, West Long Branch, New Jersey, USA

September 2005 – present ........ Graduate Research/Teaching Associate, Ohio State University, Columbus, Ohio, USA

June 2010 – September 2010 .......... Visiting Graduate Computer Science Researcher, University of Tokyo, Japan

January 2008 – September 2008 .... Visiting Graduate Student in Computing, Lancaster University, Lancaster, UK

February 4, 1979 ...................... Born – New York, New York, USA

PUBLICATIONS

Research Publications


**FIELDS OF STUDY**

Major Field: Computer Science and Engineering

Studies in:

<table>
<thead>
<tr>
<th>Field</th>
<th>Instructor</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software Engineering</td>
<td>Prof. Neelam Soundarajan</td>
</tr>
<tr>
<td>Distributed Systems</td>
<td>Prof. Paul A. G. Sivilotti</td>
</tr>
<tr>
<td>Mathematics</td>
<td>Prof. Timothy Carlson</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Dedication</td>
<td>v</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>vi</td>
</tr>
<tr>
<td>Vita</td>
<td>xi</td>
</tr>
<tr>
<td>List of Figures</td>
<td>xviii</td>
</tr>
<tr>
<td>List of Tables</td>
<td>xx</td>
</tr>
<tr>
<td>List of Algorithms</td>
<td>xxi</td>
</tr>
<tr>
<td>List of Listings</td>
<td>xxii</td>
</tr>
</tbody>
</table>

## Chapters:

1. Introduction .................................................. 1
   1.1 Automated Techniques for Software Evolution ........... 3
       1.1.1 Automated Refactoring of Legacy Java Software to Enumerated Types ............................................ 4
       1.1.2 Recovering Pointcut Expressions in Evolving Aspect-Oriented Software ...................................... 4
       1.1.3 Enforcing Behavioral Constraints in Evolving Aspect-Oriented Programs ..................................... 5
   1.2 Organization ............................................... 6
3.3.7 Suggestion Sorting

3.4 Experimental Evaluation

3.4.1 Implementation

3.4.2 Study Configuration

3.4.3 Phase I: Correlation Analysis Results

3.4.4 Phase II: Expression Recovery Results

3.4.5 Threats to Validity

3.5 Related Work

3.5.1 Concern Traceability

3.5.2 Aspects and Refactoring

3.5.3 Automated Aspect-Oriented Software Development

3.5.4 Pointcut Fragility

3.6 Future Work

3.6.1 Pointcut Expression Reconstruction

3.6.2 Performance

3.6.3 Additional Relations

3.6.4 Aspect Type and Inter-type Declaration Incorporation

3.6.5 Improving Analysis of Dynamic Pointcuts

3.6.6 Bug Detection

3.6.7 Leveraging Commonality from Historical Data

3.6.8 Framework Generalization

3.7 Conclusion

4. Enforcing Behavioral Constraints in Evolving Aspect-Oriented Programs

4.1 Introduction

4.1.1 Organization

4.2 Constraining and Enriching the Behavior of Programs

4.2.1 Constraining Behavior of Concurrent Programs

4.2.2 Constraining Behavior of Aspect-Oriented Programs

4.2.3 Deriving Effective Behavior

4.3 Specification and Verification

4.3.1 Specifications and Pointcuts

4.3.2 Join Point Traces

4.3.3 Inference Rules

4.4 Related Work

4.4.1 Aspect Restrictions

4.4.2 Aspect Classification

4.4.3 Modular Aspect Verification

4.4.4 Aspect Interfaces

4.4.5 Aspect Interference

4.4.6 Aspect-Oriented Software Design
4.5 Future Work .............................................. 164
4.6 Conclusion ................................................ 165

5. Conclusion .................................................. 166
  5.1 Contribution Summary ................................. 166
  5.2 Future Work Summary ................................. 168

Bibliography .................................................. 170
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Drive-by-wire application example</td>
<td>11</td>
</tr>
<tr>
<td>2.2</td>
<td>Contexts for a program entity α</td>
<td>24</td>
</tr>
<tr>
<td>2.3</td>
<td>Enumerizable ascender</td>
<td>26</td>
</tr>
<tr>
<td>2.4</td>
<td>Enumerizable descender</td>
<td>29</td>
</tr>
<tr>
<td>2.5</td>
<td>Extraction ascender</td>
<td>31</td>
</tr>
<tr>
<td>2.6</td>
<td>Extraction descender</td>
<td>34</td>
</tr>
<tr>
<td>2.7</td>
<td>Member constraints for transforming a group of candidate fields K</td>
<td>41</td>
</tr>
<tr>
<td>2.8</td>
<td>Initial enumeration sets</td>
<td>44</td>
</tr>
<tr>
<td>2.9</td>
<td>Enumeration sets produced by applying the enumerization algorithm</td>
<td>44</td>
</tr>
<tr>
<td>2.10</td>
<td>Screen shot of the enum refactoring wizard</td>
<td>48</td>
</tr>
<tr>
<td>2.11</td>
<td>Screen shot of the enum refactoring preview pane</td>
<td>49</td>
</tr>
<tr>
<td>3.1</td>
<td>Phase I: Pointcut analysis flow diagram</td>
<td>74</td>
</tr>
<tr>
<td>3.2</td>
<td>Phase II: Pointcut rejuvenation flow diagram</td>
<td>76</td>
</tr>
<tr>
<td>3.3</td>
<td>A graph subset computed from the example</td>
<td>79</td>
</tr>
<tr>
<td>3.4</td>
<td>Evolving the base code with a FuelCell class</td>
<td>90</td>
</tr>
</tbody>
</table>
3.5 Pattern attribute equations. ................................................. 92
3.6 Comparing a PCE with a pattern \( \pi \) in the original program. ....... 94
3.7 Comparing a PCE (PCE) with its revision (PCE') in the new program (B). ................................................................. 105
3.8 GAMBLE Phase I planned extensible architecture. ......................... 123
3.9 GAMBLE Phase II planned extensible architecture. ......................... 125
4.1 Composing parameterized specifications of base-code and aspects. .... 130
4.2 Composing parameterized specifications of base-code not influenced by advice. ........................................................... 131
4.3 Rule for method specification. ................................................. 151
4.4 Rule for method calls. ......................................................... 154
4.5 Rule for aspect application (simple version). ................................ 155
4.6 Rule for aspect application (revised version). ............................. 157
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Enumerization formalism notation</td>
<td>18</td>
</tr>
<tr>
<td>2.2 Enumerization experimental results</td>
<td>50</td>
</tr>
<tr>
<td>2.3 Contexts not considered valid for enumerated types</td>
<td>52</td>
</tr>
<tr>
<td>3.1 Analyzed program entity types and relations</td>
<td>81</td>
</tr>
<tr>
<td>3.2 Phase I: Correlation analysis experiment results</td>
<td>102</td>
</tr>
<tr>
<td>3.3 Legend for Table 3.2</td>
<td>103</td>
</tr>
<tr>
<td>3.4 Phase II: Rejuvenation experiment results</td>
<td>106</td>
</tr>
</tbody>
</table>
LIST OF ALGORITHMS

2.1 Top-level enumerization algorithm. ........................................... 21
2.2 Enumerization set building algorithm ....................................... 22
3.1 Vertex-based pattern extraction algorithm. ................................. 87
3.2 Edge-based pattern path extraction algorithm ......................... 89
LIST OF LISTINGS

2.1 Convert Constants to Enum refactoring demonstration example . . . . 47
3.1 Hybrid automobile programming example. . . . . . . . . . . . . . . . 70
3.2 Speeding prevention aspect. . . . . . . . . . . . . . . . . . . . . . . . . . 71
3.3 A new fuel cell class . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
3.4 PCE selecting database connection creation join points. . . . . . . . . 99
CHAPTER 1: Introduction

The software development life-cycle constitutes activities that involve the inception, fruition, validation, deployment, and maintenance of a particular software product. Such activities include stakeholder requirements analysis, domain analysis and design, implementation, testing, and sustenance. The focus of this thesis is the last listed activity, namely, software maintenance and evolution.

Once a software product has been initially deployed, often times, facets pertaining to the software undergo various changes prior to its retirement, i.e., the point at which the software is either no longer used supported by its developers. For example, portions of the stakeholder requirements may be changed, removed, or added. Depending on the size and complexity of the software, such a change may result in non-trivial alterations to a variety of artifacts that the software is comprised of, including design documents, user and programmer documentation, configuration files, database schemata, and, most importantly, source code. Consequently, several software development activities may need to be revisited by the development team to address requirement change in the software.

Stakeholder requirements changes are not the only reason for artifacts of a software system to be altered. Such alterations, after initial delivery, may be due to modifications to the deployment environment, migration to a new framework version that the software depends on, or restructuring to mirror an improved design (ISO/IEC 14764 2006). Unfortunately, evolution and maintenance is an expensive, time-consuming,
and error-prone task, especially when the system in question is large and complex. Typically, a change to a single program element can require changes to related, and, sometimes, seemingly unrelated, elements scattered throughout the source code.

To address such problems, approaches have emerged to mechanically assist developers with a wide range of software evolution and maintenance tasks, including migrating code to a new framework version (e.g., (Fuhrer et al., 2005; Kiežun et al., 2007) Tansey and Tilevich, 2008) von Dincklage and Diwan, 2004')) or to existing but more desirable frameworks (e.g., (Dig et al., 2009)), translating existing code to a new platform (e.g., (de Lucia et al., 1997) Kontogiannis et al., 1998 Zou and Kontogiannis, 2001)), and restructuring code to reflect a superior design philosophy (e.g., (Donovan et al., 2004; Kegel and Steimann, 2008)).

Such assistance is typically provided in the form of extensions (plug-ins) to integrated development environments (IDEs) that afford automated (or tool-supported) aid in carrying out these tasks, thus easing the burden associated with evolution and maintenance. In one approach, the corresponding plug-in keeps track of the elements relevant to the change being implemented, with the IDE displaying only those elements (Kersten and Murphy, 2006). Other approaches attempt to automatically restructure code to improve such features as type safety while preserving semantics, a process known as refactoring (Opdyke, 1992).

A comprehensive potpourri of automatic refactoring approaches exist; for example, Tip et al. (2004) automatically migrates legacy Java client code to take advantage of parameterized (generic) library classes, effectively inferring type parameters to library class instantiations and method invocations. Such an approach is helpful to developers as manually discovering the most optimal type parameter for generic
library class instances involves exploring deep library type hierarchies as well as all occurrences and usages of the instance scattered throughout heterogeneous modules. Yet, other approaches (e.g., (Aryani et al., 2009; Hassan and Holt, 2004; Mirarab et al., 2007)) take a different approach than refactoring but with the common goal of assisting in software maintenance. Such approaches predict particular changes to program elements based on, for example, historical data (Ying et al., 2004), and these changes are presented to the developer in the form of suggestions.

Although existing approaches are useful in alleviating some of the burden associated with software evolution and maintenance, there are a number of situations where developers are still required to complete evolution and maintenance tasks manually. These include but are not limited to upgrading legacy Java software to take advantage of many available features of the modern Java language and updating software composition specifications to cope with change. Automated approaches to assist developers with such cumbersome and error-prone tasks would be extremely useful in evolving and maintaining large, complex systems.

1.1 Automated Techniques for Software Evolution

In this thesis, I explore and develop a number of new techniques that can be of great value to software developers in evolving code to accommodate change. I introduce each of these in the following sections; detailed information can be found throughout the thesis as set forth in Section 1.2.
1.1.1 Automated Refactoring of Legacy Java Software to Enumerated Types

The first of these is an automated refactoring which upgrades legacy Java code to use proper language enumeration (enum) types, a feature of the modern Java language. Enums were introduced to replace various patterns employed by legacy Java software developers. These varied patterns compensated for the lack of Java language enumeration types. This leaves a large body of code to be upgraded manually to take advantage of the new enum constructs.

I have developed an approach that preserves semantics and that allows us to migrate legacy applications by automatically replacing a predominantly used pattern with suitable use of enums. This not only increases type safety, it also makes the code easier to comprehend, removes unnecessary complexity, and eliminates brittleness problems that can prevent separate compilation.

1.1.2 Recovering Pointcut Expressions in Evolving Aspect-Oriented Software

For the second technique, I explore and develop an automated approach to assist developers in maintaining pointcuts in evolving Aspect-Oriented (AO) programs. AO languages enable developers to better encapsulate crosscutting concern (CCC) implementations by allowing them to create an expression (a pointcut) which specifies well-defined points (join points) in a program’s execution where code corresponding to a CCC (an aspect) should apply. However, changes to the underlying program (base-code) may invalidate pointcuts, leaving developers to manually update pointcuts to capture the intended join points. This is a potential source of many errors as a substantial number of join points may require manual investigation by developers.
In addition, updated pointcuts may capture join points that should not be captured given the changes in the base-code or fail to capture points that should have been captured.

I have developed an approach that mechanically aids developers in suitably updating pointcuts upon changes to the base-code by analyzing arbitrarily deep structural commonalities between program elements associated with pointcuts in a particular software version. The extracted patterns are then applied to later versions to suggest additional join points that may require inclusion. My empirical evaluation suggests that the approach can be useful in assisting developers in maintaining pointcuts as base-code evolves.

### 1.1.3 Enforcing Behavioral Constraints in Evolving Aspect-Oriented Programs

The third technique I present in this thesis pertains to reasoning about the behavior of AO programs. As previously noted, AOP facilitates localized implementations of CCCs by allowing developers to encapsulate code realizing a CCC that would otherwise be scattered throughout many system modules and/or intertwined with code realizing the primary functionality of a module. Therefore, AOP allows developers to think about “one concern at a time.” However, reasoning about AOP presents unique challenges, particularly as such programs evolve. Aspects may be added, removed, or changed, consequently affecting many modules throughout the system. As a result, existing assertions may be invalidated, forcing developers to re-verify code that may not have been directly altered.

I have developed an approach to specify the behavior of aspects in such a way that they can be combined with the behavior of the base-code to arrive at the overall
system behavior without such invalidation. Although my technique is not currently automated and planned future work does not involve fully automating this technique, I nevertheless explore ways to assist developers by mechanizing portions of the associated reasoning activities.

1.2 Organization

This thesis is organized as follows. Chapter 2 discusses my work regarding automatically refactoring legacy Java software to utilize the enum construct introduced in Java 5. Chapter 3 presents work regarding pointcut rejuvenation in evolving AO software. My reasoning technique for evolving AO software is developed in Chapter 4. Finally, Chapter 5 summarizes the thesis.
CHAPTER 2: Automated Refactoring of Legacy Java Software to Enumerated Types

2.1 Introduction

Java 5 introduces a rich set of new features and enhancements such as generics, metadata annotations, boxing/unboxing, and type-safe enumerations (Sun Microsystems, 2010a). These constructs can ease software development and maintenance, possibly resulting in more efficient and robust applications. Although Java 5 is backwards compatible with code from previous releases, there are numerous advantages in migrating legacy code to these new features.

Code migration can be a laborious and expensive task both for code modification and for regression testing. The costs and dangers of migration can be reduced greatly with automated refactoring tools. This chapter presents work on a fully-automated semantics-preserving approach for migrating legacy Java code to take advantage of the new type-safe enumeration construct in Java 5.

An enumerated (enum) type (Pierce, 2002) is a data type whose legal values consist of a fixed, closely related set of items known at compile-time (Bloch, 2001). Typically, the exact values of the items are not programmatically important; what is significant is that the values are distinct from one another and perhaps ordered in a certain way, hence the term “enumeration.” Clearly, this is a desirable construct, and since it was not included in the Java language until version 5, developers were forced to use various
compensation patterns to represent enum types. These patterns produce solutions with varying degrees of uniformity, type safety, expressiveness, and functionality.

Of these patterns, one popular (as affirmed by Sun Microsystems (2010a)) way to represent an enumerated type in legacy (≤ 1.4) Java is the weak enum pattern (Bloch, 2001), also known as type codes (Fowler, 1999; Kerievsky, 2004). This pattern uses declared constants (“codes”) defined with relatively small, manually enumerated values. These constants are typically declared as static final fields. As discussed in Section 2.2, there are great advantages to migrating compensation patterns in legacy code to proper enum types.

In this chapter, I detail work on a semantics-preserving approach for identifying instances of the weak enum pattern in legacy code and migrating them to the new enum construct. At the core of the approach is an interprocedural type inferencing algorithm that tracks the flow of enumerated values. Given a set of static final fields, the algorithm computes an enumerization grouping containing fields, methods, and local variables (including formal parameters) whose types can safely be refactored to use an enum type. The algorithm identifies the fields that are being utilized as enumerated values and all other program entities that are transitively dependent upon these values.

The refactoring approach has been implemented as an Eclipse plug-in. An experimental evaluation used a set of 17 Java programs with 899 thousand lines of code. My study indicates that:

- the analysis cost is practical, with average running time of 2.48 seconds per thousand lines of code,
- the weak enum pattern is commonly used in legacy Java software, and
• the usability of the proposed algorithm is promising in that it successfully refactor-
sors a large number of static final fields participating in the weak enum pattern into proper enumerated types.

2.1.1 Contributions

I have made the following specific contributions:

Algorithm design. I present an automated refactoring approach for migration to Java 5 enum types. The approach infers fields that are possibly being used to simulate enumerated types via the weak enum pattern and identifies all code changes that need to be made in order to introduce the inferred proper enum types.

Implementation and experimental evaluation. The approach was implemented as an Eclipse plug-in to ensure real-world applicability. A study on seventeen Java programs indicates that the technique is effective and practical in migrating instances of the weak enum pattern to language enumerated types. These results advance the state of the art in automated tool support for the evolution of legacy Java code to modern Java technologies.

2.1.2 Organization

This chapter is organized as follows. Section 2.2 presents background on enumerated types and a simple example of a Java program snippet utilizing the weak enum pattern to represent enumerated types, along with the same example but refactored to use the new language enumerated types found in Java 5. I go on to compare the disadvantages of the weak enum pattern with the advantages of the refactored version. Section 2.3 briefly discusses the challenges associated with performing the refactoring
in an automated fashion and introduces my approach, while Section 2.4 discusses the details of the approach. I report on the results of the evaluation in Section 2.5. Related work is discussed in Section 2.6, future work is presented in Section 2.7, and Section 2.8 concludes.

2.2 Motivation and Example

An enumerated type has values from a fixed set of constants (Bloch, 2001). Java has historically provided no language mechanisms for defining enumerated types, leading to the emergence of various compensation patterns. However, the compiler depends on the internal representation (typically int) of the symbolically named constants, and type checking cannot distinguish between values of the enum type and those of the type internally representing those values.

2.2.1 Example

Figure 2.1(a) shows an example in which named constants are used to encode values of enumerated types. For example, field color declared at line 8 represents the color that the traffic signal is currently displaying. The values of this field come from the three static final fields RED, YELLOW, and GREEN, which map symbolic names to their associated integer representations. The compile time values of these constants are manually enumerated so that each color can be unambiguously distinguished. Of course, the integer values have no real relationship to the colors they represent. Similarly, field currentAction declared at line 23 could take its values from the

\footnote{This example was inspired by the authors’ work at the Center for Automotive Research at the Ohio State University.}
class TrafficSignal {
    public static final int RED = 0;
    public static final int YELLOW = 1;
    public static final int GREEN = 2;

    /* Current color of the traffic signal,
     * initially red by default */
    private int color = RED;

    /* Accessor for the light's current color */
    public int getColor() {return this.color;}
}

class Automobile {
    private static final int IDLE = 0;
    private static final int INCREASE_SPEED = 1;
    private static final int DECREASE_SPEED = 2;
    private static final int STOP = 3;
    private static final int MAX_SPEED = 140;

    /* The action this automobile is currently
     * performing, idle by default */
    private int currentAction = IDLE;

    /* The current speed of the automobile,
     * initially 5 mph. */
    private int currentSpeed = 5;

    private int react(TrafficSignal signal) {
        switch(signal.getColor()) {
            case TrafficSignal.RED:
                return STOP;
            case TrafficSignal.YELLOW:
                // decide whether to stop or go
                if (this.shouldGo())
                    return INCREASE_SPEED;
                else
                    return STOP;
            case TrafficSignal.GREEN:
                return this.currentAction;
            default:
                throw new IllegalArgumentException
                    ("Invalid traffic color");
        }
    }

    public void drive() {
        TrafficSignal aSignal = ...;
        int reaction = this.react(aSignal);
        if (reaction != this.currentAction &&
            reaction != Action.INCREASE_SPEED ||
            this.currentSpeed <= MAX_SPEED)
            this.performAction(reaction);
        }

        private void performAction(int action){...}
    }
}

// Improvements after refactoring

class TrafficSignal {
    public enum Color {RED, YELLOW, GREEN};

    /* Current color of the traffic signal,
     * initially red by default */
    private Color color = Color.RED;

    /* Accessor for the light's current color */
    public Color getColor() {return this.color;}
}

class Automobile {
    private enum Action {IDLE, INCREASE_SPEED, DECREASE_SPEED, STOP};

    private static final int MAX_SPEED = 140;

    /* The action this automobile is currently
     * performing, idle by default */
    private Action currentAction = Action.IDLE;

    /* The current speed of the automobile,
     * initially 5 mph. */
    private int currentSpeed = 5;

    private Action react(TrafficSignal signal) {
        switch(signal.getColor()) {
            case TrafficSignal.RED:
                return Action.STOP;
            case TrafficSignal.YELLOW:
                // decide whether to stop or go
                if (this.shouldGo())
                    return Action.INCREASE_SPEED;
                else
                    return Action.STOP;
            case TrafficSignal.GREEN:
                return this.currentAction;
            default:
                throw new IllegalArgumentException
                    ("Invalid traffic color");
        }
    }

    public void drive() {
        TrafficSignal aSignal = ...;
        Action reaction = this.react(aSignal);
        if (reaction != this.currentAction &&
            reaction != Action.INCREASE_SPEED ||
            this.currentSpeed <= MAX_SPEED)
            this.performAction(reaction);
    }

    private void performAction(Action action){...}
}

(a) Using integer constants for enumerated types. (b) Improvements after our refactoring is applied.

Figure 2.1: Running enumeration example: a hypothetical drive-by-wire application.
integer constants in static final fields \texttt{IDLE}, \texttt{INCREASE\_SPEED}, \texttt{DECREASE\_SPEED}, and \texttt{STOP}.

Field \texttt{MAX\_SPEED} (line 19) defines the maximum speed of the automobile. This field differs from the remaining static final fields: unlike their integer values, which are used only to encode enumerated values, the value of \texttt{MAX\_SPEED} has a very significant meaning. This key distinction illustrates the difference between fields that are \textit{named constants} (e.g., \texttt{MAX\_SPEED}) from those participating in the \textit{int enum pattern} \cite{Bloch2001}. I have considered a more general version of this pattern which applies to nearly all primitive types\footnote{A similar pattern called \textit{Type Codes} is described in \cite{Fowler1999,Kerievsky2004}.} I will refer to it as the \textit{weak enum pattern}, or more specifically, the \textit{weak primitive enum pattern}. The term “weak” is used to denote the lack of type safety and other features inherent to the pattern, and the term “primitive” is used to denote the types used in the pattern. In the sections that follow, I will use these terms interchangeably but will make more deliberate distinction in Section 2.7 where I discuss the possibility of refactoring additional compensation patterns.

Figure 2.1(a) illustrates the use of the weak enum pattern. Clearly, the meaning of \texttt{int} depends on the context of where values are used. The programmer is left with the responsibility of \textit{manually} inferring which \texttt{int} entities are intended to represent traffic light colors, which are automobile actions, and which are general integers. In effect, the programmer would be required to investigate transitive relationships of these program entities to other program entities and operations in which they appear.

Although the weak enum pattern provides a mechanism to make programmer intent

\footnote{I exclude boolean from this list for several reasons: (i) the type is associated with only two values, \texttt{true} and \texttt{false}, thus, any transformed enum type can only have two members and (ii) the algorithm presentation is simpler due to this exclusion.}
more explicit, it suffers from several significant weaknesses which have been well
documented (Bloch, 2001; Sun Microsystems, 2010a). I discuss some of these next.

Type Safety

The most glaring weakness is the lack of type safety. For example, there is no
mechanism to enforce the constraint that color gets its values only from the three
color fields; any integer value would be acceptable at compile time. Such problems
would not be detected until run time, when an exception would be thrown. Perhaps
worse, the execution will seem to be normal while behaving in a way not originally
intended by the programmer. Problems could also arise from the allowed operations.
For example, it would be possible to perform arbitrary integer operations, such as
addition or multiplication, upon the color values.

Program Comprehension

The weak enum pattern creates ambiguities at various levels. For example, there
are fundamental semantic differences between the constants for automobile actions
(beginning on line 15) and MAX_SPEED (line 19). Despite these differences, both
entities have essentially identical declarations. The programmer depends on docu-
mentation and/or extensive interprocedural usage investigation to determine the true
intent of the fields. This is also an issue for multiple sets of enum constants. For ex-
ample, methods getColor (line 11) and react (line 29) declare the same int return
type, even though the returned entities have very different meaning and context. In
essence, the program is less self-documented with respect to the enumerated types,
which could have negative effect on software maintenance tasks.
Verbosity

Verbosity and added complexity arises in several areas. First, there is no easy way to print the enumerated values in a meaningful way. Additional code is typically required to produce desirable results, e.g., as in

```java
if (this.color == RED)
    System.out.println("RED");
```

Second, there is no convenient way to iterate over all values of the enumerated type (Bloch, 2001), which requires the developer to manually create such machinery. Third, the weak enum pattern requires the programmer to manually enumerate the values of the constants, which increases the likelihood of errors. For example, different enum constants may be unintentionally assigned the same internal value.

Name spacing

Constant names from different enum types may collide, especially in distributed development environments, as they are not contained in their own namespace. For example, the constant RED may belong to two enum types defined in the same class. Such a collision would need to be resolved by prefixing the constants with an appropriate identifier (e.g., COLOR_RED).

Separate compilation

Finally, the weak enum pattern produces types whose values are brittle (Sun Microsystems, 2010a). Since the values are compile time constants, at compile time they are inlined into clients. Therefore, if the internal representations of the constants change, e.g., to add new constants in between existing ones using a consistent
numbering, clients must be recompiled. Otherwise, the behavior of clients upon referencing the values of the enum type is undefined. Such results are devastating for successful separate compilation, a key feature of modern compilers.

### 2.2.2 Enumerations in Java 5

The new `enum` construct supports powerful enumerated types that are completely and conveniently type safe, comparable, and serializable, saving the programmer from creating and maintaining verbose custom classes. Enum types increase self-documentation (e.g., a `getColor` method has a return type of `Color`), enable compile-time type checking, allow meaningful printed values, avoid name conflicts, and support separate compilation.

Figure 2.1(b) shows an *enumerized* version of the running example, in which the language enumerated types `TrafficSignal.Color` and `Automobile.Action` have replaced the `static final` fields. The legal values and operations of these new enumerated types are now enforced through compile-time checking. There is a clear distinction between the named constant `MAX_SPEED` and the enumerated values. It is also clear that the result of a call to `react` is an `Action`, which distinguishes it from the return type of `getColor` and makes the API more informative. Programmers are no longer required to enumerate values by hand or write extra “pretty printing” code.

After enumerization, the brittleness of the overall system is reduced. For example, suppose we wanted to make `TrafficSignal` compatible with Poland’s system, where a yellow and red combination is shown directly after red to alert drivers that a change to green is imminent. After `RED` in Figure 2.1(a) one could add a new field `RED_YELLOW` with value of 1; the remaining fields’ values would have to be incremented. Even
if we did not care to modify \texttt{Automobile} to accommodate the new color, we would still have to recompile it, since upon the original compilation the constant values for the colors were inlined. In Figure 2.1(b) on the contrary, additional values can be added easily, and only the enum or the class containing the enum would require recompilation.

2.3 Enumerization Approach

A refactoring tool which modifies legacy Java code employing the weak enum pattern to utilize the Java 5 \texttt{enum} construct faces two major challenges, namely, \textit{inferring enumerated types} and \textit{resolving dependencies}. Inferring enumerated types requires distinguishing between weak enum constants and named constants. Figure 2.1(a) illustrates this issue through fields \texttt{STOP} and \texttt{MAX\_SPEED}. Although their declarations are very similar, they are conceptually very different. That is, while the value of named constant \texttt{MAX\_SPEED} is meaningful in integer contexts (e.g., for the integer comparison at line 55), the only requirement on the value of enumerated constant \texttt{STOP} is that it should be different from the other integer values representing actions. In general, the uses of the enumerated values are limited to assignments, parameter passing, method return values, and equality comparisons. Named constants are used in a much wider context, including mathematical calculations (e.g., dividing by \texttt{java.lang.Math.PI}), various value comparisons (as in line 55), and so on. Determining the category to which a constant field belongs requires investigation of every context in which that field’s value is used.

