Automatic refactoring history reconstruction and dynamic component adaptation frameworks for refactoring-based software component evolution

Kai-Shin Lu
Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/etd
Part of the Computer Sciences Commons

Recommended Citation
Automatic refactoring history reconstruction and dynamic component adaptation frameworks for refactoring-based software component evolution

by

Kai-Shin Lu

A dissertation submitted to the graduate faculty in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Science

Program of Study Committee:
Carl K. Chang, Major Professor
Samik Basu
Ying Cai
Simanta Mitra
Wensheng Zhang

Iowa State University
Ames, Iowa
2013
Copyright © Kai-Shin Lu, 2013. All rights reserved.
DEDICATION

To Joanna,
your loving support and encouragement provide me the strength and perseverance
to complete the PhD degree.

To Dad, Mom, and Sister,
thank you, for everything.

To my dear Matt,
you are the most wonderful distraction a dad could ever ask for.
TABLE OF CONTENTS

LIST OF FIGURES .......................................................... vi
LIST OF TABLES ............................................................. ix
ACKNOWLEDGEMENTS ..................................................... x
ABSTRACT ................................................................. xi

CHAPTER 1. GENERAL INTRODUCTION ................................. 1
  1.1 Introduction ......................................................... 1
    1.1.1 The Proposed Solution ...................................... 3
  1.2 Assumptions ....................................................... 6
  1.3 Thesis Organization ............................................. 7

CHAPTER 2. ALTA ...................................................... 8
  2.1 Introduction ....................................................... 8
    2.1.1 ALTA .......................................................... 9
    2.1.2 ALTA* ...................................................... 11
  2.2 Background ....................................................... 12
  2.3 Related works .................................................... 12
    2.3.1 Adapting by Aspect Oriented Programming .................. 12
    2.3.2 Full-Automatic Solutions ................................... 13
  2.4 Method .......................................................... 17
    2.4.1 Framework and Process ...................................... 17
    2.4.2 Refactoring Nodes and Paths ................................. 18
    2.4.3 Refactoring Dependency Resolver and Path Finder .......... 18
    2.4.4 ALTA Aspect Generator ..................................... 20
CHAPTER 3. TARP

3.1 Introduction

3.1.1 The Proposed Framework

3.2 Background

3.2.1 AI Planning

3.2.2 ALTA and ALTA*

3.3 Method

3.3.1 Preliminary Modeling Strategy

3.3.2 Handling Conflicting Names

3.3.3 Handling Inheritance

3.3.4 Handling Uncertain Identities in a Goal State

3.3.5 Supporting New API

3.3.6 Modeling API Deletion

3.3.7 Supporting Variadic Refactoring Patterns

3.3.8 Handling Huge Number of Objects

3.3.9 Verifying the Correctness of a Generated Path

3.3.10 Retrieving Another Solution

3.3.11 The Final Modeling Strategy

3.4 Implementation

3.4.1 System Architecture of TARP*

3.4.2 Encoding Details

3.5 Evaluation

3.5.1 Open Source Component Refactoring Path Reconstruction Test
LIST OF FIGURES

| Figure 1.1 | Example of Temporal-Dependent Refactoring Steps (TDRS) | 4 |
| Figure 1.2 | Example of a third-party API-caller | 5 |
| Figure 1.3 | Overview of the proposed solution | 6 |
| Figure 2.1 | Main idea of fixing incompatibility problems by migrating application | 9 |
| Figure 2.2 | The main idea of ALTA’s load-time adaptation | 11 |
| Figure 2.3 | Refactoring wizard in Eclipse | 13 |
| Figure 2.4 | Exporting refactoring information in Eclipse | 14 |
| Figure 2.5 | Sample refactoring history file (in XML format) | 15 |
| Figure 2.6 | Example of a third-party API-caller | 16 |
| Figure 2.7 | A refactoring which switches the first two parameters of a method | 16 |
| Figure 2.8 | The stub generated by ReBA | 16 |
| Figure 2.9 | The stub generated by Comeback! | 17 |
| Figure 2.10 | Architecture diagram of ALTA | 17 |
| Figure 2.11 | Refactoring paths which contains many linked refactoring nodes | 18 |
| Figure 2.12 | The algorithm of Resolver | 19 |
| Figure 2.13 | Sample aspect generated by Generator | 28 |
| Figure 2.14 | The process of compound refactoring tests | 29 |
| Figure 2.15 | Performance report. X-axis: number of method adapted in one class; y-axis: performance | 29 |
| Figure 2.16 | Performance report. X-axis: number of object created; y-axis: performance. 100% of the methods in each class were adapted | 30 |
Figure 2.17 Performance report. X-axis: the number of class adapted; y-axis: performance. 100% methods in each class were adapted.

Figure 2.18 Performance report. X-axis: adapted class count; y-axis: performance. 20% of the methods in each class were adapted.

Figure 3.1 An example of Temporal-Dependent Refactoring Steps (TDRS).

Figure 3.2 Example of a third-party API-caller.

Figure 3.3 Modules and conceptual data flows of TARP.

Figure 3.4 A model with multiple solutions.

Figure 3.5 Example of an AI planning problem.

Figure 3.6 The concept of ALTA’s runtime adaptation.

Figure 3.7 Example of the preliminary modeling strategy.

Figure 3.8 Example of the same-name problem.

Figure 3.9 Example of modeling a rename action.

Figure 3.10 Solution of the same-name problem.

Figure 3.11 An example of modeling inheritance relations.

Figure 3.12 An example of the uncertain identities problem.

Figure 3.13 Goal state without API-object identities.

Figure 3.14 The concept of a signature path.

Figure 3.15 Goal state without API-object identities but with signature paths.

Figure 3.16 A sample “rename method” actions that supports name path.

Figure 3.17 An example of supporting new methods.

Figure 3.18 An example of supporting method deletions.

Figure 3.19 The concept of Pull-up methods.

Figure 3.20 Model the pull-up method pattern by refactoring transaction and parameter reducing.

Figure 3.21 Example of an adaptation-base testing with a correct input refactoring path.

Figure 3.22 Example of an adaptation-base testing.
Figure 3.23  Example of retrieving a better solution by adding path tokens . . . . . . 70
Figure 3.24  Example of using a positive path token assertion with 2 negative path
token assertions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71
Figure 3.25  A 4x4 mapping problem . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 72
Figure 3.26  A MxN mapping problem . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
Figure 3.27  A sample model: the initial state . . . . . . . . . . . . . . . . . . . . . . . . . . 74
Figure 3.28  A sample model: the goal state . . . . . . . . . . . . . . . . . . . . . . . . . . 75
Figure 3.29  System Architecture of TARP* . . . . . . . . . . . . . . . . . . . . . . . . . . 79
Figure 3.30  The result screen of the Exp. 3 from LSdiff. . . . . . . . . . . . . . . . . . . . . 86
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Table 2.1</th>
<th>Comparison of full-automatic solutions</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 2.2</td>
<td>Compound Refactoring Tests (CRT) and Open-Source Library Tests (OSLT) Report</td>
<td>25</td>
</tr>
<tr>
<td>Table 3.1</td>
<td>Open Source Component Refactoring Path Reconstruction Test Report</td>
<td>87</td>
</tr>
<tr>
<td>Table 3.2</td>
<td>Open Source Component Refactoring Path Reconstruction Test: Refactoring Details</td>
<td>88</td>
</tr>
<tr>
<td>Table 3.3</td>
<td>Open Source Component Automatic Adaptation Results</td>
<td>90</td>
</tr>
</tbody>
</table>
I would like to take this opportunity to express my thanks to those who helped me to finish the journey towards a PhD.

To Professor Carl K. Chang, my adviser, thank you for your loving guidance, patience and support throughout this research and the writing of this dissertation.

I also want to extend sincere thanks to my committee members: Dr. Samik Basu, Dr. Ying Cai, Dr. Simanta Mitra and Dr. Wensheng Zhang, for your insightful comments on this work. Dr. Mitra, it is my pleasure to be your TA for so many years. I learnt a lot from you. And thank you, Dr. Hridesh Rajan, for your suggestions for one of my research topic.

Laurel, Thank you for your support throughout these years. I truly appreciate it.

Also in thanks to Hojun, Jeyoun, Hua, Li-Yuan, Hai-Hua, Jing-Wei, Li-Ping, Hen-I, Wei, Hsin-Yi, Jin-Chun, He-Yong, and all of my lab mates and friends, for your encouragement and suggestions to this project.

To Joanna, my dear wife, it is hard to find things to say when what I am trying to express is beyond words. I am sure you know what I want to say, because you always do.

To dad, mom and sister, thanks for your unconditional love, support and prayer.

Above all, thank You, my Lord, for Your grace, wisdom, faithfulness, and protection throughout this journey. Thank You for walking with me every step of the way, through the highs and lows of my day. To You be the glory forever!
ABSTRACT

Evolution of software components may lead to compatibility problems, such as incorrect executing results, compilation errors and system crashes. Solving those problems is a big challenge in software engineering.

In the past decade, many automatic solutions to address this issue have been proposed. However, all of them rely on extra change information (i.e., the information regarding the changes of the upgraded components). Without such information, none of the existing solutions can work. Therefore, how to fully automatically solve compatibility problems without extra information is still an important open issue.

In the current study, I proposed an end-to-end solution to fully automatically adapt incompatible components without resorting to any extra information. It is composed of two parts. The first part is TARP, an AI-planning based automatic refactoring history reconstruction framework. For an upgraded component, TARP can automatically reconstruct the missing refactoring history. The second part is ALTA, an automatic load-time adaptation framework, which can adapt incompatible components on-the-fly according to the refactoring history generated from TARP. Therefore, as an integrated solution with both TARP and ALTA, compatibility problems among application and components can be fully automatically solved to a very large extent.

The implementation of ALTA as ALTA*, and TARP as TARP*, were evaluated by conducting five sets of tests. The experimental results show that the TARP* + ALTA* solution can indeed fully automatically fix compatibility problems incurred to large-scale components without any additional information.
CHAPTER 1. GENERAL INTRODUCTION

1.1 Introduction

Software components upgrade frequently and some of the changes may lead to component incompatibility. Component incompatibility may cause serious problems including incorrect execution results, compilation errors and system crashes. Therefore, how to fix component incompatibility is an important research issue. In the past decade, many solutions to address this issue have been proposed, and most of them are semi-automatic [1, 2, 3, 4, 5, 6]. These solutions require manually coded upgrade information, such as delta files, upgrading annotation, or mapping rules, in order to then automatically migrate applications to fit new components. However, developers may not be willing to manually develop such information for end users, given that the process is usually complicated, fallible and time-consuming.

To overcome this limitation, several full-automatic solutions have been proposed [7, 8, 9]. Unlike semi-automatic ones, full-automatic solutions can work without human-coded change information. One of the assumptions underlying these solutions is that developers use Eclipse to refactor their components, thus the machine-recorded refactoring history can be available. With this valuable change information, these full-automatic solutions can either replay all changes to an application (i.e., to upgrade the application to fit the upgraded component) or to components (i.e., to generate adapter/wrapping layers which provide both old and new API) and solve the compatibility problems in a full-automatic fashion.

Although full-automatic solutions are impressive, all of them need to statically modify either application or upgraded component, which may be prohibited by the license agreements of the components. Moreover, it is not reasonable to assume that every end user can get refactoring history of upgraded components from Eclipse. First of all, developers may use tools
such as VI or notepad++, which do not automatically record refactoring history to refactor their components. Second, if developers do use Eclipse but do not follow the recommended steps (i.e., to use the refactoring wizards or hot keys) to refactor their components, Eclipse cannot record the history. Therefore, in order to fix compatibility problems in general cases, it is important to find a way to get refactoring history or change information directly from the components instead of relying on machine recorded ones.

In the past decade, many static analysis methods have been proposed to get change information directly from the source code of upgraded components. Antoniol et al. [10] formalized information on APIs into linear algebra and vector compositions to infer possible refactorings. Demeyer et al. [11] traced multiple versions of components and composed change metrics to infer possible refactoring actions. Xing and Stroulia [12] applied reverse-engineering techniques to the source code of the old (i.e., before upgrade) component and the new (i.e., after upgrade) component to generate UML models of them. After that, they compared the generated models to identify the changes of components. Godfrey and Zou [13] analyzed method-calling flow in order to recognize method splitting and merging. Dig [14], Weissgerber and Diehl [15] scanned the component’s source code and checked the similarities of all parts which shed light on the changes being made. Kim et al. [16, 17, 18, 19] compared the similarities of all parts first, then converted the results into template-based logic rules in order to recognize complex refactoring activities.

Although these solutions are impressive, all of them share the same limitations:

1. **Unable to detect Temporal-Dependent Refactoring Steps (TDRS):** It is common for developers to repeatedly refactor the same part of code [20]. TDRS are refactoring steps applied to the same part of components in sequence, and each step shares at least one transient refactoring parameter with its successor. A refactoring step is different from a refactoring pattern because a refactoring step includes refactoring parameters but a refactoring pattern does not. For instance, “move method $C1.m1$ to $C2$” is a refactoring step but “move method” is a refactoring pattern. **Transient refactoring parameters** are the refactoring parameters which do not exists either in the old or new
API.

Figure 1.1 illustrates this problem. Suppose when upgrading a component, you move a method \( m1 \) from class \( C1 \) to class \( C2 \), then rename class \( C2 \) to \( C3 \) (see Figure 1.1). Since \( C2.m1 \) (the dashed bubble in Figure 1.1 (B)) does not exist in either the old API or the new API, it is a transient refactoring parameter. Thus, these two refactoring steps which share it are TDRS. Because static analysis algorithms can only gather information from the old and new API, they can never detect any refactoring steps related to transient refactoring parameters. Therefore, none of them can detect TDRS.

2. **Unable to work without source code:** All methods mentioned above require source code to do static analysis. However, compatibility problems may occur among third-party components (see Figure 1.2). If binary releases of impacted components are the only resources we can get (see the shadowed component \( X \) in the middle of Figure 1.2 (A) and (B)), all existing solutions cannot work.

3. **Unable to verify generated results:** These algorithms only generate “inferred results” without validating. Therefore, the results might contain false positives (i.e., found refactorings did not exist in the real refactoring history) and false negatives (i.e., did not find refactorings existed in the real refactoring history). Hence, it is risky to use the results to conduct automatic component adaptations.

In summary, because of the critical limitations listed above, static analysis algorithms are not applicable to discover missing change information for automated component adaptation.

1.1.1 The Proposed Solution

In the current study, a novel solution is proposed that can fully automatically adapt incompatible components without any extra information. It is composed of two parts (see Figure 1.3). The first part is TARP (Testing and AI-Planning Based Refactoring Path Reconstruction Framework), an AI-planning based automatic refactoring history reconstruction framework. TARP is a novel solution for automatically reconstructing refactoring history (also known as refactoring path), which overcomes the three limitations of static-analysis based solutions. The
Figure 1.1 An example of Temporal-Dependent Refactoring Steps (TDRS): (A) shows the old API of this component, (B) shows the intermediate API, and (C) shows the new API. Method $m1$ in class $C1$ (denoted as $C1.m1$) was moved to class $C2$ and then $C2$ was renamed to $C3$. These two refactoring steps, “move method $C1.m1$ to $C2$” and “rename class $C2$ to $C3$”, are TDRS.

The main idea of TARP is that it transfers a compatibility problems into an AI-planning problem, while all supported refactoring patterns are available AI-planning actions (operations). In this way, a generated plan is actually a refactoring path. TARP also uses an innovative technique called adaptation-based testing which can verify if the generated path is correct. If incorrect, TARP will go back to find another path, until it gets a right one. With TARP, the missing refactoring history can be reconstructed by solely processing the old and new binary jar files.

The second part is ALTA, an automatic load-time adaptation framework for refactoring-based evolution of software component. ALTA is an Aspect-Oriented-Programming (AOP) based on-the-fly automatic adaptation framework. By inputting refactoring history, ALTA can generate run-time adaptation logic according to the given refactoring history, which can dynamically weave the binary code to let an old application run with a new component without any problem and fix compatibility problems on-the-fly. In this way, no applications or components will be statically modified; therefore this solution is valid under all kinds of license agreements.

Besides, ALTA is the foundation of TARP because TARP adopts ALTA internally to perform adaptation-based testings. The main idea of adaptation-based testing is the following.
Figure 1.2  Example of a third-party API-caller. (A) Before upgrading component Y. (B) After upgrading component Y. After upgrading Y, X and Y became incompatible.

Suppose that we have a set of old test cases (i.e., the tests generated for the old component) which covers all the methods in the old API. Now, let us run the old tests directly with the new component. If the old and new components are fully compatible, all the tests shall pass. But if there are compatibility problems between the old and new components, we shall be able to see problems (either errors or failures) in the test report — unless we can find a way to automatically and fix all compatibility problems between these two components. Therefore, when TARP gets a refactoring path from the internal AI planner, TARP will assume that the path is correct, and ask ALTA to on-the-fly adapt old test cases with the new component. If there are problems showed in the test report, TARP will know the path is incorrect. On the other hand, if all the tests passed, TARP will know that the correct path has been found. In this way, TARP successfully verify a generated refactoring path by performing adaptation-based testing via ALTA.

The implementation of ALTA as ALTA*, and TARP as TARP*, were evaluated by conducting multiple sets of tests, including several open-source project’s tests. The experimental results show that the TARP* + ALTA* solution is capable of fully automatically fixing compatibility problems among large-scale components without any additional information.

In summary, the TARP + ALTA solution intends to achieve the following goals:

1. Can fully automatically adapt incompatible components without any extra information.
Figure 1.3 Overview of the proposed solution. The white arrows (with solid lines) represent data flows.

2. Can work without any source code of either applications or components.

3. Will not statically modify any application or component. In other words, all adaptations can be done dynamically.


1.2 Assumptions

This TARP + ALTA solution is under the following assumptions. For a given set of old (i.e., before upgrade) and new (i.e., after upgrade) components:

1. All the refactoring actions applied to the old component are supported by ALTA as well as TARP. In addition, no API has been deleted from the old component (i.e., no API deletion). If this assumption does not hold, TARP will not be able to generate the correct refactoring path, or TARP will not be able to use ALTA to verify the generated path.

2. The third-party AI planner included in TARP is able to generate a result, either a concrete plan or a notice saying that there is no possible solution, for every model generated from TARP as long as it is written in standard PDDL 2.1 [22]. If this assumption does not hold, TARP may not be able to produce a reconstructed refactoring history.
3. The third-party test case generator included in TARP is able to generate test cases with regression assertions [23] which cover all methods impacted by refactoring actions. In other words, the test cases generated by TARP will be able to launch each impacted method at least once and verify the correctness of the return value. If this assumption does not hold, TARP will not be able to guarantee the correctness of generated refactoring history.

1.3 Thesis Organization

The rest of this thesis is structured as follows. In Chapter 2, I introduce ALTA, and in Chapter 3, I introduce TARP. Concluding remarks and future research direction are presented in Chapter 4.
CHAPTER 2. ALTA: Automatic Load-time Adaptation Technique for Refactoring-based Evolution of Software Component

2.1 Introduction

Software evolution and maintenance is a fact of life [24, 25]. Enhancements, modifications, and bug fixes are routinely made to a software component during its usable life. Sometimes, upgrades can result in compatibility problems, such as incorrect executing results, compilation errors and system crashes. Solving those problems is a big challenge in software engineering.

In past decade, a number of solutions categorized as semi-automatic have been proposed [1, 2, 3, 4, 5, 6, 7, 8, 9]. Most of them require manually-defined “upgrading information” for applications, such as conversion/mapping rules [4], delta files [1], communication protocols [3] or upgrading annotations [2, 4, 5, 6]. With this information, these solutions can modify the applications to fit the new Application Programming Interfaces (APIs) of the upgraded components and eliminate compatibility problems in a system. Figure 2.1 illustrates this idea. In Figure 2.1, each shape represents a public method or field. API-callers are represented in black color, whereas API-providers in white color.

While semi-automatic solutions are promising methods, they are workable only when upgrading information is defined. An end user of software components may not have sufficient knowledge to define upgrading information, and the developer who upgrades the component may not be willing to manually define upgrading rules because it is a time-consuming task. Therefore, current semi-automatic solutions are not easily employed.

CatchUp! [7], ReBA [8] and Comeback! [9] are full-automatic solutions for component adaptation. All of them require machine-recorded refactoring history. ‘In principle, any change to a software program that preserves behavior can be understood as a refactoring.’ [9, P.3]
When people refactor their components in Eclipse IDE (Integrated Development Environment) [20], all refactor actions (i.e., refactorings) are automatically logged into refactoring history. By analyzing refactoring history, these techniques can gather sufficient information to adapt components, eliminating the need for manually-defined upgrading information.

Although full-automatic solutions are more practical than semi-automatic ones, these three solutions have several limitations. For example, CatchUp! requires application source code, which are not always available. ReBA and Comeback! cannot support refactorings that will lead to conflict method signatures (called conflict-making refactorings in the rest of this paper), such as changing the order of same type parameters, changing return types, hiding methods, and adding new exceptions. In addition, all of them will statically modify source or binary files, which may violate those components’ license agreements.

In this study, we proposed an automatic load-time adaptation technique for refactoring-based evolution of software component (ALTA), a full-automatic compatibility solution for refactoring-based evolution of software component, and ALTA*, an implementation of ALTA.

2.1.1 ALTA

The goal of ALTA is to overcome the limitations of previous methods. ALTA automatically analyzes the refactoring history of the upgraded component, then generates a Jar file named ALTA Aspect, which contains the logic of load-time adaptation written in AspectJ language. By simply adding ALTA Aspect into classpath and specifying AspectJ’s class loader, users can...
correctly run the old application with upgraded components on standard JVM (Java Virtual Machine).

ALTA has the following four important features:

1. **Full-automatic adaptation**: ALTA utilizes the refactoring history of upgraded components; therefore it does not require any manually-defined upgrading information.

2. **Load-time binary adaptation**: ALTA uses the load-time weaving (LTW) technique of AspectJ, which can adapt components when they are loaded. Therefore, ALTA will not modify applications or components statically. ALTA also allows users to disable this feature if there is no modification prohibition.

3. **Source code free**: ALTA does not require any source code of applications or components.

4. **Supporting conflict-making refactorings**: By using the `within` keyword of AspectJ, ALTA can change the behaviors of old method calls and preserve the behaviors of new method calls. Therefore, it can support conflict-making refactorings.

ALTA is the first full-automatic compatibility solution supporting conflict-making refactorings. (See Table 2.1). In addition, ALTA also supports newer applications designed for upgraded components. Because newer applications do not have compatibility problems with upgraded components, they only need to be launched with an empty ALTA Aspect. Figure 2.2 shows the adaptation concept of ALTA.

---

Table 2.1  Comparison of full-automatic solutions. Red and Italic fonts highlighted the parts that leave room for improvement.