Constant fields are not the only program entities that need to be refactored for enumerization. For example, in Figure 2.1(a) once it has been inferred that \texttt{STOP}
is an enumerated constant, all program entities that also require refactoring due to transitive dependencies on STOP must identified. I say that entity A is type dependent on entity B if and only if changing the type of B requires changing the type of A. An instance of such a dependency occurs with the method react, i.e., since it returns the integer form of STOP, in the refactored version it must return the enum type containing STOP. Furthermore, due to the dependence on the return value of react, the local integer variable reaction in drive (line 51, Figure 2.1(b)) must also be transformed to be of type Action.

The next section describes a interprocedural refactoring algorithm that addresses these challenges through careful categorization of the contexts in which migration from the weak enum pattern to the new enum construct is valid. The algorithm identifies all type dependent entities in those contexts, including fields, local variables, method return types, and formal parameters. After all affected entities are identified, they are classified into groups that must share the same enum type. At the end, all automatically transformed code is semantically equivalent to the original.

2.4 Algorithm

In this section, I discuss the details of my algorithm for refactoring instances of the weak enum pattern (as discussed in Section 2.2) to proper enum constructs. I begin the discussion by listing several assumptions of the algorithm regarding the source code to be refactored. Section 2.4.2 provides a high-level overview of my approach, while Section 2.4.3 presents my type inferencing algorithm that helps identify instances of the weak enum pattern. Sections 2.4.4 and 2.4.5 discuss considerations for examining transitive dependencies of candidate constants, and Section 2.4.6 presents
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{P}$</td>
<td>The original program.</td>
</tr>
<tr>
<td>$\phi(\mathcal{P})$</td>
<td>${ f \mid f$ is a static final field of primitive type in $\mathcal{P} }$.</td>
</tr>
<tr>
<td>$\mu(\mathcal{P})$</td>
<td>${ m \mid m$ is a method in $\mathcal{P} }$.</td>
</tr>
<tr>
<td>$\nu(\mathcal{P})$</td>
<td>${ l \mid l$ is a variable in $\mathcal{P} }$.</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>A variable, formal parameter, field, or method.</td>
</tr>
<tr>
<td>$\alpha_{\text{ctx}}$</td>
<td>A context in which $\alpha$ may occur.</td>
</tr>
<tr>
<td>$\mathcal{P}(ID)$</td>
<td>The program entity corresponding to the terminal identifier $ID$.</td>
</tr>
</tbody>
</table>

Table 2.1: Enumerization formalism notation.

a series of additional constraints on the final set of constants to be refactored in order to preserve semantics. Finally, Section 2.4.7 demonstrates the presented algorithm by applying it to the example code snippet given in Section 2.2.

2.4.1 Assumptions

My algorithm works on a closed-world assumption, meaning that I assume full access to all source code that could possibly affect or be affected by the refactoring. I also assume that we are able to statically identify all references to candidate fields and transitively dependent program entities. This assumption could be invalidated by reflection and custom class loaders.

Furthermore, I assume that the original source code successfully compiles under a Java 5 compiler, thus guaranteeing the following properties:

1. There are no uses of the identifier `enum` throughout the program source\(^4\)

2. The source code is type correct.

\(^4\)Although the focus of my tool is to refactor legacy Java software to utilize the new `enum` construct, I do not discriminate against current Java software (i.e., that written in Java $\geq 5$). In this case, uses of the `enum` identifier for declaring language enumerated types is acceptable.
3. All implicit primitive value conversions are lossless.

Under the Java 5 compiler, the `enum` identifier is now a reserved keyword and one that would be used in declarations of language enumerated types only. Therefore, assumption (1) allows me to use the `enum` keyword for such purposes only. Assumption (2) is essential as my algorithm is thoroughly dependent on the type relationships of each program entity in the original source. Consequently, the result of my algorithm on type-incorrect source code is undefined.

Assumption (3) is also important. Although primitive types do not share many of the same properties as reference types, such as subtype relationships, etc., there exists important relationships between these types that an inferencing algorithm must account for. In fact, this is particularly important to semantic preservation during any transformation of primitive value types to reference types. Similar to the $\leq$ relationship exploited for class type inferencing algorithms in (Palsberg and Schwartzbach, 1994), primitive types define conversion relationships between them (Sun Microsystems, 2010b). Primitives do not enjoy the same polymorphic capabilities that the subtype relationship provides reference types. However, primitives are allowed to be implicitly assigned to values of different primitive types much in the same way subtype instances can be assigned to variables of their corresponding supertypes. Such a conversion, in the context of primitives, is called an implicit widening conversion (Sun Microsystems, 2010b).

Widening conversions, which does not require explicit casts, allows primitive type values to be used interchangeably (through assignment and comparison). Thus, variables of type `double` are allowed to be assigned values of type `int`, `int` variables are allowed to be assigned values of type `char`, and so on. This relationship can be
described as $\texttt{char} \leq \texttt{int} \leq \texttt{double}$. The implicit conversion is legal so long as the value transfer is lossless, i.e., no precision of the value is lost by the conversion. Conversions in which precision can be potentially lost are called *narrowing conversions* and must be made explicit through casts. There are, however, exceptions to this rule. For example, a narrowing conversion is allowed to implicitly (i.e., sans casting) take place as long as the value of the larger, in terms of the maximum capacity in bytes held by values of the type, type can be resolved to a value that requires less than or equal to the amount of storage allocated for values of smaller types at compile-time. Although the conversion relationship between $\texttt{byte}$ and $\texttt{int}$ is $\texttt{byte} \leq \texttt{int}$, the $\texttt{int}$ constant literal $2$, whose value can be vacuously resolved at compile-time, can be stored in a variable of type $\texttt{byte}$ without a risk of loss in precision (i.e., the conversion is lossless). Since my algorithm infers enumerated types by analyzing constants, such a scenario is potentially common and the algorithm must account for this possibility. Because my algorithm assumes that the original program is type-correct, it is safe to further assume that *all* primitive, implicit conversions, widening and narrowing alike, are lossless. Henceforth, transitivity may exist between entities of *different* declared types. My algorithm does not single out *inter-primitive type* transitivity among program entities. As such, this assumption is necessary to ensure that adequate precision exists in order to preserve semantics.

### 2.4.2 Top-level Processing

Procedure *Enumerize*, shown as Algorithm 2.1, is the top-level driver of my approach. It takes as input the source code of the original program $\mathcal{P}$, as well as a set $F \subseteq \phi(\mathcal{P})$ of fields (see the notation in Table 2.1). Portions of this notation were
Algorithm 2.1 Top-level enumerization algorithm.

procedure Enumerize($F, \mathcal{P}$)
  1: $R \leftarrow$ Enumerizable($F$)
  2: $R \leftarrow$ Unique($R$) ∩ Distinct($R$) ∩ Consistent($R$)
  3: for all $T \in R$ do
  4:    Transform($T$)
  5: end for
end procedure

Inspired by (Fuhrer et al., 2005; Kieżun et al., 2007; Steimann et al., 2006). As previously mentioned, my work deals only with refactoring the so-called “standard” compensation pattern in pre-Java 5 as described in (Bloch, 2001; Fowler, 1999; Kerievsky, 2004; Sun Microsystems, 2010a). As such, Enumerize analyzes only static final fields of primitive types since they may potentially be participating in the weak enum pattern. In Section 2.7.2 I discuss plans to advance the algorithm to deal with other commonly used patterns.

The processing workflow continues as follows. Function Enumerizable (line 1) infers which candidate fields are being used as enumerated values and groups them into their corresponding inferred enum types. At line 2 certain semantics-preserving constraints are enforced (further discussed in Section 2.4.6). Finally, Transform (line 4) performs the actual code refactoring for each inferred enum type $T$, thus altering the type declarations of each corresponding program entity. The primitive constants are replaced with the new enum type declarations. The new enum constants are ordered by their original primitive values to enforce a natural ordering, thereby preserving comparability semantics.

2.4.3 Type Inferencing
Algorithm 2.2 Enumerization set building algorithm.

function \textit{Enumerizable}(C)

1: \( W \leftarrow C \) \{Seed the worklist with the input constants.\}
2: \( N \leftarrow \emptyset \) \{The non-enumerizable set list, initially empty.\}
3: \textbf{for all} \( c \in C \) \textbf{do}
4: \hspace{1em} \textit{MakeSet}(c) \ \{Initialize the union-find data structure.\}
5: \textbf{end for}
6: \textbf{while} \( W \neq \emptyset \) \textbf{do}
7: \hspace{1em} \( \alpha \leftarrow e \mid e \in W \) \{Remove an element from the worklist.\}
8: \hspace{1em} \( W \leftarrow W \setminus \{\alpha\} \) \{Remove an element from the worklist.\}
9: \hspace{1em} \textbf{for all} \( \alpha_{\text{ctxt}} \in \text{Contexts}(\alpha, \mathcal{P}) \) \textbf{do}
10: \hspace{2em} \textbf{if} \( \neg \text{EnumerizableContext}(\alpha, \alpha_{\text{ctxt}}) \) \textbf{then}
11: \hspace{3em} \( N \leftarrow N \cup \{\alpha\} \) \{Add to the non-enumerizable list.\}
12: \hspace{2em} \textbf{break}
13: \hspace{2em} \textbf{end if}
14: \hspace{2em} \textbf{for all} \( \hat{\alpha} \in \text{Extract}(\alpha, \alpha_{\text{ctxt}}) \) \textbf{do} \{For all entities to be enumerized due to \( \alpha \).\}
15: \hspace{3em} \textbf{if} \( \text{Find}(\hat{\alpha}) = \emptyset \) \textbf{then}
16: \hspace{4em} \textit{MakeSet}(\hat{\alpha})
17: \hspace{4em} \( W \leftarrow W \cup \{\hat{\alpha}\} \)
18: \hspace{3em} \textbf{end if}
19: \hspace{2em} \textit{Union}(\text{Find}(\alpha), \text{Find}(\hat{\alpha}))
20: \hspace{2em} \textbf{end for}
21: \hspace{1em} \textbf{end for}
22: \textbf{end while}
23: \( F \leftarrow \text{AllSets}() \) \{The sets to be returned.\}
24: \textbf{for all} \( \alpha' \in N \) \textbf{do} \{For all non-enumerizable elements.\}
25: \hspace{1em} \( F \leftarrow F \setminus \text{Find}(\alpha') \) \{Remove its corresponding set.\}
26: \textbf{end for}
27: \textbf{return} \( F \) \{\( F \) is all sets minus the non-enumerizable sets.\}

end function
Function *Enumerizable*, shown as Algorithm 2.2, is at the heart of the proposed approach. This type inferencing algorithm is based on a family of type inferencing approaches from [Palsberg and Schwartzbach, 1994], and has two goals:

(i) infer fields that are being used as part of enumerated types (i.e., participating in the weak enum pattern)

(ii) construct minimal sets such that members of the same set must share the same enum type after refactoring

The output of the algorithm is a set of *enumerization sets* containing fields, method declarations, and local variables (including formal parameters) and their minimal groupings that are enumerizable with respect to the input constants.

The algorithm uses a worklist $W$, which is initialized with all given constant fields, as well as a set $N$ of entities that are not amenable to enumerization. A union-find data structure maintains sets of related entities; initially, each input constant field belongs to a separate singleton set. Each worklist element $\alpha$ is a program entity whose type may have to be changed to an enum type. A helper function $\text{Contexts}$ identifies all contexts (explained next) in which $\alpha$ and its related entities $\alpha'$ appear in $\mathcal{P}$ such that each context $\alpha_{\text{ctxt}}$ needs to be examined later in the algorithm.

$\text{Contexts}(\alpha, \mathcal{P})$, depicted in Figure 2.2, includes all *inner-most* (i.e., identifier terminals in the grammar) expressions corresponding to $\alpha$ (excluding those appearing in initializations of constant fields). Furthermore, if $\alpha$ is a method, this set of contexts also includes $\text{Contexts}(\alpha', \mathcal{P})$ for every method $\alpha'$ which overrides $\alpha$ or is overridden by $\alpha$. Similarly, if $\alpha$ is a formal parameter, the set of contexts includes $\text{Contexts}(\alpha', \mathcal{P})$ for every corresponding formal parameter $\alpha'$ in an overriding or overridden method.
Contexts(α, P) = all inner-most expressions containing α ∪
\[
\begin{cases}
\bigcup_{\alpha' \in MH(\alpha)} & \text{if } \alpha \text{ is a method} \\
\bigcup_{\alpha' \in PH(\alpha)} & \text{if } \alpha \text{ is a formal parameter} \\
\emptyset & \text{o.w.}
\end{cases}
\]

Figure 2.2: Contexts for a program entity α.

Entities α’ need to be considered due to polymorphism. For example, if the return type of a method m is changed from int to an enum type, this change must be propagated to all methods overriding m or being overridden by m. Similar propagation is necessary when m’s formal parameters are changed (otherwise, method overriding would incorrectly be transformed to method overloading). I denote these sets of dependent entities as method hierarchies (abbreviated MH in Figure 2.2) and parameter hierarchies (abbreviated PH), respectively.

The predicate EnumerizableContext examines a context αctxt to determine if it is amenable to enumeration with respect to α via two helper predicates, namely, EnumerizableAscender and EnumerizableDescender. Upon application, these helper functions examine the context passed to EnumerizableContext by traversing, in disparate directions, the syntax tree of the input expression. The intent of these functions are loosely analogous to that of synthesized and inherited attributes of attribute grammars [Knuth 1967], respectively. Function Extract is responsible for determining further transitive relationships due to the enumeration of α. Extract has two helper functions, namely, ExtractionAscender and ExtractionDescender, which are
similar to the aforementioned helper predicates. For conciseness, in the following
discussion, I will use the abbreviations $EC$, $EA$, $ED$, $EX$, $XA$, and $XD$, respectively,
to refer to each of these.

Essentially, $EnumerizableContext$ and $Extract$ serve as canonical names for their
intended purpose. $EC$ has two parameters, namely, the entity $\alpha$, whose enumeriz-
ability is under question, and a context $\alpha_{ctxt}$, which is type dependent on $\alpha$. $EX$, on
the other hand, has one parameter $\alpha_{ctxt}$ whose constituent, type dependent program
entities must be examined for enumerization.

Predicate $EC$ immediately calls $EA$ passing it $\alpha_{ctxt}$, the context to be examined
and $\alpha$, the entity whose enumerization is under question. Figure 2.3 portrays many
of the rules of $EA$ which are inductively defined in the grammar. $EA$ begins at $\alpha_{ctxt}$
(e.g., $ID$) and climbs (or ascends) its way up the grammar until it reaches a significant
ancestor of $\alpha$. I say that a statement or expression is a significant ancestor of $\alpha$ if the
value of $\alpha$ can be exploited at that point. The ascent is performed via the $Parent$
function, which returns the parent expression above $\alpha_{ctxt}$ in the syntax tree. The
predicate $Contains$ helps determine which expression $EA$ ascended from.

On the way to the significant ancestor, $EA$ may find expressions that are not
amenable to enumerization. In that case, $EA$ will return $false$ and $EC$, in turn, will
return the result of $EA$. Such a situation is depicted in the rule for array access/cre-
ation in Figure 2.3. On the other hand, once $EA$ successfully reaches the significant
ancestor, it will then call $ED$ in order to commence a descent down the pivotal ex-
pression(s); that is, an expression that is consequently type dependent. Many of the
rules of $ED$ are given in Figure 2.4. As shown, $ED$ completes its descent at the leaf
nodes of the syntax tree, returning $true$ for terminal IDs and $false$ for contexts that
Identifiers

**predicate** $EA(\alpha, ID)$

1: **return** $EA(\alpha, Parent(ID))$

Parenthesized expressions

**predicate** $EA(\alpha,(ID))$

1: **return** $EA(\alpha, Parent(ID))$

Cast expressions

**predicate** $EA(\alpha, (\text{Type})\Exp)$

1: **return** false

Field access expressions

**predicate** $EA(\alpha, \Exp.ID)$

1: **return** $EA(\alpha, Parent(\Exp))$

Assignment expressions

**predicate** $EA(\alpha, \Exp_1 = \Exp_2)$

1: **return** $ED(\Exp_1) \land ED(\Exp_2)$

Subtract assignment expressions

**predicate** $EA(\alpha, \Exp_1 -\!\!\= \Exp_2)$

1: **return** false

Divide assignment expressions

**predicate** $EA(\alpha, \Exp_1 /\!\!= \Exp_2)$

1: **return** false

Infix addition expressions

**predicate** $EA(\alpha, \Exp_1 + \Exp_2)$

1: **return** false

Infix multiplication expressions

**predicate** $EA(\alpha, \Exp_1 \ast \Exp_2)$

1: **return** false

Prefix unary minus expressions

**predicate** $EA(\alpha, -\Exp)$

1: **return** false

Postfix increment expressions

**predicate** $EA(\alpha, \Exp++)$

1: **return** false

Equality expressions

**predicate** $EA(\alpha, \Exp_1 == \Exp_2)$

1: **return** $ED(\Exp_1) \land ED(\Exp_2)$

Continued
Inequality expressions

**predicate** \( EA(\alpha, \text{Exp}_1 \neq \text{Exp}_2) \)

1: **return** \( ED(\text{Exp}_1) \land ED(\text{Exp}_2) \)

Switch statements

**predicate** \( EA(\alpha, \text{switch(Exp)}) \)

1: **let** \( \text{se} = ED(\text{Exp}) \)
2: **let** \( \text{ce} = \text{true} \)
3: **for all** \( \text{case Exp}_c \in \text{cases(switch(Exp))} \) **do**
4: \( \text{ce} \leftarrow \text{ce} \land ED(\text{Exp}_c) \)
5: **end for**
6: **return** \( \text{se} \land \text{ce} \)

Switch case statements

**predicate** \( EA(\alpha, \text{case Exp}) \)

1: **return** \( EA(\alpha, \text{switchStmt(case Exp)}) \)

Conditional expressions

**predicate** \( EA(\alpha, \text{Exp}_1?\text{Exp}_2:\text{Exp}_3) \)

1: **if** \( \text{Contains(Exp}_2, \alpha) \lor \text{Contains(Exp}_3, \alpha) \) **then**
2: **return** \( EA(\alpha, \text{Parent(Exp}_1?\text{Exp}_2:\text{Exp}_3)) \)
3: **else**
4: **return** \( \text{true} \)
5: **end if**

Array access/creation expressions

**predicate** \( EA(\alpha, \text{Exp}_1[\text{Exp}_2]) \)

1: **if** \( \text{Contains(Exp}_2, \alpha) \) **then**
2: **return** \( \text{false} \)
3: **else**
4: **return** \( EA(\alpha, \text{Parent(Exp}_1)) \)
5: **end if**

Array initialization expressions

**predicate** \( EA(\alpha, \{\text{Exp}_1, \ldots, \text{Exp}_n}\}) \)

1: **let** \( ie = \text{true} \)
2: **for** \( \text{Exp}_i, 1 \leq i \leq n \) **do**
3: \( ie \leftarrow ie \land ED(\text{Exp}_i) \)
4: **end for**
5: **return** \( ie \)

Continued
Return statements
predicate $EA(\alpha, \text{return } \text{Exp})$

1: return true

Method declaration statements
predicate $EA(\alpha, \text{ID}(P_1, \ldots, P_n))$

1: let re = true
2: for all return $\text{Exp}_r \in returnStmts(\text{ID}(P_1, \ldots, P_n))$ do
3: re $\leftarrow$ re $\land$ $ED(\text{Exp}_r)$
4: end for
5: return re

Formal parameters
predicate $EA(\alpha, P_i)$

1: let ae = true
2: /*check the ith argument of each invocation of the declaring method*/
3: let $\hat{\alpha} = MethodDecl(P_i)$
4: for all $\hat{\alpha}_{ctxt} \in Invocations(\hat{\alpha}, \mathcal{P})$ do
5: ae $\leftarrow$ ae $\land$ $ED(\text{Arg}(\hat{\alpha}_{ctxt}, i))$
6: end for
7: return ae

Method invocation expressions
predicate $EA(\alpha, \text{ID}(\text{Exp}_1, \ldots, \text{Exp}_n))$

1: for $\text{Exp}_i, 1 \leq i \leq n$ do
2: if $Contains(\text{Exp}_i, \alpha)$ then
3: return true
4: end if
5: end for
6: return $EA(\alpha, Parent(\text{ID}(\text{Exp}_1, \ldots, \text{Exp}_n)))$

General statements
predicate $XA(\alpha, \text{Smt})$

1: let se = true
2: for $\text{Exp} \in \text{Children}(\text{Smt})$ do
3: se $\leftarrow$ se $\lor$ $ED(\text{Exp})$
4: end for
5: return se

Figure 2.3 continued
Integer literals
predicate \( ED(IL) \)
1: return false

Identifiers
predicate \( ED(ID) \)
1: return true

Parenthesized expressions
predicate \( ED((Exp)) \)
1: return \( ED(Exp) \)

Cast expressions
predicate \( ED((Type)Exp) \)
1: return false

Field access expressions
predicate \( ED(Exp.ID) \)
1: return \( ED(ID) \)

Assignment expressions
predicate \( ED(Exp_1 = Exp_2) \)
1: return \( ED(Exp_1) \land ED(Exp_2) \)

Subtract assignment expressions
predicate \( ED(Exp_1 -= Exp_2) \)
1: return false

Divide assignment expressions
predicate \( ED(Exp_1 /= Exp_2) \)
1: return false

Infix addition expressions
predicate \( ED(Exp_1 + Exp_2) \)
1: return false

Infix multiplication expressions
predicate \( ED(Exp_1 * Exp_2) \)
1: return false

Prefix unary minus expressions
predicate \( ED(-Exp) \)
1: return false

Postfix increment expressions
predicate \( ED(Exp++) \)
1: return false

Continued

Figure 2.4: Enumerizable descender.
Conditional expressions

\textbf{predicate} \( ED(\text{Exp}_1 \ ? \ \text{Exp}_2 : \text{Exp}_3) \)

1: \textbf{return} \( ED(\text{Exp}_2) \land ED(\text{Exp}_3) \)

Array access expressions

\textbf{predicate} \( ED(\text{Exp}_1[\text{Exp}_2]) \)

1: \textbf{return} \( ED(\text{Exp}_1) \)

Array creation expressions

\textbf{predicate} \( ED(\text{Type}[\text{Exp}][\{\text{Exp}_1, \ldots, \text{Exp}_n\}]) \)

1: \textbf{return} \( ED(\{\text{Exp}_1, \ldots, \text{Exp}_n\}) \)

Array initialization expressions

\textbf{predicate} \( ED(\{\text{Exp}_1, \ldots, \text{Exp}_n\}) \)

1: \textbf{let} \( \text{ie} \equiv \text{true} \)
2: \textbf{for} \( \text{Exp}_i, 1 \leq i \leq n \) \textbf{do}
3: \( \text{ie} \leftarrow \text{ie} \land ED(\text{Exp}_i) \)
4: \textbf{end for}
5: \textbf{return} \( \text{ie} \)

Method invocation expressions

\textbf{predicate} \( ED(\text{ID}(\text{Exp}_1, \ldots, \text{Exp}_n)) \)

1: \textbf{return} \( \text{true} \)

are not amenable to enumerization (e.g., literals). \( EA \) will then, in turn, return the result of \( ED \).

\subsection*{2.4.4 Enumerizable Contexts}

\( EC \) returns \textit{false} if the given context \( \alpha_{\text{ctx}} \) is definitively not enumerizable with respect to \( \alpha \) (e.g., \( \alpha \) being used as an array index). Otherwise, \( EC \) returns \textit{true} if \( \alpha_{\text{ctx}} \) is \textit{promising} with respect to \( \alpha \); that is, enumerizing \( \alpha \) does not adversely affect the relation between \( \alpha \) and the enclosing expressions of \( \alpha_{\text{ctx}} \). I say that such a situation is “promising” as opposed to “definite” because there may exist other program entities \( \hat{\alpha} \) that are type dependent on \( \alpha \) and I cannot yet ensure that every context \( \hat{\alpha}_{\text{ctx}} \) in
Identifiers

\textbf{function} \(XA(\alpha, \text{ID})\)

\begin{verbatim}
1: return \(XA(\alpha, \text{Parent(ID)})\)
\end{verbatim}

Parenthesized expressions

\textbf{function} \(XA(\alpha, (\text{ID}))\)

\begin{verbatim}
1: return \(XA(\alpha, \text{Parent(ID)})\)
\end{verbatim}

Cast expressions

\textbf{function} \(XA(\alpha, (\text{TYPE})\text{EXP})\)

\begin{verbatim}
1: return \\emptyset
\end{verbatim}

Field access expressions

\textbf{function} \(XA(\alpha, \text{EXP.ID})\)

\begin{verbatim}
1: return \(XA(\alpha, \text{Parent(\text{EXP})})\)
\end{verbatim}

Assignment expressions

\textbf{function} \(XA(\alpha, \text{EXP}_1 = \text{EXP}_2)\)

\begin{verbatim}
1: return \(XD(\text{EXP}_1) \cup XD(\text{EXP}_2)\)
\end{verbatim}

Subtract assignment expressions

\textbf{function} \(XA(\alpha, \text{EXP}_1 -= \text{EXP}_2)\)

\begin{verbatim}
1: return \\emptyset
\end{verbatim}

Divide assignment expressions

\textbf{function} \(XA(\alpha, \text{EXP}_1 /= \text{EXP}_2)\)

\begin{verbatim}
1: return \\emptyset
\end{verbatim}

Infix addition expressions

\textbf{function} \(XA(\alpha, \text{EXP}_1 + \text{EXP}_2)\)

\begin{verbatim}
1: return \\emptyset
\end{verbatim}

Infix multiplication expressions

\textbf{function} \(XA(\alpha, \text{EXP}_1 \ast \text{EXP}_2)\)

\begin{verbatim}
1: return \\emptyset
\end{verbatim}

Prefix unary minus expressions

\textbf{function} \(XA(\alpha, \neg\text{EXP})\)

\begin{verbatim}
1: return \\emptyset
\end{verbatim}

Postfix increment expressions

\textbf{function} \(XA(\alpha, \text{EXP}++)\)

\begin{verbatim}
1: return \\emptyset
\end{verbatim}

Equality expressions

\textbf{function} \(XA(\alpha, \text{EXP}_1 == \text{EXP}_2)\)

\begin{verbatim}
1: return \(XD(\text{EXP}_2) \cup XD(\text{EXP}_1)\)
\end{verbatim}

Continued

Figure 2.5: Extraction ascender.
Inequality expressions
function \(XA(\alpha, \text{Exp}_1 \neq \text{Exp}_2)\)
1: \(\text{return } XD(\text{Exp}_1) \cup XD(\text{Exp}_2)\)

Switch statements
function \(XA(\alpha, \text{switch}(\text{Exp}))\)
1: \(R \leftarrow XD(\text{Exp})\)
2: \(\text{for all case } \text{Exp}_c \in \text{cases}(\text{switch}(\text{Exp})) \text{ do}\)
3: \(R \leftarrow XD(\text{Exp}_c)\)
4: \(\text{end for}\)
5: \(\text{return } R\)

Switch case statements
function \(XA(\alpha, \text{case } \text{Exp})\)
1: \(\text{return } XA(\alpha, \text{switchStmt}(\text{case } \text{Exp}))\)

Conditional expressions
function \(XA(\alpha, \text{Exp}_1 \text{?} \text{Exp}_2 : \text{Exp}_3)\)
1: \(\text{if } \text{contains}(\text{Exp}_2, \alpha) \lor \text{contains}(\text{Exp}_3, \alpha) \text{ then}\)
2: \(\text{return } XA(\alpha, \text{Parent}(\text{Exp}_1 \text{?} \text{Exp}_2 : \text{Exp}_3))\)
3: \(\text{else}\)
4: \(\text{return } \emptyset\)
5: \(\text{end if}\)