<table>
<thead>
<tr>
<th>Item</th>
<th>Feature</th>
<th>CatchUp!</th>
<th>ReBA</th>
<th>Comeback!</th>
<th>ALTA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Static modification target</td>
<td>Application</td>
<td>Component</td>
<td>Component</td>
<td>None</td>
</tr>
<tr>
<td>2</td>
<td>Can work without source code?</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>Support load-time modification?</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>4</td>
<td>Support conflict-making refactorings?</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Figure 2.2 The main idea of ALTA’s load-time adaptation. (A) System after evolution. Application 1 (App 1) and the upgraded Component (Cmp ver 2) are not compatible. (B) With ALTA Aspect, App 1 can run with Cmp ver 2 correctly. App 2 also runs under ALTA, but because it was designed with Cmp ver 2, ALTA Aspect for App 2 is empty.

2.1.2 ALTA*

ALTA* is an implementation of ALTA. Currently ALTA* supports 12 categories of refactoring: 1) Change method signatures (including add/remove parameter, change the order of parameters, rename method, change exception types, and change return type), 2) Move method, 3) Rename field, 4) Move field, 5) Extract method, 6) Rename type, 7) Move type, 8) Delete type, 9) Rename package, 10) Delete Package, 11) Remove Package, and 12) Delete method.

We evaluated ALTA* with the following three types of experiments:

1. Compound refactoring tests: We consecutively applied different refactor actions to one component and then asked ALTA* to adapt it to its old test cases. The experimental results show that ALTA* can correctly adapt compound refactorings. This ability is important because people may refactor a type, method or field repeatedly.

2. Open-source library tests: We randomly applied different refactoring actions to Apache Commons library (version 3.0.1), then asked ALTA* to adapt it to its official test cases. The experimental results show that ALTA* can effectively solve incompatibility problems in real-world components.

3. Performance tests: We measured the performance of ALTA*. The experimental results
show that the performance overhead of load-time adaptation feature is around 11%.
However, if users disable this feature, the performance overhead could be negligible.

The rest of this paper is organized as follows. Section 2.2 reviews information about the refactoring process and how it is used in Eclipse. Section 2.3 discusses related works. Section 2.4 describes the proposed method. Section 2.5 shows the evaluation of our approach. Finally, we draw conclusions in Section 2.6.

2.2 Background

Eclipse supports several types of refactorings, such as Change Method Signature and Move Method. Suppose that there is a method printCode() defined as the following codes:

```java
public void printCode(int code){
    System.out.println("Code="+code);
}
```

If users want to add a String-typed parameter named message to the printCode() method in Eclipse, they just need to right click on the printCode() method in Eclipse’s text editor and click the “Refactor → Change Method Signature...” menu items. Then a GUI wizard will show up for users to change the signature, and they just need to add a parameter here (see Figure 2.3). After pressing “Ok”, Eclipse will do the rest for them, including updating all method callers. Moreover, by using refactoring wizards, Eclipse will automatically log all the refactor actions. After that, users can export the refactoring history as a separate XML file, or include the history file in exported files (see Figure 2.4). Figure 2.5 shows a sample refactoring history file, which contains a Rename Method refactoring and a Rename Type refactoring.

2.3 Related works

2.3.1 Adapting by Aspect Oriented Programming

Using the AOP (Aspect-Oriented Programming) technique to do software adaptation is not a new idea [26, 27, 28]. Camara et al. proposed a framework to support COTS composition [29], Sanchez et al. used AOP to adapt synchronization policies [27]. However, ALTA is the first
solution that conducts system-wide component adaptation without requiring any predefined rules, protocol or middleware.

2.3.2 Full-Automatic Solutions

2.3.2.1 CatchUp!

CatchUp! \cite{7} is the first full-automatic solution for solving compatibility problems. Two primary assumptions are behind the solution. The first is that people use Eclipse to refactor their components. Because Eclipse automatically logs all refector actions into refactoring history, CatchUp! can use the refactoring history rather than human-coded upgrading information to migrate incompatible applications. Another assumption, though indirect, is that developers who upgrade the components are willing to share the refactoring history with users.

With CatchUp!, if the refactoring history of component is available, CatchUp! will replay each refector action one by one to the application; therefore CatchUp! will upgrade the
application to fit the new APIs.

Although the solution is promising, it does not work if API-callers’ source code are unavailable. For example, all Eclipse plugins call the APIs of Eclipse framework but many of them are released only in binary form [8]. In addition, one binary-released component may rely on some other components. If there are any compatibility problems among those components, CatchUp! cannot function. Figure 2.6 illustrates this idea. To sum, requiring source code is a significant limitation of CatchUp!. ALTA does not have this limitation because ALTA does not require any source code.

2.3.2.2 ReBA and Comeback!

ReBA [8] and Comeback! [9] followed same assumptions of CatchUp!. They overcame limitations by instrumenting binaries of components instead of modifying their source code. ReBA starts with the upgraded components (i.e., the components which have new APIs). Next, it reads the refactoring history, then REVERSELY (i.e., from tail to head) processes each refactoring action to create a backward-compatible layer.
Comeback! is slightly different. It starts with the old component. First, it copies the old APIs into a wrapping layer. Unlike ReBA, these APIs are all empty stubs. Next, Comeback! migrates the APIs in the binary wrapping layer by repeatedly replaying the refactoring history. Finally, the stubs in the wrapping layer delegate all calls to the real (upgraded) components.

ReBA and Comeback! are both practical solutions because they can work without source code. However, they share two limitations. First, both of them need to modify or copy the binaries of components statically, which may be prohibited by the license agreements of the components. Comeback! hides the upgraded components under the wrapping layer, and thus needs to change the type information of the upgraded components. Although ReBA will not modify any components directly, it needs to copy part of the bytecodes of components to the backward-compatible layer. In other words, both solutions will be invalid under certain license agreements. ALTA has the advantage of working with all kinds of license agreements before it adapts components during the load-time.
Figure 2.6 Example of a third-party API-caller. (A) Before upgrading Cmp (i.e., component) B. (B) After upgrading Cmp B. Cmp B and A became incompatible, while only the source of App (i.e., the application) is available.

Both ReBA and Comeback! provide old APIs and new APIs simultaneously. Therefore, they are not able to support conflict-making refactorings. Figure 2.7 shows an example refactor action. To handle this situation, ReBA will insert a new stub into the compatible layer (note that it starts from the new APIs) shown in Figure 2.8.

```
1 Change method 'public int util.Math.div(int i, int j, String msg)' to 'public int div(int j, int i, String msg)'
```

Figure 2.7 A refactoring which switches the first two parameters of method div().

```
1 util.Math.div(int j, int i, String msg); //beginning
2 util.Math.div(int i, int j, String msg); //added
```

Figure 2.8 The stub generated by ReBA

However, this insertion will fail because the new stub (line 2 in Figure 2.8) has the same method signature with the existing one (line 1 in Figure 2.8). Comeback! will create a wrapping layer (note that it starts from the old APIs) shown in Figure 2.9, which also fails to put the conflicting interfaces together. If ReBA and Comeback! skip the refactoring, then the entire adaptation result will become incorrect. ALTA is unique in this aspect because it is able to adapt conflict-making refactorings.
2.4 Method

2.4.1 Framework and Process

There are three main parts of ALTA (see Figure 2.10): Refactoring Dependency Resolver and Path Finder (denoted as “Resolver” in the rest of this paper), ALTA Aspect Generator (denoted as “Generator”), and Refactoring Categories Plugins (denoted as “Plugins”). Plugins are the foundation of Resolver and Generator, because Resolver and Generator will ask Plugins to provide critical information regarding specific refactoring categories.

When a refactoring history file is given, Resolver will first convert the history into a set of refactoring paths, then Generator will use those refactoring paths to generate adapting logic written in AspectJ. Next, ALTA will use AJC (the compiler of AspectJ) to compile the aspects and produce a single Jar file (called ALTA Aspect). Finally, by indicating AspectJ class loader and the ALTA Aspect, users can run the old applications with the upgraded components on standard Java Virtual Machine (JVM).
2.4.2 Refactoring Nodes and Paths

The goal of Resolver is to analyze a given refactoring history and produce a set of refactoring paths. A refactoring path is composed of linked refactoring nodes. Refactoring nodes in one path are related to one another. Figure 2.11 (A) shows a refactoring path as well as the basic structure of a refactoring node. A refactoring node is composed of three elements: 1) the identity (signature) before this refactoring, 2) detailed information regarding this refactoring (i.e., the raw data of this XML entry) 3) the identity after refactoring. For any two linked nodes NodeX → NodeY, NodeX’s identity-after-change should always be equivalent to NodeY’s identity-before-change. Figure 2.11 (B) shows an example of this concept. The identities inside the two red circles are the same. After Resolver processes all the refactorings, the first refactoring node’s identity-before-change in each path should exist in the old component (i.e., before upgraded), and the last refactoring node’s identity-after-change should exist in the new (i.e., upgraded) component (see Figure 2.11 (C)).

![Refactoring paths which contains many linked refactoring nodes.](image)

2.4.3 Refactoring Dependency Resolver and Path Finder

Figure 2.12 shows the algorithm of Resolver.

In the beginning, Resolver will process one refactoring at a time (line 1 in Figure 2.12), then find the correspondent plugin to construct the refactoring node. A plugin knows how to retrieve the identity-before/after-change from XML. Then, Resolver will create a refactoring
for each refactoring $R$ in the refactoring history{
    Get $R$'s refactoring type $T$.
    Use $T$ to get $R$'s corresponding plugin $P$.
    Create one refactoring node $N$ which represents $R$ by $P$.
    if ($N$ can be appended into an existing path $H$){
        Append $N$ to the end of $H$.
    }
    else{
        Create a new (empty) path $H$.
        if ($R$ is not about changing package){
            Generate compensative ancient nodes.
            Add these compensative ancient nodes to $H$.
        }
        else{
            Add $N$ to $H$.
        }
        if ($R$ is not about changing method){
            Generate impacted nodes.
            Append these impacted nodes into all related lists.
        }
    }
}

Figure 2.12 The algorithm of Resolver (Refactoring Dependency Resolver and Path Finder).

node for this refactoring, then find out if this node can be appended to an existing refactoring path. If the answer is yes, Resolver will append it to that path (line 6). If not, Resolver will create a new (empty) path for it. However, before adding this node to the new path, we need to consider refactorings that happened before. For example, suppose that there are two refactoring actions in the following history file:

```plaintext
rename ClassA to ClassB;
rename ClassB.methodX() to ClassB.methodY();
```

When Resolver processes line 1 above (i.e., `rename ClassA to ClassB`), it will create one node (denoted as `nodeOfLine1`), and its identity-before-change is `ClassA` and identity-after-change is `ClassB`. Next, Resolver will create one new path, then add `nodeOfLine1` to that path. Later, when Resolver processes line 2 above, it will create a node (denoted as `nodeOfLine2`), and its identity-before-change is `ClassB.methodX()` and the identity-after-change is `ClassB.methodY()`. Because `ClassB` is not equivalent to `ClassB.methodX()`, `nodeOfLine2` cannot be appended to the path which contains `nodeOfLine2`. In this case, Resolver will create a new path for adding `nodeOfLine2`. However, `nodeOfLine1` cannot be the first node of any path because its identity-
before-change (i.e., \texttt{ClassB.methodX()}) does not exist in the old component. The reason for this nonexistence is that the refactoring in line 1 above already renamed \texttt{ClassA} to \texttt{ClassB}. Therefore, Resolver will create a special node called \textbf{compensative ancient node}, and its identity-before-change is \texttt{ClassA.methodX()} and identity-before-change is \texttt{ClassB.methodX()} (see line 11 and 12 in Figure 2.12). It is important to note that the identity-before-change of the created \textbf{compensative ancient node} exists in the old component. Finally, Resolver will add the \textbf{compensative ancient node} and the \texttt{nodeOfLine2} into the new path respectively.

Similarly, if we rename a method \texttt{ClassA.methodX()} to \texttt{ClassA.methodY()} first and then rename \texttt{ClassA} to \texttt{ClassB}, Resolver will generate an additional node given that \texttt{ClassA.methodY()} cannot be found in the new component. The identity-before-change of this new node is \texttt{ClassA.methodY()} and its identity-after-change is \texttt{ClassB.methodY()}. Later, Resolver will append it after the node which represents the \texttt{rename method} refactoring. We call this additional node an \textbf{impacted node} (see line 17 and 18 in Figure 2.12). In this manner, Resolver will produce a set of refactoring paths which satisfies Generator’s needs at the end.

\subsection{2.4.4 ALTA Aspect Generator}

The goal of Generator is to generate adaptive aspects based on a given set of refactoring paths. Figure 2.13 shows a sample aspect generated by Generator. In the following situations, Generator will apply different strategies to adapt components.

\subsubsection{2.4.4.1 If there is a missing method}

Generator will use AspectJ’s \texttt{inter-type declaration} to declare the missing method. The content of the declared method is to delegate the call to the correct target. See line 8-12 in Figure 2.13.

\subsubsection{2.4.4.2 If there are conflict method signatures}

Generator will use pointcuts with the \texttt{within} keyword in AspectJ to delegate old method calls and keep new method calls unchanged. See line 19-24 in Figure 2.13.
2.4.4.3 If there is a deleted type (i.e., a class or a interface) or package

Generator will do nothing, but ALTA can use the classpath priority to let the application search the required type or packages inside the upgraded components first, then search the old components\textsuperscript{1}. Because the application will not find the deleted type or package in the new components, the deleted type or package will be loaded from the old components.

2.4.4.4 If there is a renamed class

Generator will use inter-type declaration to declare a hidden field in the old (before renamed) class, and the hidden field’s type is the renamed class. Generator will use AspectJ’s wild card pointcuts to forward all method calls toward the old type to the hidden object’s corresponding method calls. In other words, this is indeed an AOP-based realization of an object-wrapping technique. See line 4-6, 14-17 and 26-50 in Figure 2.13.

2.4.4.5 If there is a renamed interface

Generator will use inter-type declaration’s Declare Parents technique to declare the missing interface.

2.4.4.6 If there is a deleted method

Generator will copy the method body of the deleted method (in binary form) and statically inject it into the original owner type of this method. Because it will change the upgraded component, this walk-around solution can only be applied when there is no modification restrictions.

2.4.5 Complete Example

Suppose that there is a ClassA.divide(int i, int j) API in the old component before upgrade. During upgrade, it is first renamed to ClassA.division(int i, int j), then renamed AGAIN to ClassA.div(int i, int j). Therefore, in the new component, there is

\textsuperscript{1}To support deletion of types or packages, uses need to append the paths of old (before-upgrade) components to the end of runtime classpath.
a `ClassA.div(int i, int j)` API. And the refactoring history contains these two *rename method* refactorings.

When ALTA receives the refactoring history file mentioned above, Resolver will build a path that contains only two nodes. In the first (heading) node, the identity-before-change is `ClassA.divide(int i, int j)`, and the identity-after-change is `ClassA.division(int i, int j)`. In the second (tailing) node, the identity-before-change is `ClassA.division(int i, int j)`, and the identity-after-change is `ClassA.div(int i, int j)`. After retrieving the paths, Generator will generate adaptation logics via predefined strategies. In this example, Generator will use AspectJ’s inter-type declaration to declare a `ClassA.divide(int i, int j)` method, and its content simply forwards this call to `ClassA.div(int i, int j)`. *Generator will skip all intermediate identities* so that methods calls will not be forwarded many times. With this load-time adapting rule, old applications can invoke `divide(...)` in the upgraded components without any problem.

Regarding the switching parameter example mentioned in Section 2.3.2.2, the first two parameters in method `int ClassA.div(int i,int j,String msg)` will be switched. However, because this refactoring will not change the method signature, ReBA [8] and Comeback! [9] will fail to generate adapting layers. In this case, ALTA will use AspectJ’s pointcut to define the following rules:

```java
1 around the method call
2 "int ClassA.div(int i,int j,String msg)" is invoked {
3    // internal calls
4    if (this call is invoked from the component itself){
5        invoke ClassA.div(i,j,msg), then return the result.
6    }
7    // external call
8    else{
9        invoke ClassA.div(j,i,msg), then return the result.
10    }
```

The if statement shown in line 4 above is made possible by the *within* keyword of AspectJ (you can also see line 19-24 in Figure 2.13.). With the rules above, all `div(int,int,String)`
calls invoked from the component itself will simply use the upgraded version (see line 5 above), but all external calls including old applications will call the same method while the first two parameters are swapped (see line 9 above). In this way, ALTA successfully supports conflict APIs.

2.5 Evaluation

We conducted three types of experiments to evaluate ALTA*, the implementation of the ALTA framework. All experiments were conducted on a laptop with Intel Core i5 2.50 GHz processor, and 4.00 GB of RAM.

2.5.1 Compound Refactoring Tests

Because developers may refactor a type, method or field repeatedly (these related refactorings are called compound refactorings), it is important to verify if ALTA* can correctly support compound refactorings. Therefore, we customized a set of components named Component version 1, then generated a set of test cases named Tests for Component version 1 by running Randoop [23], a state-of-the-art automatic test case generator. Randoop will generate not only the tests but also the regression assertions [23] for the components.

Next, we used Eclipse to consecutively apply different refactorings to some types or methods in the components, E.g., rename method pkg1.ClassA.methodX() to pkg1.ClassA.methodY(), rename package pkg1 to pkg2, and add one parameter to methodY(). This gave us upgraded components Component version 2, which was not compatible with Tests for Component version 1. Next, we exported the refactoring history as an XML script and passed it to ALTA* in order to generate ALTA Aspect. Finally, by designating AspectJ’s class loader and ALTA Aspect, we ran Tests for Component version 1 with Component version 2 on standard JVM. Figure 2.14 shows the test process.

Row 1 to 6 of Table 2.2 shows the test results. The “CRT 1” experiment (see row 1 in Table 3.1) shows that the component was applied for two consecutive refactorings: rename a type and then rename one of its methods (see column 2). A total of 4,299 tests were run with
this upgraded component, which took 3,747 seconds. All passed, and the branch coverage of
the tests was 100% (see column 3).

CRT 2 was a complex case that changed a single method three times. CRT 3 contained a
hide method refactoring and CRT 4 had a change return type refactoring. Both are conflict-
making refactorings. In CRT 5, we added a String-typed parameter into method methodB(),
then removed one parameter. CRT 6 is the same example discussed in Section 2.3 that ReBA
[8] and Comeback! [9] could not support. All the CRT tests passed perfectly, showing that
ALTA* can correctly adapt compound refactorings, including conflict-making refactorings. This
ability is important because people may apply different refactor actions to one method (or type)
consecutively.

2.5.2 Open-Source Library Tests

We aimed to evaluate ALTA* with real-world components and their official test cases. To
achieve this goal, we conducted open-source library tests. The test process of Open-Source
Library Tests (OSLT) was similar to the process of CRT. However, in OSLT, we used real-
world open-source libraries as the subjects rather than self-created components. In addition,
we used official test cases released with the libraries to be the applications instead of auto-
generating test cases. In this experiment, we selected Apache Commons library version 3.0.1
as our subject, and its lines of code (LOC) is 104K. We randomly applied different refactorings
to it and then asked ALTA* to adapt the refactored library to the old official tests. The results
displayed in Table 2.2 row 7 and 8 show that ALTA* can effectively solve the incompatibility
problems in real-world components.

2.5.3 Performance Tests

We measured three different aspects of ALTA*’s performance. First of all, we tried to
understand the relation between adapted method count in one class and overall execution
time. In the target component, there was only one class which contains 10 methods. During
the tests, the application called all of the 10 methods in sequence 10 to 100 times. In each
method, we just use a FOR loop to call sum+=sum*a 1,000,000 times. Figure 2.15 shows the
Table 2.2  Compound Refactoring Tests (CRT) and Open-Source Library Tests (OSLT) Report.

<table>
<thead>
<tr>
<th>Exp No.</th>
<th>Refactoring Information</th>
<th>Test Result and Branch Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>CRT 1</td>
<td>Rename type ‘b.ClassA’ to ‘ClassA_ren’; Rename method ‘b.ClassA_ren.methodB(...)’ to ‘methodB_REN’; Rename package ‘b’ to ‘b_ren’</td>
<td>Tests run: 4299, Failures: 0, Errors: 0, Time elapsed: 3.747 sec, 100%</td>
</tr>
<tr>
<td>CRT 3</td>
<td>Change method ‘public int A.ClassA.methodA(int a)’ to ‘private int methodA(int a)’</td>
<td>Tests run: 562, Failures: 0, Errors: 0, Time elapsed: 0.706 sec, 100%</td>
</tr>
<tr>
<td>CRT 4</td>
<td>Change method ‘public int b.ClassA.methodB(int c, int d, int f)’ to ‘public long methodB(int c, int d, int f)’; Change method ‘public int b.ClassA.methodA(int a)’ to ‘public long methodA(int a)’</td>
<td>Tests run: 3883, Failures: 0, Errors: 0, Time elapsed: 2.224 sec, 100%</td>
</tr>
<tr>
<td>CRT 5</td>
<td>Change method ‘public void A.ClassA.methodB(int c, int f, int d)’ to ‘public void methodB(int c, String pig, int f)’; Rename package ‘A’ to ‘A_REN’</td>
<td>Tests run: 1610, Failures: 0, Errors: 0, Time elapsed: 2.634 sec, 100%</td>
</tr>
</tbody>
</table>
result, where all methods were adapted by AspectJ’s inter-type declaration technique in (A) and the pointcuts technique in (B). There are 3 lines in Figure 2.15 (A) and (B): the blue line with the diamond-shaped legend represents the performance of NO AOP (i.e., running the compatible applications and components without any adaptation), the red line the square-shaped legend shows the performance of Static AOP adaptation (i.e., the LTW feature was disabled), and the green line the triangle-shaped legend shows the performance of LTW AOP (load-time weaving AOP adaptation). Figure 2.15 shows that if there is only one class, then the performance difference among these three modes can be ignored. This is reasonable because if there is only one class, the AspectJ’s class loader only needs to change one class definition during the load time; therefore the overhead is negligible.

Second, we wanted to know the relation between the number of created objects and performance. We generated 100 component classes, with each one containing 10 methods. All of these methods were incompatible with the application and adapted by the inter-type declaration technique. During the tests, the application called all of the 10 methods of each created object 100 to 1000 times. In this set of tests, we ran $\text{sum}+=\text{sum}*a$ 10,000 times in each method. Figure 2.16 (A) shows the results. The performance difference between No AOP and LTW AOP was close to a constant value 0.71 (second). (B) shows the overhead ratio. Because the performance difference is a constant value, the overhead ratio decreased when the number of created objects increased. This result is reasonable due to the fact that LTW AOP only change the class definition when the classes are loaded. If the number of classes is fixed, the performance overhead should be fixed as well.

Third, we wanted to know the relation between class count and performance. We generated lots of classes, each one containing 10 methods, and all the methods were incompatible with an application, so all of them needed to be adapted. We used inter-type declarations to adapt those methods. In this set of tests, we ran $\text{sum}+=\text{sum}*a$ 1,000,000 times in each method. Figure 2.17 (A) showed the result: when the class count increased, the performance difference between No AOP and LTW AOP was also increased. Figure 2.17 (B) showed the overhead ratio: when there were 700 classes, LTW AOP took almost 200% of time to finish the test. This is unacceptable.

However, the test results shown in Figure 2.17 were driven from extreme cases. In reality,
people rarely upgrade every method in all classes. Therefore, we adjusted the setting, adapting 2 (out of 10) methods in a class only. This setting was much more reasonable. According to the data shown in Figure 2.18 (A) and (B), ALTA’s LTW AOP adaptation overhead decreased when the class count was less than 300, but it slightly increased after class count ≥ 400. We think that AspectJ’s class loader did some performance optimizations so the local minimum appeared when the class count equaled to 300. The average overhead ratio of all tests shown in 2.18 (B) was 0.1135. Therefore, we conclude that the performance overhead of the load-time adaptation feature (if enabled) was around 11%. In addition, all the test results showed that the performance overhead of Static AOP was negligible.

2.6 Conclusion

Upgrading software components may lead to compatibility problems. Generally speaking, people should upgrade their applications to adopt new APIs. However, modifications of existing applications can be risky and costly. In this study, we proposed ALTA, a complete solution that can perform full-automatic load-time binary adaptation, and ALTA*, a tool that implements ALTA. As long as the refactoring history of upgraded components is available, ALTA can run old applications directly with upgraded components. In ALTA’s LTW mode, ALTA will not modify any part of the system statically. Therefore, it can work under all kinds of license agreements.
```java
import java.lang.reflect.*;

privileged aspect SampleAspect {

// inter-type declaration: define a hidden field.
public packageA.ClassA_REN
   packageA.ClassA.hiddenObj=null;

// inter-type declaration: define a method.
public static int packageA.ClassC.methodInClassC
   (java.lang.String var1) {
      return packageA.ClassC.methodInClassC_REN(var1);
   }

// pointcut and advice: handle all methods in ClassA.
Object around () : call(* packageA.ClassA.*(..)) {
   ... // skip
}

// pointcut and advice: handle conflict-APIs
int around(int var1) throws IOException:
   call(int packageA.ClassA.go(int) throws IOException)
   & args(var1) && !within(packageA.*) {
      ... // skip
   }

// pointcut and advice: handle all constructors of ClassA
packageA.ClassA around () :
   call(packageA.ClassA.new(..)) {
      ... // skip
}

// pointcut and advice: handle the 'set' actions
// of all fields in ClassA
void around(Object input, packageA.ClassA targ)
   : set(* packageA.ClassA.*) && args(input)
   && target(targ)
   && !set(* packageA.ClassA.hiddenObj) {
      ... // skip
}

// pointcut and advice: handle the 'get' actions
// of all fields in ClassA
Object around(packageA.ClassA targ)
   : get(* packageA.ClassA.*)
   && target(targ)
   && !get(* packageA.ClassA.hiddenObj) {
      ... // skip
}
}
```

Figure 2.13 Sample aspect generated by Generator.
Figure 2.14  The process of compound refactoring tests.

![Diagram]

Figure 2.15  Performance report. X-axis: number of method adapted in one class; y-axis: performance.
Figure 2.16  Performance report. X-axis: number of object created; y-axis: performance. 100% of the methods in each class were adapted.

Figure 2.17  Performance report. X-axis: the number of class adapted; y-axis: performance. 100% methods in each class were adapted.
Figure 2.18 Performance report. X-axis: adapted class count; y-axis: performance. 20% of the methods in each class were adapted.
CHAPTER 3.  TARP: A Testing and AI-Planning Based Refactoring Path
Reconstruction Framework for Full-Automatic Component Adaptation

3.1 Introduction

Software components upgrade frequently and some of the changes may lead to component incompatibility. Component incompatibility may cause serious problems including incorrect execution results, compilation errors and system crashes. Therefore, how to fix component incompatibility is an important research issue. In the past decade, many solutions to address this issue have been proposed, and most of them are semi-automatic [1, 2, 3, 4, 5, 6]. These solutions require manually coded upgrade information, such as delta files, upgrading annotation, or mapping rules, in order to automatically migrate applications to fit new components. However, developers may not be willing to manually develop such information for end users, given that the process is usually complicated, fallible and time-consuming.

To overcome this limitation, several full-automatic solutions have been proposed [7, 8, 9, 30]. Unlike semi-automatic ones, full-automatic solutions can work without human-coded change information. One of the assumptions underlying these solutions is that developers use Eclipse to refactor their components, thus the machine-recorded refactoring history can be available. With this valuable change information, these full-automatic solutions can either replay all changes to an application (i.e., to upgrade the application to fit the upgraded component) or to components (i.e., to generate adapter/wrapping layers which provide both old and new API) and solve the compatibility problems in a full-automatic fashion.

Although full-automatic solutions are impressive, it is not reasonable to assume that every end user can get refactoring history of upgraded components from Eclipse. First of all, developers may use tools such as VI or notepad++, which do not automatically record refactoring
history to refactor their components. Second, if developers do use Eclipse but do not follow the recommended steps (i.e., to use the refactoring wizards or hot keys) to refactor their components, Eclipse cannot record the history. Therefore, in order to fix compatibility problems in general cases, it is important to find a way to get refactoring history or change information directly from the components instead of relying on machine-recorded ones.

In the past decade, many static analysis methods have been proposed to get change information directly from the source code of upgraded components. Antoniol et al. [10] formalized information on APIs into linear algebra and vector compositions to infer possible refactorings. Demeyer et al. [11] traced multiple versions of components and composed change metrics to infer possible refactoring actions. Xing and Stroulia [12] applied reverse-engineering techniques to the source code of the old (i.e., before upgrade) component and the new (i.e., after upgrade) component to generate UML models of them. After that, they compared the generated models to identify the changes of components. Godfrey and Zou [13] analyzed method-calling flow in order to recognize method splitting and merging. Dig [14] scanned the component’s source code and checked the similarities of all parts which shed light on the changes being made. Kim et al. [16, 17, 18, 19] compared the similarities of all parts first, then converted the results into template-based logic rules in order to recognize complex refactorings activities.

Although these solutions are impressive, all of them share the same limitations:

1. **Unable to detect Temporal-Dependent Refactoring Steps (TDRS):** It is common for developers to repeatedly refactor the same part of code [20]. TDRS are refactoring steps applied to the same part of components in sequence, and each step shares at least one transient refactoring parameter with its successor. A refactoring step is different from a refactoring pattern because a refactoring step includes refactoring parameters but a refactoring pattern does not. For instance, “move method $C1.m1$ to $C2$” is a refactoring step but “move method” is a refactoring pattern. **Transient refactoring parameters are the refactoring parameters which do not exist in the old or new API.**

Figure 3.1 illustrates this problem. Suppose when upgrading a component, you move a
method \textit{m1} from class \textit{C1} to class \textit{C2}, then rename class \textit{C2} to \textit{C3} (see Figure 3.1). Since \textit{C2.m1} (the dashed bubble in Figure 3.1 (B)) does not exist in either the old API or the new API, it is a transient refactoring parameter. Thus, these two refactoring steps which share it are TDRS. Because static analysis algorithms can only gather information from the old and new API, they can never detect any refactoring steps related to transient refactoring parameters. Therefore, none of them can detect TDRS.

2. \textbf{Unable to work without source code}: All methods mentioned above require source code to do static analysis. However, compatibility problems may occur among third-party components (see Figure 3.2). If binary releases of impacted components are the only resources we can get (see the shadowed component \textit{X} in the middle of Figure 3.2 (A) and (B)), all existing solutions cannot work.

3. \textbf{Unable to verify generated results}: These algorithms only generate “inferred results” without validating. Therefore, it is risky to use these potentially invalid results as the input of any full-automatic compatibility solutions.
3.1.1 The Proposed Framework

In this study, we introduce TARP (Testing and AI-Planning Based Refactoring Path Reconstruction Framework). It is a novel solution for automatically reconstructing refactoring paths to overcome three limitations in the existing solutions. Unlike static-analysis based solutions, TARP embraces TDRS by adopting AI planning techniques [31]. We will briefly introduce the AI Planning technique in Section 3.2.1.

3.1.1.1 Innovation of TARP

Figure 3.3 shows an overview of TARP. When we input the old and new components into TARP, the problem modeling and solving module (PMSM) will model the APIs into an AI planning problem. Then TARP will send this model with all predefined AI planning actions (i.e., supported refactoring patterns) into an AI planner. The planner will generate a solution, which is a sequence of AI-planning actions with parameters, e.g., \{moveMethod(m1,C1,C2); renameClass(C2,C3)\}. In other words, it is a sequence of refactorings steps which changes the old API to the new API.

After getting a refactoring path, we need to verify it, because sometimes a planner will give us a wrong path. We need to make a clarification here that a path generated by a planner will always be “AI-planning correct”, which means that it really changes the given world from
Figure 3.3  Modules and conceptual data flows of TARP. There are three modules: Problem Modeling and Solving Module (PMSM), Adaptation-based Testing Module (ATM), and Result Analysis and Feedback Module (RAFM).

the initial state to arrive at the goal state. However, it does not refer to a “logically correct” solution. We used an example to illustrate this idea (see Figure 3.4). In Figure 3.4, the model (M) has more than one solutions. The solution shown in Figure 3.4 (A) contains 2 refactoring steps, and (B) contains 4 steps. Because these two solutions can lead to exactly the same goal state, they are both “AI-planning correct”. However, the meaning of these two solutions are very different in that for (A) X.add() is renamed to X.deduct(), and for (B) X.add() is changed to Y.sum(). In this case, only one solution could be correct. However, it is difficult to tell if the generated path is correct because we don’t have the correct refactoring path to compare with.

To verify if a generated result is “logically correct”, TARP’s Adaptation-based Testing
A model with multiple solutions: (M) is the model, and solution (A) and (B) are two possible solutions. While changing method X.add to Y.sum is more reasonable, they are both seen as “AI-Planning correct” solutions given that (A) and (B) have exactly the same initial state and goal state.

Module (ATM) will do the following:

1. **Generate test cases for the old component**: ATM will use a feedback-oriented test cases generator to create test cases with assertions for the old component.

2. **Run generated tests toward the new component**: ATM will use an on-the-fly adapter, such as ALTA* [30], to dynamically adapt the generated tests with the new component according to the refactoring path generated by the AI planner.

If the generated refactoring path is logically correct, then the adaptation will be correct too. Therefore, the test results will be positive. Otherwise, some tests must fail. For instance, if the planner returns the path shown in Figure 3.4 (B), all the tests which want to call add(5,2) will be bridged to sum(5,2) and return 7; therefore, it passes the assertion statement. If the planner returns the path shown in Figure 3.4 (A), add(5,2) will be adapted with deduct(5,2) and returns 3; therefore, the test result will be negative.
If the test report shows that the generated path is correct, TARP will output this path as the final product. If not, TARP will rerun the entire process to get another path. However, before doing this, we need to change something in the initial state as well as the goal state of our problem to avoid generating the same path, or the planner will definitely produce exactly the same result. To achieve this goal, the Result Analysis and Feedback Module (RAFM) will analyze the report to generate “altering notes”, named Path Tokens, to let the Problem Modeling and Solving Module know how to alter the initial state and the goal state when remodeling the problem. If, for some reasons (e.g., the component was upgraded by unsupported refactoring patterns) TARP cannot find a correct solution, it will output an empty path, meaning that TARP is unable to solve this problem.

3.1.1.2 Implementation of TARP

We have also created TARP*, a lightweight implementation of TARP. TARP* currently encoded 8 refactoring patterns, including “Rename Field”, “Move Field”, “Move Method”, “Rename Method”, “Pullup Method”, “Rename Class”, “Move Class” and “Rename Package”. TARP* chose PDDL 2.1 [22] to model AI planning problems and FF [32] as the AI planner. In ATM, TARP* selected Randoop [23] as the test case generator, and ALTA* as the on-the-fly binary adapting tool.

3.1.1.3 Evaluation of TARP*

To evaluate TARP*, we conducted the Open Source Component Refactoring Path Reconstruction Test. In this test, we selected 3 open source components: Apache POI, Apache Commons Lang, and Google Collection as our subjects. Then we carried out five experiments. In each experiment, we picked up one of the subjects and applied different refactoring steps. In two of these experiments, we even applied TDRS, which could not be detected by any solutions in the existing literature. Next, we ran TARP*, as well as Refactoring Crawler [14] and LSdiff [17], two state-of-the-art refactoring analysis tool, to detect refactoring information of the upgraded subject. Then we compared the outputs of these three solutions. Because the real refactoring history was available, we had no problems with verifying those
outputs. The experimental results showed that TARP* can work well in real world projects. More importantly, it is the only solution which can successfully detect TDRS.

In addition, to evaluate whether ALTA* can use the generated refactoring path to solve compatibility problems, we conducted the Open Source Component Official Test Cases Adaptation Test. In this test, we used the official test cases of these three open source components (before upgraded) as applications, and let ALTA* adapt these applications with the upgraded components on-the-fly. The experimental results showed that ALTA* successfully fixed all compatibility problems.

To sum up, this work makes the following contributions:

1. **Innovation**: We proposed TARP, a novel and comprehensive solution, using AI planning and on-the-fly Adaptation-based testing techniques to automatically reconstruct refactoring paths for binary components.

2. **Implementation**: We implemented TARP*, a light-weight implementation of TARP.

3. **Evaluation**: We evaluated TARP* by conducting the Open Source Component Refactoring Path Reconstruction Test and the Open Source Component Official Test Cases Adaptation Test. The experimental results showed that TARP is a workable solution for automatically reconstructing refactoring paths. More importantly, it showed that full-automatic component adapting is possible.

The rest of the paper is structured as follows. In section 3.2, we briefly introduce the background of AI planning and ALTA, followed by detailed discussions of TARP in Section 3.3. In Section 3.4, we introduce TARP*. In Section 3.5, we presented the evaluation results of TARP*. Concluding remarks and future works were discussed in Section 3.6.

### 3.2 Background

#### 3.2.1 AI Planning

AI Planning [31], or Automated planning and scheduling, is a branch of artificial intelligence. AI planning has been widely applied to different software engineering fields. For example,
Memon et al. [33, 34] used AI-Planning to generate test cases for GUI applications. Moreover, there are many studies applying AI-planning for web service compositions [35, 36, 37, 38, 39]. By modeling a planning problem into an initial state and a goal state of a specific world along with a set of available actions, we can use a standard AI planner to find a sequence of actions which will change the specific world from the initial state to arrive at the goal state. Figure 3.5 shows a common example of AI planning problems. In this problem, there are five objects: two cargoes named Ax and Bx, one rocket named Rx, and two places named Lx (ground) and Px (space). We can model this concept by the following types, objects and predicates:

```
1 (:types
2   Cargo Rocket Place – Object
3 )

4 (:predicates
5   (At ?o – Object ?p – Place)
6   (Hasfuel ?r – Rocket)
7   (In ?c – Cargo ?r – Rocket)
8 )

9 (:objects
10  Ax Bx – Cargo
```
The “At” predicate in line 6 above tells if an object \( ?o \) is at the place \( ?p \). The “Hasfuel” predicate in line 7 returns true if the rocket \( ?r \) has fuel, and the “In” predicate in line 8 tells if a cargo \( ?c \) inside a rocket \( ?r \).

In the beginning, the rocket and these two cargoes are on the the ground, and the rocket has enough fuel to fly to the space. So we can describe this initial state by the following model:

```
(: init
 (At Ax Lx)
 (At Bx Lx)
 (At Rx Lx)
 (HasFuel Rx)
)
```

After a sequence of actions, we want these two cargoes to be placed in the space. We are not concerned about the rocket in the goal state, so we don’t need to model it. Here is the goal model:

```
(: goal
 (and
  (At Ax Px)
  (At Bx Px)
 )
)
```

Now we need to provide possible actions for a planner to start planning. Suppose that we define three possible actions of the rocket: load, unload and move. “load” can move a cargo into a rocket in a certain place. “unload”, similar to load, can move a cargo out of a rocket in one place. And “move” is to launch a rocket from one place to another. The followings are the models:

```
(: action load
 : parameters
```
(:action unload
 :parameters (?r - Rocket ?p - Place ?c - Cargo)
 :precondition (and
 (At ?r ?p)
 (In ?c ?r)
 )
 :effect (and
 (At ?c ?p)
 (not (In ?c ?r))
 )
)

(:action move
 :parameters (?r - Rocket
   ?from - Place ?to - Place)
 :precondition (and
 (not (= ?from ?to))
 (At ?r ?from)
 (Hasfuel ?r)
 )
 :effect (and
 (At ?r ?to)
 (not (At ?r ?from))
 )
)
In each action, we need to define parameters, preconditions and the effects. Let us use the “move” actions as an example. Line 2 to line 4 say that the move action will consider 3 parameters: a rocket ?r and two places ?from and ?to. Line 5 to line 9 are the preconditions. Line 6 says that the two input places should not be the same. Line 7 and 8 say that the rocket ?r should be at the place ?from before performing this action and the rocket should have fuel.

Line 10 to 14 describe the post conditions. After performing this action, the rocket should be at the place ?to (line 11), and the rocket should NOT be in place ?from (line 12). Finally, it should not have fuel anymore (line 13).

Once we have the models, we can send them to a planner to get the result. The following is a possible output from a planner, which is a sequence of actions with real parameters: {load the cargoes to the rocket, launch the rocket and unload cargoes in the space}.

This solution does change the world’s status from the initial state to arrive at the goal state. Since refactoring steps are also a sequence of actions which change a component’s API from the initial state (i.e., the old API) to the goal state (i.e., the new API), it seems possible to use AI planning to reconstruct missing refactoring paths.

There are mainly three famous languages designed for modeling AI Planning problems, including STRIPS (Stanford Research Institute Problem Solver) [40], ADL (Action description language) [41] and PDDL (Planning Domain Definition Language) [22]. Current, PDDL is the most popular modeling language in AI planning area which includes all features of STRIPS and ADL. Therefore TARP* chooses PDDL (version 2.1) as its modeling language. All models shown in this section were also written in PDDL.
3.2.2 ALTA and ALTA*

ALTA [30] is a framework which can adapt incompatible components on-the-fly. ALTA* is an implementation of ALTA. ALTA relies on automatically recorded refactoring history from Eclipse IDE. Because refactoring history contains enough information to do software adaptation, the entire adaptation process is full-automatic. ALTA runs with binary files and it does not need any source code of either application or software components.

The main idea of ALTA is to use Aspect Orient Programming (AOP) technique to do adaptation during the execution time. For the example shown in Figure 3.1, if application wants to call a method C1.m1 but this method was moved to C2, and C2 was renamed to C3 during upgrade progress, then this program call will fail and throw exceptions.

To fix this problem by ALTA, we need to export the refactoring history from the Eclipse IDE. ALTA assumes that the history information is available — if this not true, then ALTA cannot help in this case. Refactoring history is recorded automatically by default in Eclipse. In the previous example, the history will show that there were two refactoring steps, first, move method C1.m1 to C2. Second, rename class C2 to C3.

After this history information has been sent to ALTA, ALTA will start creating a mapping table from the original API to the new API. The reason why ALTA wants to do that is that ALTA wants to achieve a single-hop bridge so it wants to directly bridge the old API to the new API. The followings are the generated mapping rules:

1. C1.m1 $\rightarrow$ C3.m1
2. C2.m3 $\rightarrow$ C3.m3

Note, although we did not touch m3 directly, it is also on the list because we renamed its container C2 to C3.

Next, ALTA will generate adaptation logic written in AspectJ according to this mapping table, and compile it as a Jar file. With this ALTA Jar file, the user can run their old application (which needs C1.m1) with the new component (which only has C3.m1) correctly because ALTA will dynamically redirect all method calls toward C1.m1 to C3.m1 and redirect the results back to the caller. Figure 3.6 shows this idea.
Figure 3.6 The concept of ALTA’s runtime adaptation. ALTA will redirect all Car.go() calls to Car.move(), and redirect the return value back to the caller on the fly.

The biggest limitation of ALTA is that it relies on refactoring history. However, because TARP can use AI Planning technique to reconstruct a refactoring path for ALTA, it is not a problem anymore. Besides, since ALTA is the only solution to date which can perform on-the-fly adaptation without requiring any source code, it is the best candidate for TARP to do the adaptation-based testing in module ATM (see Figure 3.3).

3.3 Method

3.3.1 Preliminary Modeling Strategy

To verify if we can really use AI planning technique to reconstruct refactoring histories of a upgraded component, we defined a preliminary modeling strategy as follows:

1. **Types**: We defined “class” and “method” as “Object” types. We omitted “field” and “package” in this preliminary design. Besides, because PDDL has used the term “types” already, we did not use this term to represent class or interface.

2. **Predicates**: We defined only one predicates: “(Contains ?parent - Object ?child - Object)” to show the “containing” relation between a parent object and a child object.
3. **Actions:** We defined only one pattern: “moveMethod” with 3 parameters: the method name, parent class and the target class.

The following is the PDDL code of these settings:

```plaintext
(:types
  APIObject  -  object
  Class  -  APIObject
  Method  -  APIObject
)

(:predicates
  (Contains
    ?c  -  Class
    ?m  -  Method
  )
)

(:action MoveMethod
  :parameters (?m1 - Method
    ?cFrom ?cTo - Class )
  :precondition (and
    (Contains ?cFrom ?m1)
  )
  :effect (and
    (Contains ?cTo ?m1)
    (not (Contains ?cFrom ?m1))
  )
)
```

In line 2, we define a type called APIObject which is a general type of package, class, interface, method, and field. In line 17, we defined the precondition for this “moveMethod” action. It says that the refactoring parameter ?cFrom must contains ?m1 in order to perform this actions. In line 20, we described that after performing this action, class ?cTo must contains ?m1, and ?cFrom should not contain ?m1 anymore. Besides, in line 22, the negative condition,
is very important. If we remove line 22, this “MoveMethod” action will simply build a new “Contains” relation between ?cTo and ?m1. Meaning that the relation between ?cFrom and ?m1 will not be removed; therefore both ?cFrom and ?cTo will contain ?m1 after performing this action. So adding negative conditions is critical when defining actions.

Moreover, for modeling a real problem, we used the following strategy to define objects, the initial state and the goal state:

1. **Objects**: All objects shown in the old API and the new API. In this design, for each object, we used its name (e.g., method name, field name, class name or package name) as its object identity. For example, for a method C1.m1, we used m1 rather than C1.m1 as its identity. We did not use a full name (i.e., package name + class name + method name) because we did not want to describe the “Contains” relation by anything else other than the “Contains” predicates.

2. **Initial state**: The relations and facts in the old API. E.g., (Contains C1 m1) — which represents that class C1 contains method m1 before refactoring.

3. **Goal state**: A SINGLE logic statement composed of relations and facts in the new API. This statement needs to be true. E.g., (and (Contains C2 m1) (Contains C2 m3)) represents that class C2 will contain method m1 and m3 after applying all refactorings steps.