Array access/creation expressions
function \(XA(\alpha, \text{Exp}_1[\text{Exp}_2])\)
1: \(\text{if } \text{contains}(\text{Exp}_2, \alpha) \text{ then}\)
2: \(\text{return } \emptyset\)
3: \(\text{else}\)
4: \(\text{return } XA(\alpha, \text{Parent}(\text{Exp}_1))\)
5: \(\text{end if}\)

Array initialization expressions
function \(XA(\alpha, \{\text{Exp}_1, \ldots, \text{Exp}_n\})\)
1: \(R \leftarrow \emptyset\)
2: \(\text{for } \text{Exp}_i, 1 \leq i \leq n \text{ do}\)
3: \(R \leftarrow R \cup XD(\text{Exp}_i)\)
4: \(\text{end for}\)
5: \(\text{return } R\)

Return statements
function \(XA(\alpha, \text{return } \text{Exp})\)
1: \(\text{return } XD(\text{MethodDecl}(\text{return } \text{Exp}))\)

Continued
Method declaration statements

function XA(α, ID(P₁, . . . , Pₙ))
1: \( R \leftarrow \emptyset \)
2: for all \( \text{return EXP}_r \in \text{returnStmts}(\text{ID}(P₁, . . . , Pₙ)) \) do
3: \( R \leftarrow R \cup XD(\text{EXP}_r) \)
4: end for
5: return \( R \)

Formal parameters

function XA(α, Pᵢ)
1: \( R \leftarrow \emptyset \)
2: /*extract the ith argument of each invocation of the declaring method*/
3: let \( \hat{\alpha} = \text{MethodDeel}(P_i) \)
4: for all \( \hat{\alpha}_\text{ctx} \in \text{Invocations}(\hat{\alpha}, \mathcal{P}) \) do
5: \( R \leftarrow R \cup XD(\text{Arg}(\hat{\alpha}_\text{ctx}, i)) \)
6: end for
7: return \( R \)

Method invocation expressions

function XA(α, ID(Exp₁, . . . , Expₙ))
1: \( R \leftarrow \emptyset \)
2: for \( \text{Exp}_i, 1 \leq i \leq n \) do
3: if contains(\( \text{Exp}_i, \alpha \)) then
4: \( R \leftarrow R \cup XD(\text{Exp}_1) \)
5: end if
6: end for
7: if \( R \neq \emptyset \) then
8: return \( R \)
9: else
10: return XA(α, Parent(\( \text{ID}(\text{Exp}_1, . . . , \text{Exp}_n) \)))
11: end if

General statements

function XA(α, SMT)
1: \( R \leftarrow \emptyset \)
2: for \( \text{Exp} \in \text{Children}(\text{SMT}) \) do
3: \( R \leftarrow R \cup XD(\text{Exp}) \)
4: end for
5: return \( R \)
Integer literals
\[ \text{function } XD(IL) \]
\[ 1: \text{return } \emptyset \]

Identifiers
\[ \text{function } XD(ID) \]
\[ 1: \text{return } P(ID) \]

Parenthesized expressions
\[ \text{function } XD((Exp)) \]
\[ 1: \text{return } XD(Exp) \]

Cast expressions
\[ \text{function } XD((Type)Exp) \]
\[ 1: \text{return } \emptyset \]

Field access expressions
\[ \text{function } XD(Exp.ID) \]
\[ 1: \text{return } XD(ID) \]

Assignment expressions
\[ \text{function } XD(Exp_1 = Exp_2) \]
\[ 1: \text{return } XD(Exp_1) \cup XD(Exp_2) \]

Subtract assignment expressions
\[ \text{function } XD(Exp_1 -= Exp_2) \]
\[ 1: \text{return } \emptyset \]

Divide assignment expressions
\[ \text{function } XD(Exp_1 /= Exp_2) \]
\[ 1: \text{return } \emptyset \]

Infix addition expressions
\[ \text{function } XD(Exp_1 + Exp_2) \]
\[ 1: \text{return } \emptyset \]

Infix multiplication expressions
\[ \text{function } XD(Exp_1 * Exp_2) \]
\[ 1: \text{return } \emptyset \]

Prefix unary minus expressions
\[ \text{function } XD(-Exp) \]
\[ 1: \text{return } \emptyset \]

Postfix increment expressions
\[ \text{function } XD(Exp++) \]
\[ 1: \text{return } \emptyset \]

Continued
Conditional expressions

\textbf{function} \( XD(\text{Exp}_1 \ ? \ \text{Exp}_2 : \text{Exp}_3) \)

1: \textbf{return} \( XD(\text{Exp}_2) \cup XD(\text{Exp}_3) \)

Array access expressions

\textbf{function} \( XD(\text{Exp}_1[\text{Exp}_2]) \)

1: \textbf{return} \( XD(\text{Exp}_1) \)

Array creation expressions

\textbf{function} \( XD(\text{Type}[\text{Exp}][\{\text{Exp}_1, \ldots, \text{Exp}_n\}] \)

1: \textbf{return} \( XD(\{\text{Exp}_1, \ldots, \text{Exp}_n\}) \)

Array initialization expressions

\textbf{function} \( XD(\{\text{Exp}_1, \ldots, \text{Exp}_n\}) \)

1: \( R \leftarrow \emptyset \)
2: \textbf{for} \( \text{Exp}_i, 1 \leq i \leq n \) \textbf{do}
3: \( R \leftarrow R \cup XD(\text{Exp}_i) \)
4: \textbf{end for}
5: \textbf{return} \( R \)

Method invocation expressions

\textbf{function} \( XD(\text{ID}(\text{Exp}_1, \ldots, \text{Exp}_n)) \)

1: \textbf{return} \( XD(\text{ID}) \)

which \( \hat{\alpha} \) appears is enumerable. This additional checking for \( \hat{\alpha} \) is performed by \( EX \), which extracts the type dependent entities that require further investigation to determine if they are enumerable with respect to a particular \( \alpha \). The \( EX \) function immediately calls the helper function \( XA \), listed in Figure 2.5. This helper function makes use of another helper function, \( XD \) which is depicted in Figure 2.6. These extracted entities will be put on the worklist and eventually checked by \( EC \).

To illustrate the type checking component mechanics, I show the application of the \( EC \) predicate at each significant ancestor discovered during the evaluation the assignment \textcolor{red}{\( \text{color=RED} \)} from line 8 of the motivating example depicted in Figure 2.1(a).
The terminal expression \( \text{RED} \) within the assignment expression \( \text{color} = \text{RED} \) would have been returned by \textit{Contexts} when \( \alpha \) is \( \text{RED} \). Applying \( EC \) for this context yields the following:

\[
EC(\text{RED}, \text{RED}) \equiv EA(\text{RED}, \text{RED}) \\
\equiv EA(\text{RED}, \text{color} = \text{RED}) \\
\equiv ED(\text{color}) \land ED(\text{RED}) \\
\equiv \text{true} \land \text{true} \\
\equiv \text{true}
\]

As a result, this expression is considered “promising”. The subsequent application of \( EX \) would extract the program entity \( \text{color} \) so that all of its contexts may be checked. A demonstration of this derivation using the rules in Figures 2.5 and 2.6 is as follows:

\[
EX(\text{RED}, \text{RED}) = XA(\text{RED}, \text{RED}) \\
= XA(\text{RED}, \text{color} = \text{RED}) \\
= XD(\text{color}) \cup XD(\text{RED}) \\
= \{P(\text{color})\} \cup \{P(\text{RED})\} \\
= \{P(\text{color}), P(\text{RED})\}
\]

where \( P(\text{color}) \) denotes the program entity corresponding to the terminal identifier expression \( \text{color} \) (see Table 2.1), in this case, the field \( \text{color} \) of class \textit{TrafficSignal}. Consequently, this field will make its way to the \textit{Contexts} function via the worklist and the entire process repeats for this entity.
Consider a hypothetical assignment color=5 when α is color; here color is type dependent on the integer literal 5. Using the rules in Figures 2.3 and 2.4 yields the following derivation:

\[ EC(color, color) \equiv EA(color, color) \]
\[ \equiv EA(color, color=5) \]
\[ \equiv ED(color) \land ED(5) \]
\[ \equiv true \land false \]
\[ \equiv false \]

Thus, \( EC(color, color) \) is determined to be false. Because the type of the integer literal cannot be altered to an enum type, color also cannot be altered and should be included in set \( N \) (see line 11 in Algorithm 2.2).

There are other situations where type dependencies prevent a program entity from being enumerated. For example, consider the following statement where α is again RED:

```java
if(color == arr[RED])
    color=GREEN;
```

The derivation using the rules would consist of the following:

\[ EC(RED, RED) \equiv EA(RED, RED) \]
\[ \equiv EA(RED, arr[RED]) \]
\[ \equiv false \]

In this case, \( EC \) returns false since it would be impossible to alter the type of RED because the index to an array access must be an integral type [Gosling et al. 37]
Note that the *then* portion of the if statement is not evaluated as it is not type dependent on $\alpha$. Although $EX$ is not called when $EC$ returns *false*, $EX$ would nevertheless return $\emptyset$ upon receiving these arguments.

As an additional example, consider a conditional expression of the form:

$$EXP_1 \ ? \ EXP_2 \ : \ EXP_3$$

where $EXP_1$ is a boolean subexpression, $EXP_2$ (the *then* expression) is the subexpression that is substituted for the entire expression if $EXP_1$ evaluates to true, and $EXP_3$ (the *else* expression) is the subexpression that is substituted if $EXP_1$ evaluates to false. Care must be taken here to distinguish between each expression in which $\alpha$ may or may not appear in. If $\alpha$ only appears in $EXP_1$, $EXP_2$ and $EXP_3$ should not be checked. However, if $\alpha$ appears in either $EXP_2$ or $EXP_3$, then both of these expressions must be enumerizable. That is, the entire expression must evaluate to that of an *enum* type in either case (i.e., the *then* or *else* case).

To illustrate, consider the following conditional expression where $\alpha$ is `DECREASE_SPEED`:

```c
action = color == GREEN ? INCREASE_SPEED : DECREASE_SPEED
```

To illustrate, consider the following conditional expression where $\alpha$ is `DECREASE_SPEED`:
This produces the following derivation, where DS and IS represent the constants DECREASE_SPEED and INCREASE_SPEED, respectively:

\[
EC(\text{DS}, \text{DS}) \equiv EA(\text{DS}, \text{DS})
\]
\[
\equiv EA(\text{DS}, \text{color} == \text{GREEN} ? \text{IS} : \text{DS})
\]
\[
\equiv EA(\text{DS}, \text{action} = \text{color} == \text{GREEN} ? \text{IS} : \text{DS})
\]
\[
\equiv ED(\text{action}) \land ED(\text{color} == \text{GREEN} ? \text{IS} : \text{DS})
\]
\[
\equiv true \land ED(\text{IS}) \land ED(\text{DS})
\]
\[
\equiv true \land true \land true
\]
\[
\equiv true
\]

The extracted set of type dependent entities would be as follows:

\[
EX(\text{DS}, \text{DS}) = XA(\text{DS}, \text{DS})
\]
\[
= XA(\text{DS}, \text{color} == \text{GREEN} ? \text{IS} : \text{DS})
\]
\[
= XA(\text{DS}, \text{action} = \text{color} == \text{GREEN} ? \text{IS} : \text{DS})
\]
\[
= XD(\text{action}) \cup XD(\text{color} == \text{GREEN} ? \text{IS} : \text{DS})
\]
\[
= \{P(\text{action})\} \cup XD(\text{IS}) \cup XD(\text{DS})
\]
\[
= \{P(\text{action})\} \cup \{P(\text{IS})\} \cup \{P(\text{DECREASE\_SPEED})\}
\]
\[
= \{P(\text{action}), P(\text{IS}), P(\text{DS})\}
\]

In general, the enumerizability of particular \(\alpha\) may depend on its occurrences within comparison expressions (see the rules for equality/inequality expressions in Figure 2.3). For comparison expressions with == and !=, as long as both operand expressions are enumerizable, both will be included in the same inferred enum type, and
the integer equality/inequality in the original code will be transformed to reference equality/inequality. For <, <=, >, and >=, the refactored code can use the methods from interface java.lang.Comparable, which is implemented by all enum types in Java 5, to preserve comparability semantics amongst the inferred type’s members. This holds true so long as the inferred enum type declarations are in the order given by their original primitive representations.

An interesting case is contexts in which polymorphic behavior may occur. In these cases, we need to consider entire hierarchies of program entities. Much of the polymorphic behavior enforcement is implemented with the help of function Contexts described earlier, however, additional checks are needed within EnumerizableContext and Extract in order to ensure the preservation of program semantics. In particular, the formal parameter expressions and the method invocation expressions require additional investigation of program entities in \( \mathcal{P} \). For example, in the case of formal parameters, \( EX \) must be certain to extract the program entities embedded in the corresponding actual argument expressions for each method invocation in the method hierarchy. These rules are depicted in Figures 2.5 and 2.6.

2.4.5 Transitive Dependencies

In function Enumerizable, if either a context that is not amenable to enumeration is encountered, or one that cannot be transformed, we mark the set containing the \( \alpha \) in question as a “non-enumerizable” set (line 11 in Algorithm 2.2). If this is not the case, the algorithm proceeds to extract other program entities that are required to undergo enumeration consideration due to the enumeration of \( \alpha \) (line 14). For each of these program entities \( \hat{\alpha} \), the following steps are taken. If \( \hat{\alpha} \) is not currently
\[ \text{Enumerizable} : \mathcal{P} [\phi(\mathcal{P})] \rightarrow \mathcal{P}^{(2)} [\phi(\mathcal{P}) \cup \mu(\mathcal{P}) \cup \nu(\mathcal{P})] \]
\[ \forall k \in K [\exists X, Y \mid k \in X \land X \in \text{Enumerizable}(Y) \land K \subseteq Y] \quad (2.1) \]

\[ \iota : \phi(\mathcal{P}) \rightarrow \Sigma^* \]
\[ \forall k_i, k_j \in K [k_i \neq k_j \Rightarrow \iota(k_i) \neq \iota(k_j)] \quad (2.2) \]

\[ \mathbb{P} \text{ is the set of all legal primitive values} \]
\[ \sigma : \phi(\mathcal{P}) \rightarrow \mathbb{P} \]
\[ \forall k_i, k_j \in K [k_i \neq k_j \Rightarrow \sigma(k_i) \neq \sigma(k_j)] \quad (2.3) \]

\[ \mathbb{V} = \{\text{public, protected, private, package}\} \]
\[ \vartheta : \phi(\mathcal{P}) \rightarrow \mathbb{V} \]
\[ \forall k_i, k_j \in K [\vartheta(k_i) = \vartheta(k_j)] \quad (2.4) \]

Figure 2.7: Member constraints for transforming a group of candidate fields \( K \).

contained in an existing set (line 15), which implies that it has not previously been seen, then a new singleton in the union-find data structure is created and consequently added to the worklist (lines 16 and 17). The two sets, namely, the set containing \( \alpha \) and the set containing \( \hat{\alpha} \), are then merged on line 19 thereby capturing the transitive dependencies between each program entity. Once the computation is complete, i.e., the worklist has emptied, the sets defined implicitly by the union-find data structure are returned minus the non-enumerizable sets (line 27).

Function \textit{Enumerizable} is responsible type inferencing; that is, it ensures that the proposed transformation is type-correct. Its result is a partitioning of program entities, limited to variables, fields, and methods, that are enumerizable with respect to a given set of \textit{static final} fields (denoted as \( K \) in Figure 2.7). This essential relationship existing between each member of each enumerizable set is expressed by the first member constraint, listed as constraint (2.1) of Figure 2.7. The premise
states that \textit{Enumerizable} is a function mapping a set of primitive, static, final fields to a set of \textit{minimal} program entity sets that are enumerizable in respect to those fields. The constraint expresses that all members of each set are enumerizable with respect to the original input constants, of who are also in the set. That is, for all elements $k$ of the set of input constants $K$, there exists two sets $X$ and $Y$ such that the element $k$ is a member of the set $X$, $X$ is a valid partition of the program elements enumerizable in respect to a set of constants $Y$, and $K$ is a subset of $Y$. This last clause allows for the flexibility to enumerate only a portion of the original constants if so desired. The partitioning captures the \textit{minimal} dependency relationships between these entities; if a transformation of one of the elements occurs, then, in order to preserve type correctness, a transformation of \textit{all} elements in its set must also occur. However, further, more subtle considerations as to which sets can be transformed are required. I discuss such considerations next.

\section*{2.4.6 Semantics-preserving Constraints}

In addition to analyzing the \textit{usage} of potential enumerated type constants, in order to preserve semantics upon transformation, it is also necessary to analyze their \textit{declarations}. Returning to the \textit{Enumerize} function listed as Algorithm \ref{Algorithm 2.1}, the functions called on line \ref{line 2} enforce program behavioral preservation by excluding sets containing constants that do not meet the remaining member constraints given in Figure \ref{Figure 2.7}. Invocation of the function \textit{Unique} corresponds to the enforcement of constraint \ref{constraint 2.2}, \textit{Distinct} to constraint \ref{constraint 2.3}, and \textit{Consistent} to constraint \ref{constraint 2.4}.

Constraint \ref{constraint 2.2} expresses that, for each set to be transformed into a corresponding enum type and for semantics to be preserved, each static final field must be uniquely
named. Here, $\iota$ is a function mapping a field to its unqualified identifier. The constraint states that no two different constants can be identically named, since no two constants in an \texttt{enum} type can be identically named. This constraint is necessary as the primitive constants may have originated from different classes.

Constraint (2.3) expresses that each candidate, primitive constant must be distinctly valued so that each originally assigned primitive value will correspond to a single memory reference. Here, $\sigma$ is a function mapping a constant to its primitive value. Such a function is computable using solely compile-time information since constants in Java must be initialized to a value when they are declared. Once a constant is initialized to a value, that value will not change during the execution of the program. This constraint is necessary to preserve behavioral semantics as, while two primitive constants can be initialized to the same value, no two \texttt{enum} constants can share a memory. This is because the \texttt{enum} construct follows the Singleton pattern (Gamma et al., 1995).

Constraint (2.4) expresses that each candidate, primitive constant has a consistent visibility. In this constraint, set $\mathcal{V}$ represents the set of legal field visibilities and $\vartheta$ is function mapping a constant to its corresponding visibility, i.e., public, protected, private, or package public. Constraint (2.4) is necessary since the new \texttt{enum} types are not allowed to have constants with independent visibilities. Thus, to preserve semantics, primitive constants cannot be grouped into a new \texttt{enum} type if they are not consistently visible.

The resulting intersection of the sets abiding to each of the member constraints is then assigned back to $R$, depicted in the \textit{Enumerize} function listed as Algorithm 2.1.
At line [4] each set $T \in R$ corresponds to the program entities that will be transformed to the new language enumeration type $T$ and the transformation takes place $\forall T \in R$.

### 2.4.7 Application

I now briefly demonstrate how my algorithm would apply to the example code snippet given earlier in Figure 2.1. A schematic depicting the results of the Enumerizable function application appear in Figures 2.8 and 2.9. The figures informally represent snapshots of the state of the union-find disjoint data structure at the beginning and the end of the algorithm, respectively. Union-find data structures may be internally represented as trees and the schematic reflects this notion.
There are two different types of nodes, namely, valued and unvalued nodes. Valued nodes represent an element, i.e., a field, method, or variable, and are used in producing the output of the algorithm. Unvalued nodes consist of <UNION> nodes, which serve as logical placeholders marking points in which the sets were merged. Edges connect nodes belonging to the same set. Edge directions depict the order in which nodes were discovered during execution of the algorithm. Furthermore, edge directions do not necessarily denote transitivity in a particular direction; transitive relationships in respect to enumerization are bidirectional.

In Figure 2.8 the initial input elements are used to seed the enumeration sets and the application of Enumerizable grows the sets as seen in Figure 2.9. During growth, the sets may be combined due to transitivity of equality/inequality comparisons on both left- and right-hand sides and/or assignments on the right-hand side. The resulting sets depicted in Figure 2.9 shows that 4 of the original 8 sets have been merged. Sets containing shaded elements designate enumerizable sets, i.e., sets that contain all elements whose usages are amenable to enumerization. Sets not shaded signify sets that contain at least one element not amenable to enumerization, such as the set containing the element MAX_SPEED.

Note that these sets portray the minimal dependency information among their elements, therefore, they may be further merged but not split. Also notice that the results produced by the algorithm as applied to the drive-by-wire example (see Figure 2.1) are not entirely optimal. Specifically, the automobile action DECREASE_SPEED is contained in a different set than that of the other automobile actions due to the current transitive nature of the elements. Surely, when performing the language enumeration type transformation, it is desirable that all automobile action be grouped together in
the same language enumeration type. For future work, I intend to examine how such a result can be automatically suggested by the refactoring. These plans are discussed in Section 2.7.

2.5 Experimental Study

In this section, I outline a experimental study aimed to quantitatively and qualitatively evaluate the effectiveness of the approach described in the previous section in migrating instances of the weak enum pattern to proper language enumeration types. Section 2.5.1 discusses facets of an implementation of the approach as an Eclipse plug-in, while Section 2.5.2 details the results of the experiment. Section 2.5.3 summarizes the findings and comments on the useful of the approach in a real-world setting.

2.5.1 Implementation

The algorithm for migrating instances of the weak enum pattern to proper language enumeration types presented in the previous section has been implemented as a plug-in in the popular Eclipse IDE. Eclipse ASTs with source symbol bindings were used as an intermediate program representation. The plug-in is built over an existing refactoring framework (Bäumer et al. 2001) and is coupled with other refactoring support in Eclipse. To increase applicability to real-world applications, the closed-world assumption described in Section 2.4.1 has been relaxed. For example, if the tool encounters a variable that is transitively dependent upon an element outside of the source code being considered, this variable and all other entities dependent on it are conservatively labeled as non-enumerizable.
The implementation has been released as an open source software project being hosted by Google Code and is publicly available for download at http://code.google.com/p/constants-to-enum-eclipse-plugin.

Although the tool is currently in a research prototype stage, it remains under active development by several developers and contains several useful features, such as a refactoring wizard, before-and-after refactoring preview pane, and automated code transformation manifested as AST rewriting. Features remaining to be implemented include a full test suite with regression tests and conformance with other refactoring plug-ins in Eclipse, e.g., undo functionality and refactoring history. The tool also has been a proposed project in the Google Summer of Code 2009 and 2010 competitions.

The main development focus has been on producing an intuitive user-interface that provides a mechanism for developers to interactively group constants together to create the new enum type. Figure 2.10 portrays a screen shot of the current refactoring wizard when applied to the following source code, more precisely, when selecting the RED and GREEN constants, and Figure 2.11 depicts the refactoring preview pane:

```java
package p;
public class A {
    public static final int RED = 0;
    public static final int GREEN = 1;
    int x() {
        int a = RED;
        a = GREEN;
        return a;
    }
    int y() {
        return x();
    }
}
```

5 http://code.google.com
6 http://code.google.com/soc
Listing 2.1: A simple example to demonstrate features of the Convert Constants to Enumerated Types Eclipse refactoring plug-in.

For future work, we plan to incorporate information visualization to enable developers to see the strength of relationships between the constants so that they may make an informed decision of how they should be grouped into enumerated types. In the current state of the tool, constant are grouped in their minimal type-dependent sets such that if one constant in the set is refactored, all other constants in the same...
set must also be refactored to the same enumerated type. Additional plans include but are not limited to providing undo functionality, refactoring history rollback, and refactoring scripting support. In addition, improvements to the refactoring wizard include conveniently allowing the user to add additional constants to the newly created enum type that were not originally part of the input set. This would better situate **Convert Constants to Enum** to conform to standard refactoring tools in

![Figure 2.11: Screen shot of the enum refactoring preview pane.](image)
<table>
<thead>
<tr>
<th>subject</th>
<th>KLOC</th>
<th>classes</th>
<th>prim</th>
<th>cands</th>
<th>enum</th>
<th>uses</th>
<th>rtypes</th>
<th>time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ArtOfIllusion</td>
<td>75</td>
<td>378</td>
<td>333</td>
<td>77</td>
<td>77</td>
<td>111</td>
<td>46</td>
<td>207</td>
</tr>
<tr>
<td>Azureus</td>
<td>272</td>
<td>1894</td>
<td>1255</td>
<td>399</td>
<td>347</td>
<td>635</td>
<td>173</td>
<td>1269</td>
</tr>
<tr>
<td>Java5</td>
<td>180</td>
<td>1586</td>
<td>1299</td>
<td>557</td>
<td>450</td>
<td>572</td>
<td>363</td>
<td>760</td>
</tr>
<tr>
<td>JavaCup</td>
<td>6</td>
<td>41</td>
<td>55</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>19</td>
</tr>
<tr>
<td>JDepend</td>
<td>3</td>
<td>28</td>
<td>13</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>JFlex</td>
<td>10</td>
<td>46</td>
<td>140</td>
<td>24</td>
<td>19</td>
<td>27</td>
<td>9</td>
<td>75</td>
</tr>
<tr>
<td>JFreeChart</td>
<td>71</td>
<td>420</td>
<td>153</td>
<td>36</td>
<td>24</td>
<td>43</td>
<td>12</td>
<td>128</td>
</tr>
<tr>
<td>JGap</td>
<td>6</td>
<td>137</td>
<td>25</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>JGraph</td>
<td>14</td>
<td>91</td>
<td>25</td>
<td>6</td>
<td>3</td>
<td>6</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>JHotDraw</td>
<td>29</td>
<td>496</td>
<td>34</td>
<td>11</td>
<td>11</td>
<td>24</td>
<td>8</td>
<td>14</td>
</tr>
<tr>
<td>JUnit</td>
<td>8</td>
<td>271</td>
<td>7</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>JWPS</td>
<td>20</td>
<td>155</td>
<td>156</td>
<td>76</td>
<td>64</td>
<td>102</td>
<td>25</td>
<td>60</td>
</tr>
<tr>
<td>SableCC</td>
<td>29</td>
<td>237</td>
<td>16</td>
<td>8</td>
<td>8</td>
<td>10</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>Tomcat6</td>
<td>153</td>
<td>1164</td>
<td>738</td>
<td>344</td>
<td>335</td>
<td>400</td>
<td>255</td>
<td>346</td>
</tr>
<tr>
<td>Verbose</td>
<td>5</td>
<td>41</td>
<td>10</td>
<td>6</td>
<td>6</td>
<td>15</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>VietPad</td>
<td>11</td>
<td>84</td>
<td>36</td>
<td>17</td>
<td>17</td>
<td>22</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>Violet</td>
<td>7</td>
<td>73</td>
<td>36</td>
<td>14</td>
<td>13</td>
<td>20</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td><strong>Total:</strong></td>
<td>899</td>
<td>7142</td>
<td>4331</td>
<td>1585</td>
<td>1384</td>
<td>1999</td>
<td>915</td>
<td>2227</td>
</tr>
</tbody>
</table>

*java package included in Java 5 JDK.

Table 2.2: Enumerization experimental results.

Eclipse. Work is currently underway to include the tool in the standard set of refactoring distributions distributed in Eclipse (see [https://bugs.eclipse.org/bugs/?id=200152](https://bugs.eclipse.org/bugs/?id=200152)).

2.5.2 Experimental Evaluation

To evaluate the effectiveness of the algorithm in migrating instances of the weak enum pattern to proper language enumeration types, seventeen open source Java applications and libraries, listed in Table 2.2, were selected as experimental subjects. The second column in the table shows the number of non-blank, non-comment lines of source code, which range from 3K for JDepend to 272K for Azureus. The third column shows the number of class files after compilation. For each subject, the
analysis was executed five times on a 2.6 GHz Pentium 4 machine with 1 GB RAM. The average running time in seconds is shown in column \textit{time} in Table 2.2. On average, the analysis time was 2.48 seconds per KLOC, which is practical even for large applications.

Column \textit{prim} shows the number of \texttt{static final} fields of primitive types\footnote{Excludes \texttt{boolean} types.}. These fields are separated into two categories. First, certain fields \textit{definitely} cannot be refactored, because the semantics of the program depends on the specific, actual values of these fields. These fields include those that were either directly or transitively dependent on operations that utilized their exact value or created a transitive dependency on an entity that could not be refactored. A complete list of filtered contexts is provided in Table 2.3. The first column, \textit{Filtered context}, displays the contexts which were filtered. The second, \textit{Example Operations}, provides a subset of operations that fall into each context, and column \textit{Example Uses} provides an example of each context.

Note here that the \texttt{v} variable depicted in the examples represents either a \texttt{static final} fields or an entity transitively type dependent on such a field. Since the weak enum pattern only allows the use of literals in the declarations of enumeration fields, other contexts that utilize them were filtered as shown in rows \textit{Character literal} and \textit{Number literal}. Since the semantics of an array access relies on the particular value of the index, any field used as such cannot be refactored. Similarly, fields whose values are utilized in the creation of a new array cannot be refactored (rows \textit{Array access} and \textit{Array creation}). There are multitudes of mathematical operations, which, of course, rely on the values of the variables being manipulated. These operations are shown in rows \textit{Infix expressions}, \textit{Postfix expressions}, and \textit{Prefix expressions}. Some of these...
Filtered Context | Example Operations | Example Uses
---|---|---
Character literals | ==, =>, !=, <, >, <= | v == ‘c’
Number literals | ==, =>, !=, <, >, <= | v != 28
Array accesses | [] | x[ v ]
Array creation | new int[], new double[] | int x[] = new int[ v ]
Infix expressions | +, -, /, *, | x = v + s
Postfix expressions | ++, -- | v++
Prefix expression | ~, ++, --, !, +, - | --v

Table 2.3: Contexts not considered valid for enumerated types.

operations may be valid for certain extensions to weak enum compensation pattern, such as those that employ a bit-vectoring over their enumeration values. Refactoring constants (transitively) involved in bit-vector operations is a planned contribution for future work and discussed in Section 2.7.2. This category also includes the fields that cannot be refactored due to lack of access to source code (e.g., a field passed as a parameter to a method defined in a library whose source code is not available).

The remaining fields are categorized to be candidate fields. The number of candidate fields per subject is shown in column cands. The fact that the actual values of these fields do not directly affect the semantics of the program provides a strong indication that they are playing the role of enumerations in the weak enum pattern. The set of candidate fields along with their corresponding, transitive type dependent entities represent the minimal set of elements a programmer would have to investigate as a prerequisite for performing the refactoring. Note that although these sets are minimal, for three of the subjects, they still contained well over 300 elements, and several others contained over 50 elements.
The number of fields that the plug-in could safely refactor is shown in column \textit{enum} in Table 2.2. The results show that the approach was able to refactor 87\% of the fields that could possibly be participating in the weak enum pattern. The tool was unable to refactor the remaining 13\% of fields because either they or an element of their dependency sets was used in explicit cast expressions. I conservatively choose not to refactor elements used in cast expressions due to the existence of possible side effects on the values of variables though narrowing conversions. For example, consider the following code:

\begin{verbatim}
short z = 128;
byte x = (byte)z;
\end{verbatim}

This is a valid cast in Java, but this cast will result in \texttt{x} having the decimal equivalent value of $-128$ and \textit{not} 128. Clearly, not accounting for such an occurrence prior to refactoring could lead to significant changes in program semantics upon migration. Detecting such changes due to explicit casts is beyond the scope of the work being considered in this thesis.

Fields are not the only program entities whose type requires alteration. Column \textit{uses} in Table 2.2 shows the total number of declaration sites that must be modified to accommodate the enumerization. The numbers motivate the need for automated tools to tackle such a problem. In particular, the large applications require hundreds of code modifications (e.g., over 600 for Azureus). These code modifications are spread across many classes and packages and occur in many distinct methods. Attempting to identify the needed modifications manually would be a labor-intensive and error-prone task.
Continuing with Table 2.2, column \textit{rtypes} shows the number of resulting \textit{enum} types produced by the tool. Note that the number of types is relatively close to the number of enum fields. This indicates that there are few, actual constants per enum type, on average about 2.2 per type. This number may not reflect the number of weak enum pattern instances intended by the programmer. My algorithm is conservative in its type creation, only grouping fields that share transitive dependencies. These are the only fields that \textit{must} share the same enum type upon refactoring. However, given the current state of the program source, dependencies may not exist between all enumerations intended to be grouped as one type. For the running example, \texttt{DECREASE\_SPEED} should intuitively be grouped with the other vehicle actions. Unfortunately, since it is not currently being referenced by the code, it does not share a dependency with any of the other fields, and, as a result, it is assigned a singleton set (as shown in Figure 2.9). Clearly, in this case, no algorithmic method could guarantee the exact grouping intended by the programmer; however, there are various heuristics that may be employed to better approximate the intended types. I plan to employ such heuristics to enable the tool to produce results that more closely mimic results that would have been produced manually by the programmer for future work. These plans are discussed in more detail in Section 2.7.1.

2.5.3 Summary

The experimental results indicate that the analysis cost is practical, that the weak enum pattern is commonly used in legacy Java software, and that the proposed algorithm successfully refactors a large number of fields (and their dependent entities) into enumerated types. Furthermore, these results suggest that the approach is
promising in its applicability to real-world software and useful in real-world software development settings.

2.6 Related Work

Both Fowler (1999) and Kerievsky (2004) present the refactoring entitled Replace Type Code With Class. Both detail a series of steps involved in transforming type codes (entities subscribing to what we label as the weak enum pattern in this thesis) into instances of custom, type-safe classes utilizing the Singleton pattern. Bloch (2001) presents a similar solution. While the pattern describes an enum class that seems effective in regards to the same criteria I have presented in this thesis, the refactoring process is manual and the transformation is not to language enumerated types. Most importantly, the developer is required to possess a priori knowledge of exactly which fields are potentially participating in the type code pattern in order to perform the refactoring. My proposed approach does not require such knowledge and is completely automated. That is, my approach infers, in an automated fashion, such fields. Furthermore, the developer is presented with the type-dependent groups of the fields, which may span multiple classes and packages.

Tip et al. (2003) propose two automated refactorings, Extract Interface and Pull Up Members, both well integrated into the Eclipse IDE. These refactorings deal with generalizing Java software in an effort to make it more reusable. Although this proposal shares similar challenges with my approach in respect to precondition checking and interprocedural dependency analysis, there are several key differences. The generalization approach manipulates the interface\(^\text{8}\) of reference types along with

\(^8\) The term interface is used here not to solely denote that of Java interfaces, but instead to denote the broader notion of interfaces which, in Java, would also include class declarations.
the means in which objects communicate through those interfaces, as my approach entails transforming primitive type entities to reference types. Moreover, a method based on type constraints (Palsberg and Schwartzbach, 1994) is used to resolve dependencies amongst program entities. Sutter et al. (2004) also use type constraints in addition to profile information to customize the use of Java library classes. A type constraint approach would have also been conceivable for my work in that similar type constraints may have been formed for primitive types. Nonetheless, a type constraint-based approach for primitive transformation may have proven to be excessive since primitive types do not share many of the same relationships as reference types (e.g., sub-typing). Therefore, I preferred more of a type checking approach as opposed to constraint solving.

Several other approaches (e.g., Donovan et al., 2004; Fuhrer et al., 2005; Kieżun et al., 2007; Tip et al., 2004; von Dincklage and Diwan, 2004) exist to migrate legacy Java source to utilize new Java 5 language features, in particular generics. Although both generics and language enumeration types serve to improve type safety, the two features are conceptually different and face unique challenges in automated migration. The instantiation problem (Fuhrer et al., 2005) entails inferring generic type arguments for generic class instances. The parameterization problem (Kieżun et al., 2007) necessitates inferring generic type parameters for non-generic class declarations. Various challenges include preserving program erasure (Bracha et al., 2003), sub-typing compatibility, inferring wild card types, etc. However, my proposal for inferring enumerated types, although not being required to address such issues, must consider other such situations. First, enumeration requires introducing a new type in the original source as opposed to introducing a type parameter or argument for an existing type.
Second, when refactoring primitives, one must consider many additional operations that may involve primitive entities that are not available to reference types. Thirdly, the dependency flow must also be taken in account across these operations. For example, in my proposal, type dependence not only flows from assignments but also from comparisons.

Steimann et al. (2003, 2006) propose an approach to decouple classes with inferred interfaces. Similar to my approach, a new type, i.e., the inferred interface, is introduced in the source, and compile-time types of program entities are altered because of the refactoring. Additionally, Steimann et al. (2003, 2006) do not leverage constraint solving mechanisms, instead, they utilize a static analysis based on the approach of Dean et al. (1995). Unlike this proposed approach, however, my approach must consider more than the transitive closure of assignments beginning on the right-hand side of assignment operations. Again, enumerization entails bidirectional dependencies not only over assignments but also over comparisons.

Automated usage analysis and type inferencing techniques similar to my proposal also exist for other languages. Eidorff et al. (1999) demonstrate a Year 2000 conversion tool utilizing type inferencing techniques for correcting problematic date variables in COBOL, a weakly-typed programming language. Ramalingam et al. (1999) also exploit usage analysis techniques to identify implicit aggregate structure and programmer intent of COBOL program entities not evident from their declarations.

Proposals for identifying enumerated types exist for both COBOL and C. Although my work applies to a significantly different source language, i.e., Java, methods for identifying enumerated types in these legacy systems share similar challenges. Deursen and Moonen (1998) present a general approach utilizing judgments and rules
for inferring type information from COBOL programs. An in-depth empirical analysis is presented {Deursen and Moonen} (1999). Both COBOL and Java $\leq 1.4$ do not provide language facilities for enumerated types, and both my proposal and that of {Deursen and Moonen} (1998) use a flow-insensitive, interprocedural data flow analysis to discover program entities intended to represent enumerated types. However, my proposed approach is focused more on the migration of these entities to a specific language enumerated type construct that contains corresponding, preexisting constraints (see Section 2.4.6). As a result, my proposal deals with different semantic preservation issues upon transformation, insuring that substitution by the new construct will produce a program that behaves identically to the original. Moreover, refactoring primitive types to reference types presents unique challenges as objects in Java cannot share the same memory location; thus, grouping program entities interacting with values from similar literals into corresponding types would not produce an applicable solution. Likewise, my approach must consider modern programming language features such as polymorphism and function overloading throughout source analysis and semantic preservation efforts.

{Gravley and Lakhotia} (1996) present an approach for identifying enumerated types in C programs that utilize a pre-compiler directive pattern similar to the weak enum pattern. While {Gravley and Lakhotia} (1996) and the approach proposed here share similar goals, {Gravley and Lakhotia} (1996) only analyze constant declarations and not the contexts where the constants and their (transitively) type dependent program entities appear. Furthermore, as will be discussed more thoroughly in the next section, the approach of {Gravley and Lakhotia} (1996) may be appropriately adapted to

\footnote{Interprocedural analysis would be referred to as \textit{inter-program} or \textit{inter-module} analysis in the case of COBOL.}
enhance the results of my algorithm by leveraging declaration characteristics during grouping.

2.7 Future Work

In this section, I detail plans for future work on the enum refactoring approach.

2.7.1 Enhanced Enumeration Constant Grouping

As mentioned earlier, the approach presented in this chapter has the potential to produce correct but suboptimal results. That is, the automatic refactoring technique may not always replicate the quality of results that, perhaps, a human developer would have produced if the refactoring was performed manually. This is due to the fact that my approach groups constants solely based on type dependencies, i.e., constants are grouped in minimal sets such that if one constant was to be converted to an enum type, then all constants (as well as transitively type dependent entities like method return types, fields, parameters, etc.) must also be converted to the same enumerated type.

However, situations exist where constants should be grouped together for reasons other than type dependence in the current state of the program. Specifically, constants may be related in ways other than their (transitive) usages in the current program text. Relating the concept to the example given in Section 2.2, all automobile actions should be grouped into a common enumerated type after the refactoring has been performed. Yet, not all automobile action constants are associated with type dependencies, thus, a singleton set was produced containing one automobile action constant.
No algorithmic approach can determine the actual intent of the developer using the tool. However, I intend to apply various heuristics that to better approximate these intended types. In fact, there are a plethora of heuristics found in the work of Gravley and Lakhotia (1996) that may be adapted for my approach. For example, I could take into account the lexical proximity of the original field declarations with the assumption that related constants are declared closely to one another. This was the case in the motivating example given earlier. Other lexical-based heuristics would include common prefixes and suffixes in the constant name (generally, edit distance measurements), ontology-based grouping (e.g., the constants GREEN, YELLOW, and RED would be related as common colors in an ontology), etc.

Other heuristic approaches may be specific to the Java programming language. One heuristic would be that all constants declared in the same class should be grouped together in an enum type. Another approach would be to state that all constants declared in classes belong to the same package should be grouped together. Other, more complex heuristic approaches may include common field annotations (if the original source code is written in Java 5 or greater), and string similarities in the javadoc comments for the original declared constants.

2.7.2 Refactoring Other Enumeration Compensation Patterns

While the weak enum pattern is indeed a popular way to represent enumerated types in legacy Java, other patterns also exist which require alternative algorithmic considerations to migrate them to language enumerated types. The approach presented here specifically dealt with the migration of primitive types that are being used
to simulate enumerated types. However, reference types, e.g., `java.lang.String` and `java.util.Date`, may also be used for these purposes.

For future work, I intend to develop ways to automatically identify and migrate reference types. Doing so will involve careful consideration and examination of all contexts where the corresponding program entities appear to preserve reference semantics upon transformation. The algorithms presented in this chapter will have to be altered to deal with contexts in which reference types may appear. Essentially, completing this task will require the `EA`, `ED`, `XA`, and `XD` to be extended with additional type inference rules to identify reference types that can be safely refactored. Including alternative patterns such as those dealing with reference types would improve the real-world applicability of the approach.

Additionally, some of filtered operations appearing in Section 2.5 may be, in fact, valid for certain extensions to weak enum compensation pattern. These patterns may include contexts that employ bit-wise operations involving constants and/or their transitively type-dependent entities. During the experimentation, I encountered several cases where constants were combined using bit-wise operations either produce new “constants” or denote that either of two constants is acceptable. Relating the motivating example, an instance of the former would be the following expression where colors are mixed:

```java
int newcolor = RED & GREEN;
```

Here, the variable `newcolor` is symbolizing the result of mixing the colors red and green, thus producing the color yellow. An example of the latter case pointed out above would consist of an expression containing a bit-wise or operation, producing a kind of “union” of two constants. In the future, I will explore ways of adding...
such contexts to my algorithm. One danger, however, is that I must be certain that
the bit-wise operations are indeed being used to intersect and union constants per
the above described examples. In other words, the algorithm must recognize when
the constants are being used for true bit-wise operations (in the traditional sense)
and when they are being used to manipulate enumeration constants. In addition,
the refactored code must compensate for the bit-wise operations, as these operations
are not valid for reference types (which the new enum Java construct amounts to).
As such, a more complex refactoring is required that would automatically produce
additional methods in the enum declaration to combine the constants in the same way
as the original code.

2.7.3 Thorough Empirical Evaluation

As remarked earlier, my evaluation reports on the ability of my approach to
transform static final fields that may be participating in the weak enum pat-
tern, i.e., the fields that pass the stated preconditions. By preconditions, I am refer-
ring to refactoring-based preconditions, which are manifested in my algorithm as the
EnumerizableContext predicate evaluating to true.

However, to evaluate the true applicability of the approach in a real-world setting,
it would be necessary to obtain the original developer’s input as to whether or not
the fields that can be transformed by the approach while preserving semantics are
the same ones that were originally intended to be enumerated types. Moreover,
a thorough investigation of the usefulness of the approach would account for the
quality of the transformation in terms of its resemblance to results that would have
been produced manually by the developer, as discussed in section 2.7.1. For future
work, I plan to perform a user study that answers these questions and report on their results.

My plans for the user study consist of implementing an automatic feedback system to be included with the plug-in, much like the user study that has been previously performed by Mick et al. (2007). After the refactoring, I plan to have a dialog window appear asking the developer to participate in the study. Study design considerations, e.g., the types of questions that the dialog will ask, would benefit from referring to the work of Chin et al. (1988); Kersten and Murphy (2006). I will then collect these results in a database connected to Internet in order to obtain results from developers around the globe.

2.8 Conclusion

In this chapter, I have presented a semantic preserving, type inferencing algorithm that migrates legacy Java code employing the weak enum pattern to instead utilize the modern, type-safe enum language construct introduced in Java 5. The algorithm has been implemented as a plug-in for the popular Eclipse IDE and evaluated on seventeen open source applications. The experiments indicate that not only did the tool scale well to large applications but also was able to refactor 87% of all fields that could possibly be participating in the weak enum pattern. I have also proposed future work that involves enhanced constant grouping to better resemble human-produced results, refactoring of other compensation patterns and/or extensions to the weak enum pattern, and an in-depth empirical evaluation utilizing a user study.
CHAPTER 3: Recovering Pointcut Expressions in Evolving Aspect-Oriented Software

3.1 Introduction

AOP 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 (Kiczales et al., 2001), 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. I 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 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 alterations, including the addition of \textit{new} elements in which existing CCCs should also apply. Without \textit{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 \textit{along} with the base-code; in these cases, the PCE is said to be \textit{fragile}. Hence, the \textit{fragile pointcut problem} \cite{Koppen2004} 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 \cite{Cazzola2006, Klose2005, Ostermann2005, Sakurai2008, Seiter2007}, limiting the scope of where advice may apply through more clearly defined interfaces \cite{Aldrich2005, Gudmundson2001}, enforcing structural and/or behavioral constraints on advice application \cite{Griswold2006, Khatchadourian2008, Sullivan2005}, making points where advice may apply more explicit in the base-code \cite{Hoffman2007}, or by removing PCEs altogether \cite{Rajan2008}. 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 \cite{Sun2010} 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 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 a 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 thesis 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, I 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 (textually) includes 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.
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 plausible that these relationships should persist in subsequent versions. Consequently, my proposal involves constructing patterns that describe these kinds of relationships, assessing their expressiveness in comparison with the PCE used to construct them, and associating them with the PCE so that they may be applied to later base-code versions to offer suggestions of new join points that may require inclusion.

My insight into the fragile pointcut problem is as follows. CCCs tend to cross-cut 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. My 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.

### 3.1.1 Contributions

This work has made the following specific contributions:

*Commonality identification.* I present a parameterized, heuristic algorithm that automatically derives arbitrarily deep structural patterns inherent to program elements corresponding to join points selected by the original PCE. This allows join points to be suggested that may require inclusion into a revised version of the PCE, ensuring that evolutionary changes can be correctly applied by mechanically assisting the developer in maintaining PCEs.
**Correlation analysis.** I empirically establish that join points selected by a single PCE typically portray a significant amount of unique structural commonality by applying the initial algorithm to automatically extract and, thereafter, analyze patterns using PCEs contained within single versions of twenty-three AspectJ programs. I found that the derived patterns were able to, on average, closely produce the majority of join points selected by the analyzed PCE in the original base code version with low $\alpha$ (false positive) and $\beta$ (false negative) error rates of 18% and 16%, respectively.

**Expression recovery.** To ensure the applicability and practicality of the approach, I implemented the algorithm as an Eclipse IDE\(^{10}\) 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. I found that, in exploiting the extracted patterns, the tool was able to accurately and automatically infer 90% 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%. 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.

\(^{10}\)http://www.eclipse.org
3.1.2 Organization

This chapter is organized as follows. Section 3.2 presents a motivating example that features a fragile PCE. Section 3.3 highlights the key algorithmic facets of my approach, while Section 3.4 details its implementation and evaluation. Related work is discussed in Section 3.5, future work is presented in Section 3.6, and Section 3.7 concludes.

3.2 Pointcut Fragility Example

Listing 3.1 shows an example AspectJ code snippet for hypothetical drive-by-wire programming of an all-wheel drive, hybrid vehicle (line 2) that draws power from two different sources, namely, a diesel engine (line 25) and an electric motor (line 32), both of which contribute to the overall speed (line 3). Fuel is distributed to the engine via the method DieselEngine.increase(Fuel) (line 27), while electricity is distributed to the motor via the method ElectricMotor.increase(Current) (line 34), whose method bodies are abbreviated.

The classes conform to the Observer pattern (Gamma et al., 1995). The DieselEngine and the ElectricMotor notify the HybridAutomobile of any change made to the energy consumption of the respective components. Here, the HybridAutomobile is considered the subject, and it is composed of a DieselEngine and a ElectricMotor. The corresponding instance fields are omitted in the HybridAutomobile for brevity. However, both the DieselEngine and ElectricMotor have a backwards reference (lines 26 and 33 respectively) to the HybridAutomobile they compose.

11This example was inspired by the authors' work at the Center for Automotive Research at Ohio State University.
package p;

class HybridAutomobile {
    private double overallSpeed; //...
    // Sets the new speed for changes in fuel.
    public void notifyChangeIn(Fuel fuel) {
        this.overallSpeed += fuel.calculateDelta(this);
        /* Update attached observers ... */
    }
    // Sets the new speed for changes in electricity.
    public void notifyChangeIn(Current current) {
        this.overallSpeed += current.calculateDelta(this);
        /* Update attached observers ... */
    }
    // Sets the new speed directly.
    public void notifyChangeIn(double mph) {
        this.overallSpeed += mph;
        /* Update attached observers ... */
    }
    public double getOverallSpeed() {
        return overallSpeed;
    }
}

class DieselEngine {
    private HybridAutomobile car;
    public void increase(Fuel fuel) {
        //...
        this.car.notifyChangeIn(fuel);
    }
}

class ElectricMotor {
    private HybridAutomobile car;
    public void increase(Current current) {
        //...
        this.car.notifyChangeIn(current);
    }
}

class Dashboard {
    private HybridAutomobile car;
    //...
    public void update() {
        this.display(car.getOverallSpeed());
    }
}

Listing 3.1: Running pointcut fragility example: hybrid automobile programing.
aspect SpeedingViolationPrevention {
    void around() :
        execution(void increase(Energy+)) {
            /* ...
            */
        }
    }

Listing 3.2: Speeding prevention aspect.

Once the HybridAutomobile is notified about energy consumption changes, it computes its new overall speed (lines 6–7, 12–13) and updates any attached observers, e.g., the Dashboard (line 39). An accessor method (line 21) retrieves the value of the private instance field overallSpeed, which the method Dashboard.update() invokes (line 43) 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 (Listing 3.2) is introduced to augment the existing functionality of the programming depicted in Listing 3.1 by limiting the vehicle’s energy intake by declaring appropriate around advice (lines 2–4) which conditionally bypasses the execution of methods that contribute to the vehicle’s overall speed. The points at 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, DieselEngine.increase(Fuel) and ElectricMotor.increase(Current). These methods have been underlined in Figure 3.1. 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

12For 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. I have omitted these designators for presentation purposes.
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 (Listing 3.1) evolves to accommodate a new vehicle energy source, namely, a fuel cell, resulting in the creation of a FuelCell class (Listing 3.3). In contrast to the existing energy sources, requests to increase power from the FuelCell require passing a numerical (double) parameter, which is the amount of acceleration (in miles/hour) that should result from the FuelCell internally generating power, to a method (line 4) that, in turn, notifies the HybridAutomobile of the change directly (line 6).

Intuitively, the SpeedingViolationPrevention aspect should also apply to the execution of this method, however, the PCE listed on line 3, Listing 3.2 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 sub-classes, 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. I will continue to use the hybrid-powered vehicle example to demonstrate how the proposed approach can be used to help identify such new join points in an automated fashion.

3.3 Harnessing Commonality

In this section, I present a parameterized, heuristic algorithm that assists developers in maintaining PCEs upon changes to the base-code by inferring new join points that may require inclusion. The algorithm works by discovering structural commonality between program elements corresponding to join points captured by a PCE in a particular software version. For instance, notice in the previous example that the two methods, namely, `DieselEngine.increase(Fuel)` and `ElectricMotor.increase(Current)`, whose corresponding method executions were selected by the PCE listed on line 3, Listing 3.2 are both declared in classes contained in package `p`. Additionally, considering solely the code snippet characterized in Listing 3.1 both method bodies contain calls to methods, namely, `notifyChangeIn(Fuel)` and `notifyChangeIn(Current)`, respectively, that read from the field `HybridAutomobile.overallSpeed`. In my approach, such commonality is captured by constructing patterns that abstractly describe the kinds of relations that the program elements have in common. The extracted patterns are then applied to later versions to offer suggestions of new join points that require inclusion as similar commonality may be exhibited in the future.
3.3.1 High-level Overview

To assist developers in maintaining PCEs over the lifetime of AO software, my approach is conceptually divided into two phases: analysis and rejuvenation. The analysis phase, whose flow diagram is depicted in Figure 3.1, 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 of the graph 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 Dagenais et al. (2007)) I 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.

I envision my approach to be most helpful in scenarios where a developer performs a series of changes to the base-code and then, at some time in the future, compares PCEs with the new base-code version to ensure that join points are captured correctly.

The benefits of the approach include information pertaining to:

(i) whether a certain PCE requires updates due to changes in the base-code, and

(ii) if a PCE update is required, suggestions as to which join points should be included.

Note that a natural alternative would consist of suggestions of join points that should not be included in an updated version of the PCE. These are join points that do not appear as suggestions, but could be captured by the original version of the PCE. However, my approach is focused on the situation where additional join points are required to be captured by the PCE but the original construction of the PCE is such that these join points are not captured. The reverse situation, i.e., join points captured by the original PCE are intended not to be included as a result of a new

\[13\] There has been an ongoing, controversial debate regarding the obliviousness facet AOP affords developers (Filman and Friedman [2000] Steimann [2006]). Indeed, the ability to concentrate on “one concern at-a-time” is very much part of the main motivation for AOP, and whether or not current AO languages materialize this notion (or whether they should completely) is a topic of much controversy in the community. So as long as fragile PCEs are possible, i.e., the PCE language includes the ability to “quantify” over the base-code execution, it will be necessary to verify that join points are captured correctly. This is the exact situation where I deem my proposed approach most applicable and useful.
Figure 3.2: Phase II: Pointcut rejuvenation flow diagram.
base-code version, but the original construction of the PCE is such that these join points are captured, is not directly considered in my approach. Whether these two situations are similar and the considerations needed for the latter situation are possible topics for future work.

The rejuvenation phase, whose flow diagram is depicted in Figure 3.2 is triggered previously to the developer manually altering the PCE so that automated assistance in performing the updates correctly can be provided (step 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 (step 2). These join points are the ones related to program elements that share structural 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 (step 4), and the list of all suggestions is sorted in decreasing order of confidence (as a result of step 3). The developer then adjusts the PCE to either incorporate or exclude the desired join points (steps 5 and 6) based on the suggestions.

In the remainder of this section, I discuss the algorithmic details of these steps, while the succeeding section describes the implementation.

3.3.2 Assumptions

Prior to continuing our discussion, I first state several simplifying assumptions about the underlying source code to be analyzed; I discuss in Section 3.4.1 how many of these have been relaxed in the implementation and how others can be dealt with as

\[\text{The evaluation my technique, however, is not biased towards assessing the performance of the implementation in a particular situation.}\]
future work in Section 3.6. Firstly, I assume that the input PCE is initially correctly
specified, i.e., it selects all and only intended join points of the original program. This
ensures that the structural commonality exhibited by the corresponding program el-
ements is correctly related to the input PCE. Furthermore, I assume that inter-type
declarations (static crosscutting) are not utilized by the analyzed aspects. Inter-
type declarations allow aspects to introduce and modify facets of the base code, e.g.,
member introduction, class hierarchy alteration, interface implementation injection,
exception softening, existing at compile-time. This assumption helps simplify the al-
gorithm presentation. Modifying current algorithm to handle inter-type declarations
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 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 me from resolving the targets of proceed calls, which may exist in around
advice. I discuss adding advice bodies to the analysis in Section 3.6. Lastly, I assume
that I can accurately resolve the declaration of the advice a PCE is bound to across
varying versions of the software. This assumption may be invalidated via the use
of refactorings, e.g., member relocation, being applied in between software versions.
Section 3.6 discusses plans for how my approach can be extended to cope with this
issue.
3.3.3 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 [Robillard and Murphy, 2002]. Concern graphs have been used in previous work, e.g., Robillard (2006) use concern graphs to discover, describe, and track concerns in evolving source code, as they allow for succinct program representations. I have chosen to use concern graphs since they include information about the structure of programs, and I am interested in unveiling underlying structural patterns. I extended concern graphs with several elements found in current Java languages, e.g., annotations, and adapted them for use with AOP.
More formally, I specify an extended concern graph \( CG^+ \) to be a labeled multidigraph consisting of a 4-tuple \( CG^+ = (V, E, R, \ell) \). The set of vertices \( V \) represent program elements contained within the analyzed program, specifically, packages, classes, interfaces, enumeration types, annotations, methods, and fields. I do not consider local variables and other parameters in my analysis as crosscutting concerns tend to crosscut a larger granularity of program elements. Set \( E \) 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, the program entities \texttt{HybridAutomobile} and \texttt{overallSpeed} from the code snippet given in Figure 3.1 are related in that the class \texttt{HybridAutomobile} declares the field \texttt{overallSpeed}. In this case, there would exist an edge in \( E \) connecting the vertex in \( V \) that represents the class \texttt{HybridAutomobile} to the vertex representing the field \texttt{overallSpeed}. \( R \) is the set of all such (binary) relations between program elements that I consider. Since two given vertices may be related in several different ways, i.e., they satisfy more than one relation, there may exist multiple edges between them. As such, \( \ell: E \to R \) serves as a labeling function that distinguishes edges by labeling them with the satisfied relations they represent. Figure 3.3 portrays a subset of the graph computed from the motivating example given in Section 3.2 diagrammatically.