Now we can start modeling real problems. The following is the PDDL code for modeling the APIs shown in Figure 3.7 (A) and (B).

```
( :objects
  C1 C2 - Class
  m1 m2 m3 - Method
)

( :init
  ( Contains C1 m1 )
  ( Contains C1 m2 )
  ( Contains C2 m3 )
)```

Figure 3.7 An example of the preliminary modeling strategy. (A) and (B) show a simplified example from Figure 3.1. Note that Figure 3.1 (C) was removed from this example so Figure 3.1 (B) became the new API. (C) and (D) are the model of this example created by the preliminary modeling strategy, where (C) is the initial state and (D) is the goal state. Note: the “C” icons represent the “Contains” relation.

```
(: goal
  (and
    (Contains C1 m2 )
    (Contains C2 m3 )
    (Contains C2 m1 )
  )
)
```

Figure 3.7 (C) and (D) illustrates this model.

When we input this model into an AI planner, we will get the following refactoring steps:

```
{ (MoveMethod m1, C1, C2) }
```

This is exactly the missing refactoring history that we want to get. By showing this simple example, we demonstrated that it is possible to reconstruct a refactoring path automatically via the AI Planning technique.
3.3.2 Handling Conflicting Names

Although the previous example shows that our preliminary modeling strategy works, this strategy has a significant drawback: it does not allow same-named objects. Let us use an example to describe this problem. Suppose that we want to model the API shown in Figure 3.8. In this diagram, both of classes $C_1$ and $C_2$ have a method $m_2$. By applying our preliminary modeling strategy (Section 3.3.1), we will define the following objects and predicates:

```
1 ( :objects
2   C1 C2  –  Class
3   m1 m2 m2  –  Method
4 )
5 ( :init
6   (Contains C1 m1)
7   (Contains C1 m2)
8   (Contains C2 m2)
9 )
```

Although this model looks correct, it is actually not. The problem here is that the redundant declarations of the two $m_2$ methods in line 3 will only create ONE object in AI planner.
Figure 3.9 Example of modeling a rename action. (A) is the precondition and (B) is the post condition. Because all object’s identity cannot be changed in an AI planning model, we model rename actions by using “HasName” predicates. In (C), the method _m001_ is related to a name object called _m1_, and in (D), this method object is related to another name object called _m1_ren_.

Therefore, although we thought that we built a model as 3.8 (B), we actually created a model 3.8 (C). In 3.8 (C), C1 and C2 share one APIObject _m2_, and this is not what we want.

To handle this problem, we modified our preliminary modeling strategy as follows. First of all, we define a new type called “name”. Second, for each API object, we generate a unique id as its identity. Third, we declare the API object’s name as a name-typed object, if it has not been declared before. Finally, we define a new predicate “(HasName ?obj - APIObject ?name - name)” to relate an API object with its name. In this manner, multiple API objects can share one name. Figure 3.10 shows this idea.

In this modeling strategy, we can model all kinds of rename patterns (such as “rename method”) into AI planning actions by manipulating the “HasName” predicates. Figure 3.9 shows this idea. In Figure 3.9 (A) and (B) we can see that a method _m1_ has been renamed to _m1_ren_. Because we cannot really “rename” a name object in an AI planning problem, what we need to do is to declare both name objects in the beginning, and use “HasName” predicate to relate the method to the old name object in the init state (see 3.9 (C)) and to the new name object (see 3.9 (D)). In addition, to support name swapping, TARP will predefined a dummy
Figure 3.10  The solution of the same-name problem. (A) shows the API and (B) shows the model. Each object has a unique serial number, and each icon “N” shows a "HasName" relation, and diamond-shaped objects are name-typed objects. Note that m0002 and m0003 share m2 because they have the same method name.

name object called DummyName in each generated model.

3.3.3 Handling Inheritance

Next challenge for us is how to model the inheritance relation in PDDL. We need inheritance information to support some refactoring patterns such as “Pull Up Method”. But the problem is: PDDL cannot support hierarchical relations. For example, suppose that a predicate (Parent ?parent ?child) means ?parent is the parent of ?child. Now, if we define “(Parent A B)” and “(Parent B C)”, the planner will NOT know A is an ancestor of C because “(Parent A C)” is still false. Of course we can define “(Parent ?grandParent ?parent ?child)” in this case, but it is not possible and not reasonable to define an exhaustive list of this kind of predicates.

To solve this problem, TARP adopted two rules:

1. Flatten inheritance tree: TARP will flatten a inheritance tree before modeling it. Flatten means that TARP will replace all indirect inheritances by direct inheritances. Then all direct inheritance will be modeled by the following predicate: (Inherit ?classChild ?classParent).
2. Ignore “out of scope” inheritance: If a parent class is not present in this model, then this inheritance relation will be neglected.

Figure 3.11 shows an example. In Figure 3.11, class Vehicle is an ancestor of class Truck, so TARP will create a predicate: (Inherit C001 C003). Besides, although class Vehicle extends class Object, but because the object class is not in either the old API or the new API, TARP will NOT mode that relation.

3.3.4 Handling Uncertain Identities in a Goal State

Although our new modeling strategy can handle conflicting names when modeling the initial state (see Figure 3.10) and actions (see Figure 3.9), it is challenging to use this strategy to model a goal state because the real identity behind an API object’s name is actually uncertain.

For example, suppose that we want to use the new modeling strategy to model the APIs shown in Figure 3.12 (A) and (B). (Note that Figure 3.12 reuses the model shown in Figure 3.4). In the beginning, we need to declare all name objects which appear in either the old API or the new API. So there will be 6 name-typed objects: X, Y, add, sub, deduct and sum. Next, we model the initial state by assigning each API-object in the old API a unique id, then use “HasName” and “Contains” predicates to describe their relationships. Figure 3.12 (C) shows
the result. In Figure 3.12 (C), the API object \textit{C001} has a name \textit{X} and it contains another API object \textit{m001} whose name is \textit{add}, which tells us that “there is a class named \textit{X} who contains a method called \textit{add}”. Similarly, this model also says that there is another class called \textit{Y} who contains a method named \textit{sub}. This part is really straightforward.

However, when we start to model the goal state, we will soon realize that there is a big problem: there are unknown identities. By observing Figure 3.12 (B), we know there are 4 API-objects, which need to be related to name objects \textit{deduct}, \textit{sum}, \textit{X} and \textit{Y}. Moreover, we know that the API object related to name object \textit{X} (denoted as a variable \textit{var3}) will contain another API object which is related to the name object \textit{deduct} (\textit{var1}). Similarly, we know that \textit{var3} has a name \textit{Y}, \textit{var3} contains \textit{var2}, and \textit{var3} has a name \textit{sum}. But the problem is how to assign real identities to those variables.

We are in a dilemma. On one hand, we should not reuse the unique identities that we assigned in the initial state because the identities behind the names might have already been changed. A critical fact is that \textbf{when we reuse an ID, we are actually binding all objects}
who share this ID. In other words, we make the planner biased. For example, if we assign a value \textit{m001} to the variable \textit{var1} in Figure 3.12, we are telling the planner “the method named \textit{add} in the initial state is actually the method named \textit{deduct} in the goal state. Under this incorrect guidance, the planner can only produce a wrong path which contains “rename X.add to deduct”. Therefore, it is risky to reuse any identities.

On the other hand, we should not assign new-generated unique identities to the API objects in the goal state either. The reason is very similar: by assigning different identities to two API objects, we are actually telling a planner that “these two objects are not the same”. For example, if we assign \textit{m777} to \textit{var1} in Figure 3.12, the planner will not be able to produce any result because the goal state is unreachable.

To solve this dilemma, we decided \textbf{not to assign any identity} to the API objects in a goal state. As we discussed above, this is the only way that we will not bias the planner. However, if we don’t assign identities to API objects, we cannot make predicates such as “contains” or “HasName” because these predicates need identities as input parameters. Figure 3.13 illustrates this idea.

Therefore, we introduced a new concept called “\textbf{signature path}”. In a sentence, “signature path” combines the concept of “HasName” and “contains” predicates while bypassing object identities. In signature paths, we simply describe a sequence of names in a structural order: a parent’s name, a child’ name, a grandchild’ name, and so on. No matter how long a path is, a parent’s name is always followed by one of its children’s name. Figure 3.14 (A) and (B) show this idea. In Figure 3.14 (A), we cannot define any “HasName” or “contains” predicates
Figure 3.14 The concept of a signature path. (A) shows a goal state without any identity of API-objects; therefore we cannot create “HasName” or “Contains” predicates. (B) shows a signature path that describes a path which includes a class name and a method name.

because we don’t know any object identities in the goal state. In Figure 3.14 (B), by defining a signature path: \{X, deduct\}, we successfully described that there is a class named X which contains a method named deduct, where both of the API objects’ names are unknown.

For realizing this concept, we defined a set of predicates called “SignaturePathTillXYZ”. “XYZ” represents the end point of this path. For example, the predicate “(SignaturePathTillMethod ?className - Name ?methodName - Name)” describes that there is class whose name is ?className, and it contains a method whose name is ?methodName. Similarly, the “(SignaturePathTillClass ?className - Name)” predicate describes that there is class whose name is ?className. Note that although a longer path may contain more information than a shorter paths (e.g., (SignaturePathTillMethod X add) v.s. (SignaturePathTillClass X)), we cannot neglect the shorter one because we still need to keep track of a container while it contains nothing. For example, if there is no “SignaturePathTillPackage”, then we have no way to describe an empty package in a goal state. This part is especially important when we do complexity reduction. We will discuss this issue in Section 3.3.8.

In this way, we can redefine predicates in the goal state of the problem shown in Figure 3.12. Figure 3.15 (B) and (D) show the results. Besides, we also need to add “signaturePath” predicates in the initial state, or the goal state will never be reachable. Figure 3.15 (A) and (C) show an example.

In addition, we also need to modify related actions so that the signature path will be modified after those actions. For example, a new “renameMethod” action can be defined as
Figure 3.15  Goal state without API-object identities but with signature paths: all the important concepts have been correctly captured.

follows:

```prolog
(:action renameClass
 :parameters (
    ?class - Class
    ?cName - Name
    ?method - Method
    ?oldMName - Name
    ?newMName - Name
  )
 :precondition (and
   ;; object structure
   (Contains ?class ?method)
   ;; name relations
   (HasName ?class ?cName)
```
Figure 3.16 A sample “rename method” actions that supports name path. (A) is the precondition and (B) is the post condition.

```
(HasName ?method ?oldMName)
(not (HasName ?method ?newMName))

;; signature paths
(SignaturePathTillMethod
  ?cName ?oldMName)
(not (SignaturePathTillMethod
  ?cName ?newMName))
)

:effect (and
  ;; object structures

  ;; name relations
  (not (HasName ?method ?oldMName))
  (HasName ?method ?newMName)
)```
Line 19 to 23 is the preconditions regarding signature paths. Line 20 and 21 say that there should be a signature path from \(?cName\) to \(?oldMName\) before performing this action, which means there is a class object named \(?cName\) which contains a method named \(?oldName\). Moreover, in line 22 and 23, we claim that there should be no signature path from \(?cName\) to \(?newMName\). Line 26 to 37 is the post condition; in line 34 to 37, we state that the old signature path does not exist anymore, and the new signature path from \(?cName\) to \(?mNewName\) appeared. In this way, signature paths in the initial state can be manipulated by different actions so that the goal state could be reachable. Figure 3.16 shows the concept of the “renameMethod” action that we discussed above. Note: because this “renameMethod” action will not modify any “signatureTillClass” predicates, there is no “signatureTillClass” shown in Figure 3.16.

### 3.3.5 Supporting New API

TARP can support method creation.

The main idea of method creation is that TARP will create and reserve a pseudo method “\(mNew\)” for method creation in each model. When TARP wants to create a new method, it will execute the action “createMethod(\(?class\), \(?className\), \(?method\), \(?methodName\))” by passing an \(mNew\) object. In this action, the predicate (Contains \(?class\), \(?method\)), (HasName \(?method\) \(?methodName\)) and (SignaturePathTillMethod \(?className\), \(?methodName\)) will be built, and therefore fulfills our needs.

Figure 3.17 shows an example. Figure 3.17 (B) is the initial state and Figure 3.17 (D) is the goal state. To reach the goal state, the “createMethod” action will be called, then all relations
showing in Figure 3.17 (F) will be built. If you compare Figure 3.17 (D) and Figure 3.17 (F), you can see that all desired predicates are true.

There are two important features of $mNew$. First, $mNew$ can be “contained” in many classes, and it can contain multiple names. Second, $mNew$ cannot be involved in any refactoring pattern except for the “createMethod” action.

Similarly, TARP also supports creating packages, classes and fields.

### 3.3.6 Modeling API Deletion

Unlike method creation, TARP cannot support method deletion.

For an upgraded component, if there are some methods removed from the new API, because the old tests which rely on the removed methods will always fail, TARP will not be able to output a verified refactoring path. Therefore, TARP cannot support API deletion. More discussion about verifying the correctness of a generated path can be found in Section 3.3.9.
Even so, TARP can still model API deletion and ask the planner to generate a refactoring path. The main idea of modeling method deletion is that when TARP wants to delete a method, it will execute the action “deleteMethod(\(\text{?class}, \text{?className}, \text{?method}, \text{?methodName}\))”. In this action, the predicate (\(\text{Contains ?class, ?method}\)), (\(\text{HasName ?method ?methodName}\)) and (\(\text{SignaturePathTillMethod ?className, ?methodName}\)) will be set to false, therefore fulfills our needs.

Figure 3.18 shows an example. Figure 3.18 (B) is the initial state and Figure 3.18 (D) is the goal state. To reach the goal state, the “deleteMethod” action will be called, then all relations show in Figure 3.18 (F) will be built. If you compare Figure 3.18 (D) and Figure 3.18 (F), you can see that all desired predicates are true. Similarly, TARP can also model the deletion of packages, classes and fields.
3.3.7 Supporting Variadic Refactoring Patterns

Another challenge for us is to model variadic refactoring patterns into AI planning actions. Variadic refactoring patterns can modify arbitrary number of API objects in one refactoring step. For example, a single pull-up method refactoring step can pull up a method from \( N \) child classes to their parent class (see Figure 3.19), where \( N \) is a positive integer. However, all AI planning modeling languages, including PDDL, do not support variadic actions, which means that an action needs to have a fixed number of parameters. For instance, to support a pull-up method action which pulls up a method from two child classes, we need to define an action which expects two child classes as its parameters. However, this action cannot pull up a method from three child classes because the number of parameters is unmatched. Furthermore, it is impossible to run a pull-up method action for multiple times to gradually pull-up child’s method to its parent because a parent class cannot own multiple copies of the (pulled-up) method.

A naïve solution for this problem is to define a set of similar actions with different number of parameters. For instance, we can prepare the following set of “pullUpMethod” actions to support pull-up methods from 1 to 5 child classes (note that we neglect some parameters regarding signature paths for saving spaces in all examples in this section. The main idea of all examples will remain the same after this simplification):

```plaintext
\begin{verbatim}
1 action pullupMethodFrom1Child:
2   (?c1m1Name ?c1m1Obj ?c1Name ?c1Obj
3   ?cParentName ?cParentObj)
\end{verbatim}
```
Line 1 to 3 define an action which can pull up a method from one child class, where line 5 and 8 define another action which can support 2 child classes. In Line 6, \(?c1m1Name\) is the method name of the method that we want to pull up to the parent class, where \(?c1m1Obj\) is the real API object which has that name. \(?c1Obj\) is the container of \(?c1m1Obj\), and \(?c1Name\) is its name. In line 7, we define \(?c2Obj\), a sibling of \(?c1Obj\), whose name is \(?c2Name\). Note that although we define \(?c2m2Obj\), we do not define \(?c2m2Name\) because all the methods that will be pulled-up to the parent should share the same name in this refactoring pattern (see Figure 3.19). Finally, in Line 8, we define the parent’s object and name.

Although this solution works in some cases, since it is impossible to provide an exhaustive list of those actions, we can never fully support this kind of refactoring patterns.

Therefore, to fully support variadic refactoring patterns, we proposed two new mechanisms called “Refactoring Transaction” and “Parameter Reducing”. The “Parameter Reducing” mechanism provides a way to gradually reduce the number of parameters of a
refactoring pattern until the minimum number of parameter of this pattern is reached. With “Parameter Reducing”, we can support a variadic refactoring pattern by defining an action with that minimum number of parameters. For example, the minimum number of parameters is 6 in the pull-up method pattern (see line 2 to 3 in the list above), we can fully support the pull-up method pattern by defining one 6-parameter action. Besides, “Refactoring Transaction” creates a pseudo atomic transaction for the parameter-reducing process so that it will not be interrupted by any other actions.

A refactoring transaction is composed of at least three refactoring actions, where the first one is a “transaction-start action”, the last one is a “transaction-end action”, and the actions in between are “in-transaction actions”.

A “transaction-start action” needs to set a semaphore (i.e., a lock) to ON to prevent a planner from executing irrelevant actions. Besides, it also needs to register this transaction by creating a predicate which contains the name of this transaction as well as some of its parameters. This step can prevent a planner from executing “in-transaction action” with different parameters. With “refactoring transaction”, we can execute multiple actions as one atomic action with arbitrary number of parameters.

A “transaction-end action” need to set the semaphore to OFF and deregister this transaction. Actions which belong to this transaction can only be executed when the semaphore is ON and the transaction is registered. In contrast, all irrelevant actions can only be executed when all semaphores are off.

With “Refactoring Transaction” and “Parameter Reducing”, we can encode a variadic refactoring pattern into a sequence of actions with a fixed number of parameters. For example, we can define the “pull up method” as the following 3 actions:

1. pullupMethod_start (?c1m1Name ?c1m1Obj ?c1Name ?c1Obj ?cParentName ?cParentObj): The “transaction-start action”. Figure 3.20 (A) shows this idea. It will do the followings:

(a) Make sure the semaphore is set to off: Check if “NotPullingMethod” is true. If not, do not continue.
(b) **Set the semaphore to on**: Set “NotPullingMethod” to false.

(c) **Register this transaction**: Register all of its parameter with predicate “CurrentPulling”.

2. **pullUpMethods_mergingSiblings** (?c1m1Name ?c1m1Obj ?c1Name ?c1Obj ?c2m2Obj ?c2Name ?c2Obj ?cParentName ?cParentObj): The “in-transaction action”. This is the place to do Parameter Reducing. Figure 3.20 (B) and (C) show this idea. In this step, it will do the followings:

   (a) **Make sure the semaphore is ON and this transaction is registered**: Check if the “NotPullingMethod” is false and if the predicate (CurrentPulling ?c1m1Name ?c1m1Obj ?c1Name ?c1Obj ?cParentName ?cParentObj) is true. If not, do not continue.

   (b) **Make sure the two input methods can be merged**: Check if ?c1m1Obj and ?c2m2Obj share ?c1m1Name, and whether both of ?c1Obj and ?c2Obj inherit ?cParentObj. If not, do not continue.

   (c) **Merge these two input methods**: Copy all necessary properties (e.g., path tokens, see Section 3.3.10) from ?c2m2Obj to ?c1m1Obj and then remove ?c2m2Obj from ?c2Obj.

3. **pullupMethods_end** (?m1Name ?m1Obj ?c1Name ?c1Obj ?cParentName ?cParentObj): The transaction-end action. Figure 3.20 (D) show this idea. This is the only step which pulls the method up. It will do the followings:

   (a) **Make sure the semaphore is ON and this transaction is registered**: Same as step 2 (a).

   (b) **Pull up the method**: Move ?c1m1Obj from ?c1Obj to ?cParentObj.

   (c) **Set semaphore to OFF and deregister the transaction**: Set “NotPulling-Method” to true and the predicate (CurrentPulling ?c1m1Name ?c1m1Obj ?c1Name ?c1Obj ?cParentName ?cParentObj) to false.

If the real refactoring history contains a pull-up method refactoring step shown in Figure 3.19, a planner will produce the following path:
Figure 3.20  Model the pull-up method pattern by refactoring transaction and parameter reducing.

Where c001 is the object identity of the class named “Salesman”, c002 is “Engineer” and c003 is “Employee”. m001 is the object identity of the Salesman.getName(), and m002 is
the identity of Engineer.getName(). As we mentioned above, these 3 steps simulate a single pull-up method refactoring step. For a pull-up method pattern which includes 3 child classes (see Figure 3.20), the planner will generate 4 steps: 1 pullMethod_start, 2 pullUpMethods_mergingSiblings, and 1 pullMethod_end.

Therefore, with the parameter reducing and refactoring transaction mechanism, TARP can support variadic refactoring patterns.

### 3.3.8 Handling Huge Number of Objects

There might be a huge number of packages, classes, methods and fields in an API. Do we need to encode everything into an AI planning model? Not really. In fact, we only need to encode the changed parts (i.e., the parts impacted by refactoring actions). Thus, TARP uses an algorithm named “Simple Diff” to remove all unchanged packages, classes, methods or fields.

The pseudo code of this algorithm is as follows:

```plaintext
1 For each package P in API_old{
2   If (exists package P' in API_new and P.name == P'.name ){
3       If(P.content == P'.content){
4           delete P and P';
5       }
6   }
7   Else{
8       For each class C in P{
9           If (exists Class C' in P' and C.name == C'.name ){
10              If(C.content == C'.content){
11                  delete C and C';
12              }
13           }
14           Else{
15              For each method M in C{
16                  If (exists method M' in C' and M.signature == M'.signature ){
17                      delete M and M';
18                  }
19              }
20           For each field F in C{
21               If (exists method F' in C' and F.signature == F'.signature ){
22                  delete F and F';
23               }
24           }
25       }
26   }
27}
```
This reduction algorithm can effectively remove unchanged parts before we start creating an AI planning model. In this manner, we can save a lot of AI-planning computation time by reducing the sizes of input models [32].

### 3.3.9 Verifying the Correctness of a Generated Path

Once we have an AI planning model, we can ask an AI planner to generate a plan for us. If the goal state is unreachable from the initial state with given actions, the planner will also tell us that there is no solution for this problem. Otherwise, we will get a plan. In our model, a plan is actually a refactoring path (or history). However, as we discussed in Section 3.1 (see Figure 3.4), there might be some incorrect paths from the old API to the new API. Therefore, we need to verify if the path is “logically correct”. However, we only have limited information: we don’t have the actual refactoring path to compare with, and we don’t have the source code of any component (TARP only receives binary jar files from the components; see Figure 3.3) either.

To achieve this goal, we invented a mechanism called “adaptation-based testing”. The main idea of adaptation-based testing is the following: First of all, TARP will generate a set of test cases named testsForOldAPI with assertions for the OLD API. This task can be done by using a feedback-directed random test generation tool such as Randoop [23] or GenRed [42]. Please note that when we execute testsForOldAPI with the OLD component, all tests will pass. Second, TARP will use ALTA*, an on-the-fly adaptation tool which relies on a given refactoring path, to adapt testsForOldAPI with the NEW component according to the generated refactoring path. If the path is logically correct, the adaptation should take effect and all tests will pass. Therefore, by running an adaptation-based testing, the rest results directly indicate the correctness of a generated refactoring path: if all the tests passed, we know the refactoring path is correct; otherwise, the path is incorrect.