Table 3.1 portrays the complete set of binary relations that I consider as well as the program entity types in which they relate as derived from a recent version of the Java language specification (Gosling et al., 2005b) (\textit{Enum} is an abbreviation for Enumerated Types). For simplicity, I group class instance creations and constructor calls with method calls. Either these relations may hold in a structural sense, e.g., field declarations, or possibly during a particular execution of the program, e.g.,
<table>
<thead>
<tr>
<th>Relation</th>
<th>From Entity</th>
<th>To Entity</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>GetsField</code></td>
<td>Methods</td>
<td>Fields</td>
</tr>
<tr>
<td><code>SetsField</code></td>
<td>Methods</td>
<td>Fields</td>
</tr>
<tr>
<td><code>CallsMethod</code></td>
<td>Methods</td>
<td>Methods</td>
</tr>
<tr>
<td><code>OverridesMethod</code></td>
<td>Methods</td>
<td>Methods</td>
</tr>
<tr>
<td><code>ImplementsMethod</code></td>
<td>Classes, Enums, Interfaces</td>
<td>Methods</td>
</tr>
<tr>
<td><code>DeclaresMethod</code></td>
<td>Classes, Enums, Interfaces</td>
<td>Methods</td>
</tr>
<tr>
<td><code>DeclaresField</code></td>
<td>Classes, Enums, Interfaces</td>
<td>Fields</td>
</tr>
<tr>
<td><code>DeclaresType</code></td>
<td>Classes, Interfaces, Annotations</td>
<td>Classes, Annotations, Interfaces, Enums</td>
</tr>
<tr>
<td><code>ExtendsClass</code></td>
<td>Classes</td>
<td>Classes</td>
</tr>
<tr>
<td><code>ExtendsInterface</code></td>
<td>Interfaces</td>
<td>Interfaces</td>
</tr>
<tr>
<td><code>ImplementsInterface</code></td>
<td>Classes, Enums</td>
<td>Interfaces, Annotations</td>
</tr>
<tr>
<td><code>ContainsType</code></td>
<td>Packages</td>
<td>Classes, Annotations, Interfaces, Enums</td>
</tr>
<tr>
<td><code>Annotates</code></td>
<td>Annotations</td>
<td>Packages, Fields, Interfaces, Enums, Classes, Methods, Annotations</td>
</tr>
</tbody>
</table>

Table 3.1: Analyzed program entity types and relations.

method calls. Section 3.4.1 discusses how I conservatively approximated the truth value of these relations in my implementation by using exclusively static information, i.e., through examination of the program text, while section 3.6 touches upon future work that could result in a more accurate approximation.

Many kinds of relations may be formulated, however, I mainly focus on popular relations as used in previous work (e.g., [Braem et al. 2006; Dagenais et al. 2007; Robillard and Murphy 2002]) with the addition of relations useful for AO languages, e.g., `Annotates`. Section 3.4 reports on the appropriateness of using such relations for PCE rejuvenation in AspectJ programs; adding additional relations is discussed in Section 3.6.
3.3.4 Concern Graph/Pointcut Association

The extended concern graph offers us a clear picture of all source code elements in the analyzed program and how they are related. Each vertex represents a source code element while each edge represents a relation between two elements. The extended concern graph offers a concise representation of source code elements and the relations satisfied between them.

The next step in my 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. Then, commonality existing between these source code elements will be discovered by traversing paths within the graph in which the associated vertices and edges lie. I then capture commonality by constructing patterns that abstractly describe the kinds of relations that the source code elements have in common.

Recall that a PCE is a means to describe a set of join points, and that a join point is a well-defined point in the program’s execution. Thus, the definition of a join point is very much dynamic in nature. A join point shadow, on the other hand, refers to base code corresponding to a join point, i.e., a point in the program text where the compiler may actually perform the weaving (Masuhara et al., 2003). Whether the base code is advised at that point is dependent on advice being applicable, which can be based on dynamic conditions being met.

In this thesis, I treat a program as consisting of a set of join point shadows that may or may not be advised. This definition differs slightly from those typically given in the literature (e.g., Hilsdale and Hugunin, 2004; Ye and Velder, 2008),
where shadows always have the potential of being influenced by advice. My definition helps simplify the algorithm presentation. Moreover, I treat a PCE as selecting a subset of these shadows; i.e., I assume that the PCE is free of dynamic conditions. This allows me to exploit solely static information in my analysis. Section 3.4.1 discusses how my implementation conservatively relaxes this assumption so that PCEs utilizing dynamic conditions may nevertheless be used as input to the tool. The evaluation results reported in Section 3.4.4 indicate that the impact of this limitation is minimal and that my approach can be useful. In fact, there is evidence that suggests that most PCEs used by AspectJ programmers do not take advantage of dynamic conditions [Apel 2010]. Shadows corresponding to method declarations enable method vertices, i.e., for a graph $CG^+ = (V, E, R, \ell)$, I 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 motivating example found in Figure 3.1 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, Listing 3.2. The graph subset in Figure 3.3 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. More specifically, I 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 would lie at the declaration of the invoked method, while in the latter, it would lie at the site of the method invocation, i.e., the client code. Section 3.4.1 discusses how my implementation leverages existing tool support to deduce enabled graph elements. Possible future work entails taking into account additional AspectJ join point types such as handler-join points.

### 3.3.5 Pattern Extraction

Once I am able to associate (enable) various graph elements with the input PCE, (see Section 3.3.4), I may 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 I only take advantage of structural commonality between program elements and not other kinds of commonality, e.g., string similarity of method names.
In this thesis, I am interested in exploiting information pertaining to the structure and organization of the base code when related to PCEs.

Recall from the motivating example that the methods \texttt{increase(Fuel)} and \texttt{increase(Current)}, whose corresponding execution was selected by the PCE execution \texttt{(void increase(Energy+))}, both contained calls to methods \texttt{notifyChangeIn(Fuel)} and \texttt{notifyChangeIn(Current)}, which read from a common field \texttt{overallSpeed}. Deliberately, this information is expressed by two paths (sequences of connected edges) \texttt{increase(Fuel) \rightarrow overallSpeed} and \texttt{increase(Energy) \rightarrow overallSpeed} in the graph snippet portrayed in Figure \ref{fig:graph_snippet}. I 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 software versions to uncover new elements displaying the captured commonality.

For each enabled (w.r.t. the input PCE) vertex \texttt{v} and edge \texttt{(u,v)}, I extract a set of patterns from finite, acyclic paths of length (in terms of edges) \( \leq k \) passing through \texttt{v} and along \texttt{(u,v)}. The \textit{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 \ref{sec:analysis_depth} discusses my choice for \( k \) in my evaluation. An example of such a path when taking the enabled vertex \( v = \texttt{increase(Fuel)} \) and \( k = 2 \) exists in the graph subset depicted
in Figure 3.3, namely, the path \texttt{increaseFuel(Fuel) } \texttt{cm} \xrightarrow{gf} \texttt{notifyChangeIn(Fuel) } \texttt{overallSpeed}, where edge labels \textit{cm} and \textit{gf} refer to the satisfied relations \textit{CallsMethod} and \textit{GetsField}, respectively.

Intuitively, patterns are constructed in a way that paths that match the pattern are ones that have common origins or sinks that are connected via similar (in terms of labels) edges. I consider two kinds of patterns, those derived from enabled vertices and those from enabled edges. A \textit{vertex}-based pattern is obtained from a path by replacing certain vertices along the path with \textit{vertex wild cards}, while an \textit{edge}-based pattern is obtained by not only replacing vertices with \textit{vertex wild cards}, but also certain edges with \textit{edge wild cards}. \textit{Vertex wild cards} only match vertices, while \textit{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 \textit{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 (see Section 3.3.4) matched by enabled wild cards are those that eventually become suggested.

A set of vertex-based patterns are extracted from a path (a sequence of edges) \(\pi = \langle e_1, e_2, \ldots, e_n \rangle\) and an enabled vertex \(v\) along \(\pi\) as depicted in Algorithm 3.1. Text appearing between curly braces 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 algorithm proceeds as follows. If \(v\) occurs in \(\pi\) as the source vertex of the first edge, a single pattern is extracted by replacing this vertex with an enabled wild card. The remaining vertices along the path are replaced by disabled wild cards except
Algorithm 3.1 Vertex-based pattern extraction algorithm.

function $\text{ExtractVertexPatterns}(\pi = (e_1, e_2, \ldots, e_n), v)$

1: $\hat{\Pi} \leftarrow \emptyset$ {Patterns to be returned, initially empty.}
2: $\hat{\pi} \leftarrow ()$ {A single pattern to be built, initially the empty sequence of edges.}
3: for $i \leftarrow 1, n$ do {For each edge along path $\pi$}
4: if $i = 1 \land s(e_i) \neq v \land t(e_i) \neq v$ 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 $i = 1 \land t(e_i) = v$ 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 an enabled wild card as the target vertex.}
8: else if $i = n \land s(e_i) \neq v \land t(e_i) \neq v$ 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 $i = n \land s(e_i) = v$ 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 $i \neq 1 \land i = n \land t(e_i) = v$ 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 $i = 1 \land i \neq n \land s(e_i) = v$ 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 $i \neq 1 \land i \neq n \land s(e_i) \neq v \land t(e_i) \neq v$ 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 $i \neq 1 \land i \neq n \land s(e_i) = v$ 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: $\hat{\pi} \leftarrow ()$ {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: $\hat{\pi} \leftarrow ()$
26: end if
27: end for
28: return $\hat{\Pi} \cup \{\hat{\pi}\}$ {Return the return set along with the last completed pattern.}
end function
for the target vertex of the last edge. To illustrate, recall the previously considered path $\text{increaseFuel}(\text{Fuel}) \xrightarrow{\text{cm}} \text{notifyChangeIn}(\text{Fuel}) \xrightarrow{gf} \text{overallSpeed}$ where the vertex representing the method $\text{increaseFuel}(\text{Fuel})$ is enabled w.r.t. the PCE on line 3, Listing 3.2. The set of patterns extracted from this path would consist of the singleton $\{? \xrightarrow{\text{cm}} ? \xrightarrow{gf} \text{overallSpeed}\}$ where $?$ denotes a disabled wild card and $?^*$ an enabled wild card.

Continuing, if $v$ occurs in $\pi$ as the target vertex of the first edge, a similar action is performed as in the previous case, however, the source vertex of the first edge is retained and instead the target vertex of the first edge is replaced with an enabled wild card. For the case that $v$ occurs in $\pi$ 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, the path is split to extract two patterns, one with $v$ as the target vertex of the last edge and the other with $v$ as the source vertex of the first edge. The algorithm then proceeds as before.

Edge-based patterns are handled in a similar manor as depicted by Algorithm 3.2. 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 curly braces offers descriptions of each of the algorithm’s steps. The key difference between the vertex and edge pattern extraction algorithms is that, in the case of edges, the corresponding algorithm is intended to construct patterns which produce other edges exhibiting commonality related to the input (enabled) edge. This requires accounting for locations of where edges appear in paths, as well as the labels of the edges.
Algorithm 3.2 Edge-based pattern path extraction algorithm.

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

1: $\hat{\Pi} \leftarrow \emptyset$ {Patterns to be returned, initially empty.}
2: $\hat{\pi} \leftarrow ()$ {A single pattern to be built, initially the empty sequence of edges.}
3: for $i \leftarrow 1, n$ do {For each edge along path $\pi$}
4:     if $i = 1 \land e_i \neq e$ 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 $i = n \land e_i \neq e$ 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 $i \neq 1 \land i \neq n \land e_i \neq e$ 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 $(i = 1 \lor i = n) \land i \neq n \land e_i = e$ 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 $i \neq 1 \land i \neq n \land e_i = e$ then {Otherwise, if it is neither the first nor the last edge and it is enabled}
13:        $\hat{\pi} \leftarrow \hat{\pi} + (?, ?)^*$
14:    end if
15:    end for
16:    $\hat{\Pi} \leftarrow \hat{\Pi} \cup \{\hat{\pi}\}$ {Add the completed pattern to the return set.}
17:    $\hat{\pi} \leftarrow (?, ?, ?)^*$ {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.}
18: else {Otherwise, it must be that it is the only edge and it is enabled}
19:    $\hat{\pi} \leftarrow \hat{\pi} + (s(e_i), ?)^*$ {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.}
20:    $\hat{\Pi} \leftarrow \hat{\Pi} \cup \{\hat{\pi}\}$
21:    $\hat{\pi} \leftarrow ((?, ?)^*, ?)^*$ {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.}
22: end function
Deciding on this specific scheme (for both vertices and edges) was pragmatic; there is a trade-off to be considered between the abstractness of patterns, i.e., the ratio of constituent wild cards to that of concrete elements, and the quality of the results produced. Particularly, highly abstract patterns are more likely to produce more (possibly spurious) results. I discuss how I dealt with this trade-off in my ranking scheme in Section 3.3.7.

3.3.6 Pattern Matching

I say that a pattern \( \hat{\pi} \) matches a path \( \pi \) iff

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

Figure 3.4: Evolving the base code with a FuelCell class.
To illustrate, suppose the graph found in Figure 3.3 is augmented with new vertices and edges representing facets of the code of the FuelCell class in Listing 3.3. The resulting situation is depicted in Figure 3.4 where a new path $\text{increase(double)} \xrightarrow{cm} \text{notifyChangeIn(double)} \xrightarrow{gf} \text{overallSpeed}$ 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 (vertices and/or edges) 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 (within code) of the method represented by the source vertex and operate (call, get, or set) on program element represented by the target vertex. For example, when matching the pattern $^* \xrightarrow{cm} ? \xrightarrow{gf} \text{overallSpeed}$ against the path $\text{increase(double)} \xrightarrow{cm} \text{notifyChangeIn(double)} \xrightarrow{gf} \text{overallSpeed}$, the method FuelCell.increase(double) is represented by a vertex that matches an enabled wild card element. The situation is emphasized in Figure 3.4 by a dashed line through the vertices that induced the wild card. As a result, my approach would suggest that the CCC being realized by the advice on lines 2–4 of Listing 3.2 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 be responsible for realizing a particular CCC, similar to how multiple methods may be responsible for realizing a particular concern in Java. Such
\[
err_\alpha(\hat{\pi}, \text{PCE}) = \begin{cases} 
0 & \text{if } |\text{Match}(\hat{\pi}, \text{Paths}(\text{CG}^+))| = 0 \\
1 - \frac{|\text{PCE} \cap \text{Match}(\hat{\pi}, \text{Paths}(\text{CG}^+))|}{|\text{Match}(\hat{\pi}, \text{Paths}(\text{CG}^+))|} & \text{o.w.}
\end{cases}
\]

(3.1)

\[
err_\beta(\hat{\pi}, \text{PCE}) = \begin{cases} 
1 & \text{if } |\text{PCE}| = 0 \\
1 - \frac{|\text{PCE} \cap \text{Match}(\hat{\pi}, \text{Paths}(\text{CG}^+))|}{|\text{PCE}|} & \text{o.w.}
\end{cases}
\]

(3.2)

\[
abs(\hat{\pi}) = \begin{cases} 
1 & \text{if } |\hat{\pi}| = 0 \\
1 - \frac{|\hat{\pi}| - |W(\hat{\pi})|}{|\hat{\pi}|} & \text{o.w.}
\end{cases}
\]

(3.3)

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

(3.4)

Figure 3.5: Pattern attribute equations.

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, \texttt{Energy} and \texttt{double}, both being the first parameter). Thus, upon the suggestion, the developer would proceed to create a new advice declaration bound to the PCE \texttt{execution(void FuelCell.increase(double))} that properly implements the CCC corresponding to speeding violation prevention.

### 3.3.7 Suggestion Sorting

Shadows suggested for incorporation are presented to the developer in descending order of the degree of \textit{confidence} in the shadow being applicable to a revised version of the input PCE. The confidence value, a real number in the interval \([0, 1]\), paired with each suggestion is inherited from the pattern that produced it. The 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 is evaluated by applying the pattern to the \textit{current} version of the base code and assessing its
performance. This assessment is performed on three different dimensions as depicted by the equations listed in Figure 3.5. I refer to each of these as pattern attributes w.r.t. the PCE to be rejuvenated.

Type I Error Assessment

To describe the attributes more precisely, I first define a function \( \text{Match}(\hat{\pi}, \Pi) \) where \( \hat{\pi} \) ranges over the set of patterns and \( \Pi \) 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 3.3.6. Then, I define the \( \text{err}_\alpha \) rate attribute, equation (3.1), 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 \( \text{Paths}(CG^+) \) to the number of shadows solely captured by the pattern. Note that \( CG^+ \) refers here to the graph computed from the base-code in which the pattern was constructed, i.e., the original, unrevised program. Furthermore, recall from Section 3.3.4 that I treat a PCE as a set of shadows, thus, \( |\text{PCE}| \) refers to the cardinality of the set denoted by PCE, i.e., the number of shadows it selects.

The \( \alpha \) signifies the metric's association with the rate of type I (or \( \alpha \)) errors which relates to the number of false positives resulting from applying the pattern to the original version of the base-code, as portrayed by region marked \( \alpha \) in the Venn diagram depicted in Figure 3.6. Conceptually, the \( \text{err}_\alpha \) rate quantifies the pattern's ability in matching solely the shadows contained within the PCE; the closer the \( \text{err}_\alpha \) 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 \( \text{err}_\alpha \) rate is one that expresses a
strong relationship amongst shadows captured by the PCE; one would expect future shadows to exhibit similar characteristics. If a pattern matches no shadows, its $err_\alpha$ rate is 0. For example, applying the pattern $?* \xrightarrow{cm} ? \xrightarrow{gf} overallSpeed$ to the original base-code version in Listing 3.1 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 $update() \xrightarrow{cm} getOverallSpeed() \xrightarrow{gf} overallSpeed$). Thus, the $err_\alpha$ rate for this pattern w.r.t. the PCE found on line 3 of Listing 3.2, which selects the execution of methods $DieselEngine.increase(Fuel)$ and $ElectricMotor.increase(Current)$ in the original base-code version, would be $\frac{1}{3}$.

**Type II Error Assessment**

The $err_\beta$ rate attribute, equation (3.2), is the ratio of the number of shadows captured by both the PCE and the pattern when applied to paths in the graph to the number of shadows captured by solely by the given PCE. The difference between the $err_\alpha$ and $err_\beta$ rates is subtle but important; the $\beta$ signifies the metric’s association
with the rate of type II (or $\beta$) errors which relates to the number of false negatives produced by the pattern (also depicted in Figure 3.6 by the region marked $\beta$). Conceptually, the $err_{\beta}$ rate quantifies the pattern’s ability in matching all of the shadows contained within the PCE; the closer the $err_{\beta}$ 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 $err_{\beta}$ rate expresses properties similar to the ones expressed by the given PCE, regardless of whether or not those properties are common to the captured shadows. Naturally, if the given PCE does not contain any shadows, the pattern’s corresponding $err_{\beta}$ 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 $err_{\beta}$ of 0 w.r.t. the PCE found on line 3, Listing 3.2 since it, when applied to the original base-code version, produces all the shadows captured by the PCE.

**Pattern Abstractness**

Recall that a pattern $\hat{\pi}$ is derived from a path $\pi$ 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, it would be desirable to take into account its abstractness (abbreviated abs), i.e., the ratio of the number of constituent wild card elements to concrete elements. Let $|\hat{\pi}|$ denote the number of elements (vertices and edges), including wild cards, at unique positions in the pattern $\hat{\pi}$. Moreover, let $\mathcal{W}(\hat{\pi})$ denote the multiset projection of wild card elements contained in pattern $\hat{\pi}$. Likewise, $|\mathcal{W}(\hat{\pi})|$ represents the number of wild card elements contained within pattern $\hat{\pi}$. Then, the abs of a pattern $\hat{\pi}$, which is independent of any particular PCE,
is given by equation 3.3. Note that an empty pattern has no concrete elements, thus, such a pattern is considered completely abstract, i.e., having an abstractness of 1. To exemplify, the aforementioned pattern would be considered $\frac{2}{5}$ abstract.

Pattern Confidence

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, the $err_\alpha$ and $err_\beta$ rates are combined via a weighted mean weighted by $abs$ for the following reasons. A pattern that is very abstract, i.e., containing many wild cards, is typically less likely to hone in on shadows that are only selected by the given PCE. Conversely, a pattern that is less abstract, i.e., more concrete, containing fewer wild cards, is less likely to cover all shadows selected by the given PCE. The combined metrics are used to derive the confidence (abbreviated $conf$) pattern attribute depicted in equation 3.4, which is a convenient, single metric in judging the confidence that the pattern can accurately detect 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 the 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. Listing 3.3).

3.4 Experimental Evaluation

In this section, I provide an overview of an experimental study conducted to quantitatively and qualitatively ascertain the usefulness of my rejuvenation approach in terms of its ability to accurately suggest shadows not captured by an original PCE.
to be incorporated into a revised version of the same PCE given evolutionary changes made to the base code.

This section is organized as follows. Section 3.4.1 discusses the implementation of the algorithm given in the previous section. Section 3.4.2 explains the experimental setting, while Sections 3.4.3 and 3.4.4 detail the results of the experiments performed. The experiments were separated into two parts, namely, Phase I and Phase II. Phase I, described in Section 3.4.3 consisted of analyzing various characteristics of the patterns produced by the algorithm regarding a single program version. Phase II, whose results are depicted in Section 3.4.4 was comprised of applying the patterns produced from a single program version to a subsequent version to study the usefulness in rejuvenating PCEs between base code versions. Finally, Section 3.4.5 reports on possible validity threats to the results of the experiments based on my assumptions.

3.4.1 Implementation

The algorithm described in the previous section was implemented as a plug-in, called Rejuvenate Pointcut, 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 fact extractor, which was extended for use with Java 5 and AspectJ. JayFX generates “facts,” using class hierarchical analysis (CHA) (Dean et al., 1995), pertaining to structural properties and relationships, e.g., field accesses, method calls. Source code and transitively referenced libraries (possibly in binary format), alike, are analyzed during graph building. The AJDT compiler was leveraged to conservatively (explained next) associate the graph with a PCE. For a given PCE, the

\[ \text{http://www.cs.mcgill.ca/~swevo/jayfx} \]
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 rules engine, which makes use of a modified version of the RETE algorithm [Forgy, 1982]. The Drools framework not only provides an efficient solution to the many-to-many matching problem the tool is faced with, as well as a natural query language, but also performance benefits such as the caching of results. Pattern descriptions were persisted as XML files, which were read and written to using the Java Domain Object Model (JDOM) translation framework. As of this writing, the project remains under active development as a Google Code project and is publicly available for download at

http://code.google.com/p/rejuvenate-pc

To increase applicability to real-world applications, several assumptions described in Section 3.3 were relaxed. For example, the tool conservatively assumes that dynamic advice, i.e., advice bound to a PCE containing run time predicates is always applied. If the tool encounters any inter-type declarations (ITDs) or any other form of static crosscutting, the associated PCE is still processed but these constructs are not factored into the analysis. That is, program elements introduced in the base code via an ITD, as well as program element relations induced by static crosscutting, are 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 my tool. Moreover, there may be relations induced by static crosscuts between program elements that my tool does not use, which may reduce pattern precision. While it

16 http://www.jboss.org/drools
17 http://jdom.org
would be reasonably straightforward to address this limitation, it did not seem to have a significant impact on the performance on my tool, as the following sections demonstrate. In addition, there is evidence that static crosscutting is not prevalent in AspectJ programs \cite{Apel2010} Section 4).

3.4.2 Study Configuration

The evaluation was conducted in two phases. For both phases, the maximum analysis depth parameter $k$ was set at two. Although setting $k$ to be less than two would theoretically improve performance, I 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 \cite{Laddad2003} that is intended to select all join points corresponding to JDBC (\url{http://java.sun.com/jdbc}) connection creations calls originating from mypackage:

\begin{verbatim}
pointcut connectionCreation(String url, String username, String password)
: call(public static Connection
    DriverManager.getConnection(String, String, String))
&& args(url, username, password)
&& within(mypackage.*);
\end{verbatim}

Listing 3.4: An example pointcut selecting join points corresponding to database connection creations.

This PCE is too specific since there are two additional \texttt{getConnection} methods in the \texttt{DriverManager} interface that can be used to create database connections \cite{Ye2008}. 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 my 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 one (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 two, on the other hand, would suffice for such a situation.

As another example, the pattern used to discover the method execution shadow corresponding to the method `increase(double)` in Section 3.2 is of length two. In this situation, the methods accessing the common field `overallSpeed` are being called by the methods whose execution shadows are selected by the PCE in the original base code version. A pattern of length one would be insufficient to capture this relationship but could capture the relationship where shadows corresponding to all methods declared in class `HybridAutomobile` are to be suggested. Such a pattern, however, would suggest a spurious shadow corresponding to the execution of the `getOverallSpeed()` method. CCCs that apply to methods that access fields indirectly through other methods are common.

In general, CCCs that apply to methods that delegate tasks to intermediate methods is common in OOP. Thus, I 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 two may result in effective rejuvenation for a wider variety of situations, but my experiments described in the forthcoming
sections suggest that it would likely result in a large run time overhead with little
gain in precision and recall. I discuss this issue further in Section 3.6.

In the first phase, I aimed to show that the motivation behind my 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 twenty-
three 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. LOC was counted using the Eclipse Metrics tool found
at [http://metrics.sourceforge.net](http://metrics.sourceforge.net). Complete source code and descriptions of as
well as references to the studied subjects can be found on my website [http://sites.
google.com/site/pointcutrejuvenation](http://sites.google.com/site/pointcutrejuvenation). I was not involved in the development
of any of the subject applications. To ensure that a certain level of quality was
maintained, I purposefully selected subjects that have been used previously in the
literature, (e.g., [Bodden and Havelund, 2008; Cacho et al. 2008; Coelho et al.
2008; Dufour et al. 2004; Kulesza et al. 2006; Marin et al. 2007; Xu and Rountev
2007, 2008; Zhao, 2002]) including empirical studies (e.g., [Figueiredo et al. 2008;
Greenwood et al. 2007]). This ensures that the subjects have achieved a particular
level of acceptance within the community.

Table 3.2, with corresponding legend in Table 3.3, 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
Table 3.2: Phase I: Correlation analysis experiment results.

<table>
<thead>
<tr>
<th>subject</th>
<th>LOC</th>
<th>class.</th>
<th>PCE</th>
<th>shad.</th>
<th>patt.</th>
<th>$\text{err}_\alpha$</th>
<th>$\text{err}_\beta$</th>
<th>t (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>AJHotDraw</td>
<td>21750</td>
<td>298</td>
<td>32</td>
<td>90</td>
<td>3662</td>
<td>0.32</td>
<td>0.06</td>
<td>73.93</td>
</tr>
<tr>
<td>Ants</td>
<td>1572</td>
<td>33</td>
<td>22</td>
<td>297</td>
<td>1254</td>
<td>0.15</td>
<td>0.23</td>
<td>9.06</td>
</tr>
<tr>
<td>Bean</td>
<td>121</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>16</td>
<td>0.24</td>
<td>0.23</td>
<td>1.75</td>
</tr>
<tr>
<td>Cactus</td>
<td>7573</td>
<td>93</td>
<td>2</td>
<td>222</td>
<td>2151</td>
<td>0.21</td>
<td>0.52</td>
<td>4.36</td>
</tr>
<tr>
<td>Contract4J</td>
<td>10722</td>
<td>199</td>
<td>15</td>
<td>350</td>
<td>1809</td>
<td>0.26</td>
<td>0.44</td>
<td>59.18</td>
</tr>
<tr>
<td>DCM</td>
<td>1680</td>
<td>29</td>
<td>8</td>
<td>343</td>
<td>2472</td>
<td>0.15</td>
<td>0.45</td>
<td>2.16</td>
</tr>
<tr>
<td>Figure</td>
<td>94</td>
<td>5</td>
<td>1</td>
<td>22</td>
<td>11</td>
<td>0.11</td>
<td>0.45</td>
<td>2.01</td>
</tr>
<tr>
<td>Glassbox</td>
<td>25940</td>
<td>430</td>
<td>55</td>
<td>208</td>
<td>2620</td>
<td>0.1</td>
<td>0.29</td>
<td>120.65</td>
</tr>
<tr>
<td>HealthWatcher</td>
<td>5716</td>
<td>76</td>
<td>27</td>
<td>122</td>
<td>1004</td>
<td>0.21</td>
<td>0.16</td>
<td>7.85</td>
</tr>
<tr>
<td>LawOfDemeter</td>
<td>1586</td>
<td>29</td>
<td>5</td>
<td>164</td>
<td>540</td>
<td>0.15</td>
<td>0.41</td>
<td>9.49</td>
</tr>
<tr>
<td>MobileMedia</td>
<td>3806</td>
<td>52</td>
<td>25</td>
<td>25</td>
<td>775</td>
<td>0.23</td>
<td>0.00</td>
<td>3.67</td>
</tr>
<tr>
<td>MySQL</td>
<td>44016</td>
<td>187</td>
<td>2</td>
<td>3016</td>
<td>17564</td>
<td>0.12</td>
<td>0.58</td>
<td>336.29</td>
</tr>
<tr>
<td>NullCheck</td>
<td>1474</td>
<td>27</td>
<td>1</td>
<td>112</td>
<td>92</td>
<td>0.17</td>
<td>0.55</td>
<td>104.66</td>
</tr>
<tr>
<td>N-Version</td>
<td>552</td>
<td>15</td>
<td>4</td>
<td>9</td>
<td>80</td>
<td>0.19</td>
<td>0.24</td>
<td>0.51</td>
</tr>
<tr>
<td>Quicksort</td>
<td>73</td>
<td>3</td>
<td>4</td>
<td>7</td>
<td>56</td>
<td>0.19</td>
<td>0.15</td>
<td>2.78</td>
</tr>
<tr>
<td>RacerAJ</td>
<td>576</td>
<td>13</td>
<td>4</td>
<td>9</td>
<td>15</td>
<td>0.23</td>
<td>0.09</td>
<td>1.82</td>
</tr>
<tr>
<td>RecoveryCache</td>
<td>222</td>
<td>3</td>
<td>4</td>
<td>14</td>
<td>72</td>
<td>0.11</td>
<td>0.21</td>
<td>1.93</td>
</tr>
<tr>
<td>Spacewar</td>
<td>1415</td>
<td>21</td>
<td>9</td>
<td>58</td>
<td>225</td>
<td>0.15</td>
<td>0.22</td>
<td>17.09</td>
</tr>
<tr>
<td>StarJ-Pool</td>
<td>38218</td>
<td>511</td>
<td>1</td>
<td>3</td>
<td>67</td>
<td>0.25</td>
<td>0.00</td>
<td>29.78</td>
</tr>
<tr>
<td>Telecom</td>
<td>277</td>
<td>10</td>
<td>4</td>
<td>5</td>
<td>32</td>
<td>0.21</td>
<td>0.02</td>
<td>3.21</td>
</tr>
<tr>
<td>Tetris</td>
<td>1043</td>
<td>8</td>
<td>18</td>
<td>27</td>
<td>498</td>
<td>0.16</td>
<td>0.01</td>
<td>6.46</td>
</tr>
<tr>
<td>TollSystem</td>
<td>5195</td>
<td>88</td>
<td>35</td>
<td>85</td>
<td>1677</td>
<td>0.26</td>
<td>0.06</td>
<td>9.47</td>
</tr>
<tr>
<td>Tracing</td>
<td>366</td>
<td>5</td>
<td>16</td>
<td>132</td>
<td>676</td>
<td>0.17</td>
<td>0.4</td>
<td>2.32</td>
</tr>
</tbody>
</table>

**Totals**: 173987 2137 298 5308 37079 0.18<sup>a</sup> 0.16<sup>b</sup> 810.43<sup>c</sup>

<sup>a</sup>MySQL Connector/J

<sup>b</sup>Average rate weighted by number of patterns.

<sup>c</sup>Average rate weighted by number of PCEs.
Table 3.3: Legend for Table 3.2.

<table>
<thead>
<tr>
<th>Legend</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>Total number of non-blank, non-commented lines of code.</td>
</tr>
<tr>
<td>class.</td>
<td>Total number of class files after compilation.</td>
</tr>
<tr>
<td>PCE</td>
<td>Total number of pointcuts analyzed.</td>
</tr>
<tr>
<td>shad.</td>
<td>Total number of selected shadows.</td>
</tr>
<tr>
<td>patt.</td>
<td>Total number of patterns extracted by our tool.</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>Average pattern $\alpha$ rate.</td>
</tr>
<tr>
<td>$\beta$</td>
<td>Average pattern $\beta$ rate.</td>
</tr>
<tr>
<td>$t$</td>
<td>Total pattern extraction time in seconds.</td>
</tr>
</tbody>
</table>

(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 my patterns is practical for even large applications. The remaining columns will be discussed in Section 3.4.3.

In Phase II, I demonstrate the usefulness of my technique in a real-world setting by rejuvenating PCEs in multiple versions of three of the aforementioned subjects. As this task was rather involved, I chose a proper subset of the subjects listed in Table 3.2 that were ripe for the analysis in a number of ways. These subjects, listed in Table 3.4, were comprised of a series of discrete releases (column vers.), which allowed the accuracy of the shadows mechanically suggested by the tool to be evaluated.
against actual modifications to PCEs, in terms of included shadows, made by human developers in subsequent versions.

I briefly introduce the subjects here; more information pertaining to the subjects can be found on my website mentioned in Section 3.4.2. HealthWatcher is a web-based application that provides various medical-related support to patients. MobileMedia is a software product line consisting of applications that manipulate photo, music, and video on mobile devices. Lastly, Contract4J is a framework that facilitates Design by Contract (DbC) (Meyer, 1992) style programming in Java (version 5 and later).

For my 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, I considered major releases as units of evolution. Moreover, I was solely interested in rejuvenating PCEs between versions that exhibited non-trivial (defined next) modifications.

I define the following conditions for PCEs regarding subsequent versions, which ensures that the performance of my tool is evaluated only in situations where the PCE recovery due to modifications to the base code is non-trivial. I say that a PCE contained in a version $A$ evolved between a version $B$ iff

(i) the textual representation of the PCE in $A$ differs from the textual representation of the PCE in $B$,

(ii) the set of shadows selected by the PCE in $A$ differs from the set of shadows selected by the PCE in $B$, and
Figure 3.7: Comparing a PCE ($PCE$) with its revision ($PCE'$) in the new program ($B$).

(iii) the set of shadows selected by the PCE in $B$ differs from the set of 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. Lastly, criterion 3 asserts that the region designated by the light-shaded arrow in the Venn diagram depicted in Figure 3.7, where:

- the outer region (universe) 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,
Table 3.4: Phase II: Rejuvenation experiment results. *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, $\sigma_{rec}$ 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.

<table>
<thead>
<tr>
<th>subject</th>
<th>vers.</th>
<th>PCE</th>
<th>targ.</th>
<th>sugg.</th>
<th>rec.</th>
<th>$\sigma_{rec}$</th>
<th>$\uparrow TP$</th>
<th>$\sigma_{\uparrow TP}$</th>
<th>t (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contract4J</td>
<td>5</td>
<td>13</td>
<td>317</td>
<td>4542</td>
<td>0.81</td>
<td>0.35</td>
<td>10.40</td>
<td>15.41</td>
<td>365.47</td>
</tr>
<tr>
<td>HealthWatcher</td>
<td>8</td>
<td>6</td>
<td>30</td>
<td>536</td>
<td>1.00</td>
<td>0.00</td>
<td>5.72</td>
<td>7.19</td>
<td>55.59</td>
</tr>
<tr>
<td>MobilePhoto</td>
<td>7</td>
<td>30</td>
<td>33</td>
<td>1714</td>
<td>0.97</td>
<td>0.18</td>
<td>0.83</td>
<td>3.64</td>
<td>115.31</td>
</tr>
<tr>
<td><strong>Totals:</strong></td>
<td>20</td>
<td>49</td>
<td>380</td>
<td>6792</td>
<td>0.93</td>
<td>0.24$^a$</td>
<td>3.97$^a$</td>
<td>9.51$^a$</td>
<td>536.37</td>
</tr>
</tbody>
</table>

$^a$Arithmetic mean

is non-empty. This last criterion excludes the situation where the PCE remained robust between versions. As such, I evaluated the performance of my 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 3.4 shows the number of PCEs across versions that met these criteria and were, consequently, selected to be rejuvenated by my tool.

Determining the region labeled *PCE* in Figure 3.7 required carefully copying the original PCE to the subsequent version and binding it to an empty advice body. This enabled me to use the AJDT compiler to deduce the set of shadows corresponding to the PCE.

Column *t* represents the total rejuvenation time in seconds, which averaged 10.95 seconds per PCE. This indicates that my tool is practical to use, especially since users will most likely rejuvenate their pointcuts between releases of their software. I discuss
ways to possibly reduce rejuvenation time in Section 3.6. The remaining columns are discussed in Section 3.4.4.

3.4.3 Phase I: Correlation Analysis Results

In Phase I, I assess the amount of unique structural commonality typically portrayed by shadows selected by a single PCE by studying attributes (cf. Figure 3.5) of patterns extracted from a single version of the subjects listed in Table 3.2. Recall that a pattern with a low $\text{err}_\alpha$, cf. equation (3.1), 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 $\text{err}_\alpha$ rate is one that expresses common structural characteristics amongst shadows selected by the PCE that are not exhibited by other shadows. Recall from Section 3.3 that my 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 $\text{err}_\alpha$ depicts the average $\text{err}_\alpha$ rate for all patterns extracted from the associated subject. I found the average, weighted by the number of patterns extracted, $\text{err}_\alpha$ rate among all subjects to be 0.18, demonstrating that a high correlation exists. Moreover, I 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 $\text{err}_\beta$, cf. equation (3.2), rate (column $\text{err}_\beta$) 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.

3.4.4 Phase II: Expression Recovery Results

In Phase II, I assess the accuracy of my technique to mirror human-produced results by rejuvenating PCEs in multiple versions of the subjects listed in Table 3.4. I then evaluate the relationship between the shadows that were suggested for inclusion by my tool and those that were actually included in (human) revised PCEs residing in a subsequent version.

I was especially interested in exploring my 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 it been applied 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 the tool could be most helpful in mechanically discovering. This “target” set of shadows is represented by the region depicted by the light-shaded arrow in Figure 3.7 \((PCE' \setminus PCE)\). The total number of shadows occupying this regions across all rejuvenations is listed by column \(try\), Table 3.4.

Quantitative Analysis

As success metrics for evaluating my approach, I defined a promising rejuvenation to be one where my tool suggests the majority of targeted shadows, i.e., a high recall. Moreover, as suggestions are ranked by confidence (cf. Section 3.3.7), the traditional notion of precision (for unranked results) does not apply to my situation (Manning et al., 2008). Instead, I defined a precise rejuvenation to be one where targeted
shadows appeared near the top of the list of suggestions after sorting by confidence. In other words, the closer true positives appear near the top of the list, the more effective I deem my tool would have been in these situations.

Column \textit{rec.}, Table 3.4 shows the average recall at which my tool was able to suggest targeted shadows, while column $\sigma_{\text{rec.}}$ 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, my tool suggested 90\% of targeted shadows with a standard deviation of 24\%, demonstrating that my tool typically resulted in promising rejuvenation. In a real-world situation, however, the developer would be left to manually discover the remaining shadows (10\% on average) that my tool did not identify. In the worst case, this activity would require a whole program analysis. Thus, for such situations where my tool does not find all shadows that must be incorporated into updated versions of pointcuts, the usefulness of my tool is limited.

While both HealthWatcher and MobileMedia had similar recall values, the average recall for ContractJ was 0.81, which was distinctly lower than the results from the other subjects. I conjecture several reasons for this difference. First, unlike the other subjects, ContractJ 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, ContractJ 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 my experiment (Section 3.4.3) when I found that the patterns produced from Contract4J had relatively high $err_{\beta}$ (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 3.4, the $\uparrow_{TP}$ value across all subjects averaged $\sim 4$, 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 my tool did not perform as well when applied to the Contract4J subject once again. I found that the difference here is because 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 my algorithm. Since substantial use of dynamic PCEs is not typical, as previously discussed in Section 3.3.4, the results indicate that the performance of my tool would be precise for many AO programs.
Qualitative Analysis

In this section, I identify potential reasons for both accurate and inaccurate suggestions made by my tool. For succinctness, I draw examples from only the Health-Watcher subject. The major contributing factor that was found to cause patterns derived by my 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 \texttt{call(* HttpSession+.putValue(String, Subject))} was affected by a modification to the base code that involved introducing the \textit{Adapter} design pattern (Gamma et al., 1995). Consequently, the \texttt{HttpSession} class was replaced, invalidating all patterns containing references to this class. Nevertheless, even in such a situation, my 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 \textit{Command} design pattern (Gamma et al., 1995), 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, my patterns proved completely effective in this situation, suggesting only and all of the targeted shadows.

I 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, I questioned their usefulness in the cases that such change does occur. Despite this, I did find scenarios involving updates to these PCEs and, surprisingly, my patterns were able to produce accurate suggestions in these situations. One particular PCE that related to synchronization required rejuvenation due to \textit{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 my approach in its ability to discover obscure structural characteristics that may have eluded a developer when manually updating PCEs.

\subsection*{3.4.5 Threats to Validity}

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

Secondly, my choice for the delta in which my 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, I 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, I chose major release points for my delta, which practically represents a subset of when PCEs would be recovered manually. Moreover, I was unaware of situations that occurred during the development of my subjects where the developer evolved the base code but neglected to update pointcuts. Since my 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.

Lastly, in Section 3.4.2, I expressed that in Phase II, I was solely interested in rejuvenating PCEs between versions that exhibited non-trivial modifications. In this way, I assessed the usefulness of my tool only in situations where it was needed as to avoid possibly overly positive results in situations where it was not needed. However, I have no way of telling if the suggestions made by my tool when used in an unneeded situation would deter the developer from correctly updating PCEs. By design, I have excluded such scenarios from the analysis and thus do not have data pertaining to the behavior of the tool when used in these situations.

3.5 Related Work

In this section, I discuss work that is related to the approach presented in this chapter and compare and contrast facets of each.
3.5.1 Concern Traceability

The closest work resembling (and inspiring) mine involves tracking (Robillard 2006) and managing (Dagenais et al. 2007) concerns in source code throughout evolution. These approaches do not specifically deal with AOP, and my approach may be seen as their adaption and extension to the paradigm. However, there are several key differences. First, Dagenais et al. (2007) 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. My patterns are also intensional descriptions but derived from other intensional descriptions, namely, PCEs. In addition, patterns produced by my approach have been made to compete with the expressiveness inherent to PCEs, which deal specifically with CCCs. For example, my confidence evaluation is obtained using three dimensions of analysis (cf. Figure 3.5). Recall from Section 3.4.2 that, due to the nature of CCCs, my graph-based approach features a general analysis depth parameter. This parameter, along with the algorithmic considerations taken, allows me to derive patterns of a specified length. Thus, concepts pertaining to algorithm development are treated more fully in this work. Lastly, I present a thorough empirical evaluation of my technique applied to evolving AO software.

3.5.2 Aspects and Refactoring

Wloka et al. (2008) 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 my tool deals with general base code modifications. Moreover, in contrary to my 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 (2007) 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, I foresee an interesting scenario where the proposed tool may be integrated with my technique to automatically cluster suggested join points to be included in a revised PCE.

### 3.5.3 Automated Aspect-Oriented Software Development

Several techniques (e.g., Anbalagan and Xie, 2006; Perez-Toledano et al., 2007; Stoerzer and Graf, 2005) 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, my 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 (Ye and Volder, 2008), 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 Volder (2008) is hard-coded by a predefined, developer-minded heuristic, while my approach ranks join point suggestions in a more custom fashion using analysis results from the previous base code version.
3.5.4 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 (Lippert and Lopes 2002), these difficulties being rooted in the inherent fragility of typical PCE languages (Koppen and Stoerzer 2004). Several approaches (e.g., (Eichberg et al. 2004; Kellens et al. 2006; Klose and Ostermann 2005; Ostermann et al. 2005; Sillito et al. 2004)) attempt to add expressiveness to help combat this problem by altering or abstracting the underlying join point model. Others (e.g., (Braem et al. 2006; Gybels and Brichau 2003)) go even further by proposing approaches that combat fragility in these models. My 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, my tool makes suggestions based off this rich model while affording the developer the luxury of using a familiar AO language. Yet, others (e.g., (Hoffman and Eugster 2007; Rajan and Leavens 2008)) propose new, hybrid languages that feature facets from both paradigms. Thus, these languages would not be considered completely AO in a traditional sense (Filman and Friedman 2000).

3.6 Future Work

3.6.1 Pointcut Expression Reconstruction

In its current state, my 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 (Wloka et al. 2008). Moreover, I plan to incorporate techniques introduced by Anbalagan and Xie (2007) 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 mine 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.

### 3.6.2 Performance

Evaluating trade-offs between performance (in terms of pattern accuracy and running time) and maximum analysis depth to more thoroughly evaluate my 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 trade-off between performance and accuracy. Exploring graph reduction techniques (e.g., those employed by Robillard and Murphy (2002) and leveraging the AspectBench Compiler for AspectJ (abc18) (Avgustinov et al., 2006) to possibly reduce rejuvenation time may be helpful in this situation.

### 3.6.3 Additional Relations

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.

18 http://abc.comlab.ox.ac.uk/
3.6.4 Aspect Type and Inter-type Declaration Incorporation

Potential future work also entails incorporating aspect types and corresponding relations into the construction of the graph, as well as inter-type declarations. Inter-type declarations allow aspects to introduce new program elements, e.g., instance fields and methods, to the base-code. This is a useful feature of AOP as shown by, e.g., Clarke and Walker (2005) and often referred to as static crosscutting, as the structure, as opposed to the behavior, of the base-code is directly altered (Laddad 2003).

The motivation for inter-type declarations is that CCC implementations may need to introduce fields and/or methods pertaining solely to the particular CCC the aspect localizes. As such, these program entities should not appear in the base-code directly, but, rather, reside in the aspect and be introduced to the base-code via the compiler.

This process would be well facilitated by the new JDT (Java Development Tools) weaving feature introduced in AJDT 1.6.2. The JDT weaving feature allows deeper integration with Eclipse, especially with handling inter-type declarations (The Eclipse Foundation 2009).

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. In addition, 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.

In incorporating aspect elements, like advice, into the concern graph, an interesting issue arises in the case of around advice. In particular, around advice may include one or more calls to proceed, which invokes the method associated with the currently advised join point. Resolving the compile-time target of such calls may be difficult in cases where the bound PCE is complex and/or contains dynamic conditions. However, it may be worthwhile doing so since various advice bodies (perhaps in different aspects) may exhibit structural commonality linked by proceed calls. For example, several pieces of advice may invoke proceed, which, in turn, executes methods that read from a common field. If an existing PCE selects join points associated with such advice, future advice exhibiting these characteristics may need to have their corresponding join points selected by a revised version of the PCE. To accomplish this, the compile-time targets of proceed calls would have to be resolved, possibly by using techniques introduced by Clifton and Leavens (2005) and/or Avgustinov et al. (2005).

### 3.6.5 Improving Analysis of Dynamic Pointcuts

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) \cite{Bacon1996}, may also be valuable in more accurately estimating the truth-values associated with relations.

My general plan of improving the analysis of dynamic PCEs is as follows. The first step is, for each analyzed subject, locate a set of test cases that can be used to execute the subject programs. Such test cases are usually in the form of JUnit\footnote{http://www.junit.org} unit test, and often are accompanied by testing scripts, typically as ant\footnote{http://ant.apache.org} scripts. This would make the test cases amenable to automatic execution. Next, I will construct an aspect whose sole purpose is to enumerate all incurred join points. These results will be saved in a file for later retrieval. Then, I will run each test case (and simultaneously measure coverage) with the incorporated aspect. Finally, the file containing the incurred join points will be used as additional input to the approach. Correspondingly, the tool will read this file during the concern graph construction in order to trim possible edges according to the input trace. A thorough investigation of the accuracy of this new approach will then follow.

### 3.6.6 Bug Detection

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.
One option is to identify PCEs that were the root cause of (fixed) bugs in real-world software, along with a corrected version of these PCEs. Then, apply my tool to an earlier (and correct) version of the maligned PCE and, subsequently, compare the output of my tool with join points selected by the corrected version of the PCE. This would help derive the ability of my approach to prevent certain kinds of bugs in AO software.

Another option is a user study, however, such studies are not easily repeatable (Lung et al., 2008).

3.6.7 Leveraging Commonality from Historical Data

As previously discussed, my work deals with exploiting structural commonalities between program elements corresponding to join points selected by a PCE. However, it is conceivable that other kinds of commonality may exist between such program elements. In the future, I intend to explore new commonalities that may be useful in improving the accuracy of my initial approach.

One possibility that I intend to further investigate is that of leveraging commonality originating from historical data such as source code repository information. My hypothesis is that program elements corresponding to join points selected by a PCE in a particular software version may have been committed together in distinct source code repository transactions within source code management systems such as CVS and SVN.

To achieve this, I plan to integrate existing source code repository interaction Eclipse plug-ins into my analysis. Eclipse plug-ins for both CVS and SVN repositories exist and come with intuitive and standard (to the Eclipse framework) APIs, which
would make their integration seamless to the end-user. As for algorithmic considerations, new relations may be added to the concern graph specifically for repository information. For example, a new relation Committed may be symbolized as an edge between two nodes representing two method declarations. Also, a tag number must be associated with such a relation, as a single project may be associated with multiple versions, i.e., “commits.” A thorough empirical evaluation (much like the one presented in Section 3.4) will follow to ensure that the extension complements my approach by improving accuracy of the suggestions.

3.6.8 Framework Generalization

Recording traceability information between software artifacts used in various phases of the software development life cycle is a valuable activity. This information can prove helpful in comprehending large, complex systems, as well as verifying that each planned requirement of the system was, in fact, implemented and, furthermore, adequately tested. However, trace links between artifacts tend to share some of the same fragility issues exhibited by pointcut expressions in AOP as the software artifact base evolves. Unfortunately, the benefits of traceability information are lost as trace links are often neglected to be maintained in this situation.

The approach presented in this chapter may be generalized into a framework for rejuvenating trace links between arbitrary software artifacts. Such a tool-supported approach would be helpful to developers in maintaining traceability information, thus retaining their benefits as the underlying requirements, design, implementation, and test cases evolve. On a more general note, I also intend to explore applying the technique to rejuvenate module specifications, such as those discussed in Chapter
Plans for this proposed extension involve producing a General frAMework for traceaBility Link rEjuvenation (GAMBLE), which would provide automated mechanisms to help with the task of maintaining trace information and specifications as the traced artifacts evolve.

My plans for this framework are again separated into two distinct phases, namely, analysis (Phase I) and rejuvenation (Phase II). I now briefly discuss plans for the framework implementation high-level architecture.

**GAMBLE Phase I Architecture**

Figure 3.8 portrays the overall, extensible planned architecture for phase I to be provided to clients. I envision GAMBLE as an Eclipse plug-ins that assists other plug-ins that deal with traceability information, for example. This will be achieved via the use of extension points in the Eclipse plug-in architecture. The numbered circles in the diagram indicate these planned extension points. Clients, i.e., other plug-ins, provide GAMBLE with three functional modules, a Trace Translator, a Fact Extractor, and an Associator. These modules may either be off the shell (i.e., commercial), or custom made for GAMBLE. Nonetheless, each module would serve as an input translation into the framework.
Extension point 1, **Trace Translation**, referred to as \( \text{EP}_1 \), acts as the input processor for the traceability information. Traceability may come in many forms from many different kinds of plug-ins. Thus, \( \text{EP}_1 \) translates the traceability information into a form that can be understood by the **Associator** (\( \text{EP}_3 \)) can understand, and corresponding match against the output of the **Fact Extractor** (\( \text{EP}_2 \)).

Extension point 2, **Fact Extractor**, referred to as \( \text{EP}_2 \), also acts as input processor, taking input as a representation of the artifact base. \( \text{EP}_2 \) derives facts about the artifact base, i.e., entities and their relationships to be fed into \( \text{EP}_3 \). These information is ultimately used to build concern graph similar to the one discussed in the previous sections.

Extension point 3, **Associator**, referred to as \( \text{EP}_3 \), serves to associate the trace link information produced by \( \text{EP}_1 \) with the facts, i.e., entities and relationships, produced by \( \text{EP}_2 \). Essentially, a trace link associates an element of an artifact produced in a previous phase with an element of an artifact produced in a later phase, the transitive source being that of a concern. Thus, these elements may consist of either entities or relationships as produced by \( \text{EP}_2 \).

The associated data emanating from \( \text{EP}_3 \) serves as input the internal **Intention Analysis** (IA) module provided by GAMBLE. The IA module is responsible for building the concern graph from the input information, associating graph elements according the trace information, and finally deriving intention patterns from the graph and trace links. The actual path matching is planned to be implemented via the Drools rules engine. Once the patterns have been derived, I plan to store them for latter use in phase II via an eXtensible Markup Language (XML) format, possibly with the aid of a Java Domain Object Model (JDOM) translation service.
GAMBLE Phase II Architecture

Figure 3.9 demonstrates the plans for the overall, extensible architecture for Phase II to be provided to clients. Input to the second phase would consist of the XML file containing the analysis results from Phase I, as well as the new version of the artifact base. Notice that the same extension points are used here in phase II as was in phase I, specifically, EP₁, EP₂, and EP₃. The extension points feed into the Associator, which matches old trace links with the facts about the new (or revised) artifact base. IA would then be repeated using this information. Finally, the results would be displayed to the user, possibly via the Eclipse framework. The display may include a table of suggested trace links sorted by decreasing order of their confidence.

3.7 Conclusion

In this chapter, I have described my approach to limiting 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 struc-
tural commonality analysis depth was set at two, were presented, indicating that my
approach would be useful in rejuvenating pointcuts in real-world situations.
CHAPTER 4: Enforcing Behavioral Constraints in Evolving Aspect-Oriented Programs

Reasoning, specification, and verification of Aspect-Oriented (AO) programs presents unique challenges especially as such programs evolve over time. Components, i.e., classes and aspects, alike, may be easily added, removed, interchanged, or presently unavailable at unpredictable frequencies. Consequently, modular reasoning of such programs is highly attractive as it enables tractable evolution, otherwise necessitating that the entire program be reexamined each time an individual component is changed.

It is well known, however, that modular reasoning about AO programs is difficult. In this chapter, I investigate techniques for reasoning about the behavior of AO programs. I have developed an approach to specify the behavior of aspects in such a way that they can be combined with the behavior of the base-code to arrive at the overall system behavior without invalidation of previously obtain results of components that have not been directly altered.

In this chapter, I discuss my work on this topic, which includes a rely-guarantee style reasoning system for AOP. The system adopts a trace-based approach to deal with the plug-n-play nature inherent to these programs, thus easing AOP evolution.