Figure 3.21 shows an example of adaptation-based testing. In Figure 3.21, the input refac-
Figure 3.21 Example of an adaptation-base testing with a correct input refactoring path. The path says the “add” method was renamed to the “sum” method, and the “sub” method was renamed to the “deduct” method; therefore there will be no errors or fails in the test report.

3.3.10 Retrieving Another Solution

In the previous section, we introduced how we verify a generated path. If the path is correct, our job is done. However, if the path is not correct, we need to ask the planner the give us a different solution. However, it is not easy. First of all, if the goal state is reachable, an AI Planner will only generate one result rather than a set of possible results, so there is no any alternative path for us to verify. Moreover, for a given model, a planner will always generate
Figure 3.22 Example of an adaptation-base testing. The input refactoring is incorrect (it says the “add” method was renamed to the “deduct” method); therefore there must be some errors or failures in the test report.

exactly the same result no matter how many times we run it. Therefore, how to ask for another solution is a big problem.

In fact, for a given AI-planning model, the answer from a planner is already fixed. It will be either “the problem is proven unsolvable” or a concrete path from the initial state to the goal state. Hence, to ask for another solution, we need to provide a slightly different model. Therefore, we introduce a mechanism called “path token” to address this issue of an incorrect mapping ($\text{Src} \rightarrow \text{Dest}$) is generated and we don’t want a planner to generate any path which results in this mapping again. To achieve this goal, before rerunning the planner, we just need to put a special token in $\text{Src}$’s hand. Then we claim that $\text{Dest}$ will not hold that token in the goal state. In this case, for reaching the goal state, a planner has no choice but to generate another solution which maps $\text{Src}$ to anywhere but $\text{Dest}$. We named this kind of tokens “path tokens” and this kind of claims “negative path token assertions”.

Figure 3.23 shows an example. In Figure 3.23 (A), a planner generated a refactoring path which maps $M.$add to $MU.$deduct. However, the path is wrong. In fact, $M.$add should be mapped to $MU.$sum, and $M.$sub should be mapped to $MU.$deduct. Therefore, in (B), to prevent a planner from generating any path which leads to these two maps, we added a path token
Figure 3.23  Example of retrieving a better solution by adding path tokens. Dashed (red) arrows are incorrect mappings and others (blue) are correct mappings.

named PT1 and related it to M.add in the initial state, and made a negative path token assertions to ensure MU.deduct will not be related to PT1. In (C), because of the negative path token assertion, the planner could not generate a path which maps M.add to Mu.deduct again, so it generated a path which maps M.add to Mu.sum.

In TARP, when we want to relate a path token with an API object in the initial state, we will use two predicates: a “contains” predicate, which connects the token and the API object’s identity, and a “(signatureTillPathToken ?className ?methodName ?pathToken)” predicate, which builds a signature path until that path token. Regarding the goal state, because we do not know the real identity of any API object, we just need to make the following negative path token assertions: “(not (SignaturePathTillPathToken ?className ?methodName ?pathToken))”.

There are two types of Path Token Assertions: **Negative Path Token Assertion (NPTA)** and **Positive Path Token Assertion (PPTA)**. Both of them use the “SignaturePathTillPathToken” predicate, but unlike a PPTA, a NPTA adds a negation symbol (i.e., “not” in PDDL) in front of the predicate. The example shown in Figure 3.23 (B) includes two NPTAs. As we mentioned above, the main function of NPTA is to prevent a planner from generating a path
which leads to a specific mapping. On the other hand, a PPTA is to ask a planner to preserve a specific mapping. Figure 3.24 shows a hybrid example. In Figure 3.24 (A), a generated path leads to one correct mapping and two incorrect ones. In (B), we added 3 path tokens (PT1, PT2, and PT3) in the initial state, and one PPTA to “lock” the correct mapping, and two NPTA to “exclude” the two incorrect mappings. In (C), \textit{M.add} maps to \textit{MU.sum} because this mapping is required in the goal state. Regarding \textit{M.m1}, because the planner cannot map it to \textit{MU.m3} (since there is already a NPTA for this mapping) or \textit{MU.sum} (since \textit{M.add} needs to be mapped to \textit{MU.sum}), it will be mapped to a new target \textit{M.m1}. Similarly, a planner will map \textit{M.m1} to a new target \textit{MU.m3}. By comparing Figure 3.24 (A) and (C), it is clear that we successfully enforced a planner to give us a better solution.

**3.3.10.1 Complexity Analysis: N\times N**

Suppose that the computation time for a planner to produce a solution is a constant, and there are \(N\) same-parameter-types methods in the old API that needs to be mapped to \(N\)
By using negative path token assertions, there are at most $N - 1 = 3$ combinations which contains a wrong target of $m_1$, and $N - 2 = 2$ combinations for $m_2$, $N - 3 = 1$ combination for $m_3$, and 0 combination for $m_4$. So the total number of combinations in the worst case is $3 + 2 + 1 + 1 = 7$. Bold-typed fonts such as $m_{12}$ and $m_{11}$ in the first data row of (B) indicate the swapping of correct answers.

For example, suppose that after reduction (see Section 3.3.8) there are 15 methods in the old API but there are only 3 methods which have the same parameter types: $C1.m1$ (int, float, String, File), $C1.m5$ (int, float, String, File) and $C7.m1$ (int, float, String, File). Moreover, suppose that there are 18 methods in the new API but there are only 3 methods which have identical parameter types: $C1.m1$ (int, float, String, File), $C1.m4$ (int, float, String, File) and $C5.m5$ (int, float, String, File). In this case, there are 3 methods in the old API that need be mapped to 3 possible targets in the new API. Therefore, there are $3! = 6$ possible mapping results.

By adding path tokens with negative path token assertions, we just need to try at most $1 + \sum_{m=1}^{N-1} m$ times since for each method, whenever it maps to a wrong target, NPTA removes this target from its candidate list. Moreover, when that method mapped to a wrong target, it means that there exists another method also mapped to a wrong method (i.e., they “swapped” their correct targets). Figure 3.25 shows this idea. Figure 3.25 shows a 4x4 method mapping where the correct solution is $(m_1 \to m_{11}), (m_2 \to m_{12}), (m_3 \to m_{13})$ and $(m_4 \to m_{14})$. For $m_1$, a planner can map it to a wrong target for $N - 1 = 3$ times (see (B)’s combination 1.
to 3). However, for \( m_2 \), it only has \( N - 2 = 2 \) incorrect targets because after the combination 1 TARP already eliminated the \( (m_2 \rightarrow m_{11}) \) mapping. Similarly, for \( m_3 \), it only has \( N - 3 = 1 \) wrong target, and \( m_4 \) does not have any incorrect target. After all of the incorrect targets were found invalid, we will definitely get the correct answer in the next trial. Therefore, the time complexity retrieving a correct answer in the worst case is \( O(N^2) \).

If we use the PPTA and NPTA together, in the worst case, we can still get the correct answer in the \( N^{th} \) time, because during the first \( N - 1 \) trials, all methods already went through all wrong targets (NPTAs will prevent any incorrect mapping from showing up twice.) So, for each method, the \( N^{th} \) trial will always come with the correct answer. Therefore, the time complexity in the worst case will be \( O(N) \).

In normal cases, the correct solution may show up early. For example, in Figure 3.25 (B), by adopting PPTA (it already adopted NPTA), after processing the combination #1, the planner will skip #2, #4 and #6 because \( m_3 \) needs to be mapped to \( m_{13} \). Similarly, the planner will also skip #3 and #5 because \( m_4 \) needs to be mapped to \( m_{14} \). Therefore, we will get the correct result in the second trial.

### 3.3.10.2 Complexity Analysis: MxN

Suppose that the old API has \( M \) methods but the new API has \( N \) methods, where \( M < N \) (i.e., there are some newly added API; see Figure 3.26). By using negative path token assertions
Figure 3.27  A sample model: the initial state

(NPTA), in the worst case TARP need to try $M \times (N - 1)$ rounds to get the correct path. This is because there are maximum $(N - 1)$ rounds for each method in the old API to try out their incorrect mapping targets. So the total maximum number of rounds will be $M \times (N - 1)$. Thus, the complexity in the worst case is $O(MN)$.

By using positive path token assertions (PPTA), in the worst case, every round will remove one incorrect mapping target of every method in the old API. So the total maximum number of rounds will be $N$. Thus, the complexity in the worst case is $O(N)$.

### 3.3.10.3 Complexity Analysis: Multiple Groups

Suppose that there are $p$ groups of methods. Each group has $M_i$ methods and $N_i$ possible targets, where $0 < i \leq p$ and $M \leq N$. Because after each round the NTPA or PPTA will simultaneously remove at least one candidate in each group, the worst case will be found in the $j^{th}$ group where \( \max_{0 < i \leq p} (M_i \times N_i) = M_j \times N_j, 0 < j \leq p \), and it will be $O(M_jN_j)$ (when using PPTA) or $O(N_j)$ (when using PPTA).
3.3.11 The Final Modeling Strategy

Our final modeling strategy is composed of the strategies discussed in Section 3.3.2, Section 3.3.3, Section 3.3.4, Section 3.3.5, Section 3.3.7 and Section 3.3.10.

All examples in previous sections are simplified by hiding many types such as package and field, or properties such as modifiers. In this section, we will introduce the following new types and predicates for modeling those details:

1. **Root**: For each model, TARP will create a pseudo node as the root of the entire object tree. Its type is APIObject. It has no name, and its object identity is `Root`. An API model can have only one `Root`. `Root` is very important because it is the origin of all signature paths. A `Root` may contain one or more packages.

2. **Package**: A package is an APIObject. It is similar to a class object: it has a unique identity and it is related to a name. Moreover, there is a “SignaturePathTillPackage” predicate to track signature paths. A package may contain one or more classes.

3. **Modifier**: Modifier is a new type. TARP only defines two Modifier-typed object, one is
“static” and the other is “non-static”. If a class, method or field is static, TARP will use “(HasModifier ?obj - APIObject ?Mod - Modifier)” predicate to model this concept.

4. **Class**: For a class object, TARP will use “HasName” to define its name, “Contains” to claim its methods, “HasModifier” to declare its modifier, and “Inherits” to model its ancestors. Besides, the “SignaturePathTillClass” predicate is used to track a signature path until reaching a class.

5. **Field**: For a field object, TARP will use “HasName” to define its name, “HasModifier” to declare its modifier. TARP will not model field types because it is not a part of a field’s signature. Besides, the “SignaturePathTillField” predicate is used to track a signature path until reaching a field.

6. **MethodParameterTypes**: MethodParameterTypes is a new type. TARP will use MethodParameterTypes to define a method’s parameter types. Note that TARP will model the entire parameter list as a single object, for example, “int,int” or “int,String,Car”. TARP will use the same mapping table discussed above to store the mapping from the name shown in a parameter list to its full name. Lastly, the predicate “SignaturePathTillMethod” will end with a MethodParameterTypes object, not a method’s name object.

7. **Method**: For a method object, TARP will use “HasName” to define its name, “HasModifier” to declare its modifier, and “(HasMethodParamTypes ?method ?parameterTypes)” to keep its parameter information. Besides, the “SignaturePathTillMethod” predicate is used to track a signature path until reaching a method.

8. **Path Token**: There will be no path token objects in the original model because path token is designed to request another AI-planning result. In an altered model, a method, field, class or package may “Contains” one or more path tokens. Besides, “SignaturePathTillPathToken” is the predicate to track the signature path in the initial state, and to make NTPA or PPTA in the goal state.

Figure 3.27 shows a sample initial-state encoding tree generated by our final modeling strategy. Figure 3.28 shows a sample goal-state encoding tree generated by the same strategy.
The “C” icons in Figure 3.27 represent the “Contains” predicates, the “N” icons show the “HasName” predicate, the “P” icons denote the “HasMethodParamType” predicates, and the “M” icons indicate the “HasModifier” predicates. It should be noted that there are no such icons shown in in Figure 3.28 because we do not know any object identity in the goal state as discussed in Section 3.3.4.

3.4 Implementation

3.4.1 System Architecture of TARP*

To verify TARP, we created TARP*, a lightweight implementation of TARP. As we mentioned in Section 3.1, TARP* supports the following patterns: “Rename Field”, “Move Field”, “Move Method”, “Rename Method”, “Pullup Method”, “Rename Class”, “Move Class” and “Rename Package”. Supporting “Pullup Method” confirms that TARP* is capable of supporting variadic refactoring patterns. Besides, TARP* is using three third-party tools: FF [32] as the planner, Randoop [23] as the test case generator, and ALTA* as the on-the-fly adapter. However, TARP* does not have the “adding new API” and “remove API feature”.

There are 3 modules in TARP (see Figure 3.3). Insides these modules, there are a total 6 of sub-modules and 3 third-party tools (see Figure 3.29). Details now follow:

1. Problem Modeling and Solving Module: This is the module to convert a pair of incompatible components (the old jar and the new jar) into an AI-Planning problem and use a planner to retrieve a solution. It contains 3 parts:

   (a) Component Context Extractor and Simplifier: In this is part, the content of input jars will be extracted and simplified in order to reduce the computational complexity.

   (b) PDDL Fact File Generator: In this part, a PDDL fact file (i.e., object declarations, initial state and the goal state) will be generated. Note: the domain PDDL (i.e., the type definitions, actions and predicate definitions) is predefined.

   (c) (Third Party) AI Planner Engine: TARP* will use a third-party tool called FF (Fast Forward), an award-winning AI-planner which supports PDDL 2.1 [43], to generate
a plan according to given domain and fact PDDL files.

2. **Adaptation-based Testing Module**: The main goal of this module is to verify whether a generated plan is correct. It will assume that plan is correct, and use it as the true refactoring history to do on-the-fly adaptation. The adaptation is to connect the tests cases designed for the old component with the new component. If the refactoring path is not correct, then the adaptation will fail. Reasonably, if all the tests passed, we know that the generated plan is correct. It contains 4 parts:

   (a) **Refactoring Path Converter**: It will convert the generated plan to a refactoring history in the Eclipse format. The output of this part is an XML file.

   (b) **(Third Party) Test Case Generator**: TARP will use Randoop to automatically generate test cases for the old component. Randoop can not only generate the tests but also create regression assertions. The idea is the following: Randoop will randomly launch method calls of the old component, and record all return values. Moreover, Randoop will assume that the old component is perfect (has no bug) so the collected return values can be used as assertion values. For example, if Randoop called a method add(5,3) and got 8, then Randoop will generate a test which assert the return value of calls add(5,3) is 8. In this way, Randoop can efficiently generate a lot of test cases.

   (c) **(Third Party) ALTA***: Once we have test cases, refactoring history in the Eclipse format and the old and new components, TARP will use ALTA*, an implementation of ALTA, to generate ALTA aspect, which is a jar file which contains on-the-fly adaptation logic.

   (d) **Test Executor**: TARP will then put the ALTA aspect, the new component and the generated test cases together, and run those test cases by standard JUnit executor. A test report will be generated.

3. **Result Analysis and Feedback Module**: Once the test report is ready, we can decide what to do next. If all the tests passed, we can let the user know that the correct plan
Figure 3.29 System Architecture of TARP*. There are six internal modules and three third-party tools (i.e., the dashed bubbles).

(i.e., refactoring path) is successfully reconstructed. If not, we need to alter our model so the planner can give us a different path. There are two parts in this module:

(a) Test Report analyzer: This tool will analyze the test report, and create a mapping correctness report that tells us which mapping (e.g., method A $\rightarrow$ method B) is correct and which is not.

(b) Path Token Generator: This tool will generate path tokens into our old model according to the mapping correctness report. The idea of path token was discussed in Section 3.3.10. Then TARP* will run the entire process again to get a new plan.

3.4.2 Encoding Details

For reducing computation complexity, we reduced the granularity of many concepts. For example, we define “PackageName”, “ClassName”, “MethodName” and “FieldName” to represent a general concept “name”. Similarly, for real API objects, we defined “Package”, “Class”,

```plaintext
3.4.2 Encoding Details

For reducing computation complexity, we reduced the granularity of many concepts. For example, we define “PackageName”, “ClassName”, “MethodName” and “FieldName” to represent a general concept “name”. Similarly, for real API objects, we defined “Package”, “Class”,
```
“Method” and “Field” rather than just a general type “APIObject”. In this way, we can redefine predicates and actions with specific parameter types. For example, “ContainsPackage ?r - APIRoot ?p - Package)”. With specific parameter types, an AI planner can directly eliminate many incorrect combinations of solutions.

The following is a complete list of all types and predicates. After this list, we provided the definition of the “rename method” action as a sample of standard actions in TARP.

```
(:types

;; object structures
APIObject - object
APIRoot - APIObject
Package - APIObject
Class - APIObject
Method - APIObject
Field - APIObject

;; names
PackageName - object
ClassName - object
MethodName - object
FieldName - object

;; types
MethodParamTypes - object

;; modifier
MethodModifier - object
FieldModifier - object

;; path tokens
PackagePathToken - Object
ClassPathToken - Object
MethodPathToken - Object
FieldPathToken - Object
```
(:predicates

;; tree structures
(ContainsPackage ?r - APIRoot ?p - Package)
(ContainsClass ?p - Package ?c - Class)
(ContainsMethod ?c - Class ?m - Method)
(ContainsField ?c - Class ?f - Field)

;; path token
(ContainsPackageNamePathToken ?p - Package ?t - PackagePathToken)
(ContainsClassNamePathToken ?c - Class ?t - ClassPathToken)
(ContainsMethodNamePathToken ?m - Method ?t - MethodPathToken)
(ContainsFieldPathToken ?f - Field ?t - FieldPathToken)

(HasPackageName ?p - Package ?pName - PackageName)
(HasClassName ?c - Class ?cName - ClassName)
(HasMethodName ?m - Method ?mName - MethodName)
(HasFieldName ?f - Field ?fName - FieldName)

;; types
(HasMethodParamTypes ?m - Method ?types - MethodParamTypes)

;; modifiers
(HasMethodModifier ?m - Method ?mod - MethodModifier)
(HasFieldModifier ?f - Field ?mod - FieldModifier)

;; signature paths
;; package
(SignaturePathTillPackage ?r - APIRoot ?pName - PackageName)
(SignaturePathTillPackagePathToken ?r - APIRoot ?pName - PackageName ?pToken - PackagePathToken)
:: class
(SignaturePathTillClass ?r – APIRoot ?pName – PackageName ?cName –
   ClassName)
(SignaturePathTillClassPathToken ?r – APIRoot ?pName – PackageName ?cName
   – ClassName ?cToken – ClassPathToken)

:: method — remember to include types
(SignaturePathTillMethod ?r – APIRoot ?pName – PackageName ?cName –
   ClassName ?mName – MethodName ?types – MethodParamTypes)
(SignaturePathTillMethodPathToken ?r – APIRoot ?pName – PackageName ?cName
   – ClassName ?mName – MethodName ?mParamTypes – MethodParamTypes ?
   mToken – MethodPathToken)

:: field
(SignaturePathTillField ?r – APIRoot ?pName – PackageName ?cName –
   ClassName ?fName – FieldName)
(SignaturePathTillFieldPathToken ?r – APIRoot ?pName – PackageName ?cName
   – ClassName ?fName – FieldName ?fToken – FieldPathToken)

:: inherit
(Inherit ?classChild – Class ?classParent – Class)

:: pull-up transactions
(notPullingUpMethods)

:: note: all objects, not names
(MethodDuringPullingUp ?m1 – Method ?m1ParamTypes – MethodParamTypes ?
   cFrom – Class ?cTo – Class)

:: rename method
(:action renameMethod

:parameters (?
   ?root = APIRoot
   ?p1 = Package


?p1Name – PackageName
?c1 – Class
?c1Name – ClassName
?m1 – Method
?m1Name – MethodName
?m1NewName – MethodName
?m1ParamTypes – MethodParamTypes

): precondition (and

(notPullingUpMethods)

;; Object structure check
(ContainsPackage ?root ?p1)
(ContainsClass ?p1 ?c1)
(ContainsMethod ?c1 ?m1)

;; name check
(HasPackageName ?p1 ?p1Name)
(HasClassName ?c1 ?c1Name)
(HasMethodName ?m1 ?m1Name)
(not (HasMethodName ?m1 ?m1NewName))
(HasMethodParamTypes ?m1 ?m1ParamTypes)

;; signature path
(SignaturePathTillPackage ?root ?p1Name)
(SignaturePathTillClass ?root ?p1Name ?c1Name)
(SignaturePathTillMethod ?root ?p1Name ?c1Name ?m1Name ?m1ParamTypes)
(not (SignaturePathTillMethod ?root ?p1Name ?c1Name ?m1Name ?m1NewName ?
     m1ParamTypes))

)

): effect (and
;; change unique id object structure
;; nothing needs to be changed in this case

;; change name part
( not (HasMethodName ?m1 ?m1Name))
(HasMethodName ?m1 ?m1NewName)

;; change related signature paths
;; 1. sigpathTillmethod m1, via c1
( not (SignaturePathTillMethod ?root ?p1Name ?c1Name ?m1Name ?m1ParamTypes))
(SignaturePathTillMethod ?root ?p1Name ?c1Name ?m1NewName ?m1ParamTypes)

;; 2. sigpathTillMethodPathToken: m1’s token
(forall (?oneMethodPathToken − MethodPathToken)
 (when (and
 (ContainsMethodPathToken ?m1 ?oneMethodPathToken)
 )
 (and
 ;; remove the sig path
 (not (SignaturePathTillMethodPathToken ?root ?p1Name ?c1Name ?m1Name ?m1ParamTypes ?oneMethodPathToken))
 ;; adding the path
 (SignaturePathTillMethodPathToken ?root ?p1Name ?c1Name ?m1NewName ?m1ParamTypes ?oneMethodPathToken)
 )
)
)
)
3.5 Evaluation

3.5.1 Open Source Component Refactoring Path Reconstruction Test

To verify TARP*, we chose three open source components as our subjects. They were: Apache POI version 3.1, whose lines of code (LOC) is 136K, Google Collections version 1.0, whose LOC is 32K, and Apache Commons Lang version 3.0.1, whose LOC is 55K. We designed 5 experiments toward these 3 components. In each experiment, we manually refactored one subject, and asked TARP* to reconstruct the refactoring path. Because we knew the real refactoring history, we could precisely verify if the generated results from TARP* is correct. Besides, because we have claimed in section 3.1 that TARP is capable of handling TDRS, we designed several experiments which contains TDRS. In addition, in each experiment, we also used Refactoring Crawler and LSdiff, two state-of-the-art static analysis tools, to find the path. In this way, we can compare TARP* with these two solutions to know its performance and effectiveness.

Table 3.1 shows the result. The first column of Table 3.1 is the experiment number, and the second column shows the subject of this experiment and its LOC. The third column shows the summary of the real refactoring steps that we applied to the subject. Column 4 tells us if there are TDRS in real refactoring paths. Columns 5-6 are about Refactoring Crawler. Column 5 shows the summary of the results retrieved by Refactoring crawler, and column 6 is the computation time. Similarly we have columns 7 and 8 for LSdiff. Columns 9 to 11 are about TARP*. Column 9 is the summary of the results, column 10 is the computation time, and column 11 shows the test report of the output path. Moreover, we recorded detailed refactoring steps regarding the columns 3, 5 7 and 9 of Table 3.1 in Table 3.2.

From the two tables, the results of Exp. 1 show that all of these three solutions successfully detected two independent “Rename Method” steps. LSdiff, however, produced 2 false positives: 1 “Inline Method” and 1 “Extract Method”. If we go check Table 3.2 (in the 3rd row of Exp. 1, line 2 and 4), we can see that these two actions are simply counteractions. Besides, Refactoring Crawler used 50.110 seconds and LSdiff used 49.610 seconds to get these results.

In Exp. 2, we renamed 1 package, and renamed 1 irrelevant class. In Exp. 3, we renamed
2 classes and renamed 4 methods. There are no TDRS in Exp. 2 and Exp. 3, so Refactoring Crawler and LSdiff should work in these two cases. However, In Exp. 2, Refactoring Crawler returned 2 false positives, and LSdiff produced 2 false negatives and 11 false positives. In Exp. 3, Refactoring Crawler returned 4 false negatives and 3 false positives, while LSdiff produced 5 false negatives and 21 false positives (see Table 3.2, Exp. 3, row3, and Figure 3.30).

Exp. 4 and 5 contains TDRS. In Exp. 4, we moved 1 static method from one class to another, then renamed that method. In Exp. 5, we renamed 1 method in a class, and renamed that class. As we expected, in these two experiments, both of Refactoring Crawler and LSdiff did not detect anything. On the other hand, TARP* successfully reconstruct the refactoring paths. Actually, in all of these 5 experiments, TARP* returned correct answer.

Regarding TARP*'s computation time, if we compare Table 3.1 row 1 and row 2, we can realize the LOC of the component is not the key factor of computation time. The key factor is the patterns. Because “Rename Package” will affect all the classes, methods and fields inside that package, the planner spent almost double amount of time to produce a plan.

As a conclusion, the verification results show that TARP* could really reconstruct the
Table 3.1 Open Source Component Refactoring Path Reconstruction Test Report

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Apache POI 3.1 (136K)</td>
<td>Renamed 2 independent methods in a class.</td>
<td>No</td>
<td>No</td>
<td>50.110</td>
<td>Partially, found all but also found 1 Inline Method and 1 Extract Method</td>
<td>49.010</td>
<td>Yes</td>
<td>66.102</td>
</tr>
<tr>
<td>2</td>
<td>Google Collection 1.0 (32K)</td>
<td>Renamed 1 package and renamed 1 irrelevant class.</td>
<td>No</td>
<td>Partially, found all but also found 2 Change Method Signature</td>
<td>30.544</td>
<td>No</td>
<td>Only found 11 Move Method.</td>
<td>Yes</td>
<td>116.031</td>
</tr>
<tr>
<td>3</td>
<td>Apache Commons Lang 3.0.1 (55K)</td>
<td>Renamed 2 classes and renamed 4 methods (all independent)</td>
<td>No</td>
<td>Partially, correctly found 2 Rename Method, but also found 3 Move Method inside the renamed class.</td>
<td>17.975</td>
<td>Partially, correctly found 1 Rename Method, but also found 13 Move Method, 6 More Field, 1 Inline Method, and 1 Extract Method</td>
<td>11.197</td>
<td>Yes</td>
<td>55.196</td>
</tr>
<tr>
<td>4</td>
<td>Apache POI 3.1 (136K)</td>
<td>Renamed a static method and moved it to another class.</td>
<td>Yes</td>
<td>No</td>
<td>91.300</td>
<td>No</td>
<td>Found nothing.</td>
<td>Yes</td>
<td>618 test, 100% passed</td>
</tr>
<tr>
<td>5</td>
<td>Apache POI 3.1 (136K)</td>
<td>Renamed a class and renamed one method inside this class.</td>
<td>Yes</td>
<td>No</td>
<td>91.526</td>
<td>No</td>
<td>Only found 2 Move Field.</td>
<td>Yes</td>
<td>42.253</td>
</tr>
</tbody>
</table>

refactoring path in large-scale components, even if there are TDRS in the refactoring history.

3.5.2 Open Source Component Official Test Cases Adaptation Test

To evaluate whether ALTA* can use the generated refactoring path to solve compatibility problems, we conducted the Open Source Component Official Test Cases Adaptation Test for each experiment shown in Table 3.1 and Table 3.2. In this test, we used the official test cases of these three open source components (before upgraded) as applications, and let ALTA* adapt these applications with the upgraded components on-the-fly according to the refactoring paths generated by the Open Source Component Refactoring Path Reconstruction Test.

Before we started, we ran those tests with the old components, and removed unsuccessful
Table 3.2  Open Source Component Refactoring Path Reconstruction Test: Refactoring Details

<table>
<thead>
<tr>
<th>Exp. ID</th>
<th>Tool</th>
<th>Detected Refactoring History</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Real History</td>
<td>Rename method ‘org.apache.poi.hpsf.Section.getOffset()’ to ‘getOffset_REN’</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rename method ‘org.apache.poi.hpsf.Section.getCodepage()’ to ‘getCodepage_REN’</td>
</tr>
<tr>
<td></td>
<td>Refactoring Crawler</td>
<td>Rename method ‘org.apache.poi.hpsf.Section.getOffset()’ to ‘getOffset_REN’</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rename method ‘org.apache.poi.hpsf.Section.getCodepage()’ to ‘getCodepage_REN’</td>
</tr>
<tr>
<td></td>
<td>LSdiff</td>
<td>Consolidate duplicate cond fragments (invalid refactoring step)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Inline method ‘org.apache.poi.hpsf.Section.toString()’</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Extract method ‘org.apache.poi.hpsf.Section.toString()’</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rename method ‘org.apache.poi.hpsf.Section.getOffset()’ to ‘getOffset_REN’</td>
</tr>
<tr>
<td></td>
<td>TARP*</td>
<td>Rename method ‘org.apache.poi.hpsf.Section.getOffset()’ to ‘getOffset_REN’</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rename method ‘org.apache.poi.hpsf.Section.getCodepage()’ to ‘getCodepage_REN’</td>
</tr>
<tr>
<td>2</td>
<td>Real History</td>
<td>Rename package ‘com.google.common.annotations’ to ‘com.google.common.annotations_REN’</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rename class ‘com.google.common.collect.ForwardingList’ to ‘ForwardingList_REN’</td>
</tr>
<tr>
<td></td>
<td>Refactoring Crawler</td>
<td>Rename package ‘com.google.common.annotations’ to ‘com.google.common.annotations_REN’</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rename class ‘com.google.common.collect.ForwardingList’ to ‘ForwardingList_REN’</td>
</tr>
<tr>
<td></td>
<td>LSdiff</td>
<td>Change method signature ‘com.google.common.collect.ForwardingList.iterator()` to ‘ForwardingList_REN.iterator()’</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Move method ‘com.google.common.collect.ForwardingList.iterator()’ to ‘ForwardingList_REN’</td>
</tr>
<tr>
<td></td>
<td>TARP*</td>
<td>Rename package ‘com.google.common.annotations’ to ‘com.google.common.annotations_REN’</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rename class ‘com.google.common.collect.ForwardingList’ to ‘ForwardingList_REN’</td>
</tr>
</tbody>
</table>

For each experiment shown in Table 3.1 and Table 3.2 we conducted three sub-tests. First, we ran the official tests with the old (i.e., before upgrade) component. Because we already removed all unsuccessful cases, all tests passed (see column 5 of Table 3.3).

Second, we ran the official tests with the new (i.e., after upgrade) component. In column 6 of Table 3.3), the test report of Exp. 2 shows 44 errors, Exp. 3 shows 7 errors, and Exp. 5 shows 3 errors. Those errors indicated compatibility problems which resulted from component upgrades. Besides, there was no compatibility problem in Exp. 1 and 4 because the official tests did not cover (i.e., execute) the changed methods.

Finally, the column 7 of Table 3.3 shows the test results via ALTA* on-the-fly adaptations. It shows that ALTA* successfully fixed all compatibility problems in Exp 2, Exp. 3, and Exp.
<table>
<thead>
<tr>
<th>Exp. ID</th>
<th>Tool</th>
<th>Detected Refactoring History</th>
</tr>
</thead>
</table>
Table 3.2 (Continued)
Open Source Component Refactoring Path Reconstruction Test Report

<table>
<thead>
<tr>
<th>Exp. ID</th>
<th>Tool</th>
<th>Detected Refactoring History</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Real History</td>
<td>Rename method <code>org.apache.poi.util.StringUtil.hasMultibyte(java.lang.String)</code> to <code>hasMultibyte_ren</code></td>
</tr>
<tr>
<td></td>
<td>Refactoring Crawler</td>
<td>Found Nothing</td>
</tr>
<tr>
<td></td>
<td>LSdiff</td>
<td>Found Nothing</td>
</tr>
<tr>
<td></td>
<td>TARP*</td>
<td>Rename method <code>org.apache.poi.util.StringUtil.hasMultibyte(java.lang.String)</code> to <code>hasMultibyte_ren</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Move method <code>org.apache.poi.util.StringUtil.hasMultibyte_ren(java.lang.String)</code> to <code>org.apache.poi.util.IOUtils</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Exp. ID</th>
<th>Tool</th>
<th>Detected Refactoring History</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Real History</td>
<td>Rename class <code>org.apache.poi.util.TempFile</code> to <code>TempFile_REN</code></td>
</tr>
<tr>
<td></td>
<td>Refactoring Crawler</td>
<td>Found Nothing</td>
</tr>
<tr>
<td></td>
<td>LSdiff</td>
<td>Move Field <code>org.apache.poi.util.TempFile.rnd</code> to <code>TempFile_REN.rnd</code></td>
</tr>
<tr>
<td></td>
<td>LSdiff</td>
<td>Move Field <code>org.apache.poi.util.TempFile.dir</code> to <code>TempFile_REN</code></td>
</tr>
<tr>
<td></td>
<td>TARP*</td>
<td>Rename class <code>org.apache.poi.util.TempFile</code> to <code>TempFile_REN</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Rename method <code>org.apache.poi.util.TempFile_REN.createTempFile(java.lang.String, java.lang.String)</code> to <code>createTempFile_REN</code></td>
</tr>
</tbody>
</table>

Table 3.3 Open Source Component Automatic Adaptation Results

<table>
<thead>
<tr>
<th>Exp. ID</th>
<th>Component (LOC)</th>
<th>Applied Refactorings</th>
<th>Has TDRS?</th>
<th>Official Tests (as Applications)</th>
<th>Execution Results</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Run with the Old Component</td>
<td>Run with the New Component without ALTA*</td>
<td>Run with the New Component with ALTA*</td>
</tr>
<tr>
<td>1</td>
<td>Apache POI 3.1 (136K)</td>
<td>Renamed 2 independent methods in a class.</td>
<td>No</td>
<td>927 Tests, 100% Pass, 0 Errors, 0 Failures. Time: 1.596 (Sec)</td>
<td>927 Tests, 100% Pass, 0 Errors, 0 Failures. Time: 1.677 (Sec)</td>
</tr>
<tr>
<td>2</td>
<td>Google Collection 1.0 (32K)</td>
<td>Renamed 1 package and renamed 1 irrelevant class.</td>
<td>No</td>
<td>220 Tests, 100% Pass, 0 Errors, 0 Failures. Time: 0.247 (Sec)</td>
<td>220 Tests, 80% Pass, 44 Errors, 0 Failures. Time: 0.247 (Sec)</td>
</tr>
<tr>
<td>3</td>
<td>Apache Commons Lang 3.0.1 (55K)</td>
<td>Renamed 2 classes and renamed 4 methods (all independent).</td>
<td>No</td>
<td>2013 Tests, 100% Pass, 0 Failures. Time: 7.596 (Sec)</td>
<td>2013 Tests, 99.7% Pass, 7 Errors, 0 Failures. Time: 7.596 (Sec)</td>
</tr>
<tr>
<td>4</td>
<td>Apache POI 3.1 (136K)</td>
<td>Moved 1 static method from one class to another, then renamed that method.</td>
<td>Yes</td>
<td>927 Tests, 100% Pass, 0 Errors, 0 Failures. Time: 1.510 (Sec)</td>
<td>927 Tests, 100% Pass, 0 Errors, 0 Failures. Time: 1.489 (Sec)</td>
</tr>
<tr>
<td>5</td>
<td>Apache POI 3.1 (136K)</td>
<td>Renamed 1 method in a class, and renamed that class.</td>
<td>Yes</td>
<td>927 Tests, 100% Pass, 0 Errors, 0 Failures. Time: 1.596 (Sec)</td>
<td>927 Tests, 99.7% Pass, 3 Errors, 0 Failures. Time: 1.571 (Sec)</td>
</tr>
</tbody>
</table>

5. For all tests, we also recorded the execution time. For example, in Exp. 3 (see the third data row of Table 3.3), column 5 shows that the execution time was 7.596 seconds when the official tests ran with the old components. Column 7 shows that the execution time was 8.213 seconds when the official tests ran with the new (i.e., incompatible) components via ALTA* on-the-fly adaptation. The average delay of all the 5 experiments (shown in Table 3.3) is 16.7%.

3.6 Conclusion

Refactoring history of upgraded components is valuable for automatic software adaptation. However, it is usually not available in the real world. In this study, we presented TARP, a
comprehensive framework which can fully automatically reconstruct missing refactoring history. TARP has three significant features. First, it supports temporal-dependent refactoring step (TDRS). Second, it can guarantee that the output results are correct. Third, it does not require any components source code.

We also evaluated TARP* by adopting it to discover refactoring paths for three well-known open source projects: Apache Commons Lang 3.0.1, Apache POI 3.1.0, and Google Commons 1.0. In addition, we used two state-of-the-art static analysis tools, Refactoring Crawler and LSdiff, to solve the same set of problems. The experimental results showed that TARP* can work well in large-scale projects, and it is the only current solution which can detect TDRS.

Furthermore, we also used the official test cases of the 3 open source components to verify if ALTA* can really solve compatibility problems according to the reconstructed refactoring path generated by TARP*. The experimental results showed that ALTA* successfully fixed all the compatibility problems in those experiments.

Future work is required to handle unsupported refactoring types in TARP. In our current design, TARP will generate an empty path if there is any unsupported refactoring step in the real refactoring history. This is the main limitation of TARP. Our goal for the next generation of TARP is to allow it to skip unsupported ones and generate a partial refactoring path.
CHAPTER 4. CONCLUSIONS AND FUTURE WORK

In the current work, two frameworks were proposed, ALTA (Chapter 2) and TARP (Chapter 3), to solve compatibility problems fully automatically. ALTA is an Aspect-Oriented Programming (AOP) based on-the-fly component adaptation framework. By inputting the refactoring history of an upgraded component, ALTA can generate a binary jar file, called ALTA Aspect, which contains run-time adaptation logic. With this jar file, applications created for the old (i.e., before upgrade) API can run smoothly with the new (i.e., after upgrade) component.

The main limitation of ALTA is that it relies on given refactoring history. Because refactoring history is not always available, TARP was proposed to automatically reconstruct the missing refactoring history.

TARP is a testing and AI-Planning based refactoring path reconstruction framework. By inputting the binary jar files of old and new version of the upgraded component, TARP can extract APIs from both components, model the APIs as an AI-Planning problem, and use an AI planner to solve it. The solution generated from the planner is actually a refactoring path from the old API to the new API. Then TARP will use a novel technique called adaptation-based testing to verify the generated path. If the path is correct, TARP will export it as an Eclipse-styled refactoring history. Otherwise, TARP will keep generating another solutions until it gets a correct one.

In addition, we implemented ALTA as ALTA*, and TARP as TARP*. We also evaluated these two tools separately in Chapter 2 and Chapter 3. In addition, we evaluated the combined solution TARP* + ALTA* in Section 3.5.2 of Chapter 3. The experimental results show that not only these two tools are both applicable of solving compatibility problems in real-world projects, but also they can work together to perform fully automatically component adaptation.

To sum up, the proposed TARP + ALTA solution has the following features:
1. **It can perform full-automatic component adaptation without any extra information**: The TARP + ALTA solution can fully automatically solve compatibility problems which resulted from refactoring-based software component evolution, without any extra information.

2. **It can work without any source code of components**: The TARP + ALTA solution only requires the binary jar file of applications and components, so it can fix compatibility problems among binary components. Moreover, it will not statically modify any component or application, so this solution is valid under all kinds of license agreements.

3. **It will not statically modify any application or component**: The TARP + ALTA solution will adapt incompatible parts dynamically, so it can work under all kinds of license agreements.

4. **It can handle Temporal-Dependent Refactoring Steps (TDRS)**: The TARP + ALTA solution is the only solution to date which is able to handle TDRS.

4.1 **General Discussions**

In the past decade, component adaptation without extra human-coded or machine recorded change information emerged an open issue, because we need some clue to either upgrade application source code or to generate adapters for incompatible parts. In the current work, I showed a possible solution composed of the TARP and ALTA frameworks for this issue.

This work could be useful for self-evolving software frameworks such as Situ [44]. In a self-evolving software framework, components will change their API by themselves automatically; therefore we will not have software specifications, requirement documents, human-coded change information or machine-recorded refactoring information after a self-evolving process. In this case, if there is any compatibility problem found among components after a self-evolving process, people can apply the TARP + ALTA solution to fully automatically fix it.
4.2 Recommendations for Future Research

Unlike all existing solutions in this field, the TARP + ALTA solution transfers an compatibility problem into an AI-Planning problem, then use the adaptation-based testing technique to verify it. Regarding this solution, future research is required to:

1. **Simplify the contents of an API before converting it into a model**: An API may contain a lot of packages, classes, methods and fields. Because the computation time for an AI-planner to process an input model has a positive correlation with the size of the input model [32], it is critical to develop algorithms to simplify the contents of an API before converting them into an AI-planning model. Currently, TARP uses the “Simple Diff” algorithm to remove unchanged parts (see Section 3.3.8). However, this algorithm cannot remove anything inside a changed container.

   For example, suppose that there is a class $X$ which contains 3 methods: $m1$, $m2$ and $m3$ in the old API. In the new API, $X$ is renamed to $X_{ren}$ while it still contains $m1$, $m2$ and $m3$. In this case, when we run the “Simple Diff” algorithm, nothing will be removed because signatures of the class and its methods are all changed. For instance, the original method signature of $m2$ was $X.m2$ but now it becomes $X_{ren}.m2$.

   Therefore, it is important to design a new algorithm to handle changed containers. One important fact that we found in the previous example is that we can get exactly the same result from a planner without encoding $m2$ and $m3$. It is because $m1$, as “a representative of same-classed methods”, has already provided enough information for an AI-planner to generate the correct plan. Therefore, it is possible to use this “representative” concept to create a more effective simplification algorithm.

2. **Enhance the modeling strategy**: The current modeling strategy adds a lot of nodes to the encoding tree for a small component (see Figure 3.27 in Chapter 3). Thus, it is valuable to improve the modeling strategy to reduce the sizes of encoding trees. For example, it is possible to create NPTA or PPTA (see Section 3.3.10) by using object identities rather than creating additional path token objects.
3. **Create a customized test case generator**: There are many test case generators such as Randoop [23] and GenRed [42] which can automatically generate test cases with regression assertions for components. The common goal of these tools is to achieve high code coverage (with a minimum set of test cases) [42]. However, in our application, it is more important for a set of test cases to touch every method rather than to cover every line. For example, for a method which requires one special object as its parameter, say, an instance of a class *Coelacanth*, in order to cover more lines inside this method and to get a reusable return value from it, Randoop will not generate any test case for this method until it got a real instance of *Coelacanth*. If Randoop cannot get a *Coelacanth* object in a given period of processing time (e.g., 10 minutes), it will not generate a test case for this method. In this case, the generated test cases cannot help TARP to verify any mapping related to this method.

We can solve this problem by creating a customized test case generator that always generates test cases for all methods. For a method that requires hard-to-get parameters, our test case generator may simply pass *null* objects into it. In this way, TARP can get basic test cases for all methods. Moreover, our customized test case generator can generate test cases for private and protected methods for TARP to reconstruct internal refactoring paths.

4. **Support more patterns**: Currently the TARP* + ALTA* solution only supports 8 refactoring patterns (see Section 3.1.1.2). It is important to support more patterns.