Although my technique is not presently automated and planned future work does not involve fully automating this technique, in this chapter, I overview possible ways assist developers by mechanizing portions of the associated reasoning activities for future work.
4.1 Introduction

Aspect-oriented programming (AOP) ([Kiczales et al., 1997]) allows for modular implementations of crosscutting concerns. Since its inception, many authors (e.g., [Deng et al., 2004; Laddad, 2003; Lippert and Lopes, 2002; Rashid and Chitchyan, 2003]) have shown how aspects may be used to write localized implementations of important crosscutting concerns such as process synchronization, event logging, data persistence, exceptional situation handling, etc. The separation of concerns that AOP enables helps produce programs whose components are increasingly decoupled from one another as a direct consequence of the reduction of scattered and tangled code. As such, AO programs tend to enjoy plug-n-play-type capabilities where base and/or aspect components may be introduced, removed, and interchanged easily ([Laddad, 2003]). Thus, AO programs may evolve in a “non-invasive” fashion simply by “switching” features on and off.

Program evolution, however, is bound to occur at unpredictable frequencies; therefore, programmers are often required to make key decisions and come to conclusions about software components, base-code and aspect alike, utilizing either incomplete or highly volatile information at hand. Thus, the ability to reason about individual AO program components modularly and to then compose these reasoning efforts, just as we would compose the components themselves, to obtain the actual behavior of the overall program becomes extremely desirable. This ability would permit AO programmers to avoid the unfortunate situation where the entire program must be reexamined upon each component change, thereby facilitating tractable evolution.

Unfortunately, modular reasoning about AO programs indeed presents significant challenges ([Aldrich, 2005; Clifton and Leavens, 2002; Clifton et al., 2007; Dantas])
and Walker, 2006; Devereux, 2003; Griswold et al., 2006; Gudmundson and Kiczales, 2001; Kiczales and Mezini, 2005; Krishnamurthi et al., 2004; Ostermann, 2007, 2008; Ostermann et al., 2005; Sihman and Katz, 2003; Steimann, 2006; Sullivan et al., 2005; Xu et al., 2004). The problem is that circumscribing core concerns into classes and crosscutting concerns into aspects essentially creates two different (sub) systems, a baseline system (base-code) and an augmented system, which is the result of applying aspects that alter the behavior of the baseline. Indeed, the ability of an aspect to change the behavior of the base-code that it advises, which is the very reason for much of the power of AOP, is also what causes difficulties for reasoning about the behavior of such software. In fact, as aspects “weave” in and out of (or “plugged” then “played”) a software system, we may be forced to reason about the entire system, accounting for the interleaved execution of various pieces of advice with the base-code. What is needed is a way to draw meaningful and useful conclusions about component code, e.g., base-code that may reside in a method or an advice body that is itself subject to advice, without considering the actual advice code.

Ideally, one would like to specify the behavior of AO components without any particular advice in mind. That way, to arrive at the behavior of the augmented system, just as aspects are plugged-in to enhance or enrich the behavior of the advised components, the specifications of applicable advice would be “plugged” into the matching behavioral specifications of the base-code. Furthermore, to arrive at useful conclusions that remain valid despite the addition of advice, it may be necessary to constrain possible advice behavior as to preserve the intended semantics of the advised component. In other words, for a component to function correctly, assumptions may need to be made about potentially applicable advice such that these assumptions
Figure 4.1: Schematic of behavior derivation using parameterized specifications of a component C under the influence of advice of A₁ and/or A₂.

hold throughout evolution, with aspects entering, leaving, and possibly re-entering the software.

Adopting such an idealized approach would allow developers, as AO programs evolve, to deduce the behavior of the augmented software without reexamining the internals of each component. In essence, the specified behavior of a component would be parameterized over the behaviors of all possibly applicable advice. Figure 4.1 helps illustrate this notion, portraying schematically at a bird’s-eye-view how these parameterized specifications can be hypothetically combined with the specifications of
Figure 4.2: Schematic of behavior derivation using parameterized specifications of a component C not under the influence of advice.

advice to obtain the overall system behavior. Here, \(Spec(X)\) refers to the behavioral specifications of component \(X\).

Although the above outlined approach may seem desirable, several key obstacles must be overcome in its achievement:

Usefulness. As previously mentioned, one would like to draw useful conclusions about component code that is subject to the application of advice without considering the actual advice code. As the focus of this proposal is on *evolving* AO software, the advice code may not yet exist, e.g., it may be added a later time. It is not clear, however, exactly how useful these conclusions can possibly be considering
that they should hold upon general advice application. One potential difficulty is that drawn conclusions are too general, resulting in specifications that are too complicated. This is discussed in the next bullet.

**Complexity.** Since component specifications would be written in terms of any applicable advice, there is a strong possibility of these specifications becoming unwieldy. As such, any changes made to the *internals* (i.e., the implementation) of the advised component would require a rather involved effort to rebuild the component’s parameterized specifications. Also, situations may arise where a component $C$ may not be under the influence of advice, yet $Spec(C)$ would still be specified over all possibly applicable advice. Therefore, the complexity of the specification may be unnecessarily complicated. This situation is pictorially represented in 4.2. Future work may entail finding ways to curb complexity in this and other situations.

**Obliviousness.** Associating component code with parameterized specifications by its very nature compromises the traditional *oblivious* ([Filman and Friedman](2000)) property intrinsic to AOP, in particular with languages such as AspectJ ([Kiczales et al.](2001)). Thus, by allowing AOP developers to construct base-code specifications in a parameterized fashion, and to further constrain the behavior of absent advice (which would constitute the *actual* parameters), they would be at least aware of crosscutting concerns (CCCs). Nevertheless, it has been shown by [Kiczales and Mezini](2005) that even in ordinary (i.e., non-AOP) software, developers must still be aware of CCCs, and [Sullivan et al.](2005) suggest that designing components subject to advice also requires the cognition of CCCs.
**Modeling.** Modeling specifications that abstract adequate information from the internal details of components while simultaneously constraining the effects of potentially applicable advice that manipulates these details can be a daunting task. In my work, the uses of *traces* enables developers to write such specifications for AOP.

**Composition.** Given the specifications of a component and its applicable aspects, a reasoning system must provide the proper techniques for matching the *formal* parameters (in this case, contained in the base-code specification) with the actual parameters (in this case, the behavioral specifications of advice). Performing this activity, however, requires the ability to decide if advice is indeed applicable to the given base-code. Such a task may be especially difficult in situations where advice is bound to complex pointcuts, e.g., lexical and dynamic pointcuts. Lexical pointcuts, e.g., pointcuts containing such AspectJ constructs as `within()` and `withincode()`, contain conditions that are resolved using textual and/or structural properties of the source code. Dynamic pointcuts, e.g., pointcuts containing such AspectJ constructs as `cflow()` and `if()`, on the other hand, contain conditions that can only be completely and accurately resolved at run time in the general case. Once the advice is deemed applicable, the reasoning system must also provide the proper machinery to utilize the specifications of the advice and that of the parameterized base-code specifications to arrive at the overall behavior of the system, thus enabling the developer to verify that the composed software behaves as intended. Conversely, one must be able to, using the reasoning system, derive the behavior of base-code in which no advice is applicable. In fact, the schematic in Figure [4.1](#) is somewhat misleading in
that it fails to mention the situation where no advice is applicable. My work has not directly confronted such issues; nevertheless, such considerations will be taken into account for future work.

4.1.1 Organization

This chapter is organized as follows. Section 4.2 discusses the background of the issue. Section 4.3 details my proposed reasoning system. Related work is presented in Section 4.4, while future work is discussed in Section 4.5. Finally, I conclude with Section 4.6.

4.2 Constraining and Enriching the Behavior of Programs

Several approaches (e.g., [Aldrich 2005, Dantas and Walker 2006, Griswold et al. 2006, Gudmundson and Kiczales 2001]) have been developed to restrict the behavior of AO components to reduce the efforts required in reasoning (both formally and informally) about such systems. The focus of my work, however, is in specifying the intended run time behavior of components under the possible influence of advice that accounts for the unique evolutionary nature of AOP. Hence, assertions of these components should reflect this notion.

The motivation behind my proposal is to develop a reasoning technique that is, in comparison with existing approaches, more flexible in the way that constraints on acting advice are expressed. In this section, I briefly discuss how this is achieved in settings other than AOP in an effort to draw parallels between previous approaches and my proposed approach. In Section 4.2.1, I will briefly discuss how the behavior of processes within concurrent programs is suitably constrained to achieve interference
freedom using a rely/guarantee approach. Section 4.2.2 will draw a parallel with concurrent programs and that of AOP, and outline my work in adapting a rely/guarantee approach for AOP. Section 4.2.3 will overview my proposal in how assertions for evolving AO programs can be written, and how specifications can be composed to arrive at the effective behavior of the overall system, i.e., the behavior of the components augmented with the behavior of applicable aspects.

### 4.2.1 Constraining Behavior of Concurrent Programs

Consider a concurrent program with two processes $P_1$ and $P_2$ that share some variables that either of them may read or write. Standard modular reasoning would require one to reason about each process independently of the other and then combine the results of the two reasoning tasks to arrive at the behavior of the overall program, namely, $[P_1//P_2]$. However, since the two processes will be interleaved during execution, any conclusions that have been drawn about each process independently may not, in fact, be valid. In effect, the actions of each process may interfere with the other, thereby invalidating any results established by reasoning about an individual process. The rely-guarantee approach (Jones, 1983; Xu et al., 1997) addresses the problem of interference in concurrent programs in a compositional manner.21

The rely-guarantee approach is constructed as follows. Let $\sigma$ be the state, i.e., the set of all program variables of the program consisting of two processes $P_1$ and $P_2$ running in parallel. When reasoning about $P_1$, one recognizes that the actions of $P_2$ may modify the state. Hence, assertions are written in the proof outline of $P_1$ in such

---

21 There is ongoing controversy regarding whether or not the rely-guarantee method is truly compositional, especially in relation to ghost variables (Jones, 2003). Such issues have also been addressed by Feng (2009). Adopting facets of the approach developed by Feng (2009) in the AOP setting will be explored for future work and discussed in Section 4.5.
a manner that they continue to be satisfied even in the presence of such actions. To enable this, a binary relation $\text{rely}_1$, a predicate over two states, $\sigma_a$ and $\sigma_b$, is utilized.

This relation is applied as follows:

(i) Suppose that at some point in the execution of $P_1$, the current state is $\sigma_a$.

(ii) Suppose now that some part of $P_2$ is interleaved in the execution of $P_1$.

(iii) Suppose that the state when $P_1$ gets control back is $\sigma_b$.

(iv) At this point, $\text{rely}_1(\sigma_a, \sigma_b)$ must be satisfied.

In summary, when reasoning about the behavior of $P_1$, it is assumed that any interleaved action that $P_2$ (or any other process in the case of programs with more than two processes) may change the state but only within the constraints specified by $\text{rely}_1$. If this is satisfied, the correctness of the proof outline of $P_1$ will not be affected by the actions of $P_2$. Conversely, when reasoning about $P_2$, a relation $\text{rely}_2$ is introduced that imposes constraints on the changes in the state that may be caused by $P_1$’s actions.

Next, the fact that the individual executions of $P_2$ and $P_1$ meet the requirements contained, respectively, in $\text{rely}_1$ and $\text{rely}_2$ must be verified. To enable this, when reasoning about each process, a guarantee clause is established. This clause, denoted as $\text{guar}_1$ in the case of $P_1$, is again a binary relation over two states; it is a guarantee provided by $P_1$ that any change it makes in the state when executing any command in it will obey the constraints specified in $\text{guar}_1$. The resulting specification of $P_1$ is of the form:

$$(\text{pre}_1, \text{rely}_1, \text{guar}_1, \text{post}_1)$$
which symbolizes the following:

(i) if $P_1$ commences execution in a state that satisfies $pre_1$ and

(ii) if all transitions, i.e., state changes, made by $P_2$ satisfy the constraints specified in $rely_1$;

(iii) then, each transition made by $P_1$ will satisfy the constraints specified in $guar_1$,

(iv) and the state, when $P_1$ completes execution, will satisfy $post_1$.

When composing the specifications of the two processes, the parallel composition rule requires a check, using $guar_1$ and $guar_2$, that the $rely$ clauses of both processes are indeed satisfied.

### 4.2.2 Constraining Behavior of Aspect-Oriented Programs

In this section, I overview my work in specifying AO programs. First, I draw analogies between situations in concurrent programming and that of AOP. Next, I introduce my proposal in specifying AO program components using relations over two states. Lastly, I briefly discuss existing modularity techniques for AOP and explain how my proposal extends them.

**Concurrency vs. Aspect Orientation**

Reasoning in concurrent programs seems to have some resemblance to reasoning about AOP. Suppose, for example, that the code of a method in a class, say a class $C$, subject to advice contains an assignment statement assigning a value $v$ returned

\[\text{guar} \]

The $guar$ clause of a process may, in general, contain additional information about the behavior of the process beyond that needed to show that the process meets the $rely$ clause requirements of the other process(es).
from a method call \texttt{m()} on an object \texttt{obj} to a particular instance variable \texttt{x} of \texttt{C}.

When reasoning about the code of the method in \texttt{C}, one might have established an assertion following the assignment that states that the value of \texttt{C.x} would be equal to \textit{v}. Suppose now that an aspect is added containing an \textit{after}-advice that applies at the \textit{call}-join point associated with the invocation of \texttt{obj.m()}. Immediately following the execution of the assignment of the returned value \textit{v} to \texttt{C.x}, the \textit{after}-advice would execute and, possibly, invoke a mutator method of \texttt{C} (or manipulate the instance variable directly if the advice is contained in a so-called “privileged” aspect) that assigns a new value to \texttt{C.x}. When control returns to \texttt{C}, at this point, any assertion previously established may no longer satisfied. In other words, the aspect has \textit{interfered} with the component code in an analogous manner to processes of a concurrent program interfering with each other.

While this seems highly comparable to the case of the concurrent program, there are notable differences between the situation in AOP and that of concurrent programs. Firstly, AO programs are intrinsically sequential, making the interleaving of statements more predictable. Nevertheless, when reasoning about components independently, it must be recognized that advice may be weaved in, out, or around each join point in the base-code. This detracts the otherwise innate predictability previously mentioned. Secondly, the severity of possible interference is governed by the join point model of the underlying AO language. Possible interference is thus dictated by the types of join points, control structures, and mutable contexts (e.g., instance fields, variables, return values) that are exposed and available for advice to manipulate. Thirdly, there is a notable asymmetry between components that lack the ability to advise other components and those that do. In particular, while advice
can intercept the execution of a method, a method cannot intercept the execution of
dvice. In this case, advice may dictate the flow of control more implicitly than, for
instance, concurrent programs, where semaphores or monitors may be used to protect
values of shared variables, and coroutines (Conway 1963), where control is explicitly
released and suppressed at various points. Lastly, in the case of parallel programs,
concurrent processes are typically designed hand-in-hand, while in AO programs, as
detailed in section 4.1 aspects may be added, removed, and/or changed to a system
at unpredictable intervals as the software evolves.

Recognizing the Possibility of Behavior Modifications

A key observation is that assertions contained within a component subject to
advice should be in the form of a relation over two states $\sigma_a$ and $\sigma_b$. Here, $\sigma_a$
refers to the state of the advised component prior to control transferring to advice, and $\sigma_b$
refers to the state of the advised component immediately following the point where it
reacquires control. Therefore, components subject to advice can effectively detail the
sorts of constraints on advice behavior required for it to behave properly regardless
if advice is applicable now. Principally, when reasoning about a component $C$, one
recognizes that its behavior may be modified because of aspect(s) being applied to it.
As $C$ executes, if control were to reach a join point that matches a pointcut at which
a particular advice is applicable, control will transfer to the advice before, after, or
around (potentially bypassing) the statement at that point. The advice would then
execute, possibly changing the values of instance variables of $C$ and/or other accessible
parameters. Finally, control would then return to $C$, which would continue execution.

23This situation does not consider auxiliary technologies such as @AspectJ (see http://tinyurl.com/yhabbsa), which utilize Java Annotations to mark certain methods as advising other methods.
Extending Existing AOP Modularity Mechanisms

The approach discussed in this thesis is based on augmenting an existing technique made for improving modularity in AO programs, specifically, I extend the notion of pointcut interfaces. A pointcut interface is a programming modularity construct originally proposed by Gudmundson and Kiczales [2001] that serves as a kind of contract between base-code advise. Specifically, base-code, e.g., a class, lists, in its declaration, a set of pointcuts in which it is acceptable for advice to be applied. Advice declarations, then, can bind only to the listed pointcuts in the base-code and no others. The use of such an interface is associated with several benefits over traditional (AspectJ-style) pointcut declarations. For example, base-code developers are more cognizant of the portions of the base-code in which advice may be applied. In addition, advice developers are relieved from writing and, perhaps more importantly, maintaining pointcuts due to changes in this base-code. While the use of pointcut interfaces places some burden on base-code developers in terms of separation of concerns, the benefits of explicitly declaring where advice can apply (i.e., the loss of some obliviousness, namely, knowing exactly “where” advice may apply but not exactly knowing “what” it may do) and the ability of the base-code developer to “export” explicit events occurring within the execution of the base-code to the advice developers is considered valuable.

My proposal involves annotating listed pointcuts with associated specifications that must be met during the execution of the matching join points by both the component (through a guar clause) and applicable advice (through a rely clause). Although related work will be discussed in more detail in Section 4.4 it is worth mentioning that the contractual obligations between advice code and advised code
proposed here is similar in spirit to that of Crosscutting Interfaces (XPIs) \cite{Griswold et al. 2006}. However, unlike the approach of Griswold et al. \cite{Griswold et al. 2006}, who illustrate the use of XPIs in enforcing structural restrictions at compile-time and behavioral restrictions at run time, my interest lies in establishing behavioral properties through use of an axiomatic proof method.

### 4.2.3 Deriving Effective Behavior

Unlike concurrent programs, where the prime concern (in terms of reasoning efforts) is the preservation of interference freedom \cite{Owicki and Gries 1976}, the addition of aspects typically corresponds to enriching existing program behavior. In fact, it is the possibility of such enrichment that makes AOP an attractive technology. For this purpose, I introduce the concept of join point traces (JPTs). A JPT is used in reasoning about a component \( C \) under the potential influence of advice to record the “flow-of-control” through various join points contained within methods of \( C \). These join points are the ones captured by the pointcut interface of \( C \), where advice may be applied to enrich or otherwise alter its behavior.

While the structure of JPTs are discussed in more detail in Section 4.3.2, it is worth noting here that the central idea is to specify the behavior of a component (class or aspect\(^{24}\)) in terms of assertions involving not just the variables of \( C \) but also abstractly in terms of the state changes caused by various advice that could possibly be applied at join points recorded in \( C \)'s JPT, \textit{without} referencing the actual advice.

When reasoning about \( C \), one will not, of course, know what these state changes will be since the aspect(s) in question may not yet have been constructed (or even

\(^{24}\)Note that aspects themselves may be considered “base-code” if advice exists whose bound pointcut captures a join point associated with either a method or advice declared in an aspect."
if they have been, they have not yet been reasoned about). Hence, in the reasoning process, the reasoning system has to allow for a range of possible state changes that, subject to the constraints of the appropriate rely clauses, these items of advice may carry out. Essentially, the assertions characterizing the behavior of C will allow for various such changes and, corresponding to each, specify how C will behave. In effect, the behavior of C will be parameterized with respect to the possible behaviors that each item of advice code may engage in at the various join points, with the JPT being used to record the “parameter values” representing these behaviors. The next step, given a particular set of advice specifications, is to compose the JPT-based specification of C with the specified behaviors of the advice to arrive at the resulting enriched behavior of the composed system as illustrated earlier in Figures 4.1 and 4.2. Of course, to do so, one must verify that the advice respect the requirements specified in the rely clause corresponding to the bound pointcut and that the advised component respects the associated guar clause. Formally, this will be carried out by appealing to the rule of composition of aspect and the components they advise (discussed in Section 4.3.3).

4.3 Specification and Verification

In this section, I will overview my work in developing ways to specify and curtail the behavior of AO programs to improve reasoning in these systems that is akin to their evolutionary patterns, which is intuitively captured in Figure 4.1. Several inference rules will be presented next; these rules are written using a simplified and incomplete AspectJ-like AO language. The purpose of these rules is to allow the reasoner to show that the behavioral composition of AO components (i.e., base and/or
aspect) meets a certain specification. My work on this topic is lacking a complete formal set of rules. My work, rather, serves to indicate the types of considerations that would be involved with creating a full system of rules. For future work, I intend to define the syntax for a complete but simplified version of AspectJ, present its operational semantics, extend my set of proof rules to apply to this language, define a formal operational model based on the notion of JPTs, and address questions about soundness and completeness of the rules with respect to the model.

4.3.1 Specifications and Pointcuts

To demonstrate the crux of my work, I will only consider \textit{call}-join points and \textit{after} advice. \textit{Before}-advice could be theoretically handled in a symmetric manor; \textit{around}-advice, however, poses some interesting complications as advice could alter both the calling and callee objects and avoid the execution at the join point entirely by opting not to call \texttt{proceed()}, an AspectJ keyword which is replaced with a call to the advised method at run time. Incorporating \textit{around}- advice will be considered as part of future work. My work also does not take into account such constructs as lexical pointcuts but will be considered as future work. Also, my work does not deal with all types of dynamic pointcuts, such as those containing the \texttt{if()} run time predicate, however, augmenting the approach to incorporate such pointcuts in a na"ive way is reasonably straightforward. More sophisticated techniques will also be the subject for future work.

As suggested by Section 3.1, the desire for both flexible and expressive specifications may lead to undesired complexity as many join points may occur as a result of a method invocation. This complexity, however, depends heavily on the strength...
of the associated \textit{rely} clause since any applicable advice must respect it. Thus, although the specifications will be over the behavior of any applicable advice, there is no requirement to consider potential behavior that does not abide the \textit{rely} assertion. Another possibility would be to follow the conventions in the approach taken by Aldrich (2005), allowing only external calls to methods within a component $C$ listed on the interface of $C$ to be subject to advice, i.e., $C$ would be considered “sealed.” A sealed component is one where internal calls, i.e., calls to methods of the \textit{same} component, cannot be intercepted by advice unless they are included within pointcuts listed on the component pointcut interface. Thus, pointcuts appear on $C$’s interface to “export” important internal events within $C$ that the author of the component feels aspects may be interested in advising.

Note that, importantly, the author does not examine existing aspects to come to the decision to export particular, internal events occurring within the execution of the component. Instead, one does so by solely examining $C$ to determine which internal events should be exported on its interface. In much the same way, in the context of my proposal, the author of the component determines the necessary constraints to place on possible advice that would apply to it based on the internals of that component alone. Fundamentally, the author would be deciding what essential constraints are necessary to place on the behavior of advice either currently in existence or to be developed in the future so that the component may continue function properly. Moreover, as a possible extension to my approach, it may be worthwhile to break the sealing of a component for \textit{spectator} advice (Clifton and Leavens 2002), i.e., advice that does not alter the effective specification of the components they advise. Then, this advice would be free to “spectate” any join point within the component. That
way, less intrusive advice, e.g., ones realizing a logging concern, would be allowed to advise the execution of the entire program.

A common challenge with providing a reasoning scheme for programming languages that deal with objects is aliasing, which tends obscure the precise representation of an object’s state. A component \( C \), say a class, will, in general, define a number of (instance) variables. Some of these will be of primitive types (\texttt{int}, \texttt{boolean}, etc.), and others will be of reference types. Consider an instance \( \text{obj} \) of \( C \). The state of \( \text{obj} \) at any time will consist of the values of the variables of primitive types plus the values of references to objects.

Generally, the reasoning system will not consider the states of objects that \( \text{obj} \) contains references to as part of the state of \( \text{obj} \). Only changes to the values of its primitive variables resulting from execution of methods invoked on \( \text{obj} \) will be reflected as changes in the state of \( \text{obj} \). As these methods execute, they will, in general, invoke methods on objects that \( \text{obj} \) has references to, resulting in changes in the states of those objects. These latter changes are not part of the changes in the state of \( \text{obj} \). In effect, I am assuming that there is a heap in the background that holds all the objects, retains their current states, and makes them available to us as needed. These considerations are, of course, common to reasoning frameworks for all object-oriented languages. Hence, aliasing will not considered in any detail when presenting the formalism. Despite this, incorporating aliasing into the reasoning system will be the subject of future work, perhaps by utilizing Separation Logic (Ishtiaq and O’Hearn, 2001; O’Hearn et al., 2001).
4.3.2 Join Point Traces

From this point forward, I will typically consider the situation where one wishes to specify and verify a method in a class that may potentially be under the influence of advice. It is not inconceivable to conversely apply my proposal to reasoning about advice, which itself may be open to advice (possibly itself), however, as noted earlier, `around`-advice poses several interesting problems (e.g., constraining the behavior of calls to `proceed()`). Such endeavors will be the subject of future work.

Uses for JPTs

Consider, in general, a method `m()` of a class `C`. The JPT for this method will record the flow-of-control through the various join points in the body of `m()` where advice defined in various (perhaps yet-to-be-developed) aspects may apply. Each join point is a call to a method either of the same class `C` or of a different class. Note that the reasoner does not have to worry about every call that appears in the body of `m()`. For example, suppose there is a call to a method `n()` of a class `D`. If calls to `n()` appear in an exported pointcut on the pointcut interface of `D`, then this call will be recorded on the JPT of `m()`. If not, however, it will not be recorded since this call, in this case, would have no advice applicable to it. The designers of `D` are responsible for deciding whether calls to `D.n()` should be included in one of the pointcuts of `D`. The specifications developed using my proposed reasoning system, using JPTs, will reflect these decisions.

Traces and Specifications

Traces of various kinds have been widely used for specifying the behaviors of different types of systems ranging from ADTs.
and Sekerinski, 2001) to processes in a distributed system (Hoare, 1978; Misra and
Chandy, 1981). In each case, elements in the traces are used to record information
about important events in the system; the ordering of the elements in the trace
represents the order in which the corresponding events took place. Specifications of
the systems are written in terms of conditions that must be satisfied by the structure
of the trace and by the information recorded in the individual elements. In the case of
JPTs, the events of interest are the arrival of control at various join points. Since the
only kind of join point we are considering is the call-join point and since the only type
of advice we are considering is after-advice, each element of the JPT will correspond
to the completion of a call\textsuperscript{25} In order to deal with dynamic pointcuts such as those
containing cflow ad cflowbelow constructs, it is also convenient, from the point of
view of specifying methods, to record on the JPT the events corresponding to the
start of each method execution as well as its end.

**JPT General Structure**

The structure of elements that appear in a JPT is as follows. Consider the com-
pletion of a call of a method $m()$ of a class $C$ invoked on object $obj$. Suppose that a
pointcut defined in the pointcut interface of $C$ (or inherited from the interface that $C$
implements) includes calls to $m()$. Further suppose that $A$ is an aspect that includes
(after)-advice that applies to this pointcut. In addition, for simplicity, I assume that
any variables defined in $A$ will be of primitive types. As previously mentioned, future
work will consider types that are more complex.

\textsuperscript{25}To allow for other types of join points, the JPT would have to include elements that record
control arriving at each of those join points as well.
Continuing, the code of the advice may update the values of these variables, as well as update the state of \texttt{obj} by updating values of the primitive variables in that state. My work only considers the possibility of \texttt{A} invoking additional methods on \texttt{obj} (or on objects of \texttt{obj} or on the object on which the method that called \texttt{m}() was invoked). Trace models are, in general, powerful enough to handle such complications but the resulting specifications tend to be rather complex. The exclusion of other such calls means that the reasoner need not worry about additional advice associated with calls to such methods being triggered. Nevertheless, relaxation of these restrictions will be the focus of future work.

During actual execution of the system, if an aspect such as \texttt{A} considered above had been defined, control will transfer to the corresponding advice code. That code will execute, possibly resulting in changes in the state of the aspect as well as in the state of \texttt{obj} (e.g., through exposing context at the join point using the \texttt{target}() AspectJ construct), or even the caller of \texttt{obj.m}() (e.g., through exposing context at the join point using the \texttt{this}() AspectJ construct). Hence, this element of the JPT contain the state of \texttt{obj} at the time that control reached the completion of the call to \texttt{obj.m}() and its state when control returns from the advice, and likewise for the calling object of \texttt{obj.m}(). If there is no applicable advice, either because calls to \texttt{obj.m}() are not included in any pointcut, \texttt{obj.m}() is an internal call in \texttt{C}, no aspect such as \texttt{A} had been defined to apply at the pointcut, or the conditions for the advice to be applied are not satisfied, these two states will be identical, since the state of \texttt{obj} will not, in this case, change between the time that the call completes to \texttt{ob.m}() and the code of the calling method continues execution.
Global JPTs

The completion of the call to \texttt{obj.m()} is not the only point at which advice may apply. That is, calls made to additional methods within the body of \texttt{c.m()} may, themselves, be subject to advice interception. As such, the specifications of these method calls will also be parameterized over applicable aspects.