5. **Handle unsupported refactoring types**: In the current design, TARP will generate an empty path if there is any unsupported refactoring step in the real refactoring history. Therefore, it is important to find a way to let TARP skip unsupported refactoring steps and generate a partial refactoring path. Although a partial refactoring path cannot be used to conduct automatic component adaptations, it is still valuable for people to read in order to understand what happened to the modified component.
APPENDIX A. Real Refactoring History of Exp. 3 in Table 3.1

<xml version="1.0" encoding="UTF-8"/>
- <session version="1.0">
  <refactoring version="1.0" textual="false" similarDeclarations="false" references="true" qualified="false" project="/opt/apache/commons-lang3.0.1-manyChanges_new" name="IDKey_REN" matchStrategy="1">
  </refactoring>
  <refactoring version="1.0" textual="false" similarDeclarations="false" references="true" qualified="false" project="/opt/apache.commons.lang3.0.1-manyChanges_new" name="MultiBackgroundInitializer_REN" input="/src/org.apache.commons.lang3.concurrent.MultiBackgroundInitializer.java [MultiBackgroundInitializer" id="org.eclipse.jdt.ui.rename.type.flags=589830" description="Rename type 'MultiBackgroundInitializer'" comment="Rename type 'org.apache.commons.lang3.concurrent.MultiBackgroundInitializer' to 'MultiBackgroundInitializer_REN'" Original project: '89_ApacheCommons.lang3.0.1-manyChanges_new' - Original element: 'org.apache.commons.lang3.concurrent.MultiBackgroundInitializer' - Renamed element: 'org.apache.commons.lang3.concurrent.MultiBackgroundInitializer_REN' - Update references to refactored element - Update textual occurrences in comments and strings/>
</session>
(define (domain api_refactoring)

(:constants
  Static_Method  MethodModifier
  Instance_Method MethodModifier
  Static_Field FieldModifier
  Instance_Field FieldModifier
)

(:types
  ;; structures
  APIObject object
  APIRoot  APIObject
  Package  APIObject
  Class    APIObject
  Method   APIObject
  Field    APIObject

  ;; names
  PackageName object
  ClassName  object
  MethodName object
  FieldName  object

  ;; types
  MethodParamTypes object

  ;; modifier
)
MethodModifier – object
FieldModifier – object

;; path tokens
MethodPathToken – Object

)

(:predicates
  ;; tree structures
  (ContainsPackage ?r – APIRoot ?p – Package)
  (ContainsClass ?p – Package ?c – Class)
  (ContainsMethod ?c – Class ?m – Method)
  (ContainsField ?c – Class ?f – Field)

  ;; path token
  (ContainsMethodPathToken ?m – Method ?t – MethodPathToken)

  (HasPackageName ?p – Package ?pName – PackageName)
  (HasClassName ?c – Class ?cName – ClassName)
  (HasMethodName ?m – Method ?mName – MethodName)
  (HasFieldName ?f – Field ?fName – FieldName)

  ;; types
  (HasMethodParamTypes ?m – Method ?types – MethodParamTypes)

  ;; modifiers
  (HasMethodModifier ?m – Method ?mod – MethodModifier)
  (HasFieldModifier ?f – Field ?mod – FieldModifier)

  ;; signature paths
  ;; package
  (SignaturePathTillPackage ?r – APIRoot ?pName – PackageName)

  ;; class
(SignaturePathTillClass ?r – APIRoot ?pName – PackageName ?cName –
  ClassName)

;; method — remember to include types
(SignaturePathTillMethod ?r – APIRoot ?pName – PackageName ?cName –
  ClassName
  ?mName – MethodName ?types – MethodParamTypes)
(SignaturePathTillMethodPathToken ?r – APIRoot ?pName – PackageName ?
  cName – ClassName
  ?mName – MethodName ?mParamTypes – MethodParamTypes ?mToken –
  MethodPathToken)

;; field
(SignaturePathTillField ?r – APIRoot ?pName – PackageName ?cName –
  ClassName
  ?fName – FieldName)

;; inherit
(Inherit ?classChild – Class ?classParent – Class)

;; pull-up transactions
(notPullingUpMethods)

;; note: all objects, not names
(MethodDuringPullingUp ?m1 – Method ?m1ParamTypes – MethodParamTypes
  ?cFrom – Class ?cTo – Class)

)

;;

;; --------------------------------------------------- actions ---------------------------------------------------

;;

;; rename field
(:action renameField
   :parameters (
     ?root – APIRoot
     ?p1 – Package
     ?p1Name – PackageName
     ?c1 – Class
     ?c1Name – ClassName
     ?f1 – Field
     ?f1Name – FieldName
     ?f1NewName – FieldName
   )
   :precondition (and
     (notPullingUpMethods)
     ;; uuid structure check
     (ContainsPackage ?root ?p1)
     (ContainsClass ?p1 ?c1)
     (ContainsField ?c1 ?f1)
     ;; name check
     (HasPackageName ?p1 ?p1Name)
     (HasClassName ?c1 ?c1Name)
     (HasFieldName ?f1 ?f1Name)
     (not (HasFieldName ?f1 ?f1NewName))
     ;; signature path
     (SignaturePathTillPackage ?root ?p1Name)
     (SignaturePathTillClass ?root ?p1Name ?c1Name)
     (SignaturePathTillField ?root ?p1Name ?c1Name ?f1Name)
     (not (SignaturePathTillField ?root ?p1Name ?c1Name ?f1NewName))
   )
   :effect (and
%; change unique id object structure
%; nothing needs to be changed

%; change name part
(not (HasFieldName ?f1 ?f1Name))
(HasFieldName ?f1 ?f1NewName)

%; change related signature paths
   ;; 1. sigpathTillmethod m1, via c1
(not (SignaturePathTillField ?root ?p1Name ?c1Name ?f1Name))
(SignaturePathTillField ?root ?p1Name ?c1Name ?f1NewName)

); we only allow to move static field right now.
(:action moveField
   :parameters (?
   ?root – APIRoot
   ?p1 – Package
   ?p1Name – PackageName
   ?CFrom – Class
   ?cFromName – ClassName
   ?f1 – Field
   ?f1Name – FieldName
   ?f1Modifier – FieldModifier
   ?p2 – Package
   ?p2Name – PackageName
   ?cTo – Class
   ?cToName – ClassName
)
   :precondition (and
(notPullingUpMethods)

;; uuid structure check
(ContainsPackage ?root ?p1)
(ContainsPackage ?root ?p2)
  (ContainsClass ?p1 ?cFrom)
  (ContainsClass ?p2 ?cTo)
  (ContainsField ?cFrom ?f1)
  (not (ContainsField ?cTo ?f1))

;; name check
(HasPackageName ?p1 ?p1Name)
(HasPackageName ?p2 ?p2Name)
(HasClassName ?cFrom ?cFromName)
(HasClassName ?cTo ?cToName)
(HasFieldName ?f1 ?f1Name)
(HasFieldModifier ?f1 ?f1Modifier)

;; critical part
(= ?f1Modifier Static_Field)

;; signature path
(SignaturePathTillPackage ?root ?p1Name)
(SignaturePathTillPackage ?root ?p2Name)
(SignaturePathTillClass ?root ?p1Name ?cFromName)
(SignaturePathTillClass ?root ?p2Name ?cToName)
(SignaturePathTillField ?root ?p1Name ?cFromName ?f1Name)
(not (SignaturePathTillField ?root ?p2Name ?cToName ?f1Name))

)
: effect (and

;; change unique id object structure
(not (ContainsField ?cFrom ?f1))
(ContainsField ?cTo ?f1)
;; change related signature paths

;; 1. sigpathTillField f1, via cFrom

(not (SignaturePathTillField ?root ?p1Name ?cFromName ?f1Name))

;; 2. 1. sigpathTillField f1, via cTo

(SignaturePathTillField ?root ?p2Name ?cToName ?f1Name)

)

)

;; we only allow to move static method right now.

(:action moveMethod

  :parameters (
    ?root – APIRoot
    ?p1 – Package
    ?p1Name – PackageName
    ?CFrom – Class
    ?cFromName – ClassName
    ?m1 – Method
    ?m1Name – MethodName
    ?m1ParamTypes – MethodParamTypes
    ?m1Modifier – MethodModifier

    ?p2 – Package
    ?p2Name – PackageName
    ?cTo – Class
    ?cToName – ClassName
    )

  :precondition (and

    (notPullingUpMethods)

    ;; uuid structure check
    (ContainsPackage ?root ?p1)
(ContainsPackage ?root ?p2)
(ContainsClass ?p1 ?cFrom)
(ContainsClass ?p2 ?cTo)
(ContainsMethod ?cFrom ?m1)
(not (ContainsMethod ?cTo ?m1))

;; name check
(HasPackageName ?p1 ?p1Name)
(HasPackageName ?p2 ?p2Name)
(HasClassName ?cFrom ?cFromName)
(HasClassName ?cTo ?cToName)
(HasMethodName ?m1 ?m1Name)
(HasMethodParamTypes ?m1 ?m1ParamTypes)
(HasMethodModifier ?m1 ?m1Modifier)

;; critical part
 (= ?m1Modifier Static_Method)

;; signature path
(SignaturePathTillPackage ?root ?p1Name)
(SignaturePathTillPackage ?root ?p2Name)
(SignaturePathTillClass ?root ?p1Name ?cFromName)
(SignaturePathTillClass ?root ?p2Name ?cToName)
(SignaturePathTillMethod ?root ?p1Name ?cFromName ?m1Name ?m1ParamTypes)
(not (SignaturePathTillMethod ?root ?p2Name ?cToName ?m1Name ?m1ParamTypes))

)

):effect (and

;; change unique id object structure
(not (ContainsMethod ?cFrom ?m1))
(ContainsMethod ?cTo ?m1)

;; change related signature paths
;; 1. sigpathTillMethod m1, via cFrom
(not (SignaturePathTillMethod ?root ?p1Name ?cFromName ?m1Name ?m1ParamTypes))

;; 2.1. sigpathTillMethod m1, via cTo
(SignaturePathTillMethod ?root ?p2Name ?cToName ?m1Name ?m1ParamTypes)

;; 3. sigpathTillMethodPathToken: m1's token, remove all via cFrom and add all via cTo
(forall (?oneMethodPathToken - MethodPathToken)
  (when (and
    (ContainsMethodPathToken ?m1 ?oneMethodPathToken)
  )
  )
  (and
    ;; remove the sig path via cFrom
    (not (SignaturePathTillMethodPathToken ?root ?p1Name ?cFromName ?m1Name ?m1ParamTypes ?oneMethodPathToken))
    ;; adding the path via cTo
    (SignaturePathTillMethodPathToken ?root ?p2Name ?cToName ?m1Name ?m1ParamTypes ?oneMethodPathToken)
  )
)
)
)
)
;
;; rename method
(:action renameMethod
  :parameters (
    ?root - APIRoot
    ?p1 - Package
    ?p1Name - PackageName
    ?c1 - Class
    ?c1Name - ClassName
    ?m1 - Method
    ?m1Name - MethodName
    ?m1NewName - MethodName
  )
)
?m1ParamTypes = MethodParamTypes

:precondition (and

(notPullingUpMethods)

;; uuid structure check
(ContainsPackage ?root ?p1)
(ContainsClass ?p1 ?c1)
(ContainsMethod ?c1 ?m1)

;; name check
(HasPackageName ?p1 ?p1Name)
(HasClassName ?c1 ?c1Name)
(HasMethodName ?m1 ?m1Name)
(not (HasMethodName ?m1 ?m1NewName))
(HasMethodParamTypes ?m1 ?m1ParamTypes)

;; signature path
(SignaturePathTillPackage ?root ?p1Name)
(SignaturePathTillClass ?root ?p1Name ?c1Name)
(SignaturePathTillMethod ?root ?p1Name ?c1Name ?m1Name ?m1ParamTypes)
(not (SignaturePathTillMethod ?root ?p1Name ?c1Name ?m1Name ?m1NewName ?m1ParamTypes))

)

:effect (and

;; change unique id object structure
;; nothing needs to be changed

;; change name part
(not (HasMethodName ?m1 ?m1Name))
(HasMethodName ?m1 ?m1NewName)
333 ;; change related signature paths
334 ;; 1. sigpathTillMethod m1, via c1
335 (not (SignaturePathTillMethod ?root ?p1Name ?c1Name ?m1Name ?m1ParamTypes 
336 ))
337 (SignaturePathTillMethod ?root ?p1Name ?c1Name ?m1NewName ?m1ParamTypes)
338
339 ;; 3. sigpathTillMethodPathToken: m1’s token
340 (forall (?oneMethodPathToken − MethodPathToken)
341 (when (and
342 (ContainsMethodPathToken ?m1 ?oneMethodPathToken)
343 )
344 (and
345 ;; remove the sig path
346 (not (SignaturePathTillMethodPathToken ?root ?p1Name ?c1Name ?
347 m1Name ?m1ParamTypes ?oneMethodPathToken)))
348 ;; adding the path
349 (SignaturePathTillMethodPathToken ?root ?p1Name ?c1Name ?m1NewName
350 ?m1ParamTypes ?oneMethodPathToken)
351 )
352 )
353
354 (:action pullupMethods_start
355 :parameters ( 
356 ?root − APIRoot
357 ?p1 − Package
358 ?p1Name − PackageName
359 ?CFrom − Class
360 ?cFromName − ClassName
361 ?m1 − Method
362 ?m1Name − MethodName
363 ?m1ParamTypes − MethodParamTypes
364 )
: precondition (and

;; not running pulling up method
(notPullingUpMethods)

;; uuid structure check
(ContainsPackage ?root ?p1)
(ContainsPackage ?root ?p2)
  (ContainsClass ?p1 ?cFrom)
  (ContainsClass ?p2 ?cTo)
  (ContainsMethod ?cFrom ?m1)
  (not (ContainsMethod ?cTo ?m1))

;; name check
(HasPackageName ?p1 ?p1Name)
(HasPackageName ?p2 ?p2Name)
(HasClassName ?cFrom ?cFromName)
(HasClassName ?cTo ?cToName)
(HasMethodName ?m1 ?m1Name)
(HasMethodParamTypes ?m1 ?m1ParamTypes)

;; cFrom extends cTo
(Inherit ?cFrom ?cTo)

;; signature path
(SignaturePathTillPackage ?root ?p1Name)
(SignaturePathTillPackage ?root ?p2Name)
(SignaturePathTillClass ?root ?p1Name ?cFromName)
(SignaturePathTillClass ?root ?p2Name ?cToName)
(SignaturePathTillMethod ?root ?p1Name ?cFromName ?m1Name ?m1ParamTypes)
(not (SignaturePathTillMethod ?root ?p2Name ?cToName ?m1Name ?m1ParamTypes))

:effect (and
    ;; just start up
    (MethodDuringPullingUp ?m1 ?m1ParamTypes ?cFrom ?cTo)
    (not (notPullingUpMethods))
)

(:action pullUpMethods_mergingSiblings
    :parameters (
        ?root – APIRoot

        ?p1 – Package
        ?p1Name – PackageName
        ?cFrom – Class
        ?cFromName – ClassName
        ?m1 – Method
        ?m1Name – MethodName
        ?m1ParamTypes – MethodParamTypes
        ?m1Modifier – MethodModifier

        ?p2 – Package
        ?p2Name – PackageName
        ?cSibling – Class
        ?cSiblingName – ClassName
        ?m2 – Method
    ;; name note: m2 should be in m1Name!
    ;; type note: m2 should has the same param types!

        ?cTo – Class
    )
)
:precondition (and

;; must during pulling up process
(MethodDuringPullingUp ?m1 ?m1ParamTypes ?cFrom ?cTo)
(not (notPullingUpMethods))

;; uuid structure check
(ContainsPackage ?root ?p1)
(ContainsPackage ?root ?p2)
(ContainsClass ?p1 ?cFrom)
(ContainsClass ?p2 ?cSibling)
(ContainsMethod ?cFrom ?m1)
(ContainsMethod ?cSibling ?m2)

;; name check
(HasPackageName ?p1 ?p1Name)
(HasPackageName ?p2 ?p2Name)
(HasClassName ?cFrom ?cFromName)
(HasClassName ?cSibling ?cSiblingName)
(HasMethodName ?m1 ?m1Name)
(HasMethodName ?m2 ?m1Name) ;; this part is very critical

;; types
(HasMethodParamTypes ?m1 ?m1ParamTypes)
(HasMethodParamTypes ?m2 ?m1ParamTypes) ;; this part is very critical

;; modifier
;; we don’t need to check the static part.
;; this is okay because pull up can be mixed.
;; anyway, I think that eclipse has a bug in this issue
;; so I am going to forbid it.
(HasMethodModifier ?m1 ?m1Modifier)
(HasMethodModifier ?m2 ?m1Modifier)

;; inherit
cFrom extends cTo

(Inherit ?cFrom ?cTo)

(Inherit ?cSibling ?cTo)

; ; signature path

(SignaturePathTillPackage ?root ?p1Name)

(SignaturePathTillPackage ?root ?p2Name)

(SignaturePathTillClass ?root ?p1Name ?cFromName)

(SignaturePathTillClass ?root ?p2Name ?cSiblingName)

(SignaturePathTillMethod ?root ?p1Name ?cFromName ?m1Name ?m1ParamTypes)

; ; this part is very critical

(SignaturePathTillMethod ?root ?p2Name ?cSiblingName ?m1Name ?m1ParamTypes)

)

: effect (and

; ; move all its path token

(forall (?oneMethodPathToken MethodPathToken)

  (when (and

    (ContainsMethodPathToken ?m2 ?oneMethodPathToken)

  )

)

  (and

    ; ; 1. move path token to m1

    (not (ContainsMethodPathToken ?m2 ?oneMethodPathToken))

    (ContainsMethodPathToken ?m1 ?oneMethodPathToken)

  )

)

; ; remove all method path token signature paths through m2

(not (SignaturePathTillMethodPathToken ?root ?p2Name ?cSiblingName ?m1Name ?m1ParamTypes ?oneMethodPathToken))

; ; add new method path token signature paths through m1

(SignaturePathTillMethodPathToken ?root ?p1Name ?cFromName ?m1Name ?m1ParamTypes ?oneMethodPathToken)

)

)
;; representative
;; remove all rest relations with m2
;; with container
(not (ContainsMethod ?cSibling ?m2))
;; with name
(not (HasMethodName ?m2 ?m1Name))
;; with types
(not (HasMethodParamTypes ?m2 ?m1ParamTypes))
;; with modifier
(not (HasMethodModifier ?m2 ?m1Modifer))
;; remove signature path till m2
(not (SignaturePathTillMethod ?root ?p2Name ?cSiblingName ?m1Name ?m1ParamTypes))

(:action pullupMethods_end
 :parameters (~root ~APIRoot
 ~p1 ~Package
 ~p1Name ~PackageName
 ~CFrom ~Class
 ~CFromName ~ClassName
 ~m1 ~Method
 ~m1Name ~MethodName
 ~m1ParamTypes ~MethodParamTypes

 ~p2 ~Package
 ~p2Name ~PackageName
 ~cTo ~Class
 ~cToName ~ClassName
)
:precondition (and

;; must during pulling up process
(MethodDuringPullingUp ?m1 ?m1ParamTypes ?cFrom ?cTo)
(not (notPullingUpMethods))

;; uuid structure check
(ContainsPackage ?root ?p1)
(ContainsPackage ?root ?p2)
(ContainsClass ?p1 ?cFrom)
(ContainsClass ?p2 ?cTo)
(ContainsMethod ?cFrom ?m1)
(not (ContainsMethod ?cTo ?m1))

;; name check
(HasPackageName ?p1 ?p1Name)
(HasPackageName ?p2 ?p2Name)
(HasClassName ?cFrom ?cFromName)
(HasClassName ?cTo ?cToName)
(HasMethodName ?m1 ?m1Name)
(HasMethodParamTypes ?m1 ?m1ParamTypes)

;; cFrom extends cTo
(Inherit ?cFrom ?cTo)

;; signature path
(SignaturePathTillPackage ?root ?p1Name)
(SignaturePathTillPackage ?root ?p2Name)
(SignaturePathTillClass ?root ?p1Name ?cFromName)
(SignaturePathTillClass ?root ?p2Name ?cToName)
(SignaturePathTillMethod ?root ?p1Name ?cFromName ?m1Name ?m1ParamTypes)
(not (SignaturePathTillMethod ?root ?p2Name ?cToName ?m1Name ?m1ParamTypes) )
); effect (and

;; change unique id object structure

(not (ContainsMethod ?cFrom ?m1))
(ContainsMethod ?cTo ?m1)

;; change related signature paths

;; 1. sigpathTillmethod m1, via cFrom

(not (SignaturePathTillMethod ?root ?p1Name ?cFromName ?m1Name ?m1ParamTypes))

;; 2. 1. sigpathTillmethod m1, via cTo

(SignaturePathTillMethod ?root ?p2Name ?cToName ?m1Name ?m1ParamTypes)

;; 3. sigpathTillMethodPathToken: m1’s token, remove all via cFrom and add all via cTo

(forall (?oneMethodPathToken − MethodPathToken)

  (when (and

    (ContainsMethodPathToken ?m1 ?oneMethodPathToken)

  )

  (and

    ;; remove the sig path via cFrom

    (not (SignaturePathTillMethodPathToken ?root ?p1Name ?cFromName ?m1Name ?m1ParamTypes ?oneMethodPathToken))

    ;; adding the path via cTo

    (SignaturePathTillMethodPathToken ?root ?p2Name ?cToName ?m1Name ?m1ParamTypes ?oneMethodPathToken)

  ))
)
)

;; end this process

(not (MethodDuringPullingUp ?m1 ?m1ParamTypes ?cFrom ?cTo))
(notPullingUpMethods)
)
}
(:action renameClass

  :parameters (
      ?root – APIRoot
      ?p1 – Package
      ?p1Name – PackageName
      ?C1 – Class
      ?c1Name – ClassName
      ?c1NewName – ClassName
    )

  :precondition (and
    (notPullingUpMethods)

    ;; object structure
    (ContainsPackage ?root ?p1)
    (ContainsClass ?p1 ?c1)

    ;; name
    (HasPackageName ?p1 ?p1Name)
    (HasClassName ?c1 ?c1Name)
    (not (HasClassName ?c1 ?c1NewName))

    ;; sig path
    (SignaturePathTillPackage ?root ?p1Name)
    (SignaturePathTillClass ?root ?p1Name ?c1Name)
    (not (SignaturePathTillClass ?root ?p1Name ?c1NewName))
  )

  :effect (and
    ;; the object structure didn't change.
    ;; change object c1's name relation
    (not (HasClassName ?c1 ?c1Name))
    (HasClassName ?c1 ?c1NewName)
;; change sig path till class
(n (SignaturePathTillClass ?root ?pName ?cName))

(SignaturePathTillClass ?root ?pName ?cNewName)

;; change sig path till method
(forall (?oneMethod - Method

  ?oneMethodName - MethodName
  ?oneMethodParamTypes - MethodParamTypes)

  (when (and

    (ContainsMethod ?c1 ?oneMethod)
    (HasMethodName ?oneMethod ?oneMethodName)
    (HasMethodParamTypes ?oneMethod ?oneMethodParamTypes)
    )

    (and

      (SignaturePathTillMethod ?root ?pName ?cNewName ?oneMethodName ?
       oneMethodParamTypes)
      (not (SignaturePathTillMethod ?root ?pName ?cName ?oneMethodName ?
       oneMethodParamTypes))
      )

    )

  )

)

;; change sig path till method's path token

(forall (?oneMethod - Method

  ?oneMethodName - MethodName
  ?oneMethodParamTypes - MethodParamTypes
  ?oneMethodPathToken - MethodPathToken)

  (when (and

    (ContainsMethod ?c1 ?oneMethod)
    (HasMethodName ?oneMethod ?oneMethodName)
    (HasMethodParamTypes ?oneMethod ?oneMethodParamTypes)
    (ContainsMethodPathToken ?oneMethod ?oneMethodPathToken)
    )

  )

)
(and
  (SignaturePathTillMethodPathToken ?root ?p1Name ?c1NewName ?oneMethodName ?oneMethodParamTypes ?oneMethodPathToken)
  (not (SignaturePathTillMethodPathToken ?root ?p1Name ?c1Name ?oneMethodName ?oneMethodParamTypes ?oneMethodPathToken))
)
)
)

;; change sig path till field
(forall (?oneField – Field
  ?oneFieldName – FieldName)
  (when (and
    (ContainsField ?c1 ?oneField)
    (HasFieldName ?oneField ?oneFieldName)
  )
  (and
    (SignaturePathTillField ?root ?p1Name ?c1NewName ?oneFieldName)
    (not (SignaturePathTillField ?root ?p1Name ?c1Name ?oneFieldName))
  )
)
)
)
)

(:action moveClass
  :parameters (?
    ?root – APIRoot
    ?p1 – Package
    ?p1Name – PackageName
    ?C1 – Class
    ?c1Name – ClassName
    ;; target
    ?p2 – Package
    ?p2Name – PackageName
  )
:precondition (and

( notPullingUpMethods )

;; uuid structure check
(ContainsPackage ?root ?p1)
(ContainsPackage ?root ?p2)
(ContainsClass ?p1 ?c1)
(not (ContainsClass ?p2 ?c1))

;; name check
(HasPackageName ?p1 ?p1Name)
(HasPackageName ?p2 ?p2Name)
(HasClassName ?c1 ?c1Name)