The JPT has been described as if it corresponded to just (a single call to) this method, there is, in fact, a single JPT for the \textit{entire} system. This JPT is initialized, at the start of the system's execution, to the empty sequence. The symbol $\gamma_T$ will be used to denote this (global) trace. Each time a method is called, an element is added to the JPT to record the start of this invocation. Moreover, when the method call completes, an element corresponding to the completion is added to the JPT. In addition, the effect of any (after-) advice that applies at this call-join point is recorded in the completion element.

JPT Completion Elements

The completion elements of the JPT will have the following structure:

$$(oid, mid, aid, args, res, \sigma, \sigma')$$

where \texttt{mid} is the identity of the method whose call just completed; \texttt{oid} is the identity of the object on which the method \texttt{mid} was invoked; \texttt{aid} is the identity of an applicable aspect instance, \texttt{args} and \texttt{res} are the arguments and result (if any) of this method. \texttt{\sigma} and \texttt{\sigma'} are state vectors as follows:

- $\sigma[oid]$ is the state of the callee object at the time the method call completed (immediately prior to transferring control to the advice if any).
• $\sigma'[oid]$ is the state at the time immediately following the completion of the advice code.

• $\sigma[this]$ is the state of the calling object at the time the method call completed.

• $\sigma'[this]$ is the state at the time immediately following the execution of advice.

Note that if there is no applicable advice at this join point, then $\sigma[this] = \sigma'[this]$. These state vectors also contain the state of aspects in a similar fashion. Specifically, $\sigma[aid]$ refers to the aspect state immediately following the completion of the execution of the method, whereas $\sigma'[aid]$ is the aspect state immediately following the completion of the advice code. If there is no applicable advice at this join point, these elements will not be included. While the above description presents the JPT as it was an actual program entity, the JPT is, in effect, a ghost variable whose sole purpose is to assist with reasoning activities. That is, the JPT does not actually exist at run time but serves as a mechanism for understanding run time behavior in a compositional manner.

**JPT Method Invocation Records**

The structure of the JPT elements that record method invocations is as follows. Consider again the call of a method $m()$ invoked on object $obj$. If the reasoner desired to account for before-advice that might apply at this point, the element of the JPT recording this invocation must include information very similar to the above. As previously mentioned, in this thesis, however, I only consider after-advice. Hence, such information omitted in this case. The only information that is required to be included in the JPT is the identities of $m()$ and $obj$, since these may be used to control the applicability of some advice.
\[ pre \land \lambda \tau = \langle (\text{inv}, \text{C.m}) \rangle \Rightarrow p \]
\[ \{p\}S\{q\} \]
\[ q \Rightarrow \text{guar}(\sigma[\text{this}]) \]
\[ q \land \text{rely}(\sigma[\text{this}], \sigma'[\text{this}]) \Rightarrow \text{post}[\lambda \tau \leftarrow \lambda \tau^-(\text{this}, \text{C.m}, \?, \text{args}, \text{res}, \sigma, \sigma'), \sigma \leftarrow \sigma'] \]

\[ \text{C.m} :: \langle \text{pre}, \text{post}, \text{guar}, \text{rely} \rangle \]

Figure 4.3: Rule for method specification.

4.3.3 Inference Rules

In this section, I will describe three rules corresponding to accounting for advice that may apply when a method call completes, for the call a method body makes to another method, and for combining the specification of a method \text{C.m()} with that of the aspects that apply. For the last point, these aspects include those that apply to the various methods that \text{m()} calls during its executions, so that the reasoner may arrive at the resulting “enriched” behavior of \text{m()}.

Specifying the Behavior of Methods

First, I will consider the rule depicted in Figure 4.3 corresponding to completion of a method call to \text{C.m()}. Recall that the specification of a method will be a 4-tuple, \((\text{pre}, \text{post}, \text{guar}, \text{rely})\). Here, \text{guar} is obtained from the disjunction of each \text{guar} clause of each exported pointcut that corresponds to calls to \text{C.m()}. Likewise, \text{rely} is obtained in a similar manor except that it is derived from the conjunction of each \text{rely} clause. This specification annotation means that if \text{pre} is satisfied when \text{C.m()} is called; and if the methods called in the body of \text{C.m()} satisfy their respective specifications (these calls may be to methods of \text{C} or other classes or both); and if
any advice applicable to calls to \texttt{C.m()} satisfy the \textit{rely} clause; then, when the body of \texttt{C.m()} finishes execution, it will satisfy \textit{post} as well as the requirements specified in the \textit{guar} clause. Note that the postcondition will, in general, involve the JPT since it is a key mechanism for enabling the reasoner to later enrich this specification, thus accounting for the action of advice defined in any aspect that may be developed to apply to calls to \texttt{C.m()}. It is also important to note that this JPT will not be the global JPT, rather, it will correspond to a single execution of the method. I will refer to this JPT as the “local” (in respect to a method execution) JPT and use the symbol \( \lambda \tau \) to refer to it.

The body of the method in the rule depicted in Figure 4.3 is denoted by \( S \). Likewise, the precondition is denoted by \( p \). Importantly, this differs from \( pre \) in the conclusion since \( pre \) does not provide any information regarding \( \lambda \tau \); thus, the first line essentially conveys that when the method body starts execution, \( \lambda \tau \) has been initialized to contain the element representing the start of the invocation (\textit{inv}) of the method \texttt{C.m()}.

The postcondition of \( S \) is as denoted \( q \). The second line of the rule requires the reasoner to show that when \( S \) completes, \( q \) will indeed hold. Moreover, \texttt{C.m()} will, in general, provide a guarantee to any advice that may apply when the call to \texttt{C.m()} completes. This guarantee is represented by \textit{guar} and this has to be satisfied when \( S \) finishes; i.e., the postcondition \( q \) of \( S \) must imply this assertion.

The next requirement, split over the next two lines of the rule, essentially allow the reasoner to go from the postcondition \( q \) of the body of the method to \textit{post}, the postcondition of the method. The difference between these two assertions arises due to the JPT having an extra element added to it to represent the completion of the
method. Moreover, the state might be modified from $\sigma$ to $\sigma'$ because of an advice that has been defined to apply at the completion of the method call. Any such advice is required, as indicated in the third line, to satisfy the rely clause. The element being appended to $\lambda\tau$ at this point, corresponds to the completion of the method call. The first element denotes the fact that the object in question is simply the this object.

To enable composition of the method specification with that of any (future) applicable advice, the aid element represents the identity of the advice, if any, that may apply at this point. Since the reasoner would only be considering the method now, there are no requirements with respect to an aspect state, hence the question mark. However, the state of the current object may itself change; this change is represented by the elements denoted $\sigma[this]$ and $\sigma'[this]$. These elements may, as just noted, be assumed to satisfy the rely clause, since, if not, the advice is considered unacceptable. Once the method completion element has been added to the JPT, the state of the object is changed to whatever the advice assigns to it. The rule requires the reasoner to show that, following this assignment, $post$, the specified postcondition of the method, is satisfied.

The rule seems rather involved but much of it is notational complexity, whose corresponding simplification will be part of the planned future work. Intuitively, however, the rule may be summarized by the following. The rule requires the reasoner to show that the body $S$ of $C.m()$ behaves according to its specification; and that when $S$ finishes, the state satisfies the requirements specified by the guar clause so any after-advice that is defined can legitimately assume this clause. The rule also requires the reasoner to take account of the fact that when the method is invoked, before the body starts execution, the trace must be appropriately initialized. Moreover, when

153
the method finishes, the trace must be finalized by adding a completion element that also records the effect of any advice that may be applied corresponding to calls to this method.

**Specifying Method Invocations**

I now consider the rule portrayed in Figure 4.4 for dealing with a method call. Suppose the method invocation is of \texttt{obj.m(args)} where \texttt{m()} is a method of the class \texttt{C}. Furthermore, assume that the specification of \texttt{C.m()} is \texttt{⟨pre, post, guar, rely⟩}. Moreover, assume that the (local) JPTs for the calling and called methods are named \texttt{λτ}_1 and \texttt{λτ}_2, respectively. At the start of the method, the reasoner must account for the fact that the object \texttt{obj} of the calling method will play the role of the \texttt{this} object of the called method and substitute the actual arguments for the formal parameters of \texttt{C.m()}. Let \texttt{p[x/e]} denote \texttt{p} where all free occurrences of a variable in the list \texttt{x} are replaced by its respective expression in \texttt{e}. When the method finishes, the reasoner must substitute in the reverse direction and append \texttt{λτ}_2 of \texttt{C.m()} to \texttt{λτ}_1 of the caller.

Fundamentally, the JPT records appropriate information about the various (potential) join points through which control flows. The value of \texttt{λτ}_1 at the time of the method call, i.e., just prior to control being transferred to \texttt{C.m()}, represents all join points encountered thus far in the calling method. Now control continues in the body...
\{ \text{guar}(\sigma) \land \text{ap} \} A_{\text{adv}} \{ \text{rely}[\sigma/\sigma@\text{pre},\sigma'/\sigma] \land \text{aq} \}

C.m :: \{ \text{pre}, \text{post}, \text{guar}, \text{rely} \}

\{ \text{pre} \land \text{ap} \} C.m() + A \{ \text{post} \land \text{aq} \}

Figure 4.5: Rule for aspect application (simple version).

of \text{C.m}(). \text{As that body is executed, additional methods may be called and information about the corresponding call-join points should be accumulated in } \lambda \tau_2, \text{the local JPT of } \text{C.m}(). \text{Nevertheless, this is, in fact, simply a record of the additional join points that we are encountering as execution continues. Hence, as control returns from } \text{C.m}() \text{ to its caller, this record is appended to the JPT obtained immediately prior to the call, i.e., to } \lambda \tau_1. \text{This will ensure that, in } \lambda \tau_1, \text{the JPT will contain a complete record of the control-flow along join points that occurs during the entire execution of the caller of } \text{C.m}(), \text{including the flow that occurs during the execution of the methods that are called.}

\textbf{Composing Base-code with Advice}

The final rule listed in Figure 4.5 is for applying an aspect to a method to arrive at the resulting enriched behavior. The situation is as follows. An aspect has been defined to apply at a pointcut that includes a call join point to \text{C.m}(), and the reasoner desires to arrive at the enriched behavior of the method as a result of the aspect. Figure 4.5 depicts a simpler form of the rule, ignoring the possibility of any dynamic pointcuts. Here, \text{A} refers to the aspect being applied and \text{A}_{\text{adv}} \text{ is the applied advice defined in the aspect.}
Recall that the *rely* and *guar* clauses are part of the specification of \( C.m() \). The \( \sigma@pre \) notation denotes the state at the start of the execution of method. The first line requires the reasoner to check that the advice meets the *rely* and *guar* clauses. In particular, the postcondition of \( A_{adv} \) ensures that the *rely* clause with \( \sigma@pre \) playing the role of the starting state of the clause and \( \sigma \) playing the role of ending state, is satisfied. The \( ap \) and \( aq \) are assertions over the state of the *aspect*. The second line simply states that the required result about \( C.m() \) has already been established. The postcondition in the conclusion of the rule shows the effect of the enrichment resulting from the application of the aspect.

There are several problems with this rule, however. First, the precondition requires not only that the expected precondition of the *method* is satisfied, but also \( ap \), which is a condition over the aspect state. The aspect state would not be affected by the execution of the body of \( C.m() \), so if it is satisfied at the start of the execution of this body, it will also be satisfied when \( A_{adv} \) starts execution.

In order for the rule to be consistent, it must be ensured that \( ap \) is, in fact, satisfied at the start of \( C.m() \). In general, this state will be modified only because of executions of various advice code declared in the aspect and not by the body of \( C.m() \). Nevertheless, there is nothing in the precondition of \( C.m() \), as currently presented, that will ensure that the aspect state, as it existed at the start of \( C.m() \), satisfies the assertion. Hence, an additional part to the precondition of \( C.m() \) is required, and this information needs to be provided as part of the *invocation* element that is added to the JPT at the start of \( C.m() \).

However, although the execution of the body of \( C.m() \) will not directly modify the state of the aspect, there may be calls in this body to other methods, and those
methods might be subject to this same advice as well. Thus, the state of the aspect when the execution of the body of `C.m()` completes and control is transferred to the advice may not be the same as it was when `C.m()` started execution. Instead, it will be the remnants of the most recent call after it has finished execution. These calls will, of course, be recorded on the JPT as per the method call rule. Further, the effect of the advice acting on the called methods will result in the aspect state that exists at the end of each such call to be recorded on the JPT. These considerations can be addressed by making two changes to the above rule. First, the precondition of `A` can be modified so that the assertion `ap` applies to the state of the aspect as recorded in the invocation element of the JPT. Second, the postcondition can be modified so that the assertion `aq` applies to the (final) aspect state recorded in the JPT when this method returns to its caller.

The third problem with the rule pertains to the pointcut in which the advice is bound to. The rule above assumes that the advice code `A_{adv}` will apply to the execution join point of `C.m()`. However, it may not. Or, rather, there may be a dynamic condition that depends on the call stack (as is the case with such AspectJ constructs as `cflow`) that will determine whether it is applicable. In addition, this, of course, cannot depend on anything that is contained in the body of `C.m()` nor can it be specified as part of the precondition of `C.m()`. Instead, it will depend on

\[
\{\text{guar}(\sigma) \land ap\} \ A_{adv} \ \{\text{rely}[\sigma/\sigma@pre, \sigma'/\sigma] \land aq\}
\]

\[
\begin{align*}
\text{C.m} & \:: \ \{\text{pre, post, guar, rely}\} \\
\{\text{pre} \land ap\} \ C.m() & + \ A \ \{(d \Rightarrow \text{post} \land aq) \land (\neg d \Rightarrow \text{post} \land ap)\}
\end{align*}
\]

Figure 4.6: Rule for aspect application (revised version).
the state of the call stack for each call to C.m() that is required to deal with in reasoning about the behavior of the overall system. To handle this, when reasoning about C.m(), if the pointcut associated with this advice is dynamic, the reasoning system will allow for both possibilities, namely, for the possibility that the associated condition is satisfied and for when it is not satisfied. For this purpose, there are two (possibly) distinct postconditions with the method, corresponding respectively to the cases when the method is called with the state of the call stack satisfying the condition of the dynamic pointcut and when this condition is not satisfied. These are marked, in the rule depicted in Figure 4.6, with the labels d and ¬d respectively, d being the condition specified in the dynamic pointcut for deciding whether the advice should apply when the execution of C.m() finishes.

There is one final complication, namely, the combination of the two problems identified above. That is, advice applicable to methods called within the body of C.m() may also have dynamic pointcuts associated with them. This means that the effect of dynamic pointcuts will not result in just two possibilities in the postcondition of C.m() but, rather, all possible combinations of the conditions corresponding to various dynamic pointcuts being or not being satisfied. In the worst case, this would give rise to $2^n$ possibilities, n being the number of calls in the body of C.m(). What is indicated here is that, while dynamic pointcuts are undoubtedly powerful, using them too liberally may lead to systems that are extremely difficult to specify or reason about. In fact, it is precisely this problem that forced Krishnamurthi et al. (2004), in their model checking approach, to introduce a depth parameter that is used as a threshold value to combat the explosion previously mentioned. This problem is indeed more general as illustrated above as it would also be encountered when a join point
resides either in a loop or is traversed multiple times because of recursion. Developing ways to address this problem is beyond the scope of this work.

4.4 Related Work

In this section, I discuss work that is related to the approach presented in this chapter and compare and contrast facets of each.

4.4.1 Aspect Restrictions

Several authors have proposed restrictions to AOP in order to address the complexity of the associated reasoning. Clifton et al. (2007) present MAO, a language that extends AspectJ with concern domains and control-limited advice. MAO, via static analysis, allows developers to restrict the behavior of advice, e.g., to allow accesses to only certain parts of the heap belonging to a particular concern domain. MAO also allows for restricting the manipulation of control-flow by advice thereby forbidding it to perturb the control-flow of the base-code in inappropriate ways. Such restrictions are expected to help simplify reasoning about AOP since developers can examine the signatures of each advice declaration to reason about its potential effects. Similar restrictions are also conceivable using my proposed rely clauses; however, my advice restrictions are more flexible and fine-grained in that rely clauses take an arbitrary assertion over two states \( \sigma \) and \( \sigma' \). These two states correspond to the state at the point in which advice obtained control in the former case and the state corresponding to the point when advises releases control in the later case. Furthermore, MAO does not provide the proper facilities to combine the specifications of the base-code and the advice to arrive at the overall behavior exhibited by the augmented system.
Nevertheless, it should be possible to adapt several of MAO’s ideas to help simplify my formalism.

Dantas and Walker (2006) propose “Harmless Advice,” a restricted form of advice that has minimal effects on the base-code, and develop a type system that enforces such behavior statically. Harmless advice cannot alter state (with the exception of I/O) and control-flow that is visible to the base-code. What is interesting about such advice is that, although highly constrained, it is shown to be quite useful especially in the domain of security. With the use of rely clauses, my approach could conceivably be adopted to relax some of the constraints on harmless advice to make it more “helpful” while maintaining local reasoning. This would allow the base-code developer to explicitly state, on a fine-grained level, the kinds of advice behavior that is considered “harmless” by means of a less restrictive rely assertion. Furthermore, JPTs can then used to reason about the overall effects of the advice applied to the base-code.

4.4.2 Aspect Classification

Various approaches (Clifton and Leavens 2002, Katz 2006, Rinard et al. 2004, Sihman and Katz 2003) attempt to categorize aspects based on their effects on the underlying system. The idea is that once these aspects have been classified, the burden associated with reasoning about their effects is alleviated for certain categories of aspects. The approach presented in this thesis, however, is focused on being very general and widely applicable in that reasoning can be performed about potentially advised code independent of any particular kind (with respect to its effects) of aspect.
4.4.3 Modular Aspect Verification

Krishnamurthi et al. (2004) propose a verification technique that can, using model-checking (Clarke et al., 2000), modularly verify advice independent of the base-code. Their proposal, given the base-code represented as a finite-state model, a set of properties that the augmented system (i.e., the base-code combined with the aspects) must satisfy, and a set of pointcuts where potential advice may be applicable, automatically generates enhanced interfaces that can be used for verifying the advice when it becomes available. Essentially, the interface captures the state of the model checking process prior to advice being added to the system. Goldman and Katz (2006) present a related technique using their MAVEN tool. While these approaches, as well as the approach presented in this thesis, all employ techniques that do not require repeated analysis of the entire augmented system each time a developer adds, removes, or changes advice, there are several key differences. Firstly, my proposed proof technique relies on deductive logical reasoning, while model checking entails a fundamentally different approach in that an abstract model is exhaustively examined for violations of a particular property. Furthermore, my approach is centered on combining the specifications of the base-code and that of the aspects using JPTs in order to assist developers in obtaining the overall behavior of the system. As such, my approach does not require a specific property that neither the base-code nor the augmented system must exhibit.

Comparable to my approach, Devereux (2003) also attempts to exploit the similarities between AOP and concurrent programs. The approach translates an aspect-oriented program into an equivalent, low-level concurrent program in an alternating-time logic formalism. The reasoning then is performed on this concurrent program
using an assume-guarantee paradigm (Henzinger et al., 1998). The focus here is on preserving particular properties of the base-code despite the addition of advice. My approach, through the use of JPTs, on the other hand, allows a developer to reason about the behavior of the base-code parameterized over any applicable aspect; therefore, reasoning about the base-code does not need to be reconstructed for each property being verified nor a specific property that the base-code must evince. My interests lie in obtaining the enriched behavior of the combined system as opposed to solely verifying the existence of interference freedom. Moreover, transformation from an aspect-oriented program to a concurrent program may cause the task of reasoning about the original program to be more difficult. That is, a change to either the base-code or an aspect could possibly result in previous reasoning efforts being invalidated. In addition, assume-guarantee reasoning in concurrent programs is normally leveraged with the acknowledgment that other processes may exist in the system. In AOP, however, aspects may not even have been developed yet or may be interchanged between different systems. As such, my proposed technique is designed more towards how AOP is used, especially to the plug-n-play style use inherent to aspects.

4.4.4 Aspect Interfaces

Several approaches (Aldrich, 2005; Griswold et al., 2006; Gudmundson and Kiczales, 2001; Lieberherr et al., 2003) attempt to augment traditional interfaces with various degrees of information regarding crosscutting concerns in order to improve reasoning. In particular, Kiczales and Mezini (2005) argue that in the presence of crosscutting concerns we cannot expect to work with the standard interfaces provided
by a class’ methods and their behaviors. Instead, we must define a more detailed interface for the class that includes information pertaining to how the system is intended to be deployed. These aspect-aware interfaces, which include the various join points at which advices defined in the aspects are applicable, accompany traditional interfaces, thus adding to their usefulness.

Aldrich (2005) introduces the notion of open modules. The idea is that the base-code designer should explicitly identify a specific set of points in the base-code at which advices will be allowed to apply; these are the points at which the base-code is open to accepting advice. Any attempt to define an aspect that contains an advice applicable at a point that is not included in this set will result in a (compile-time) error. Aldrich (2005) creates a language centered upon this concept and defines its operational semantics. These approaches, however, do not consider behavioral specifications of the classes nor how the enriched behavior of the classes resulting from the application of the aspects may be deduced.

4.4.5 Aspect Interference

Prior research (e.g., Durr et al., 2005; Katz and Katz, 2008) has considered aspects semantically interfering with one another. While my approach is focused on the obtaining the enriched behavior of both the base-code and the aspects together, and properly restricting advice via the rely clause, it is conceivable to extend my approach to dealing with the composition of advice specifications as well.

4.4.6 Aspect-Oriented Software Design

Modularity in the context of AOP has also been discussed in the realm of design. Sullivan et al. (2005) propose an approach that decouples the design of advice from the
design of the base-code. Their technique represents an departure from the traditional obliviousness (Filman and Friedman, 2000) method to the design of AOP. Sullivan et al. (2005) challenge the notion that base-code can be effectively designed and implemented with absolutely no knowledge of the influence of crosscutting concerns. Their strategy, which leverages information-hiding mechanisms, is then compared with the traditional obliviousness approach via a case study. The results show that utilizing the proposal in AOP designs yields an architecture in which modules are increasingly decoupled. Clemente et al. (2005) have a similar view expressing that ignoring crosscutting concerns during base-code design may result in a situation where, during implementation, the base-code must be “redesigned” in order to effectively accommodate crosscutting concern implementations. Both efforts raise serious questions as to how AO design methodologies may affect the reasoning process of such systems.

4.5 Future Work

The specifications resulting from my approach, as well as the method of combining aspect specifications with those of the base code to arrive at the overall behavior, tend to be complex. For future work, I will consider ways to simplify these. One possibility is to restrict the kinds of aspects that the reasoning system will deal with. In fact, Section 4.4.1 discusses several approaches made to restrict the behavior of aspects. I will examine each of these in detail to decide if such restrictions yield simplifications in my proposed specification scheme. As long as the restrictions are of the kinds that are satisfied in practice, the work will be helpful to AOP developers.

Furthermore, my technique is not currently automated. While my planned future work does not involve fully automating my reasoning technique, in the future, I will
explore ways to assist developers by mechanizing portions of the associated reasoning activities. This may be achieved using theorem provers, such as Isabelle\textsuperscript{26} and PVS\textsuperscript{27}.

4.6 Conclusion

Reasoning, specification, and verification of AO programs present unique challenges, especially as such programs evolve over time. Constructing an approach general enough to reason about components subject to the unpredictable frequency of advice applicability poses many obstacles including but not limited to usefulness, complexity, obliviousness, abstraction, and composition. In this chapter, I have presented my work, as well outlined future work, in developing such a technique that attempts to overcome these obstacles in an effort to enable tractable evolution of AOP. My proposed approach is aimed at tailoring specifications of these systems to their evolutionary plug-n-play nature and enhancing the expressiveness of constraints made on their constituent components.

\textsuperscript{26} http://isabelle.in.tum.de
\textsuperscript{27} http://pvs.csl.sri.com
CHAPTER 5: Conclusion

For a variety of reasons, modern, non-trivial software systems must evolve to cope with change, including alterations in stakeholder requirements, environments in which the software is deployed, and dependent technologies. Unfortunately, evolution and maintenance is an expensive, time-consuming, and error-prone task, especially when the system in question is large and complex.

In this thesis, I have outlined my work in exploring and developing a number of new techniques that will be of great value to software developers in evolving code to accommodate change. These techniques include automatically migrating legacy Java source code to utilize the new \texttt{enum} construct provided in Java 5, rejuvenating pointcut expressions upon changes to the base-code in Aspect-Oriented software, and specifying Aspect-Oriented program modules so that such specifications are more robust to change.

5.1 Contribution Summary

The main contributions documented in this thesis are summarized as follows. In Chapter \(^2\) I presented a semantic preserving, type inferencing algorithm that migrates legacy Java code employing the weak enum pattern to instead utilize the modern, type-safe \texttt{enum} language construct introduced in Java 5. The algorithm is implemented as a publicly available, open source plug-in for the Eclipse IDE. The tool was evaluated on seventeen open source applications, and the experimental results
indicate that not only did the it scale well to large applications but also was able to refactor 87% of all fields that could possibly be participating in the weak enum pattern.

In Chapter 3 I presented an approach that limits the problems associated with pointcut fragility in evolving Aspect-Oriented programs. The approach is also implemented as a publicly available, open source Eclipse IDE plug-in that can be used as a tool by AspectJ developers in maintaining pointcuts as the base code evolves. The tool does so by providing automated assistance to developers. 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 results of a corresponding experimental study, where the maximum structural commonality analysis depth set at two, were presented. These results indicate that the approach would be useful in rejuvenating pointcuts in real-world situations.

As mentioned in Chapter 4 reasoning, specification, and verification of Aspect-Oriented programs present unique challenges, especially as these programs evolve. In Chapter 4 I presented my work on developing a reasoning approach general enough to reason about components that may be under the influence of applicable advice. Without a global analysis, advice may intercept the base code at unpredictable frequencies and have an unpredictable behavior. While striving to be general, the specification notation of my approach aims to be useful, simple to understand, and easy to compose in an effort to enable tractable evolution of Aspect-Oriented programs. I have presented the fundamental concepts behind my specification technique and described
its notation and composition, including the novel notion of a join point trace (JPT). JPTs allow Aspect-Oriented software developers to reason about software components in a way that is akin to their evolutionary plug-n-play nature.

5.2 Future Work Summary

The main future work set forth in this thesis is summarized as follows. In Chapter 2, I have proposed future work that involves enhanced constant grouping to better resemble human-produced results, refactoring of other compensation patterns and/or extensions to the weak enum pattern, and an in-depth empirical evaluation utilizing a user study.

The future work mentioned in Chapter 3 includes integrating existing approaches to automatically reconstruct pointcut expressions to include the join point shadows suggested by my tool. I have also proposed future work that may contribute to improving the overall performance on the pointcut rejuvenation tool, including applying graph reduction techniques and faster intermediate representations. A general traceability link rejuvenation framework was also proposed.

The specifications resulting from the approach presented in Chapter 4 tend to be complex. In this chapter, I set forth future work that will consider ways to simplify the process of combining aspect specifications with those of the base code to arrive at the overall program behavior. One possible method is to restrict the kinds of aspects that the reasoning system will deal with. As long as the restrictions are of the kinds that are satisfied in practice, the work will be helpful to Aspect-Oriented software developers. Furthermore, my technique is not currently automated. Future
work also consists of exploring ways to assist developers by mechanizing portions of the associated reasoning activities.


171


E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: elements of reusable Object-Oriented software. Addison-Wesley, Mar. 1995.
M. Goldman and S. Katz. Modular generic verification of LTL properties for aspects.


J. Sillito, C. Dutchyn, A. D. Eisenberg, and K. D. Volder. Use case level pointcuts.


F. Steimann, P. Mayer, and A. Meißner. Decoupling classes with inferred interfaces.


Sun Microsystems. Java 2 Platform SE 5.0 API Specification, 2010b. URL [http://java.sun.com/j2se/1.5.0/docs/api](http://java.sun.com/j2se/1.5.0/docs/api) Last checked: March 26, 2011.