;; signature path
(SignaturePathTillPackage ?root ?p1Name)
(SignaturePathTillPackage ?root ?p2Name)
(SignaturePathTillClass ?root ?p1Name ?c1Name)
(not (SignaturePathTillClass ?root ?p2Name ?c1Name)) ;; this prevents name-conflict after moving

;; without a forall !!!!
)

:effect (and

;; uuid structure change
(not (ContainsClass ?p1 ?c1))
(ContainsClass ?p2 ?c1)

;; change sig path till class
(not (SignaturePathTillClass ?root ?p1Name ?c1Name))
(SignaturePathTillClass ?root ?p2Name ?c1Name)

;; change sig path till method
(forall (?oneMethod - Method

?oneMethodName - MethodName

?oneMethodParamTypes - MethodParamTypes)
(when (and
    (ContainsMethod ?c1 ?oneMethod)
    (HasMethodName ?oneMethod ?oneMethodName)
    (HasMethodParamTypes ?oneMethod ?oneMethodParamTypes)
)
(and
    (SignaturePathTillMethod ?root ?p2Name ?c1Name ?oneMethodName ?oneMethodParamTypes)
    (not (SignaturePathTillMethod ?root ?p1Name ?c1Name ?oneMethodName ?oneMethodParamTypes))
)
)
)
)

;; change sig path till method's path token

(forall (?oneMethod - Method
    ?oneMethodName - MethodName
    ?oneMethodParamTypes - MethodParamTypes
    ?oneMethodPathToken - MethodPathToken)
(when (and
    (ContainsMethod ?c1 ?oneMethod)
    (HasMethodName ?oneMethod ?oneMethodName)
    (HasMethodParamTypes ?oneMethod ?oneMethodParamTypes)
    (ContainsMethodPathToken ?oneMethod ?oneMethodPathToken)
)
)
(and
    (SignaturePathTillMethodPathToken ?root ?p2Name ?c1Name ?oneMethodName ?oneMethodParamTypes ?oneMethodPathToken)
    (not (SignaturePathTillMethodPathToken ?root ?p1Name ?c1Name ?oneMethodName ?oneMethodParamTypes ?oneMethodPathToken))
)
)
)
;; change sig path till field
(forall (?oneField – Field
  ?oneFieldName – FieldName )
  (when (and
    (ContainsField ?c1 ?oneField)
    (HasFieldName ?oneField ?oneFieldName)
  )
  (and
    (SignaturePathTillField ?root ?p2Name ?c1Name ?oneFieldName)
    (not (SignaturePathTillField ?root ?p1Name ?c1Name ?oneFieldName))
  )
)
)
)
)
):action renamePackage
  :parameters (?
    ?root – APIRoot
    ?p1 – Package
    ?p1Name – PackageName
    ?p1NewName – PackageName
  )
  :precondition (and
(NO)

  (ContainsPackage ?root ?p1)
  (HasPackageName ?p1 ?p1Name)
  (not (HasPackageName ?p1 ?p1NewName))

  (SignaturePathTillPackage ?root ?p1Name)
  (not (SignaturePathTillPackage ?root ?p1NewName))
)
  :effect (and

;; change uuid object structure
(not (HasPackageName ?p1 ?p1Name))
(HasPackageName ?p1 ?p1NewName)

;; change sig path till package
(SignaturePathTillPackage ?root ?p1NewName)
(not (SignaturePathTillPackage ?root ?p1Name ))

;; change sig path till class
(forall (?oneClass - Class

   ?oneClassName - ClassName)
   (when (and
      (ContainsClass ?p1 ?oneClass)
      (HasClassName ?oneClass ?oneClassName)
   )
   (and
      (SignaturePathTillClass ?root ?p1NewName ?oneClassName)
      (not (SignaturePathTillClass ?root ?p1Name ?oneClassName ))
   )
   )
)

;; change sig path till method
(forall ( ?oneClass - Class

   ?oneClassName - ClassName
   ?oneMethod - Method
   ?oneMethodName - MethodName
   ?oneMethodParamTypes - MethodParamTypes)
   (when (and
      (ContainsClass ?p1 ?oneClass)
      (HasClassName ?oneClass ?oneClassName)
      (ContainsMethod ?oneClass ?oneMethod)
      (HasMethodName ?oneMethod ?oneMethodName)
   )
   (and
      (SignaturePathTillMethod ?root ?p1NewName ?oneClassName ?oneMethodName)
      (not (SignaturePathTillMethod ?root ?p1Name ?oneClassName ?oneMethodName )))
   )
)
(HasMethodParamTypes ?oneMethod ?oneMethodParamTypes)
)
(and
(SignaturePathTillMethod ?root ?p1NewName ?oneClassName ?
oneMethodName ?oneMethodParamTypes)
(not (SignaturePathTillMethod ?root ?p1Name ?oneClassName ?
oneMethodName ?oneMethodParamTypes))
)
)

;; change sig path till method's path token

(forall (

?oneClass - Class

?oneClassName - ClassName

?oneMethod - Method

?oneMethodName - MethodName

?oneMethodParamTypes - MethodParamTypes

?oneMethodPathToken - MethodPathToken)

(when (and

(ContainsClass ?p1 ?oneClass)
(HasClassName ?oneClass ?oneClassName)
(ContainsMethod ?oneClass ?oneMethod)
(HasMethodName ?oneMethod ?oneMethodName)
(HasMethodParamTypes ?oneMethod ?oneMethodParamTypes)
(ContainsMethodPathToken ?oneMethod ?oneMethodPathToken)
)
(and

(SignaturePathTillMethodPathToken ?root ?p1NewName ?oneClassName ?
oneMethodName ?oneMethodParamTypes ?oneMethodPathToken)
(not (SignaturePathTillMethodPathToken ?root ?p1Name ?oneClassName 
?oneMethodName ?oneMethodParamTypes ?oneMethodPathToken))
)
)
)
; change sig path till field
(forall ( ?oneClass - Class
    ?oneClassName - ClassName
    ?oneField - Field
    ?oneFieldName - FieldName )
    (when (and
        (ContainsClass ?pl ?oneClass)
        (HasClassName ?oneClass ?oneClassName)
        (ContainsField ?oneClass ?oneField)
        (HasFieldName ?oneField ?oneFieldName)
    )
    (and
        (SignaturePathTillField ?root ?plNewName ?oneClassName ?oneField)
        (not (SignaturePathTillField ?root ?plName ?oneClassName ?oneField))
    )
    )
  )
)
APPENDIX C. Fact File of Exp. 3

(define (problem pb1)
  (:domain api_refactoring)
  (:requirements :strips :adl)
  (:objects

; ; general part
  dummyMTk - MethodPathToken
  dummyFieldName - FieldName ; ; for swapping method names
  dummyMethodObject - Method
  dummyFieldObject - Field
  VOID - MethodParamTypes

; ; Object and names used in Old API
  RT_root - APIRoot
  RT_root_PKG_PKGorgapachecommonslang3builder - Package
  PKG_PKGorgapachecommonslang3builder - PackageName
  RT_root_PKG_PKGorgapachecommonslang3builder_CLS_CLSidkey - Class
  CLS_CLSidkey - ClassName

  MethodParamTypes_MTDTYPECLASSjavalangobject - MethodParamTypes
  RT_root_PKG_PKGorgapachecommonslang3builder_CLS_CLSidkey_MTD_MTDequals - Method
  MTD_MTDequals - MethodName

  MethodParamTypes_MTDTYPEVOID - MethodParamTypes
  RT_root_PKG_PKGorgapachecommonslang3builder_CLS_CLSidkey_MTD_MTDhashcode - Method
  MTD_MTDhashcode - MethodName

  RT_root_PKG_PKGorgapachecommonslang3concurrent - Package
  PKG_PKGorgapachecommonslang3concurrent - PackageName

  RT_root_PKG_PKGorgapachecommonslang3concurrent_CLS_CLSmultibackgroundinitializer - Class
  CLS_CLSmultibackgroundinitializer - ClassName

  MethodParamTypes_MTDTYPECLASSjavalangstringclassorgapachecommonslang3concurrent(LineWrapMark)_backgroundinitializer - MethodParamTypes
  RT_root_PKG_PKGorgapachecommonslang3concurrent_CLS_CLSmultibackgroundinitializer(LineWrapMark)_MTD_MTDaddinitializer - Method
  MTD_MTDaddinitializer - MethodName

; ; (already declared !) MethodParamTypes_MTDTYPEVOID - MethodParamTypes
  RT_root_PKG_PKGorgapachecommonslang3concurrent_CLS_CLSmultibackgroundinitializer(LineWrapMark)_MTD_MTDgettaskcount - Method
  MTD_MTDgettaskcount - MethodName
)
45;; (already declared!) MethodParamTypes_MTDTYPESvoid = MethodParamTypes
46RT_root_PKG_PKGorgapachecommonslang3concurrent_CLS_CLSmultibackgroundinitializer_<LineWrapMark>
47  MTDMTDinitialize = Method
48  MTDMTDinitialize = MethodName
49
50RT_root_PKG_PKGorgapachecommonslang3math = Package
51PKG_PKGorgapachecommonslang3math = PackageName
52
53RT_root_PKG_PKGorgapachecommonslang3math_CLS_CLSnumberutils = Class
54CLS_CLSnumberutils = ClassName
55
56MethodParamTypes_MTDTYPESClassjavalangstring = MethodParamTypes
57RT_root_PKG_PKGorgapachecommonslang3math_CLS_CLSnumberutils_MTD_MTDcreatebigdecimal = Method
58MTDMTDcreatebigdecimal = MethodName
59
60MethodParamTypes_MTDTYPESClassjavalangstring = MethodParamTypes
61RT_root_PKG_PKGorgapachecommonslang3math_CLS_CLSnumberutils_MTD_MTDmax = Method
62MTDMTDmax = MethodName
63
64MethodParamTypes_MTDTYPESClassjavalangstring = MethodParamTypes
65RT_root_PKG_PKGorgapachecommonslang3math_CLS_CLSnumberutils_MTD_MTDmin = Method
66MTDMTDmin = MethodName
67
68MethodParamTypes_MTDTYPESClassjavalangstring = MethodParamTypes
69RT_root_PKG_PKGorgapachecommonslang3math_CLS_CLSnumberutils_MTD_MTDtobyte = Method
70MTDMTDtobyte = MethodName
71
72;; names used in New API (we don’t care object in the goal)
73;; (already declared!) RT_root = APIRoot
74
75;; (already declared!) PKG_PKGorgapachecommonslang3builder = PackageName
76
77CLS_CLSidkeyren = ClassName
78
79;; (already declared!) MTD_MTDequals = MethodName
80
81;; (already declared!) MTD_MTDhashcode = MethodName
82
83;; (already declared!) PKG_PKGorgapachecommonslang3concurrent = PackageName
84
85CLS_CLSmultibackgroundinitializerren = ClassName
86
87;; (already declared!) MTD_MTDaddinitializer = MethodName
88
89;; (already declared!) MTD_MTDgettaskcount = MethodName
90
91;; (already declared!) MTD_MTDinitialize = MethodName
92
93;; (already declared!) PKG_PKGorgapachecommonslang3math = PackageName
94
95;; (already declared!) CLS_CLSnumberutils = ClassName
96
97MTDMTDcreatebigdecimalren = MethodName
MTDmaxren MethodName

MTDminren MethodName

MTDtobyteren MethodName

}\n
{ init
  (notPullingUpMethods)
  (ContainsPackage RT_root PKG_PKGorgapachecommonslang3builder)
  (SignaturePathTillPackage RT_root PKG_PKGorgapachecommonslang3builder)
  (ContainsPackage RT_root PKG_PKGorgapachecommonslang3concurrent)
  (SignaturePathTillPackage RT_root PKG_PKGorgapachecommonslang3concurrent)
  (ContainsPackage RT_root PKG_PKGorgapachecommonslang3math)
  (SignaturePathTillPackage RT_root PKG_PKGorgapachecommonslang3math)
  ; ; package name
  (HasPackageName RT_root PKG_PKGorgapachecommonslang3builder
    PKG_PKGorgapachecommonslang3builder)
  (ContainsClass RT_root PKG_PKGorgapachecommonslang3builder
    RT_root_PKG_PKGorgapachecommonslang3builder_CLS_CLSidkey)
  (SignaturePathTillClass RT_root PKG_PKGorgapachecommonslang3builder_CLS_CLSidkey)
  ; ; class name
  (HasClassName RT_root PKG_PKGorgapachecommonslang3builder_CLS_CLSidkey
    CLS_CLSidkey)
  (ContainsMethod RT_root PKG_PKGorgapachecommonslang3builder_CLS_CLSidkey
    RT_root_PKG_PKGorgapachecommonslang3builder_CLS_CLSidkey_MTD_MTDequals)
  (SignaturePathTillMethod RT_root PKG_PKGorgapachecommonslang3builder_CLS_CLSidkey
    MTD_MTDequals MethodParamTypes MTDTYPESclassjavalangobject)
  (ContainsMethod RT_root PKG_PKGorgapachecommonslang3builder_CLS_CLSidkey
    RT_root_PKG_PKGorgapachecommonslang3builder_CLS_CLSidkey_MTD_MTDhashcode)
  (SignaturePathTillMethod RT_root PKG_PKGorgapachecommonslang3builder_CLS_CLSidkey
    MTD_MTDhashcode MethodParamTypes MTDTYPESvoid)
  ; ; method name
  (HasMethodName RT_root PKG_PKGorgapachecommonslang3builder_CLS_CLSidkey
    MTD_MTDequals)
  (HasMethodName RT_root PKG_PKGorgapachecommonslang3builder_CLS_CLSidkey
    MTD_MTDhashcode)
  (HasMethodParamTypes RT_root PKG_PKGorgapachecommonslang3builder_CLS_CLSidkey
    MTD_MTDequals MethodParamTypes MTDTYPESclassjavalangobject)
  (HasMethodModifier RT_root PKG_PKGorgapachecommonslang3builder_CLS_CLSidkey
    MTD_MTDequals Instance Method)
  ; ; method name
  (HasMethodName RT_root PKG_PKGorgapachecommonslang3builder_CLS_CLSidkey
    MTD_MTDhashcode)
  (HasMethodName RT_root PKG_PKGorgapachecommonslang3builder_CLS_CLSidkey
    MTD_MTDhashcode)
  (HasMethodParamTypes RT_root PKG_PKGorgapachecommonslang3builder_CLS_CLSidkey
    MTD_MTDhashcode MethodParamTypes MTDTYPESvoid)
  (HasMethodModifier RT_root PKG_PKGorgapachecommonslang3builder_CLS_CLSidkey
    MTD_MTDhashcode Instance Method)
  ; ; package name
  (HasPackageName RT_root PKG_PKGorgapachecommonslang3concurrent
    PKG_PKGorgapachecommonslang3concurrent)
  (ContainsClass RT_root PKG_PKGorgapachecommonslang3concurrent
    RT_root_PKG_PKGorgapachecommonslang3concurrent_CLSmultibackgroundinitializer)
  (SignaturePathTillClass RT_root PKG_PKGorgapachecommonslang3concurrent
    CLS_CLSmultibackgroundinitializer)
  ; ; class name
182 (ContainsMethod RT_root_PKG_PKGorgapachecommonslang3math_CLS_CLSnumberutils
   RT_root_PKG_PKGorgapachecommonslang3math_CLS_CLSnumberutils_MTD_MTDmax)

183 (SignaturePathTillMethod RT_root_PKG_PKGorgapachecommonslang3math_CLS_CLSnumberutils MTD_MTDmax
   MethodParamTypes_MTDTYPESlonglonglong)

184 (ContainsMethod RT_root_PKG_PKGorgapachecommonslang3math_CLS_CLSnumberutils
   RT_root_PKG_PKGorgapachecommonslang3math_CLS_CLSnumberutils_MTD_MTDmin)

185 (SignaturePathTillMethod RT_root_PKG_PKGorgapachecommonslang3math_CLS_CLSnumberutils MTD_MTDmin
   MethodParamTypes_MTDTYPESclassd)

186 (ContainsMethod RT_root_PKG_PKGorgapachecommonslang3math_CLS_CLSnumberutils
   RT_root_PKG_PKGorgapachecommonslang3math_CLS_CLSnumberutils_MTD_MTDtobyte)

187 (SignaturePathTillMethod RT_root_PKG_PKGorgapachecommonslang3math_CLS_CLSnumberutils MTD_MTDtobyte
   MethodParamTypes_MTDTYPESclassjavalongstringbyte)

188 ;; method name

189 (HasMethodName
   RT_root_PKG_PKGorgapachecommonslang3math_CLS_CLSnumberutils_MTD_MTDcreatebigdecimal
   "createbigdecimal")

190 (HasMethodParamTypes
   RT_root_PKG_PKGorgapachecommonslang3math_CLS_CLSnumberutils_MTD_MTDcreatebigdecimal
   MethodParamTypes_MTDTYPESclassjavalongstring)

191 (HasMethodModifier
   RT_root_PKG_PKGorgapachecommonslang3math_CLS_CLSnumberutils_MTD_MTDcreatebigdecimal
   Static_Method)

192 ;; method name

193 (HasMethodName
   RT_root_PKG_PKGorgapachecommonslang3math_CLS_CLSnumberutils_MTD_MTDmax
   "createbigdecimal")

194 (HasMethodParamTypes
   RT_root_PKG_PKGorgapachecommonslang3math_CLS_CLSnumberutils_MTD_MTDmax
   MethodParamTypes_MTDTYPESlonglonglong)

195 (HasMethodModifier
   RT_root_PKG_PKGorgapachecommonslang3math_CLS_CLSnumberutils_MTD_MTDmax
   Static_Method)

196 ;; method name

197 (HasMethodName
   RT_root_PKG_PKGorgapachecommonslang3math_CLS_CLSnumberutils_MTD_MTDmin
   "createbigdecimal")

198 (HasMethodParamTypes
   RT_root_PKG_PKGorgapachecommonslang3math_CLS_CLSnumberutils_MTD_MTDmin
   MethodParamTypes_MTDTYPESclassd)

199 (HasMethodModifier
   RT_root_PKG_PKGorgapachecommonslang3math_CLS_CLSnumberutils_MTD_MTDmin
   Static_Method)

200 ;; method name

201 (HasMethodName
   RT_root_PKG_PKGorgapachecommonslang3math_CLS_CLSnumberutils_MTD_MTDtobyte
   "tobyte")

202 (HasMethodParamTypes
   RT_root_PKG_PKGorgapachecommonslang3math_CLS_CLSnumberutils_MTD_MTDtobyte
   MethodParamTypes_MTDTYPESclassjavalongstringbyte)

203 (HasMethodModifier
   RT_root_PKG_PKGorgapachecommonslang3math_CLS_CLSnumberutils_MTD_MTDtobyte
   Static_Method)

204 )

205 (: goal (and
   (notPullingUpMethods)
   (SignaturePathTillPackage RT_root_PKG_PKGorgapachecommonslang3builder)
   (SignaturePathTillPackage RT_root_PKG_PKGorgapachecommonslang3concurrent)
   (SignaturePathTillPackage RT_root_PKG_PKGorgapachecommonslang3math)
   (SignaturePathTillClass RT_root_PKG_PKGorgapachecommonslang3builder_CLS_CLSidkeyren)
   (SignaturePathTillMethod RT_root_PKG_PKGorgapachecommonslang3builder_CLS_CLSidkeyren
    MTD_MTDequals Static_Method)
   (SignaturePathTillMethod RT_root_PKG_PKGorgapachecommonslang3builder_CLS_CLSidkeyren
    MTD_MTDhashcode MethodParamTypes_MTDTYPESclassjavalongobject)

206 )
APPENDIX D. Planning Results of Exp. 3

<table>
<thead>
<tr>
<th>Time 155530</th>
</tr>
</thead>
<tbody>
<tr>
<td>(RENAMECLASS RT_ROOT RT_ROOT PKG PKGORGAPACHECOMMONSLANG3CONCURRENT PKG PKGORGAPACHECOMMONSLANG3CONCURRENT CLS CLSMULTIBACKGROUNDINITIALIZER CLS CLSMULTIBACKGROUNDINITIALIZER)</td>
</tr>
<tr>
<td>(RENAMECLASS RT_ROOT RT_ROOT PKG PKGORGAPACHECOMMONSLANG3BUILDER PKG PKGORGAPACHECOMMONSLANG3BUILDER RT_ROOT PKG PKGORGAPACHECOMMONSLANG3BUILDER CLS CLSKEY CLS CLSKEY)</td>
</tr>
<tr>
<td>(RENAMEMETHOD RT_ROOT RT_ROOT PKG PKGORGAPACHECOMMONSLANG3MATH PKG PKGORGAPACHECOMMONSLANG3MATH RT_ROOT PKG PKGORGAPACHECOMMONSLANG3MATH CLS CLSNUMBERUTILS CLS CLSNUMBERUTILS RT_ROOT PKG PKGORGAPACHECOMMONSLANG3MATH CLS CLSNUMBERUTILS MTD MTD TOBYTE MTD MTD TOBYTE MTD MTD TOBYTE)</td>
</tr>
<tr>
<td>(RENAMEMETHOD RT_ROOT RT_ROOT PKG PKGORGAPACHECOMMONSLANG3MATH PKG PKGORGAPACHECOMMONSLANG3MATH RT_ROOT PKG PKGORGAPACHECOMMONSLANG3MATH CLS CLSNUMBERUTILS CLS CLSNUMBERUTILS RT_ROOT PKG PKGORGAPACHECOMMONSLANG3MATH CLS CLSNUMBERUTILS MTD MTD MIN MTD MTD MIN MTD MTD MIN)</td>
</tr>
<tr>
<td>(RENAMEMETHOD RT_ROOT RT_ROOT PKG PKGORGAPACHECOMMONSLANG3MATH PKG PKGORGAPACHECOMMONSLANG3MATH RT_ROOT PKG PKGORGAPACHECOMMONSLANG3MATH CLS CLSNUMBERUTILS CLS CLSNUMBERUTILS RT_ROOT PKG PKGORGAPACHECOMMONSLANG3MATH CLS CLSNUMBERUTILS MTD MTD MAX MTD MTD MAX MTD MTD MAX)</td>
</tr>
<tr>
<td>(RENAMEMETHOD RT_ROOT RT_ROOT PKG PKGORGAPACHECOMMONSLANG3MATH PKG PKGORGAPACHECOMMONSLANG3MATH RT_ROOT PKG PKGORGAPACHECOMMONSLANG3MATH CLS CLSNUMBERUTILS CLS CLSNUMBERUTILS RT_ROOT PKG PKGORGAPACHECOMMONSLANG3MATH CLS CLSNUMBERUTILS MTD MTD CREATE BIGDECIMAL MTD MTD CREATE BIGDECIMAL MTD MTD CREATE BIGDECIMAL)</td>
</tr>
</tbody>
</table>
APPENDIX E. Readable Planning Results of Exp. 3

2. rename class: org.apache.commons.lang3.builder>IDKey → org.apache.commons.lang3.builder.IDKey_REN
REFERENCES


